# Algorithms
{fundamental techniques}

Contents

Chapter 1

Introduction

This book covers techniques for the design and analysis of algorithms. The algorithmic techniques covered include: divide and conquer, backtracking, dynamic programming, greedy algorithms, and hill-climbing.

Any solvable problem generally has at least one algorithm of each of the following types:

1. the obvious way;
2. the methodical way;
3. the clever way; and
4. the miraculous way.

On the first and most basic level, the "obvious" solution might try to exhaustively search for the answer. Intuitively, the obvious solution is the one that comes easily if you're familiar with a programming language and the basic problem solving techniques.

The second level is the methodical level and is the heart of this book: after understanding the material presented here you should be able to methodically turn most obvious algorithms into better performing algorithms.

The third level, the clever level, requires more understanding of the elements involved in the problem and their properties or even a reformulation of the algorithm (e.g., numerical algorithms exploit mathematical properties that are not obvious). A clever algorithm may be hard to understand by being non-obvious that it is correct, or it may be hard to understand that it actually runs faster than what it would seem to require.

The fourth and final level of an algorithmic solution is the miraculous level: this is reserved for the rare cases where a breakthrough results in a highly non-intuitive solution.

Naturally, all of these four levels are relative, and some clever algorithms are covered in this book as well, in addition to the methodical techniques. Let's begin.

1.1 Prerequisites

To understand the material presented in this book you need to know a programming language well enough to translate the pseudocode in this book into a working solution. You also need to know the basics about the following data structures: arrays, stacks, queues, linked-lists, trees, heaps (also called priority queues), disjoint sets, and graphs.

Additionally, you should know some basic algorithms like binary search, a sorting algorithm (merge sort, heap sort, insertion sort, or others), and breadth-first or depth-first search.

If you are unfamiliar with any of these prerequisites you should review the material in the *Data Structures* book first.

1.2 When is Efficiency Important?

Not every problem requires the most efficient solution available. For our purposes, the term efficient is concerned with the time and/or space needed to perform the task. When either time or space is abundant and cheap, it may not be worth it to pay a programmer to spend a day or so working to make a program faster.

However, here are some cases where efficiency matters:

- When resources are limited, a change in algorithms could create great savings and allow limited machines (like cell phones, embedded systems, and sensor networks) to be stretched to the frontier of possibility.
- When the data is large a more efficient solution can mean the difference between a task finishing in two days versus two weeks. Examples include physics, genetics, web searches, massive online stores, and network traffic analysis.
- Real time applications: the term "real time applications" actually refers to computations that give time guarantees, versus meaning "fast." However,

the quality can be increased further by choosing the appropriate algorithm.

- Computationally expensive jobs, like fluid dynamics, partial differential equations, VLSI design, and cryptanalysis can sometimes only be considered when the solution is found efficiently enough.

- When a subroutine is common and frequently used, time spent on a more efficient implementation can result in benefits for every application that uses the subroutine. Examples include sorting, searching, pseudorandom number generation, kernel operations (not to be confused with the operating system kernel), database queries, and graphics.

In short, it's important to save time when you do not have any time to spare.

When is efficiency unimportant? Examples of these cases include prototypes that are used only a few times, cases where the input is small, when simplicity and ease of maintenance is more important, when the area concerned is not the bottle neck, or when there's another process or area in the code that would benefit far more from efficient design and attention to the algorithm(s).

1.3 Inventing an Algorithm

Because we assume you have some knowledge of a programming language, let's start with how we translate an idea into an algorithm. Suppose you want to write a function that will take a string as input and output the string in lowercase:

// tolower -- translates all alphabetic, uppercase characters in str to lowercase function **tolower**(string *str*): string

What first comes to your mind when you think about solving this problem? Perhaps these two considerations crossed your mind:

1. Every character in *str* needs to be looked at
2. A routine for converting a single character to lower case is required

The first point is "obvious" because a character that needs to be converted might appear anywhere in the string. The second point follows from the first because, once we consider each character, we need to do something with it. There are many ways of writing the **tolower** function for characters:

function **tolower**(character *c*): character

There are several ways to implement this function, including:

- look *c* up in a table -- a character indexed array of characters that holds the lowercase version of each character.

- check if *c* is in the range 'A' ≤ *c* ≤ 'Z', and then add a numerical offset to it.

These techniques depend upon the character encoding. (As an issue of separation of concerns, perhaps the table solution is stronger because it's clearer you only need to change one part of the code.)

However such a subroutine is implemented, once we have it, the implementation of our original problem comes immediately:

// tolower -- translates all alphabetic, uppercase characters in str to lowercase function **tolower**(string *str*): string let *result* := "" for-each *c* in *str*: *result*.append(**tolower**(*c*)) repeat return *result* end
This code sample is also available in Ada.

The loop is the result of our ability to translate "every character needs to be looked at" into our native programming language. It became obvious that the **tolower** subroutine call should be in the loop's body. The final step required to bring the high-level task into an implementation was deciding how to build the resulting string. Here, we chose to start with the empty string and append characters to the end of it.

Now suppose you want to write a function for comparing two strings that tests if they are equal, ignoring case:

// equal-ignore-case -- returns true if s and t are equal, ignoring case function **equal-ignore-case**(string *s*, string *t*): boolean

These ideas might come to mind:

1. Every character in strings *s* and *t* will have to be looked at
2. A single loop iterating through both might accomplish this
3. But such a loop should be careful that the strings are of equal length first
4. If the strings aren't the same length, then they cannot be equal because the consideration of ignoring case doesn't affect how long the string is
5. A tolower subroutine for characters can be used again, and only the lowercase versions will be compared

These ideas come from familiarity both with strings and with the looping and conditional constructs in your language. The function you thought of may have looked something like this:

```
// equal-ignore-case -- returns true if s or t are equal, ignoring case
function equal-ignore-case(string s[1..n], string t[1..m]): boolean
  if n != m:
    return false \if they aren't the same length, they aren't equal\
  fi
  for i := 1 to n:
    if tolower(s[i]) != tolower(t[i]):
      return false
    fi
  repeat
  return true
end
```

This code sample is also available in Ada.

Or, if you thought of the problem in terms of functional decomposition instead of iterations, you might have thought of a function more like this:

```
// equal-ignore-case -- returns true if s or t are equal, ignoring case
function equal-ignore-case(string s, string t): boolean
  return tolower(s).equals(tolower(t))
end
```

Alternatively, you may feel neither of these solutions is efficient enough, and you would prefer an algorithm that only ever made one pass of *s* or *t*. The above two implementations each require two-passes: the first version computes the lengths and then compares each character, while the second version computes the lowercase versions of the string and then compares the results to each other. (Note that for a pair of strings, it is also possible to have the length precomputed to avoid the second pass, but that can have its own drawbacks at times.) You could imagine how similar routines can be written to test string equality that not only ignore case, but also ignore accents.

Already you might be getting the spirit of the pseudocode in this book. The pseudocode language is not meant to be a real programming language: it abstracts away details that you would have to contend with in any language. For example, the language doesn't assume generic types or dynamic versus static types: the idea is that it should be clear what is intended and it should not be too hard to convert it to your native language. (However, in doing so, you might have to make some design decisions that limit the implementation to one particular type or form of data.)

There was nothing special about the techniques we used so far to solve these simple string problems: such techniques are perhaps already in your toolbox, and you may have found better or more elegant ways of expressing the solutions in your programming language of choice. In this book, we explore general algorithmic techniques to expand your toolbox even further. Taking a naive algorithm and making it more efficient might not come so immediately, but after understanding the material in this book you should be able to methodically apply different solutions, and, most importantly, you will be able to ask yourself more questions about your programs. Asking questions can be just as important as answering questions, because asking the right question can help you reformulate the problem and think outside of the box.

1.4 Understanding an Algorithm

Computer programmers need an excellent ability to reason with multiple-layered abstractions. For example, consider the following code:

```
function foo(integer a):
  if (a / 2) * 2 == a:
    print "The value " a " is even."
  fi
end
```

To understand this example, you need to know that integer division uses truncation and therefore when the if-condition is true then the least-significant bit in *a* is zero (which means that *a* must be even). Additionally, the code uses a string printing API and is itself the definition of a function to be used by different modules. Depending on the programming task, you may think on the layer of hardware, on down to the level of processor branch-prediction or the cache.

Often an understanding of binary is crucial, but many modern languages have abstractions far enough away “from the hardware” that these lower-levels are not necessary. Somewhere the abstraction stops: most programmers don't need to think about logic gates, nor is the physics of electronics necessary. Nevertheless, an essential part of programming is multiple-layer thinking.

But stepping away from computer programs toward algorithms requires another layer: mathematics. A program may exploit properties of binary representations. An algorithm can exploit properties of set theory or other mathematical constructs. Just as binary itself is not explicit in a program, the mathematical properties used in an algorithm are not explicit.

Typically, when an algorithm is introduced, a discussion (separate from the code) is needed to explain the mathematics used by the algorithm. For example, to really understand a greedy algorithm (such as Dijkstra’s algorithm) you should understand the mathematical properties that show how the greedy strategy is valid for all cases. In a way, you can think of the mathematics as its own kind of subroutine that the algorithm invokes. But this “subroutine” is not present in the code because there’s nothing to call. As you read this book try to think about mathematics as an implicit subroutine.

1.5 Overview of the Techniques

The techniques this book covers are highlighted in the following overview.

- **Divide and Conquer**: Many problems, particularly when the input is given in an array, can be solved by cutting the problem into smaller pieces (*divide*), solving the smaller parts recursively (*conquer*), and then combining the solutions into a single result. Examples include the merge sort and quicksort algorithms.

- **Randomization**: Increasingly, randomization techniques are important for many applications. This chapter presents some classical algorithms that make use of random numbers.

- **Backtracking**: Almost any problem can be cast in some form as a backtracking algorithm. In backtracking, you consider all possible choices to solve a problem and recursively solve subproblems under the assumption that the choice is taken. The set of recursive calls generates a tree in which each set of choices in the tree is considered consecutively. Consequently, if a solution exists, it will eventually be found.

  Backtracking is generally an inefficient, brute-force technique, but there are optimizations that can be performed to reduce both the depth of the tree and the number of branches. The technique is called backtracking because after one leaf of the tree is visited, the algorithm will go back up the call stack (undoing choices that didn't lead to success), and then proceed down some other branch. To be solved with backtracking techniques, a problem needs to have some form of "self-similarity," that is, smaller instances of the problem (after a choice has been made) must resemble the original problem. Usually, problems can be generalized to become self-similar.

- **Dynamic Programming**: Dynamic programming is an optimization technique for backtracking algorithms. When subproblems need to be solved repeatedly (i.e., when there are many duplicate branches in the backtracking algorithm) time can be saved by solving all of the subproblems first (bottom-up, from smallest to largest) and storing the solution to each subproblem in a table. Thus, each subproblem is only visited and solved once instead of repeatedly. The "programming" in this technique's name comes from programming in the sense of writing things down in a table; for example, television programming is making a table of what shows will be broadcast when.

- **Greedy Algorithms**: A greedy algorithm can be useful when enough information is known about possible choices that "the best" choice can be determined without considering all possible choices. Typically, greedy algorithms are not challenging to write, but they are difficult to prove correct.

- **Hill Climbing**: The final technique we explore is hill climbing. The basic idea is to start with a poor solution to a problem, and then repeatedly apply optimizations to that solution until it becomes optimal or meets some other requirement. An important case of hill climbing is network flow. Despite the name, network flow is useful for many problems that describe relationships, so it's not just for computer networks. Many matching problems can be solved using network flow.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

1.6 Algorithm and code example

1.6.1 Level 1 (easiest)

1. *Find maximum* With algorithm and several different programming languages

2. *Find minimum* With algorithm and several different programming languages

3. *Find average* With algorithm and several different programming languages

4. *Find mode* With algorithm and several different programming languages

5. *Find total* With algorithm and several different programming languages

6. *Counting* With algorithm and several different programming languages

1.6.2 Level 2

1. *Talking to computer Lv 1* With algorithm and several different programming languages

2. *Sorting-bubble sort* With algorithm and several different programming languages

1.6.3 Level 3

1. *Talking to computer Lv 2* With algorithm and several different programming languages

1.6.4 Level 4

1. *Talking to computer Lv 3* With algorithm and several different programming languages

2. *Find approximate maximum* With algorithm and several different programming languages

1.6.5 Level 5

1. *Quicksort*

Chapter 2

Mathematical Background

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

Before we begin learning algorithmic techniques, we take a detour to give ourselves some necessary mathematical tools. First, we cover mathematical definitions of terms that are used later on in the book. By expanding your mathematical vocabulary you can be more precise and you can state or formulate problems more simply. Following that, we cover techniques for analysing the running time of an algorithm. After each major algorithm covered in this book we give an analysis of its running time as well as a proof of its correctness

2.1 Asymptotic Notation

In addition to correctness another important characteristic of a useful algorithm is its time and memory consumption. Time and memory are both valuable resources and there are important differences (even when both are abundant) in how we can use them.

How can you measure resource consumption? One way is to create a function that describes the usage in terms of some characteristic of the input. One commonly used characteristic of an input dataset is its size. For example, suppose an algorithm takes an input as an array of n integers. We can describe the time this algorithm takes as a function f written in terms of n . For example, we might write:

$$f(n) = n^2 + 3n + 14$$

where the value of $f(n)$ is some unit of time (in this discussion the main focus will be on time, but we could do the same for memory consumption). Rarely are the units of time actually in seconds, because that would depend on the machine itself, the system it's running, and its load. Instead, the units of time typically used are in terms of the number of some fundamental operation performed. For example, some fundamental operations we might care about are: the number of additions or multiplications needed; the number of element comparisons; the number of memory-location swaps performed; or the raw number of machine instructions executed. In general we might just refer to these fundamental operations performed as steps taken.

Is this a good approach to determine an algorithm's resource consumption? Yes and no. When two different algorithms are similar in time consumption a precise function might help to determine which algorithm is faster under given conditions. But in many cases it is either difficult or impossible to calculate an analytical description of the exact number of operations needed, especially when the algorithm performs operations conditionally on the values of its input. Instead, what really is important is not the precise time required to complete the function, but rather the degree that resource consumption changes depending on its inputs. Concretely, consider these two functions, representing the computation time required for each size of input dataset:

$$f(n) = n^3 - 12n^2 + 20n + 110$$

$$g(n) = n^3 + n^2 + 5n + 5$$

They look quite different, but how do they behave? Let's look at a few plots of the function ($f(n)$ is in red, $g(n)$ in blue):

In the first, very-limited plot the curves appear somewhat different. In the second plot they start going in sort of the same way, in the third there is only a very small difference, and at last they are virtually identical. In fact, they approach n^3 , the dominant term. As n gets larger, the other terms become much less significant in comparison to n^3.

As you can see, modifying a polynomial-time algorithm's low-order coefficients doesn't help much. What really matters is the highest-order coefficient. This is why we've adopted a notation for this kind of analysis. We say that:

$$f(n) = n^3 - 12n^2 + 20n + 110 = O(n^3)$$

We ignore the low-order terms. We can say that:

$$O(\log n) \leq O(\sqrt{n}) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n)$$

This gives us a way to more easily compare algorithms with each other. Running an insertion sort on n elements takes steps on the order of $O(n^2)$. Merge sort sorts in $O(n \log n)$ steps. Therefore, once the input dataset is large enough, merge sort is faster than insertion sort.

In general, we write

$$f(n) = O(g(n))$$

when

$$\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n).$$

That is, $f(n) = O(g(n))$ holds if and only if there exists some constants c and n_0 such that for all $n > n_0$ $f(n)$ is positive and less than or equal to $cg(n)$.

Note that the equal sign used in this notation describes a relationship between $f(n)$ and $g(n)$ instead of reflecting a true equality. In light of this, some define Big-O in terms of a set, stating that:

$$f(n) \in O(g(n))$$

when

$$f(n) \in \{f(n) : \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq c{\cdot}g(n)\}.$$

Big-O notation is only an upper bound; these two are both true:

$$n^3 = O(n^4)$$

$$n^4 = O(n^4)$$

If we use the equal sign as an equality we can get very strange results, such as:

$$n^3 = n^4$$

which is obviously nonsense. This is why the set-definition is handy. You can avoid these things by thinking of the equal sign as a one-way equality, i.e.:

$$n^3 = O(n^4)$$

does not imply

$$O(n^4) = n^3$$

Always keep the O on the right hand side.

2.1.1 Big Omega

Sometimes, we want more than an upper bound on the behavior of a certain function. Big Omega provides a lower bound. In general, we say that

$$f(n) = \Omega(g(n))$$

when

$$\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n).$$

i.e. $f(n) = \Omega(g(n))$ if and only if there exist constants c and n_0 such that for all n>n_0 f(n) is positive and **greater** than or equal to cg(n).

So, for example, we can say that

$n^2 - 2n = \Omega(n^2)$, (c=1/2, n_0=4) or
$n^2 - 2n = \Omega(n)$, (c=1, n_0=3),

but it is false to claim that

$$n^2 - 2n = \Omega(n^3).$$

2.1.2 Big Theta

When a given function is both O(g(n)) and Ω(g(n)), we say it is Θ(g(n)), and we have a tight bound on the function. A function f(n) is Θ(g(n)) when

$$\exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c_1{\cdot}g(n) \leq f(n) \leq c_2{\cdot}g(n),$$

but most of the time, when we're trying to prove that a given $f(n) = \Theta(g(n))$, instead of using this definition, we just show that it is both O(g(n)) and Ω(g(n)).

2.1.3 Little-O and Omega

When the asymptotic bound is not tight, we can express this by saying that $f(n) = o(g(n))$ or $f(n) = \omega(g(n))$. The definitions are:

f(n) is o(g(n)) iff $\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 :$ $0 \leq f(n) < c \cdot g(n)$ and

f(n) is ω(g(n)) iff $\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 :$ $0 \leq c \cdot g(n) < f(n)$.

Note that a function f is in o(g(n)) when for any coefficient of g, g eventually gets larger than f, while for O(g(n)), there only has to exist a single coefficient for which g eventually gets at least as big as f.

2.1.4 Big-O with multiple variables

Given a functions with two parameters $f(n,m)$ and $g(n,m)$,

$f(n,m) = O(g(n,m))$ if and only if $\exists c > 0, \exists n_0 > 0, \exists m_0 > 0, \forall n \geq n_0, \forall m \geq m_0 : 0 \leq f(n,m) \leq c \cdot g(n,m)$.

For example, $5n + 3m = O(n+m)$, and $n + 10m + 3nm = O(nm)$.

2.2 Algorithm Analysis: Solving Recurrence Equations

Merge sort of n elements: $T(n) = 2*T(n/2)+c(n)$ This describes one iteration of the merge sort: the problem space n is reduced to two halves ($2*T(n/2)$), and then merged back together at the end of all the recursive calls ($c(n)$). This notation system is the bread and butter of algorithm analysis, so get used to it.

There are some theorems you can use to estimate the big Oh time for a function if its recurrence equation fits a certain pattern.

[TODO: write this section]

2.2.1 Substitution method

Formulate a guess about the big Oh time of your equation. Then use proof by induction to prove the guess is correct.

[TODO: write this section]

2.2.2 Summations

[TODO: show the closed forms of commonly needed summations and prove them]

2.2.3 Draw the Tree and Table

This is really just a way of getting an intelligent guess. You still have to go back to the substitution method in order to prove the big Oh time.

[TODO: write this section]

2.2.4 The Master Theorem

Consider a recurrence equation that fits the following formula:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^k)$$

for $a \geq 1$, $b > 1$ and $k \geq 0$. Here, a is the number of recursive calls made per call to the function, n is the input size, b is how much smaller the input gets, and k is the polynomial order of an operation that occurs each time the function is called (except for the base cases). For example, in the merge sort algorithm covered later, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

because two subproblems are called for each non-base case iteration, and the size of the array is divided in half each time. The $O(n)$ at the end is the "conquer" part of this divide and conquer algorithm: it takes linear time to merge the results from the two recursive calls into the final result.

Thinking of the recursive calls of T as forming a tree, there are three possible cases to determine where most of the algorithm is spending its time ("most" in this sense is concerned with its asymptotic behaviour):

1. the tree can be **top heavy**, and most time is spent during the initial calls near the root;
2. the tree can have a **steady state**, where time is spread evenly; or
3. the tree can be **bottom heavy**, and most time is spent in the calls near the leaves

Depending upon which of these three states the tree is in T will have different complexities:

The Master Theorem

Given $T(n) = aT\left(\frac{n}{b}\right) + O(n^k)$ for $a \geq 1$, $b > 1$ and $k \geq 0$:

- If $a < b^k$, then $T(n) = O(n^k)$ (top heavy)
- If $a = b^k$, then $T(n) = O(n^k \cdot \log n)$ (steady state)
- If $a > b^k$, then $T(n) = O(n^{\log_b a})$ (bottom heavy)

For the merge sort example above, where

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

we have

$$a = 2, b = 2, k = 1 \implies b^k = 2$$

thus, $a = b^k$ and so this is also in the "steady state": By the master theorem, the complexity of merge sort is thus

$$T(n) = O(n^1 \log n) = O(n \log n)$$

2.3 Amortized Analysis

[Start with an adjacency list representation of a graph and show two nested for loops: one for each node n, and nested inside that one loop for each edge e. If there are n nodes and m edges, this could lead you to say the loop takes O(nm) time. However, only once could the inner-loop take that long, and a tighter bound is O(n+m).]

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

Chapter 3

Divide and Conquer

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

The first major algorithmic technique we cover is **divide and conquer**. Part of the trick of making a good divide and conquer algorithm is determining how a given problem could be separated into two or more similar, but smaller, subproblems. More generally, when we are creating a divide and conquer algorithm we will take the following steps:

The first algorithm we'll present using this methodology is the merge sort.

3.1 Merge Sort

The problem that **merge sort** solves is general sorting: given an unordered array of elements that have a total ordering, create an array that has the same elements sorted. More precisely, for an array *a* with indexes 1 through *n*, if the condition

for all i, j such that $1 \le i < j \le n$ then $a[i] \le a[j]$

holds, then *a* is said to be **sorted**. Here is the interface:

// sort -- returns a sorted copy of array a function **sort**(array *a*): array

Following the divide and conquer methodology, how can *a* be broken up into smaller subproblems? Because *a* is an array of *n* elements, we might want to start by breaking the array into two arrays of size *n*/2 elements. These smaller arrays will also be unsorted and it is meaningful to sort these smaller problems; thus we can consider these smaller arrays "similar". Ignoring the base case for a moment, this reduces the problem into a different one: Given two sorted arrays, how can they be combined to form a single sorted array that contains all the elements of both given arrays:

// merge -- given a and b (assumed to be sorted) returns a merged array that // preserves order function **merge**(array *a*, array *b*): array

So far, following the methodology has led us to this point, but what about the base case? The base case is the part of the algorithm concerned with what happens when the problem cannot be broken into smaller subproblems. Here, the base case is when the array only has one element. The following is a sorting algorithm that faithfully sorts arrays of only zero or one elements:

// base-sort -- given an array of one element (or empty), return a copy of the // array sorted function **base-sort**(array *a*[1..*n*]): array assert (*n* <= 1) return *a*.copy() end

Putting this together, here is what the methodology has told us to write so far:

// sort -- returns a sorted copy of array a function **sort**(array *a*[1..*n*]): array if *n* <= 1: return *a*.copy() else: let *sub_size* := *n* / 2 let *first_half* := **sort**(*a*[1,..,*sub_size*]) let *second_half* := **sort**(*a*[*sub_size* + 1,..,*n*]) return **merge**(*first_half*, *second_half*) fi end

And, other than the unimplemented merge subroutine, this sorting algorithm is done! Before we cover how this algorithm works, here is how merge can be written:

// merge -- given a and b (assumed to be sorted) returns a merged array that // preserves order function **merge**(array *a*[1..*n*], array *b*[1..*m*]): array let *result* := new array[*n* + *m*] let *i*, *j* := 1 for *k* := 1 to *n* + *m*: if *i* >= *n*: *result*[*k*] := *b*[*j*]; *j* += 1 else-if *j* >= *m*: *result*[*k*] := *a*[*i*]; *i* += 1 else: if *a*[*i*] < *b*[*j*]: *result*[*k*] := *a*[*i*]; *i* += 1 else: *result*[*k*] := *b*[*j*]; *j* += 1 fi fi repeat end

[TODO: how it works; including correctness proof] This algorithm uses the fact that, given two sorted arrays, the smallest element is always in one of two places. It's either at the head of the first array, or the head of the second.

3.1.1 Analysis

Let $T(n)$ be the number of steps the algorithm takes to run on input of size n .

Merging takes linear time and we recurse each time on two sub-problems of half the original size, so

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n).$$

By the master theorem, we see that this recurrence has a "steady state" tree. Thus, the runtime is:

$T(n) = O(n \cdot \log n)$.

This can be seen intuitivey by asking how may times does n need to be divided by 2 before the size of the array for sorting is 1? Why m times of course !

More directly, $2^m = n$, equivalent to $\log 2^m = \log n$, equivalent to m x $\log_2 2 = \log_2 n$, and since $\log_2 2 = 1$, equivalent to $m = \log_2 n$.

Since m is the number of halvings of an array before the array is chopped up into bite sized pieces of 1-element arrays, and then it will take m levels of merging a sub-array with its neighbor where the sum size of sub-arrays will be n at each level, it will be exactly n/2 comparisons for merging at each level, with m ($\log_2 n$) levels, thus O(n/2 x log n) <=> **O (n log n).**

3.1.2 Iterative Version

This merge sort algorithm can be turned into an iterative algorithm by iteratively merging each subsequent pair, then each group of four, et cetera. Due to a lack of function overhead, iterative algorithms tend to be faster in practice. However, because the recursive version's call tree is logarithmically deep, it does not require much run-time stack space: Even sorting 4 gigs of items would only require 32 call entries on the stack, a very modest amount considering if even each call required 256 bytes on the stack, it would only require 8 kilobytes.

The iterative version of mergesort is a minor modification to the recursive version - in fact we can reuse the earlier merging function. The algorithm works by merging small, sorted subsections of the original array to create larger subsections of the array which are sorted. To accomplish this, we iterate through the array with successively larger "strides".

```
// sort -- returns a sorted copy of array a
function sort_iterative(array a[1..n]): array
  let result := a.copy()
  for power := 0 to log2(n−1)
    let unit := 2^power
    for i := 1 to n by unit*2
      if i+unit−1 < n:
        let a1[1..unit] := result[i..i+unit−1]
        let a2[1..unit] := result[i+unit..min(i+unit*2-1, n)]
        result[i..i+unit*2-1] := merge(a1,a2)
      fi
    repeat
  repeat
  return result
end
```

This works because each sublist of length 1 in the array is, by definition, sorted. Each iteration through the array (using counting variable *i*) doubles the size of sorted sub-lists by merging adjacent sublists into sorted larger versions. The current size of sorted sublists in the algorithm is represented by the *unit* variable.

3.1.3 space inefficiency

Straight forward merge sort requires a space of 2 x n , n to store the 2 sorted smaller arrays , and n to store the final result of merging. But merge sort still lends itself for batching of merging.

3.2 Binary Search

Once an array is sorted, we can quickly locate items in the array by doing a binary search. Binary search is different from other divide and conquer algorithms in that it is mostly divide based (nothing needs to be conquered). The concept behind binary search will be useful for understanding the partition and quicksort algorithms, presented in the randomization chapter.

Finding an item in an already sorted array is similar to finding a name in a phonebook: you can start by flipping the book open toward the middle. If the name you're looking for is on that page, you stop. If you went too far, you can start the process again with the first half of the book. If the name you're searching for appears later than the page, you start from the second half of the book instead. You repeat this process, narrowing down your search space by half each time, until you find what you were looking for (or, alternatively, find where what you were looking for would have been if it were present).

The following algorithm states this procedure precisely:

```
// binary-search -- returns the index of value in the given array, or
// −1 if value cannot be found. Assumes array is sorted in ascending order
function binary-search(value, array A[1..n]): integer
  return search-inner(value, A, 1, n + 1)
end

// search-inner -- search subparts of the array; end is one past the
// last element
function search-inner(value, array A, start, end): integer
  if start == end:
    return −1 // not found
  fi
  let length := end - start
  if length == 1:
    if value == A[start]:
      return start
    else:
      return −1
    fi
  fi
  let mid := start + (length / 2)
  if value == A[mid]:
    return mid
  else-if value > A[mid]:
    return search-inner(value, A, mid + 1, end)
  else:
    return search-inner(value, A, start, mid)
  fi
end
```

Note that all recursive calls made are tail-calls, and thus the algorithm is iterative. We can explicitly remove the tail-calls if our programming language does not do that for us already by turning the argument values passed to the recursive call into assignments, and then looping to the top of the function body again:

```
// binary-search -- returns the index of value in the given array, or
// −1 if value cannot be found. Assumes array is sorted in ascending order
function binary-search(value, array A[1,..n]): integer
  let start := 1
  let end := n + 1
  loop:
    if start == end:
      return −1
    fi // not found
    let length := end - start
    if length == 1:
      if value == A[start]:
        return start
      else:
        return −1
      fi
    fi
    let mid := start + (length / 2)
    if value == A[mid]:
      return mid
    else-if value > A[mid]:
      start := mid + 1
    else:
      end := mid
    fi
  repeat
end
```

Even though we have an iterative algorithm, it's easier to reason about the recursive version. If the number of steps the algorithm takes is $T(n)$, then we have the following recurrence that defines $T(n)$:

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + O(1).$$

The size of each recursive call made is on half of the input size (n), and there is a constant amount of time spent outside of the recursion (i.e., computing *length* and *mid* will take the same amount of time, regardless of how many elements are in the array). By the master theorem, this recurrence has values $a = 1, b = 2, k = 0$, which is a “steady state” tree, and thus we use the steady state case that tells us that

$$T(n) = \Theta(n^k \cdot \log n) = \Theta(\log n).$$

Thus, this algorithm takes logarithmic time. Typically, even when *n* is large, it is safe to let the stack grow by $\log n$ activation records through recursive calls.

difficulty in initially correct binary search implementations

The article on wikipedia on Binary Search also mentions the difficulty in writing a correct binary search algorithm: for instance, the java Arrays.binarySearch(..) overloaded function implementation does an interative binary search which didn't work when large integers overflowed a simple expression of mid calculation mid = (end + start) / 2 i.e. end + start > max_positive_integer . Hence the above algorithm is more correct in using a length = end - start, and adding half length to start. The java binary Search algorithm gave a return value useful for finding the position of the nearest key greater than the search key, i.e. the position where the search key could be inserted.

i.e. it returns *- (keypos+1)* , if the search key wasn't found exactly, but an insertion point was needed for the search key (insertion_point = *-return_value - 1*). Looking at boundary values, an insertion point could be at the front of the list (ip = 0, return value = -1), to the position just after the last element, (ip = length(A), return value = *- length(A) - 1*) .

As an exercise, trying to implement this functionality on the above iterative binary search can be useful for further comprehension.

3.3 Integer Multiplication

If you want to perform arithmetic with small integers, you can simply use the built-in arithmetic hardware of your machine. However, if you wish to multiply integers larger than those that will fit into the standard “word” integer size of your computer, you will have to implement a multiplication algorithm in software or use a software implementation written by someone else. For example, RSA encryption needs to work with integers of very large size (that is, large relative to the 64-bit word size of many machines) and utilizes special multiplication algorithms.[1]

3.3.1 Grade School Multiplication

How do we represent a large, multi-word integer? We can have a binary representation by using an array (or an allocated block of memory) of words to represent the bits of the large integer. Suppose now that we have two integers, X and Y , and we want to multiply them together. For simplicity, let’s assume that both X and Y have n bits each (if one is shorter than the other, we can always pad on zeros at the beginning). The most basic way to multiply the integers is to use the grade school multiplication algorithm. This is even easier in binary, because we only multiply by 1 or 0:

x6 x5 x4 x3 x2 x1 x0 × y6 y5 y4 y3 y2 y1 y0 ---------------------- x6 x5 x4 x3 x2 x1 x0 (when y0 is 1; 0 otherwise) x6 x5 x4 x3 x2 x1 x0 0 (when y1 is 1; 0 otherwise) x6 x5 x4 x3 x2 x1 x0 0 0 (when y2 is 1; 0 otherwise) x6 x5 x4 x3 x2 x1 x0 0 0 0 (when y3 is 1; 0 otherwise) ... et cetera

As an algorithm, here’s what multiplication would look like:

// multiply -- return the product of two binary integers, both of length n function **multiply**(bitarray *x*[1,..*n*], bitarray *y*[1,..*n*]): bitarray bitarray *p* = 0 for *i*:=1 to *n*: if *y*[*i*] == 1: *p* := **add**(*p*, *x*) fi *x* := **pad**(*x*, 0) *// add another zero to the end of x* repeat return *p* end

The subroutine **add** adds two binary integers and returns the result, and the subroutine **pad** adds an extra digit to the end of the number (padding on a zero is the same thing as shifting the number to the left; which is the same as multiplying it by two). Here, we loop *n* times, and in the worst-case, we make *n* calls to **add**. The numbers given to **add** will at most be of length $2n$. Further, we can expect that the **add** subroutine can be done in linear time. Thus, if *n* calls to a $O(n)$ subroutine are made, then the algorithm takes $O(n^2)$ time.

3.3.2 Divide and Conquer Multiplication

As you may have figured, this isn't the end of the story. We've presented the “obvious” algorithm for multiplication; so let’s see if a divide and conquer strategy can give us something better. One route we might want to try is breaking the integers up into two parts. For example, the integer *x* could be divided into two parts, x_h and x_l , for the high-order and low-order halves of x . For example, if x has *n* bits, we have

$$x = x_h \cdot 2^{n/2} + x_l$$

We could do the same for y :

$$y = y_h \cdot 2^{n/2} + y_l$$

But from this division into smaller parts, it's not clear how we can multiply these parts such that we can combine the results for the solution to the main problem. First, let's write out $x \times y$ would be in such a system:

$$x \times y = x_h \times y_h \cdot (2^{n/2})^2 + (x_h \times y_l + x_l \times y_h) \cdot (2^{n/2}) + x_l \times y_l$$

This comes from simply multiplying the new hi/lo representations of x and y together. The multiplication of the smaller pieces are marked by the " $\times$ " symbol. Note that the multiplies by $2^{n/2}$ and $(2^{n/2})^2 = 2^n$ does not require a real multiplication: we can just pad on the right number of zeros instead. This suggests the following divide and conquer algorithm:

```
// multiply -- return the product of two binary integers, both of length n
function multiply(bitarray x[1,..n], bitarray y[1,..n]): bitarray
if n == 1: return x[1] * y[1] fi // multiply single digits: O(1)
let xh := x[n/2 + 1, .., n] // array slicing, O(n)
let xl := x[0, .., n / 2] // array slicing, O(n)
let yh := y[n/2 + 1, .., n] // array slicing, O(n)
let yl := y[0, .., n / 2] // array slicing, O(n)
let a := multiply(xh, yh) // recursive call; T(n/2)
let b := multiply(xh, yl) // recursive call; T(n/2)
let c := multiply(xl, yh) // recursive call; T(n/2)
let d := multiply(xl, yl) // recursive call; T(n/2)
b := add(b, c) // regular addition; O(n)
a := shift(a, n) // pad on zeros; O(n)
b := shift(b, n/2) // pad on zeros; O(n)
return add(a, b, d) // regular addition; O(n)
end
```

We can use the master theorem to analyze the running time of this algorithm. Assuming that the algorithm's running time is $T(n)$, the comments show how much time each step takes. Because there are four recursive calls, each with an input of size $n/2$, we have:

$$T(n) = 4T(n/2) + O(n)$$

Here, $a = 4, b = 2, k = 1$, and given that $4 > 2^1$ we are in the "bottom heavy" case and thus plugging in these values into the bottom heavy case of the master theorem gives us:

$$T(n) = O(n^{\log_2 4}) = O(n^2).$$

Thus, after all of that hard work, we're still no better off than the grade school algorithm! Luckily, numbers and polynomials are a data set we know additional information about. In fact, we can reduce the running time by doing some mathematical tricks.

First, let's replace the $2^{n/2}$ with a variable, z:

$$x \times y = x_h * y_h z^2 + (x_h * y_l + x_l * y_h) z + x_l * y_l$$

This appears to be a quadratic formula, and we know that you only need three co-efficients or points on a graph in order to uniquely describe a quadratic formula. However, in our above algorithm we've been using four multiplications total. Let's try recasting x and y as linear functions:

$$P_x(z) = x_h \cdot z + x_l$$

$$P_y(z) = y_h \cdot z + y_l$$

Now, for $x \times y$ we just need to compute $(P_x \cdot P_y)(2^{n/2})$. We'll evaluate $P_x(z)$ and $P_y(z)$ at three points. Three convenient points to evaluate the function will be at $(P_x \cdot P_y)(1), (P_x \cdot P_y)(0), (P_x \cdot P_y)(-1)$:

[TODO: show how to make the two-parts breaking more efficient; then mention that the best multiplication uses the FFT, but don't actually cover that topic (which is saved for the advanced book)]

3.4 Base Conversion

[TODO: Convert numbers from decimal to binary quickly using DnC.]

Along with the binary, the science of computers employs bases 8 and 16 for it's very easy to convert between the three while using bases 8 and 16 shortens considerably number representations.

To represent 8 first digits in the binary system we need 3 bits. Thus we have, 0=000, 1=001, 2=010, 3=011, 4=100, 5=101, 6=110, 7=111. Assume M=$(2065)_8$. In order to obtain its binary representation, replace each of the four digits with the corresponding triple of bits: 010 000 110 101. After removing the leading zeros, binary representation is immediate: M=$(10000110101)_2$. (For the hexadecimal system conversion is quite similar, except that now one should use 4-bit representation of numbers below 16.) This fact follows from the general conversion algorithm and the observation that 8= 2^3 (and, of course, 16= 2^4). Thus it appears that the shortest way to convert numbers into the binary system is to first convert them into either octal or hexadecimal representation. Now let see how to implement the general algorithm programmatically.

For the sake of reference, representation of a number in a system with base (radix) N may only consist of digits that are less than N.

More accurately, if

$$(1) M = a_k N^k + a_{k-1} N^{k-1} + ... + a_1 N^1 + a_0$$

with $0 <= a_i < N$ we have a representation of M in base N system and write

$M = (a_k a_{k-1} ... a_0)N$

If we rewrite (1) as

$(2) M = a_0 + N * (a_1 + N * (a_2 + N * ...))$

the algorithm for obtaining coefficients ai becomes more obvious. For example, $a_0 = M\ modulo\ n$ and $a_1 = (M/N)\ modulo\ n$, and so on.

3.4.1 Recursive Implementation

Let's represent the algorithm mnemonically: (result is a string or character variable where I shall accumulate the digits of the result one at a time)

result = "" if M < N, result = 'M' + result. Stop. S = M mod N, result = 'S' + result M = M/N goto 2

A few words of explanation.

"" is an empty string. You may remember it's a zero element for string concatenation. Here we check whether the conversion procedure is over. It's over if M is less than N in which case M is a digit (with some qualification for N>10) and no additional action is necessary. Just prepend it in front of all other digits obtained previously. The '+' plus sign stands for the string concatenation. If we got this far, M is not less than N. First we extract its remainder of division by N, prepend this digit to the result as described previously, and reassign M to be M/N. This says that the whole process should be repeated starting with step 2. I would like to have a function say called Conversion that takes two arguments M and N and returns representation of the number M in base N. The function might look like this

1 String Conversion(int M, int N) // return string, accept two integers 2 { 3 if (M < N) // see if it's time to return 4 return new String(""+M); // ""+M makes a string out of a digit 5 else // the time is not yet ripe 6 return Conversion(M/N, N) + new String(""+(M mod N)); // continue 7 }

This is virtually a working Java function and it would look very much the same in C++ and require only a slight modification for C. As you see, at some point the function calls itself with a different first argument. One may say that the function is defined in terms of itself. Such functions are called recursive. (The best known recursive function is factorial: n!=n*(n-1)!.) The function calls (applies) itself to its arguments, and then (naturally) applies itself to its new arguments, and then ... and so on. We can be sure that the process will eventually stop because the sequence of arguments (the first ones) is decreasing. Thus sooner or later the first argument will be less than the second and the process will start emerging from the recursion, still a step at a time.

3.4.2 Iterative Implementation

Not all programming languages allow functions to call themselves recursively. Recursive functions may also be undesirable if process interruption might be expected for whatever reason. For example, in the Tower of Hanoi puzzle, the user may want to interrupt the demonstration being eager to test his or her understanding of the solution. There are complications due to the manner in which computers execute programs when one wishes to jump out of several levels of recursive calls.

Note however that the string produced by the conversion algorithm is obtained in the wrong order: all digits are computed first and then written into the string the last digit first. Recursive implementation easily got around this difficulty. With each invocation of the Conversion function, computer creates a new environment in which passed values of M, N, and the newly computed S are stored. Completing the function call, i.e. returning from the function we find the environment as it was before the call. Recursive functions store a sequence of computations implicitly. Eliminating recursive calls implies that we must manage to store the computed digits explicitly and then retrieve them in the reversed order.

In Computer Science such a mechanism is known as LIFO - Last In First Out. It's best implemented with a stack data structure. Stack admits only two operations: push and pop. Intuitively stack can be visualized as indeed a stack of objects. Objects are stacked on top of each other so that to retrieve an object one has to remove all the objects above the needed one. Obviously the only object available for immediate removal is the top one, i.e. the one that got on the stack last.

Then iterative implementation of the Conversion function might look as the following.

1 String Conversion(int M, int N) // return string, accept two integers 2 { 3 Stack stack = new Stack(); // create a stack 4 while (M >= N) // now the repetitive loop is clearly seen 5 { 6 stack.push(M mod N); // store a digit 7 M = M/N; // find new M 8 } 9 // now it's time to collect the digits together 10 String str = new String(""+M); // create a string with a single digit M 11 while (stack.NotEmpty()) 12 str = str+stack.pop() // get from the stack next digit 13 return str; 14 }

The function is by far longer than its recursive counterpart; but, as I said, sometimes it's the one you want to use, and sometimes it's the only one you may actually use.

3.5 Closest Pair of Points

For a set of points on a two-dimensional plane, if you want to find the closest two points, you could compare all of them to each other, at $O(n^2)$ time, or use a divide and conquer algorithm.

[TODO: explain the algorithm, and show the n^2 algorithm]

[TODO: write the algorithm, include intuition, proof of correctness, and runtime analysis]

Use this link for the original document.

http://www.cs.mcgill.ca/~{ }cs251/ClosestPair/ClosestPairDQ.html

3.6 Closest Pair: A Divide-and-Conquer Approach

3.6.1 Introduction

The brute force approach to the closest pair problem (i.e. checking every possible pair of points) takes quadratic time. We would now like to introduce a faster divide-and-conquer algorithm for solving the closest pair problem. Given a set of points in the plane S, our approach will be to split the set into two roughly equal halves (S1 and S2) for which we already have the solutions, and then to merge the halves in linear time to yield an O(nlogn) algorithm. However, the actual solution is far from obvious. It is possible that the desired pair might have one point in S1 and one in S2, does this not force us once again to check all possible pairs of points? The divide-and-conquer approach presented here generalizes directly from the one dimensional algorithm we presented in the previous section.

3.6.2 Closest Pair in the Plane

Alright, we'll generalize our 1-D algorithm as directly as possible (see figure 3.2). Given a set of points S in the plane, we partition it into two subsets S1 and S2 by a vertical line l such that the points in S1 are to the left of l and those in S2 are to the right of l.

We now recursively solve the problem on these two sets obtaining minimum distances of d1 (for S1), and d2 (for S2). We let d be the minimum of these.

Now, identical to the 1-D case, if the closes pair of the whole set consists of one point from each subset, then these two points must be within d of l. This area is represented as the two strips P1 and P2 on either side of l

Up to now, we are completely in step with the 1-D case. At this point, however, the extra dimension causes some problems. We wish to determine if some point in say P1 is less than d away from another point in P2. However, in the plane, we don't have the luxury that we had on the line when we observed that only one point in each set can be within d of the median. In fact, in two dimensions, all of the points could be in the strip! This is disastrous, because we would have to compare n2 pairs of points to merge the set, and hence our divide-and-conquer algorithm wouldn't save us anything in terms of efficiency. Thankfully, we can make another life saving observation at this point. For any particular point p in one strip, only points that meet the following constraints in the other strip need to be checked:

- those points within d of p in the direction of the other strip
- those within d of p in the positive and negative y directions

Simply because points outside of this bounding box cannot be less than d units from p (see figure 3.3). It just so happens that because every point in this box is at least d apart, there can be at most six points within it.

Now we don't need to check all n2 points. All we have to do is sort the points in the strip by their y-coordinates and scan the points in order, checking each point against a maximum of 6 of its neighbors. This means at most 6*n comparisons are required to check all candidate pairs. However, since we sorted the points in the strip by their y-coordinates the process of merging our two subsets is not linear, but in fact takes O(nlogn) time. Hence our full algorithm is not yet O(nlogn), but it is still an improvement on the quadratic performance of the brute force approach (as we shall see in the next section). In section 3.4, we will demonstrate how to make this algorithm even more efficient by strengthening our recursive sub-solution.

3.6.3 Summary and Analysis of the 2-D Algorithm

We present here a step by step summary of the algorithm presented in the previous section, followed by a performance analysis. The algorithm is simply written in list form because I find pseudo-code to be burdensome and unnecessary when trying to understand an algorithm. Note that we pre-sort the points according to their x coordinates, and maintain another structure which holds the points sorted by their y values(for step 4), which in itself takes O(nlogn) time.

ClosestPair of a set of points:

1. Divide the set into two equal sized parts by the line l, and recursively compute the minimal distance in each part.
2. Let d be the minimal of the two minimal distances.

3. Eliminate points that lie farther than d apart from l.
4. Consider the remaining points according to their y-coordinates, which we have precomputed.
5. Scan the remaining points in the y order and compute the distances of each point to all of its neighbors that are distanced no more than d(that's why we need it sorted according to y). Note that there are no more than 5(there is no figure 3.3 , so this 5 or 6 doesnt make sense without that figure . Please include it .) such points(see previous section).
6. If any of these distances is less than d then update d.

Analysis:

- Let us note T(n) as the efficiency of out algorithm
- Step 1 takes 2T(n/2) (we apply our algorithm for both halves)
- Step 3 takes O(n) time
- Step 5 takes O(n) time (as we saw in the previous section)

so,

$T(n) = 2T(n/2) + O(n)$

which, according the Master Theorem, result

$T(n) \in O(nlogn)$

Hence the merging of the sub-solutions is dominated by the sorting at step 4, and hence takes O(nlogn) time.

This must be repeated once for each level of recursion in the divide-and-conquer algorithm,

hence the whole of algorithm ClosestPair takes O(logn*nlogn) = O(nlog2n) time.

3.6.4 Improving the Algorithm

We can improve on this algorithm slightly by reducing the time it takes to achieve the y-coordinate sorting in Step 4. This is done by asking that the recursive solution computed in Step 1 returns the points in sorted order by their y coordinates. This will yield two sorted lists of points which need only be merged (a linear time operation) in Step 4 in order to yield a complete sorted list. Hence the revised algorithm involves making the following changes: Step 1: Divide the set into..., and recursively compute the distance in each part, returning the points in each set in sorted order by y-coordinate. Step 4: Merge the two sorted lists into one sorted list in O(n) time. Hence the merging process is now dominated by the linear time steps thereby yielding an O(nlogn) algorithm for finding the closest pair of a set of points in the plane.

3.7 Towers Of Hanoi Problem

[TODO: Write about the towers of hanoi algorithm and a program for it]

There are n distinct sized discs and three pegs such that discs are placed at the left peg in the order of their sizes. The smallest one is at the top while the largest one is at the bottom. This game is to move all the discs from the left peg

3.7.1 Rules

1) Only one disc can be moved in each step.

2) Only the disc at the top can be moved.

3) Any disc can only be placed on the top of a larger disc.

3.7.2 Solution

Intuitive Idea

In order to move the largest disc from the left peg to the middle peg, the smallest discs must be moved to the right peg first. After the largest one is moved. The smaller discs are then moved from the right peg to the middle peg.

Recurrence

Suppose n is the number of discs.

To move n discs from peg a to peg b,

1) If n>1 then move n-1 discs from peg a to peg c

2) Move n-th disc from peg a to peg b

3) If n>1 then move n-1 discs from peg c to peg a

Pseudocode

void hanoi(n,src,dst){ if (n>1) hanoi(n-1,src,pegs-{src,dst}); print "move n-th disc from src to dst"; if (n>1) hanoi(n-1,pegs-{src,dst},dst); }

Analysis

The analysis is trivial. $T(n) = 2T(n-1) + O(1) = O(2^n)$

3.8 Footnotes

[1] A (mathematical) integer larger than the largest "int" directly supported by your computer's hardware is often

called a “BigInt”. Working with such large numbers is often called “multiple precision arithmetic”. There are entire books on the various algorithms for dealing with such numbers, such as:

- Modern Computer Arithmetic, Richard Brent and Paul Zimmermann, Cambridge University Press, 2010.
- Donald E. Knuth, The Art of Computer Programming , Volume 2: Seminumerical Algorithms (3rd edition), 1997.

People who implement such algorithms may

- write a one-off implementation for one particular application
- write a library that you can use for many applications, such as GMP, the GNU Multiple Precision Arithmetic Library or McCutchen’s Big Integer Library or various libraries used to demonstrate RSA encryption
- put those algorithms in the compiler of a programming language that you can use (such as Python and Lisp) that automatically switches from standard integers to BigInts when necessary

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

Chapter 4

Randomization

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

As deterministic algorithms are driven to their limits when one tries to solve hard problems with them, a useful technique to speed up the computation is **randomization**. In randomized algorithms, the algorithm has access to a *random source*, which can be imagined as tossing coins during the computation. Depending on the outcome of the toss, the algorithm may split up its computation path.

There are two main types of randomized algorithms: Las Vegas algorithms and Monte-Carlo algorithms. In Las Vegas algorithms, the algorithm may use the randomness to speed up the computation, but the algorithm must always return the correct answer to the input. Monte-Carlo algorithms do not have the former restriction, that is, they are allowed to give *wrong* return values. However, returning a wrong return value must have a *small probability*, otherwise that Monte-Carlo algorithm would not be of any use.

Many approximation algorithms use randomization.

4.1 Ordered Statistics

Before covering randomized techniques, we'll start with a deterministic problem that leads to a problem that utilizes randomization. Suppose you have an unsorted array of values and you want to find

- the maximum value,
- the minimum value, and
- the median value.

In the immortal words of one of our former computer science professors, "How can you do?"

4.1.1 find-max

First, it's relatively straightforward to find the largest element:

// find-max -- returns the maximum element function **find-max**(array *vals*[1..*n*]): element let *result* := *vals[1]* for *i* from *2* to *n*: *result* := max(*result*, *vals[i]*) repeat return *result* end

An initial assignment of $-\infty$ to *result* would work as well, but this is a useless call to the max function since the first element compared gets set to *result*. By initializing result as such the function only requires *n-1* comparisons. (Moreover, in languages capable of metaprogramming, the data type may not be strictly numerical and there might be no good way of assigning $-\infty$; using vals[1] is type-safe.)

A similar routine to find the minimum element can be done by calling the min function instead of the max function.

4.1.2 find-min-max

But now suppose you want to find the min and the max at the same time; here's one solution:

// find-min-max -- returns the minimum and maximum element of the given array function **find-min-max**(array *vals*): pair return pair {**find-min**(*vals*), **find-max**(*vals*)} end

Because **find-max** and **find-min** both make *n-1* calls to the max or min functions (when *vals* has *n* elements), the total number of comparisons made in **find-min-max** is $2n - 2$.

However, some redundant comparisons are being made. These redundancies can be removed by "weaving" together the min and max functions:

// find-min-max -- returns the minimum and maximum element of the given array function **find-min-max**(array *vals*[1..*n*]): pair let *min* := ∞ let *max* := $-\infty$ if *n* is odd: *min* := *max* := *vals*[1] *vals* := *vals*[2,..,*n*] *// we can now assume n is even* *n* := *n* - 1 fi for *i*:=1 to *n* by 2: *// consider pairs of values in vals* if *vals*[*i*] < *vals*[*i* + 1]: let *a* := *vals*[*i*] let *b* := *vals*[*i* + 1] else: let *a* := *vals*[*i* + 1] let *b* := *vals*[*i*] *// invariant: a <= b* fi if *a* < *min*: *min* := *a* fi if *b* > *max*: *max* := *b* fi repeat return pair {*min*, *max*} end

Here, we only loop $n/2$ times instead of *n* times, but for each iteration we make three comparisons. Thus, the number of comparisons made is $(3/2)n = 1.5n$, resulting in a $3/4$ speed up over the original algorithm.

Only three comparisons need to be made instead of four because, by construction, it's always the case that $a \leq b$. (In the first part of the "if", we actually know more specifically that $a < b$, but under the else part, we can only conclude that $a \leq b$.) This property is utilized by noting that *a* doesn't need to be compared with the current maximum, because *b* is already greater than or equal to *a*, and similarly, *b* doesn't need to be compared with the current minimum, because *a* is already less than or equal to *b*.

In software engineering, there is a struggle between using libraries versus writing customized algorithms. In this case, the min and max functions weren't used in order to get a faster **find-min-max** routine. Such an operation would probably not be the bottleneck in a real-life program: however, if testing reveals the routine should be faster, such an approach should be taken. Typically, the solution that reuses libraries is better overall than writing customized solutions. Techniques such as open implementation and aspect-oriented programming may help manage this contention to get the best of both worlds, but regardless it's a useful distinction to recognize.

4.1.3 find-median

Finally, we need to consider how to find the median value. One approach is to sort the array then extract the median from the position *vals*[*n*/2]:

// find-median -- returns the median element of vals function **find-median**(array *vals*[1..*n*]): element assert (*n* > 0) sort(*vals*) return *vals*[*n* / 2] end

If our values are not numbers close enough in value (or otherwise cannot be sorted by a radix sort) the sort above is going to require $O(n \log n)$ steps.

However, it is possible to extract the *n*th-ordered statistic in $O(n)$ time. The key is eliminating the sort: we don't actually require the entire array to be sorted in order to find the median, so there is some waste in sorting the entire array first. One technique we'll use to accomplish this is randomness.

Before presenting a non-sorting **find-median** function, we introduce a divide and conquer-style operation known as **partitioning**. What we want is a routine that finds a random element in the array and then partitions the array into three parts:

1. elements that are less than or equal to the random element;
2. elements that are equal to the random element; and
3. elements that are greater than or equal to the random element.

These three sections are denoted by two integers: *j* and *i*. The partitioning is performed "in place" in the array:

// partition -- break the array three partitions based on a randomly picked element function **partition**(array *vals*): pair{*j*, *i*}

Note that when the random element picked is actually represented three or more times in the array it's possible for entries in all three partitions to have the same value as the random element. While this operation may not sound very useful, it has a powerful property that can be exploited: When the partition operation completes, the randomly picked element will be in the same position in the array as it would be if the array were fully sorted!

This property might not sound so powerful, but recall the optimization for the **find-min-max** function: we noticed that by picking elements from the array in pairs and comparing them to each other first we could reduce the total number of comparisons needed (because the current min and max values need to be compared with only one value each, and not two). A similar concept is used here.

While the code for **partition** is not magical, it has some tricky boundary cases:

// partition -- break the array into three ordered partitions from a random element function **partition**(array *vals*): pair{*j*, *i*} let *m* := 0 let *n* := *vals*.length - 2 // for an array vals, vals[vals.length-1] is the last element, which holds the partition, // so the last sort element is vals[vals.length-2] let *irand* := random(*m*, *n*) *// returns any value from m to n* let *x* := *vals*[*irand*] swap(*irand*,*n*+ 1) // n+1 = vals.length-1 , which is the right most element, and acts as store for partition element and sentinel for m // values in *vals*[*n*..] are greater than *x* // values in *vals*[0..*m*] are less than *x* while (m <= n) // see explanation in quick sort why should be m <= n instead of m < n in the 2 element case, // vals.length -2 = 0 = n = m, but if the 2-element case is out-of-order vs. in-order, there must be a different action. // by implication, the different action occurs within this loop, so must process the m = n case before exiting. while *vals*[*m*] <= *x* // in the 2-element case, second element is partition, first element at m. If in-order, m will increment *m*++ endwhile while *x* < *vals*[*n*] && *n* > 0 // stops if vals[n] belongs in left partition or hits start of array *n*-- endwhile if (m >= n) break; swap(*m*,*n*) // exchange *vals*[*n*] and *vals*[*m*] *m*++ // don't rescan swapped elements *n*-- endwhile // partition: [0..*m*$-$1] [] [*n*+1..] note that *m*=*n*+1 // if you need non empty sub-arrays: swap(*m*,*vals*.length - 1) // put the partition element in the between left and right partitions // in 2-element out-of-order case, m=0 (not incremented in loop), and the first and last(second) element will swap. // partition: [0..*n*$-$1] [*n*..*n*] [*n*+1..] end

We can use **partition** as a subroutine for a general **find** operation:

// find -- moves elements in vals such that location k holds the value it would when sorted function **find**(array *vals*,

integer *k*) assert (0 <= *k* < *vals*.length) *// k it must be a valid index* if *vals*.length <= 1: return fi let pair (*j*, *i*) := **partition**(*vals*) if *k* <= *i*: **find**(*a*[0,..,*i*], *k*) else-if *j* <= *k*: **find**(*a*[*j*,..,*n*], *k* - *j*) fi TODO: debug this! end

Which leads us to the punch-line:

// find-median -- returns the median element of vals function **find-median**(array *vals*): element assert (*vals*.length > 0) let *median_index* := *vals*.length / 2; **find**(*vals*, *median_index*) return *vals*[*median_index*] end

One consideration that might cross your mind is “is the random call really necessary?" For example, instead of picking a random pivot, we could always pick the middle element instead. Given that our algorithm works with all possible arrays, we could conclude that the running time on average for *all of the possible inputs* is the same as our analysis that used the random function. The reasoning here is that under the set of all possible arrays, the middle element is going to be just as “random” as picking anything else. But there’s a pitfall in this reasoning: Typically, the input to an algorithm in a program isn't random at all. For example, the input has a higher probability of being sorted than just by chance alone. Likewise, because it is real data from real programs, the data might have other patterns in it that could lead to suboptimal results.

To put this another way: for the randomized median finding algorithm, there is a very small probability it will run suboptimally, independent of what the input is; while for a deterministic algorithm that just picks the middle element, there is a greater chance it will run poorly on some of the most frequent input types it will receive. This leads us to the following guideline:

Note that there are “derandomization” techniques that can take an average-case fast algorithm and turn it into a fully deterministic algorithm. Sometimes the overhead of derandomization is so much that it requires very large datasets to get any gains. Nevertheless, derandomization in itself has theoretical value.

The randomized **find** algorithm was invented by C. A. R. “Tony” Hoare. While Hoare is an important figure in computer science, he may be best known in general circles for his quicksort algorithm, which we discuss in the next section.

4.2 Quicksort

The median-finding partitioning algorithm in the previous section is actually very close to the implementation of a full blown sorting algorithm. Building a Quicksort Algorithm is left as an exercise for the reader, and is recommended first, before reading the next section (Quick sort is diabolical compared to Merge sort, which is a sort not improved by a randomization step) .

A key part of quick sort is choosing the right median. But to get it up and running quickly, start with the assumption that the array is unsorted, and the rightmost element of each array is as likely to be the median as any other element, and that we are entirely optimistic that the rightmost doesn't happen to be the largest key , which would mean we would be removing one element only (the partition element) at each step, and having no right array to sort, and a n-1 left array to sort.

This is where **randomization** is important for quick sort, i.e. *choosing the more optimal partition key*, which is pretty important for quick sort to work efficiently.

Compare the number of comparisions that are required for quick sort vs. insertion sort.

With insertion sort, the average number of comparisons for finding the lowest first element in an ascending sort of a randomized array is n /2 .

The second element’s average number of comparisons is (n-1)/2;

the third element (n- 2) / 2.

The total number of comparisons is [n + (n - 1) + (n - 2) + (n - 3) .. + (n - [n-1])] divided by 2, which is [n x n - (n-1)!] /2 or about O(n squared) .

In Quicksort, the number of comparisons will halve at each partition step if the true median is chosen, since the left half partition doesn't need to be compared with the right half partition, but at each step , the number elements of all partitions created by the previously level of partitioning will still be n.

The number of levels of comparing n elements is the number of steps of dividing n by two , until n = 1. Or in reverse, 2 ^ m ~ n, so m = $\log_2$ n.

So the total number of comparisons is n (elements) x m (levels of scanning) or n x $\log_2$n ,

So the number of comparison is O(n x log $_2$(n)) , which is smaller than insertion sort’s O(n^2) or O(n x n).

(Comparing O(n x log $_2$(n)) with O(n x n) , the common factor n can be eliminated , and the comparison is $\log_2$(n) vs n , which is exponentially different as n becomes larger. e.g. compare n = 2^16 , or 16 vs 32768, or 32 vs 4 gig).

To implement the partitioning in-place on a part of the array determined by a previous recursive call, what is needed a scan from each end of the part , swapping whenever the value of the left scan’s current location is greater than the partition value, and the value of the right scan’s current location is less than the partition value. So the initial step is :-

Assign the partition value to the right most element, swapping if necessary.

So the partitioning step is :-

increment the left scan pointer while the current value is less than the partition value. decrement the right scan pointer while the current value is more than the partition value , or the location is equal to or more than the left most location. exit if the pointers have crossed (l >= r), OTHERWISE perform a swap where the left and right pointers have stopped , on values where the left pointer's value is greater than the partition, and the right pointer's value is less than the partition. Finally, after exiting the loop because the left and right pointers have crossed, *swap the* rightmost *partition value,* with the last location of the **left** forward scan pointer *,* and hence ends up between the left and right partitions.

Make sure at this point , that after the final swap, the cases of a 2 element in-order array, and a 2 element out-of-order array , are handled correctly, which should mean all cases are handled correctly. This is a good debugging step for getting quick-sort to work.

For the in-order two-element case, the left pointer stops on the partition or second element , as the partition value is found. The right pointer , scanning backwards, starts on the first element before the partition, and stops because it is in the leftmost position.

The pointers cross, and the loop exits before doing a loop swap. Outside the loop, the contents of the left pointer at the rightmost position and the partition , also at the right most position , are swapped, achieving no change to the in-order two-element case.

For the out-of-order two-element case, The left pointer scans and stops at the first element, because it is greater than the partition (left scan value stops to swap values greater than the partition value).

The right pointer starts and stops at the first element because it has reached the leftmost element.

The loop exits because left pointer and right pointer are equal at the first position, and the contents of the left pointer at the first position and the partition at the rightmost (other) position , are swapped , putting previously out-of-order elements , into order.

Another implementation issue, is to how to move the pointers during scanning. Moving them at the end of the outer loop seems logical.

partition(a,l,r) { v = a[r]; i = l; j = r −1; while (i <= j) { // need to also scan when i = j as well as i < j , // in the 2 in-order case, // so that i is incremented to the partition // and nothing happens in the final swap with the partition at r. while (a[i] < v) ++i; while (v <= a[j] && j > 0) --j; if (i >= j) break; swap(a,i,j); ++i; --j; } swap(a, i, r); return i;

With the pre-increment/decrement unary operators, scanning can be done just before testing within the test condition of the while loops, but this means the pointers should be offset −1 and +1 respectively at the start : so the algorithm then looks like:-

partition (a, l, r) { v=a[r]; // v is partition value, at a[r] i=l-1; j=r; while(true) { while(a[++i] < v); while(v <= a[--j] && j > l); if (i >= j) break; swap (a, i, j); } swap (a,i,r); return i; }

And the qsort algorithm is

qsort(a, l, r) { if (l >= r) return ; p = partition(a, l, r) qsort(a , l, p-1) qsort(a, p+1, r)

}

Finally, randomization of the partition element.

random_partition (a,l,r) { p = random_int(r-l) + l; // median of a[l], a[p] , a[r] if (a[p] < a[l]) p =l; if (a[r]< a[p]) p = r; swap(a, p, r); }

this can be called just before calling partition in qsort().

4.3 Shuffling an Array

This keeps data in during shuffle temporaryArray = { } **This records if an item has been shuffled** usedItemArray = { } **Number of item in array** itemNum = 0 while (itemNum != lengthOf(inputArray)){ usedItemArray[itemNum] = false **None of the items have been shuffled** itemNum = itemNum + 1 } itemNum = 0 **we'll use this again** itemPosition = randdomNumber(0 --- (lengthOf(inputArray) - 1)) while(itemNum != lengthOf(inputArray)){ while(usedItemArray[itemPosition] != false){ itemPosition = randdomNumber(0 --- (lengthOf(inputArray) - 1)) } temporaryArray[itemPosition] = inputArray[itemNum] itemNum = itemNum + 1 } inputArray = temporaryArray

4.4 Equal Multivariate Polynomials

[TODO: as of now, there is no known deterministic polynomial time solution, but there is a randomized polytime solution. The canonical example used to be IsPrime, but a deterministic, polytime solution has been found.]

4.5 Hash tables

Hashing relies on a hashcode function to randomly distribute keys to available slots evenly. In java , this is done in a fairly straight forward method of adding a moderate sized prime number (31 * 17) to a integer key , and then modulus by the size of the hash table. For string keys, the initial hash number is obtained by adding the products of each character's ordinal value multiplied by 31.

The wikibook Data Structures/Hash Tables chapter covers the topic well.

4.6 Skip Lists

[TODO: Talk about skips lists. The point is to show how randomization can sometimes make a structure easier to understand, compared to the complexity of balanced trees.]

Dictionary or Map , is a general concept where a value is inserted under some key, and retrieved by the key. For instance, in some languages , the dictionary concept is built-in (Python), in others , it is in core libraries (C++ S.T.L. , and Java standard collections library). The library providing languages usually lets the programmer choose between a hash algorithm, or a balanced binary tree implementation (red-black trees). Recently, skip lists have been offered, because they offer advantages of being implemented to be highly concurrent for multiple threaded applications.

Hashing is a technique that depends on the randomness of keys when passed through a hash function, to find a hash value that corresponds to an index into a linear table. Hashing works as fast as the hash function, but works well only if the inserted keys spread out evenly in the array, as any keys that hash to the same index , have to be deal with as a hash collision problem e.g. by keeping a linked list for collisions for each slot in the table, and iterating through the list to compare the full key of each key-value pair vs the search key.

The disadvantage of hashing is that in-order traversal is not possible with this data structure.

Binary trees can be used to represent dictionaries, and in-order traversal of binary trees is possible by visiting of nodes (visit left child, visit current node, visit right child, recursively). Binary trees can suffer from poor search when they are "unbalanced" e.g. the keys of key-value pairs that are inserted were inserted in ascending or descending order, so they effectively look like *linked lists* with no left child, and all right children. *self-balancing* binary trees can be done probabilistically (using randomness) or deterministically (using child link coloring as red or black) , through local 3-node tree **rotation** operations. A rotation is simply swapping a parent with a child node, but preserving order e.g. for a left child rotation, the left child's right child becomes the parent's left child, and the parent becomes the left child's right child.

Red-black trees can be understood more easily if corresponding **2-3-4 trees** are examined. A 2-3-4 tree is a tree where nodes can have 2 children, 3 children, or 4 children, with 3 children nodes having 2 keys between the 3 children, and 4 children-nodes having 3 keys between the 4 children. 4-nodes are actively split into 3 single key 2 -nodes, and the middle 2-node passed up to be merged with the parent node , which , if a one-key 2-node, becomes a two key 3-node; or if a two key 3-node, becomes a 4-node, which will be later split (on the way up). The act of splitting a three key 4-node is actually a re-balancing operation, that prevents a string of 3 nodes of grandparent, parent , child occurring , without a balancing rotation happening. 2-3-4 trees are a limited example of **B-trees**, which usually have enough nodes as to fit a physical disk block, to facilitate caching of very large indexes that can't fit in physical RAM (which is much less common nowadays).

A **red-black tree** is a binary tree representation of a 2-3-4 tree, where 3-nodes are modeled by a parent with one red child, and 4 -nodes modeled by a parent with two red children. Splitting of a 4-node is represented by the parent with 2 red children, **flipping** the red children to black, and itself into red. There is never a case where the parent is already red, because there also occurs balancing operations where if there is a grandparent with a red parent with a red child , the grandparent is rotated to be a child of the parent, and parent is made black and the grandparent is made red; this unifies with the previous **flipping** scenario, of a 4-node represented by 2 red children. Actually, it may be this standardization of 4-nodes with mandatory rotation of skewed or zigzag 4-nodes that results in re-balancing of the binary tree.

A newer optimization is to left rotate any single right red child to a single left red child, so that only right rotation of left-skewed inline 4-nodes (3 red nodes inline) would ever occur, simplifying the re-balancing code.

Skip lists are modeled after single linked lists, except nodes are multilevel. Tall nodes are rarer, but the insert operation ensures nodes are connected at each level.

Implementation of skip lists requires creating randomly high multilevel nodes, and then inserting them.

Nodes are created using iteration of a random function where high level node occurs later in an iteration, and are rarer, because the iteration has survived a number of random thresholds (e.g. 0.5, if the random is between 0 and 1).

Insertion requires a temporary previous node array with the height of the generated inserting node. It is used to store the last pointer for a given level , which has a key less than the insertion key.

The scanning begins at the head of the skip list, at highest level of the head node, and proceeds across until a node is found with a key higher than the insertion key, and the previous pointer stored in the temporary previous node array. Then the next lower level is scanned from that node , and so on, walking zig-zag down, until the lowest level is reached.

Then a list insertion is done at each level of the temporary previous node array, so that the previous node's next node at each level is made the next node for that level for the inserting node, and the inserting node is made the previous node's next node.

Search involves iterating from the highest level of the head node to the lowest level, and scanning along the next pointer for each level until a node greater than the search key is found, moving down to the next level , and proceeding with the scan, until the higher keyed node at the lowest level has been found, or the search key found.

The creation of less frequent-when-taller , randomized height nodes, and the process of linking in all nodes at every level, is what gives skip lists their advantageous overall structure.

4.6.1 a method of skip list implementation : implement lookahead single-linked linked list, then test , then transform to skip list implementation , then same test, then performance comparison

What follows is a implementation of skip lists in python. A single linked list looking at next node as always the current node, is implemented first, then the skip list implementation follows, attempting minimal modification of the former, and comparison helps clarify implementation.

#copyright SJT 2014, GNU #start by implementing a one lookahead single-linked list : #the head node has a next pointer to the start of the list, and the current node examined is the next node. #This is much easier than having the head node one of the storage nodes. class LN: “a list node, so don't have to use dict objects as nodes” def __init__(self): self.k=None self.v = None self.next = None class single_list2: def __init__(self): self.h = LN() def insert(self, k, v): prev = self.h while not prev.next is None and k < prev.next.k : prev = prev.next n = LN() n.k, n.v = k, v n.next = prev.next prev.next = n def show(self): prev = self.h while not prev.next is None: prev = prev.next print prev.k, prev.v, ' ' def find (self,k): prev = self.h while not prev.next is None and k < prev.next.k: prev = prev.next if prev.next is None: return None return prev.next.k #then after testing the single-linked list, model SkipList after it. # The main conditions to remember when trying to transform single-linked code to skiplist code: # * multi-level nodes are being inserted # * the head node must be as tall as the node being inserted # * walk backwards down levels from highest to lowest when inserting or searching, # since this is the basis for algorithm efficiency, as taller nodes are less frequently and widely dispersed. import random class SkipList3: def __init__(self): self.h = LN() self.h.next = [None] def insert(self, k , v): ht = 1 while random.randint(0,10) < 5: ht +=1 if ht > len(self.h.next) : self.h.next.extend([None] * (ht - len(self.h.next))) prev = self.h prev_list = [self.h] * len(self.h.next) # instead of just prev.next in the single linked list, each level i has a prev.next for i in xrange(len(self.h.next)−1, −1, −1): while not prev.next[i] is None and prev.next[i].k > k: prev = prev.next[i] #record the previous pointer for each level prev_list[i] = prev n = LN() n.k,n.v = k,v # create the next pointers to the height of the node for the current node. n.next = [None] * ht #print “prev list is ", prev_list # instead of just linking in one node in the single-linked list , ie. n.next = prev.next, prev.next =n # do it for each level of n.next using n.next[i] and prev_list[i].next[i] # there may be a different prev node for each level, but the same level must be linked, # therefore the [i] index occurs twice in prev_list[i].next[i]. for i in xrange(0, ht): n.next[i] = prev_list[i].next[i] prev_list[i].next[i] = n #print “self.h ", self.h def show(self): #print self.h prev = self.h while not prev.next[0] is None: print prev.next[0].k, prev.next[0].v prev = prev.next[0] def find(self, k): prev = self.h h = len(self.h.next) #print “height ", h for i in xrange(h-1, −1, −1): while not prev.next[i] is None and prev.next[i].k > k: prev = prev.next[i] #if prev.next[i] <> None: #print “i, k, prev.next[i].k and .v”, i, k, prev.next[i].k, prev.next[i].v if prev.next[i] <> None and prev.next[i].k == k: return prev.next[i].v if pref.next[i] is None: return None return prev.next[i].k def clear(self): self.h= LN() self.h.next = [None] #driver if __name__ == "__main__": #l = single_list2() l = SkipList3() test_dat = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' pairs = enumerate(test_dat) m = [(x,y) for x,y in pairs] while len(m) > 0: i = random.randint(0,len(m)−1) print “inserting ", m[i] l.insert(m[i][0], m[i][1]) del m[i] # l.insert(3, 'C') # l.insert(2, 'B') # l.insert(4, 'D') # l.insert(1, 'A') l.show() n = int(raw_input(“How many elements to test?")) if n <0: n = -n l.clear() import time l2 = [x for x in xrange(0, n)] random.shuffle(l2) for x in l2: l.insert(x , x) l.show() print print “finding..” f = 0 t1 = time.time() nf = [] for x in l2: if l.find(x) == x: f += 1 else: nf.append(x) t2 = time.time() print “time”, t2 - t1 td1 = t2 - t1 print “found ", f print “didn't find”, nf dnf = [] for x in nf: tu = (x,l.find(x)) dnf.append(tu) print “find again ", dnf sl = single_list2() for x in l2: sl.insert(x,x) print “finding..” f = 0 t1 = time.time() for x in l2: if sl.find(x) == x: f += 1 t2 = time.time() print “time”, t2 - t1 print “found ", f td2 = t2 - t1 print “factor difference time”, td2/td1

4.6.2 Role of Randomness

The idea of making higher nodes geometrically randomly less common, means there are less keys to compare with the higher the level of comparison, and since these are randomly selected, this should get rid of problems of degenerate input that makes it necessary to do tree balancing in tree algorithms. Since the higher level list have more widely separated elements, but the search algorithm moves down a level after each search terminates at a level, the higher levels help “skip” over the need to search earlier elements on lower lists. Because there are multiple levels of skipping, it becomes less likely that a meagre skip at a higher level won't be compensated by better skips at lower levels, and Pugh claims O(logN) performance overall.

Conceptually , is it easier to understand than balancing trees and hence easier to implement ? The development of ideas from binary trees, balanced binary trees, 2-3 trees, red-black trees, and B-trees make a stronger conceptual network but is progressive in development, so arguably, once red-black trees are understood, they have more conceptual context to aid memory , or refresh of memory.

4.6.3 concurrent access application

Apart from using randomization to enhance a basic memory structure of linked lists, skip lists can also be extended as a global data structure used in a multiprocessor application. See supplementary topic at the end of the chapter.

4.6.4 Idea for an exercise

Replace the Linux completely fair scheduler red-black tree implementation with a skip list , and see how your brand of Linux runs after recompiling.

4.7 Treaps

A treap is a two keyed binary tree, that uses a second randomly generated key and the previously discussed tree operation of parent-child rotation to randomly rotate the tree so that overall, a balanced tree is produced. Recall that binary trees work by having all nodes in the left subtree small than a given node, and all nodes in a right subtree greater. Also recall that node rotation does not break this order (some people call it an invariant), but changes the relationship of parent and child, so that if the parent was smaller than a right child, then the parent becomes the left child of the formerly right child. The idea of a tree-heap or treap, is that a binary heap relationship is maintained between parents and child, and that is a parent node has higher priority than its children, which is not the same as the left , right order of keys in a binary tree, and hence a recently inserted leaf node in a binary tree which happens to have a high random priority, can be rotated so it is relatively higher in the tree, having no parent with a lower priority.

A treap is an alternative to both red-black trees, and skip lists, as a self-balancing sorted storage structure.

4.7.1 java example of treap implementation

// Treap example: 2014 SJT, copyleft GNU . import java.util.Iterator; import java.util.LinkedList; import java.util.Random; public class Treap1<K extends Comparable<K>, V> { public Treap1(boolean test) { this.test = test; } public Treap1() {} boolean test = false; static Random random = new Random(System.currentTimeMillis()); class TreapNode { int priority = 0; K k; V val; TreapNode left, right; public TreapNode() { if (!test) { priority = random.nextInt(); } } } TreapNode root = null; void insert(K k, V val) { root = insert(k, val, root); } TreapNode insert(K k, V val, TreapNode node) { TreapNode node2 = new TreapNode(); node2.k = k; node2.val = val; if (node == null) { node = node2; } else if (k.compareTo(node.k) < 0) { node.left = insert(k, val, node.left); } else { node.right = insert(k, val, node.right); } if (node.left != null && node.left.priority > node.priority) { // right rotate (rotate left node up, current node becomes right child) TreapNode tmp = node.left; node.left = node.left.right; tmp.right = node; node = tmp; } else if (node.right != null && node.right.priority > node.priority) { // left rotate (rotate right node up , current node becomes left child) TreapNode tmp = node.right; node.right = node.right.left; tmp.left = node; node = tmp; } return node; } V find(K k) { return findNode(k, root); } private V findNode(K k, Treap1<K, V>.TreapNode node) { // TODO Auto-generated method stub if (node == null) return null; if (k.compareTo(node.k) < 0) { return findNode(k, node.left); } else if (k.compareTo(node.k) > 0) { return findNode(k, node.right); } else { return node.val; } } public static void main(String[] args) { LinkedList<Integer> dat = new LinkedList<Integer>(); for (int i = 0; i < 15000; ++i) { dat.add(i); } testNumbers(dat, true); // no random priority balancing testNumbers(dat, false); } private static void testNumbers(LinkedList<Integer> dat, boolean test) { Treap1<Integer, Integer> tree= new Treap1<>(test); for (Integer integer : dat) { tree.insert(integer, integer); } long t1 = System.currentTimeMillis(); Iterator<Integer> iter = dat.iterator(); int found = 0; while (iter.hasNext()) { Integer j = desc.next(); Integer i = tree.find(j); if (j.equals(i)) { ++found; } } long t2 = System.currentTimeMillis(); System.out.println("found = " + found + " in " + (t2 - t1)); } }

4.7.2 Treaps compared and contrasted to Splay trees

Splay trees are similiar to treaps in that rotation is used to bring a higher priority node to the top without changing the main key order, except instead of using a random key for priority, the last accessed node is rotated to the root of the tree, so that more frequently accessed nodes will be near the top. This means that in treaps, inserted nodes will only rotate upto the priority given by their random priority key, whereas in splay trees, the inserted node is rotated to the root, and every search in a splay tree will result in a re-balancing, but not so in a treap.

4.8 Derandomization

[TODO: Deterministic algorithms for Quicksort exist that perform as well as quicksort in the average case and are guaranteed to perform at least that well in all cases. Best of all, no randomization is needed. Also in the discussion should be some perspective on using randomization: some randomized algorithms give you better confidence probabilities than the actual hardware itself! (e.g. sunspots can randomly flip bits in hardware, causing failure, which is a risk we take quite often)]

[Main idea: Look at all blocks of 5 elements, and pick the median (O(1) to pick), put all medians into an array (O(n)), recursively pick the medians of that array, repeat until you have < 5 elements in the array. This recursive median constructing of every five elements takes time T(n)=T(n/5) + O(n), which by the master theorem is O(n). Thus, in O(n) we can find the right pivot. Need to show that this pivot is sufficiently good so that we're still O(n log n) no matter what the input is. This version of quicksort doesn't need rand, and it never performs poorly. Still need to show that element picked out is sufficiently good for a pivot.]

4.9 Exercises

1. Write a **find-min** function and run it on several different inputs to demonstrate its correctness.

4.10 Supplementary Topic: skip lists and multiprocessor algorithms

Multiprocessor hardware provides CAS (compare-and-set) or CMPEXCHG(compare-and-exchange)(intel manual 253666.pdf, p 3-188) atomic operations, where an expected value is loaded into the accumulator register, which is compared to a target memory location's contents, and if the same, a source memory location's contents is loaded into the target memories contents, and the zero flag set, otherwise, if different, the target memory's contents is returned in the accumulator, and the zero flag is unset, signifying , for instance, a lock contention. In the intel architecture, a LOCK instruction is issued before CMPEXCHG , which either locks the cache from concurrent access if the memory location is being cached, or locks a shared memory location if not in the cache , for the next instruction.

The CMPEXCHG can be used to implement locking, where spinlocks , e.g. retrying until the zero flag is set, are simplest in design.

Lockless design increases efficiency by avoiding spinning waiting for a lock .

The java standard library has an implementation of non-blocking concurrent skiplists, based on a paper titled "a pragmatic implementation of non-blocking single-linked lists".

The skip list implementation is an extension of the lock-free single-linked list , of which a description follows :-

The **insert** operation is : X -> Y insert N , N -> Y, X -> N ; expected result is X -> N -> Y .

A race condition is if M is inserting between X and Y and M completes first , then N completes, so the situation is X -> N -> Y <- M

M is not in the list. The CAS operation avoids this, because a copy of -> Y is checked before updating X -> , against the current value of X -> .

If N gets to update X -> first, then when M tries to update X -> , its copy of X -> Y , which it got before doing M -> Y , does not match X -> N , so CAS returns non-zero flag set. The process that tried to insert M then can retry the insertion after X, but now the CAS checks ->N is X's next pointer, so after retry, X->M->N->Y , and neither insertions are lost.

If M updates X-> first, N 's copy of X->Y does not match X -> M , so the CAS will fail here too, and the above retry of the process inserting N, would have the serialized result of X ->N -> M -> Y .

The **delete** operation depends on a separate 'logical' deletion step, before 'physical' deletion.

'Logical' deletion involves a CAS change of the next pointer into a 'marked' pointer. The java implementation substitutes with an atomic insertion of a proxy marker node to the next node.

This prevents future insertions from inserting after a node which has a next pointer 'marked' , making the latter node 'logically' deleted.

The **insert** operation relies on another function , *search* , returning 2 **unmarked** , at the time of the invocation, node pointers : the first pointing to a node , whose next pointer is equal to the second.

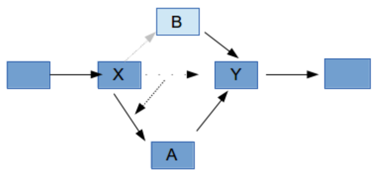
The first node is the node before the insertion point.

The *insert* CAS operation checks that the current next pointer of the first node, corresponds to the unmarked reference of the second, so will fail 'logically' if the first node's *next* pointer has become marked *after* the call to the *search* function above, because the first node has been concurrently logically deleted.

This meets the aim to prevent a insertion occurring concurrently after a node has been deleted.

If the insert operation fails the CAS of the previous node's next pointer, the search for the insertion point starts from the **start of the entire list** again, since a new unmarked previous node needs to be found, and there are no previous node pointers as the list nodes are singly-linked.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*



A race insertion between A and B may cause the pointer from X to B to be lost, if B inserts, then A inserts, without a CAS checking that X's next pointer is still to Y when inserting A.

The **delete** operation outlined above, also relies on the *search* operation returning two *unmarked* nodes, and the two CAS operations in delete, one for logical deletion or marking of the second pointer's next pointer, and the other for physical deletion by making the first node's next pointer point to the second node's unmarked next pointer.

The first CAS of delete happens only after a check that the copy of the original second nodes' next pointer is unmarked, and ensures that only one concurrent delete succeeds which reads the second node's current next pointer as being unmarked as well.

The second CAS checks that the previous node hasn't been logically deleted because its next pointer is not the same as the unmarked pointer to the current second node returned by the search function, so only an active previous node's next pointer is 'physically' updated to a copy of the original unmarked next pointer of the node being deleted (whose next pointer is already marked by the first CAS).

If the second CAS fails, then the previous node is logically deleted and its next pointer is marked, and so is the current node's next pointer. A call to *search* function again, tidies things up, because in endeavouring to find the key of the current node and return adjacent unmarked previous and current pointers, and while doing so, it truncates strings of logically deleted nodes .

Lock-free programming issues

Starvation could be possible , as failed inserts have to restart from the front of the list. Wait-freedom is a concept where the algorithm has all threads safe from starvation.

The ABA problem exists, where a garbage collector recycles the pointer A , but the address is loaded differently, and the pointer is re-added at a point where a check is done for A by another thread that read A and is doing a CAS to check A has not changed ; the address is the same and is unmarked, but the contents of A has changed.

Chapter 5

Backtracking

Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. Backtracking is also known as **depth-first search** or **branch and bound**. By inserting more knowledge of the problem, the search tree can be pruned to avoid considering cases that don't look promising. While backtracking is useful for hard problems to which we do not know more efficient solutions, it is a poor solution for the everyday problems that other techniques are much better at solving.

However, dynamic programming and greedy algorithms can be thought of as optimizations to backtracking, so the general technique behind backtracking is useful for understanding these more advanced concepts. Learning and understanding backtracking techniques first provides a good stepping stone to these more advanced techniques because you won't have to learn several new concepts all at once.

This methodology is generic enough that it can be applied to most problems. However, even when taking care to improve a backtracking algorithm, it will probably still take exponential time rather than polynomial time. Additionally, exact time analysis of backtracking algorithms can be extremely difficult: instead, simpler upperbounds that may not be tight are given.

5.1 Longest Common Subsequence (exhaustive version)

Note that the solution to the longest common subsequence (LCS) problem discussed in this section is not efficient. However, it is useful for understanding the dynamic programming version of the algorithm that is covered later.

The LCS problem is similar to what the Unix “diff” program does. The diff command in Unix takes two text files, *A* and *B*, as input and outputs the differences line-by-line from *A* and *B*. For example, diff can show you that lines missing from *A* have been added to *B*, and lines present in *A* have been removed from *B*. The goal is to get a list of additions and removals that could be used to transform *A* to *B*. An overly conservative solution to the problem would say that all lines from *A* were removed, and that all lines from *B* were added. While this would solve the problem in a crude sense, we are concerned with the minimal number of additions and removals to achieve a correct transformation. Consider how you may implement a solution to this problem yourself.

The LCS problem, instead of dealing with lines in text files, is concerned with finding common items between two different arrays. For example,

let *a* := array {"The”, “great”, “square”, “has”, “no”, “corners"} let *b* := array {"The”, “great”, “image”, “has”, “no”, “form"}

We want to find the longest subsequence possible of items that are found in both *a* and *b* in the same order. The LCS of *a* and *b* is

“The”, “great”, “has”, “no”

Now consider two more sequences:

let *c* := array {1, 2, 4, 8, 16, 32} let *d* := array {1, 2, 3, 32, 8}

Here, there are two longest common subsequences of *c* and *d*:

1, 2, 32; and

1, 2, 8

Note that

1, 2, 32, 8

is *not* a common subsequence, because it is only a valid subsequence of *d* and not *c* (because *c* has 8 before the 32). Thus, we can conclude that for some cases, solutions to the LCS problem are not unique. If we had more information about the sequences available we might prefer one subsequence to another: for example, if the sequences were lines of text in computer programs, we might choose the subsequences that would keep function definitions or paired comment delimiters intact (instead of choosing delimiters that were not paired in the syntax).

On the top level, our problem is to implement the following function

// lcs -- returns the longest common subsequence of a and b function **lcs**(array *a*, array *b*): array

which takes in two arrays as input and outputs the subsequence array.

How do you solve this problem? You could start by noticing that if the two sequences start with the same word, then the longest common subsequence always contains that word. You can automatically put that word on your list, and you would have just reduced the problem to finding the longest common subset of the rest of the two lists. Thus, the problem was made smaller, which is good because it shows progress was made.

But if the two lists do not begin with the same word, then one, or both, of the first element in *a* or the first element in *b* do not belong in the longest common subsequence. But yet, one of them might be. How do you determine which one, if any, to add?

The solution can be thought in terms of the back tracking methodology: Try it both ways and see! Either way, the two sub-problems are manipulating smaller lists, so you know that the recursion will eventually terminate. Whichever trial results in the longer common subsequence is the winner.

Instead of "throwing it away" by deleting the item from the array we use array slices. For example, the slice

a[1,..,5]

represents the elements

{*a*[1], *a*[2], *a*[3], *a*[4], *a*[5]}

of the array as an array itself. If your language doesn't support slices you'll have to pass beginning and/or ending indices along with the full array. Here, the slices are only of the form

a[1,..]

which, when using 0 as the index to the first element in the array, results in an array slice that doesn't have the 0th element. (Thus, a non-sliced version of this algorithm would only need to pass the beginning valid index around instead, and that value would have to be subtracted from the complete array's length to get the pseudo-slice's length.)

// lcs -- returns the longest common subsequence of a and b function **lcs**(array *a*, array *b*): array if *a*.length == 0 OR *b*.length == 0: *// if we're at the end of either list, then the lcs is empty* return new array {} else-if *a*[0] == *b*[0]: *// if the start element is the same in both, then it is on the lcs, // so we just recurse on the remainder of both lists.* return append(new array {*a*[0]}, **lcs**(*a*[1,..], *b*[1,..])) else *// we don't know which list we should discard from. Try both ways, // pick whichever is better.* let *discard_a* := **lcs**(*a*[1,..], *b*) let *discard_b* := **lcs**(*a*, *b*[1,..]) if *discard_a*.length > *discard_b*.length: let *result* := *discard_a* else let *result* := *discard_b* fi return *result* fi end

5.2 Shortest Path Problem (exhaustive version)

To be improved as Dijkstra's algorithm in a later section.

5.3 Largest Independent Set

5.4 Bounding Searches

If you've already found something "better" and you're on a branch that will never be as good as the one you already saw, you can terminate that branch early. (Example to use: sum of numbers beginning with 1 2, and then each number following is a sum of any of the numbers plus the last number. Show performance improvements.)

5.5 Constrained 3-Coloring

This problem doesn't have immediate self-similarity, so the problem first needs to be generalized. Methodology: If there's no self-similarity, try to generalize the problem until it has it.

5.6 Traveling Salesperson Problem

Here, backtracking is one of the best solutions known.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

Chapter 6

Dynamic Programming

Dynamic programming can be thought of as an optimization technique for particular classes of backtracking algorithms where subproblems are repeatedly solved. Note that the term *dynamic* in dynamic programming should not be confused with dynamic programming languages, like Scheme or Lisp. Nor should the term *programming* be confused with the act of writing computer programs. In the context of algorithms, dynamic programming always refers to the technique of filling in a table with values computed from other table values. (It's dynamic because the values in the table are filled in by the algorithm based on other values of the table, and it's programming in the sense of setting things in a table, like how television programming is concerned with when to broadcast what shows.)

6.1 Fibonacci Numbers

Before presenting the dynamic programming technique, it will be useful to first show a related technique, called **memoization**, on a toy example: The Fibonacci numbers. What we want is a routine to compute the *n*th Fibonacci number:

// fib -- compute Fibonacci(n) function **fib**(integer *n*): integer

By definition, the *n*th Fibonacci number, denoted F_n is

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

How would one create a good algorithm for finding the nth Fibonacci-number? Let's begin with the naive algorithm, which codes the mathematical definition:

// fib -- compute Fibonacci(n) function **fib**(integer *n*): integer assert (n >= 0) if *n* == 0: return 0 fi if *n* == 1: return 1 fi return **fib**(*n* - 1) + **fib**(*n* - 2) end

This code sample is also available in Ada.

Note that this is a toy example because there is already a mathematically closed form for F_n :

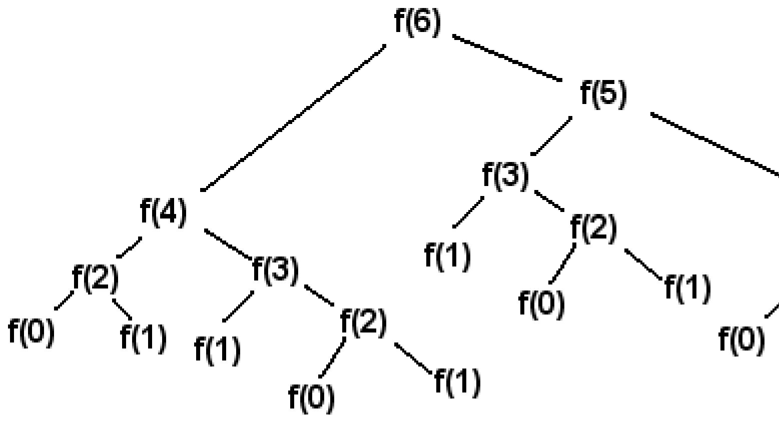
$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

where:

$$\phi = \frac{1+\sqrt{5}}{2}$$

This latter equation is known as the Golden Ratio. Thus, a program could efficiently calculate F_n for even very large *n*. However, it's instructive to understand what's so inefficient about the current algorithm.

To analyze the running time of fib we should look at a call tree for something even as small as the sixth Fibonacci number:



Every leaf of the call tree has the value 0 or 1, and the sum of these values is the final result. So, for any *n,* the number of leaves in the call tree is actually F_n itself! The closed form thus tells us that the number of leaves in **fib**(*n*) is approximately equal to

$$\left(\frac{1+\sqrt{5}}{2}\right)^n \approx 1.618^n = 2^{\lg(1.618^n)} = 2^{n\lg(1.618)} \approx 2^{0.69n}.$$

(Note the algebraic manipulation used above to make the base of the exponent the number 2.) This means that there are far too many leaves, particularly considering the repeated patterns found in the call tree above.

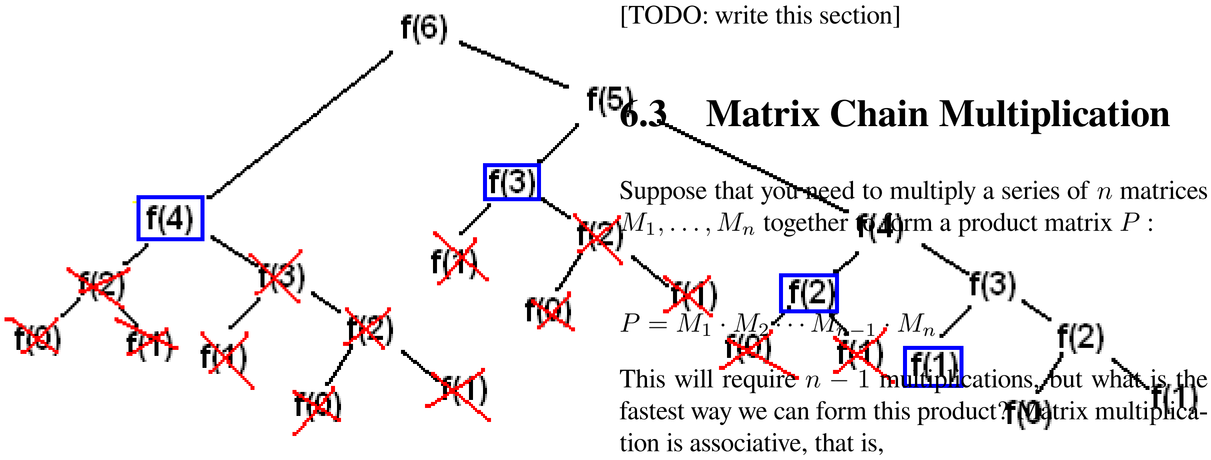
One optimization we can make is to save a result in a table once it’s already been computed, so that the same result needs to be computed only once. The optimization process is called memoization and conforms to the following methodology:

Consider the solution presented in the backtracking chapter for the Longest Common Subsequence problem. In the execution of that algorithm, many common subproblems were computed repeatedly. As an optimization, we can compute these subproblems once and then store the result to read back later. A recursive memoization algorithm can be turned “bottom-up” into an iterative algorithm that fills in a table of solutions to subproblems. Some of the subproblems solved might not be needed by the end result (and that is where dynamic programming differs from memoization), but dynamic programming can be very efficient because the iterative version can better use the cache and have less call overhead. Asymptotically, dynamic programming and memoization have the same complexity.

So how would a fibonacci program using memoization work? Consider the following program ($f[n]$ contains the nth Fibonacci-number if has been calculated, -1 otherwise):

function **fib**(integer *n*): integer if *n* == 0 **or** n == 1: return *n* else-if $f[n]$!= -1: return $f[n]$ else $f[n]$ = **fib**(*n* - 1) + **fib**(*n* - 2) return $f[n]$ fi end
This code sample is also available in Ada.

The code should be pretty obvious. If the value of fib(n) already has been calculated it’s stored in f[n] and then returned instead of calculating it again. That means all the copies of the sub-call trees are removed from the calculation.



The values in the blue boxes are values that already have been calculated and the calls can thus be skipped. It is thus a lot faster than the straight-forward recursive algorithm. Since every value less than n is calculated once, and only once, the first time you execute it, the asymptotic running time is $O(n)$. Any other calls to it will take $O(1)$ since the values have been precalculated (assuming each subsequent call’s argument is less than n).

The algorithm does consume a lot of memory. When we calculate fib(*n*), the values fib(0) to fib(n) are stored in main memory. Can this be improved? Yes it can, although the $O(1)$ running time of subsequent calls are obviously lost since the values aren't stored. Since the value of fib(*n*) only depends on fib(*n-1*) and fib(*n-2*) we can discard the other values by going bottom-up. If we want to calculate fib(*n*), we first calculate fib(2) = fib(0) + fib(1). Then we can calculate fib(3) by adding fib(1) and fib(2). After that, fib(0) and fib(1) can be discarded, since we don't need them to calculate any more values. From fib(2) and fib(3) we calculate fib(4) and discard fib(2), then we calculate fib(5) and discard fib(3), etc. etc. The code goes something like this:

function **fib**(integer *n*): integer if *n* == 0 **or** n == 1: return *n* fi let *u* := 0 let *v* := 1 for *i* := 2 to *n*: let *t* := *u* + *v* *u* := *v* *v* := *t* repeat return *v* end
This code sample is also available in Ada.

We can modify the code to store the values in an array for subsequent calls, but the point is that we don't *have* to. This method is typical for dynamic programming. First we identify what subproblems need to be solved in order to solve the entire problem, and then we calculate the values bottom-up using an iterative process.

6.2 Longest Common Subsequence (DP version)

This will remind us of the backtracking version and then improve it via memoization. Finally, the recursive algorithm will be made iterative and be full-fledged DP. [TODO: write this section]

6.3 Matrix Chain Multiplication

Suppose that you need to multiply a series of n matrices $M_1, \ldots, M_n$ together to form a product matrix P :

$$P = M_1 \cdot M_2 \cdots M_{n-1} \cdot M_n$$

This will require $n-1$ multiplications, but what is the fastest way we can form this product? Matrix multiplication is associative, that is,

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

for any A, B, C , and so we have some choice in what multiplication we perform first. (Note that matrix multiplication is *not* commutative, that is, it does not hold in general that $A \cdot B = B \cdot A$.)

Because you can only multiply two matrices at a time the product $M_1 \cdot M_2 \cdot M_3 \cdot M_4$ can be paranthesized in these ways:

$$((M_1 M_2) M_3) M_4$$

$$(M_1 (M_2 M_3)) M_4$$

$$M_1 ((M_2 M_3) M_4)$$

$$(M_1 M_2)(M_3 M_4)$$

$$M_1 (M_2 (M_3 M_4))$$

Two matrices M_1 and M_2 can be multiplied if the number of columns in M_1 equals the number of rows in M_2 . The number of rows in their product will equal the number rows in M_1 and the number of columns will equal the number of columns in M_2 . That is, if the dimensions of M_1 is $a \times b$ and M_2 has dimensions $b \times c$ their product will have dimensions $a \times c$.

To multiply two matrices with each other we use a function called matrix-multiply that takes two matrices and returns their product. We will leave implementation of this function alone for the moment as it is not the focus of this chapter (how to multiply two matrices in the fastest way has been under intensive study for several years [TODO: propose this topic for the *Advanced* book]). The time this function takes to multiply two matrices of size $a \times b$ and $b \times c$ is proportional to the number of scalar multiplications, which is proportional to abc . Thus, paranthezation matters: Say that we have three matrices M_1 , M_2 and M_3 . M_1 has dimensions 5×100 , M_2 has dimensions 100×100 and M_3 has dimensions 100×50 . Let's paranthezise them in the two possible ways and see which way requires the least amount of multiplications. The two ways are

$$((M_1 M_2) M_3)$$

$$(M_1 (M_2 M_3))$$

To form the product in the first way requires 75000 scalar multiplications (5*100*100=50000 to form product $(M_1 M_2)$ and another 5*100*50=25000 for the last multiplications.) This might seem like a lot, but in comparison to the 525000 scalar multiplications required by the second parenthesization (50*100*100=500000 plus 5*50*100=25000) it is miniscule! You can see why determining the parenthesization is important: imagine what would happen if we needed to multiply 50 matrices!

6.3.1 Forming a Recursive Solution

Note that we concentrate on finding a how many scalar multiplications are needed instead of the actual order. This is because once we have found a working algorithm to find the amount it is trivial to create an algorithm for the actual parenthesization. It will, however, be discussed in the end.

So how would an algorithm for the optimum parenthesization look? By the chapter title you might expect that a dynamic programming method is in order (not to give the answer away or anything). So how would a dynamic programming method work? Because dynamic programming algorithms are based on optimal substructure, what would the optimal substructure in this problem be?

Suppose that the optimal way to parenthesize

$$M_1 M_2 \ldots M_n$$

splits the product at k :

$$(M_1 M_2 \ldots M_k)(M_{k+1} M_{k+2} \ldots M_n)$$

Then the optimal solution contains the optimal solutions to the two subproblems

$$(M_1 \ldots M_k)$$

$$(M_{k+1} \ldots M_n)$$

That is, just in accordance with the fundamental principle of dynamic programming, the solution to the problem depends on the solution of smaller sub-problems.

Let's say that it takes $c(n)$ scalar multiplications to multiply matrices M_n and M_{n+1} , and $f(m, n)$ is the number of scalar multiplications to be performed in an optimal parenthesization of the matrices $M_m \ldots M_n$. The definition of $f(m, n)$ is the first step toward a solution.

When $n - m = 1$, the formulation is trivial; it is just $c(m)$. But what is it when the distance is larger? Using the observation above, we can derive a formulation. Suppose an optimal solution to the problem divides the matrices at matrices k and k+1 (i.e. $(M_m \ldots M_k)(M_{k+1} \ldots M_n)$) then the number of scalar multiplications are.

$$f(m, k) + f(k + 1, n) + c(k)$$

That is, the amount of time to form the first product, the amount of time it takes to form the second product, and the amount of time it takes to multiply them together. But what is this optimal value k? The answer is, of course, the value that makes the above formula assume its minimum value. We can thus form the complete definition for the function:

$$f(m,n) = \begin{cases} \min_{m \le k < n} f(m,k) + f(k+1,n) + c(k) & \text{if } n - m > 1 \\ 0 & \text{if } n = m \end{cases}$$

A straight-forward recursive solution to this would look something like this *(the language is Wikicode)*:

```
function f($m$, $n$) {
    if $m$ == $n$
        return 0
    let $minCost$ := $\infty$
    for $k$ := $m$ to $n$ - 1 {
        v := f($m$, $k$) + f($k$ + 1, $n$) + $c(k)$
        if $v$ < $minCost$
            $minCost$ := $v$
    }
    return $minCost$
}
```

This rather simple solution is, unfortunately, not a very good one. It spends mountains of time recomputing data and its running time is exponential.

Using the same adaptation as above we get:

```
function f($m$, $n$) {
    if $m$ == $n$
        return 0
    else-if $f[m,n]$ != $-1$:
        return $f[m,n]$
    fi
    let $minCost$ := $\infty$
    for $k$ := $m$ to $n$ - 1 {
        v := f($m$, $k$) + f($k$ + 1, $n$) + $c(k)$
        if $v$ < $minCost$
            $minCost$ := $v$
    }
    $f[m,n]$=minCost
    return $minCost$
}
```

6.4 Parsing Any Context-Free Grammar

Note that special types of context-free grammars can be parsed much more efficiently than this technique, but in terms of generality, the DP method is the only way to go.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

Chapter 7

Greedy Algorithms

In the backtracking algorithms we looked at, we saw algorithms that found decision points and recursed over all options from that decision point. A **greedy algorithm** can be thought of as a backtracking algorithm where at each decision point "the best" option is already known and thus can be picked without having to recurse over any of the alternative options.

The name "greedy" comes from the fact that the algorithms make decisions based on a single criterion, instead of a global analysis that would take into account the decision's effect on further steps. As we will see, such a backtracking analysis will be unnecessary in the case of greedy algorithms, so it is not greedy in the sense of causing harm for only short-term gain.

Unlike backtracking algorithms, greedy algorithms can't be made for every problem. Not every problem is "solvable" using greedy algorithms. Viewing the finding solution to an optimization problem as a hill climbing problem greedy algorithms can be used for only those hills where at every point taking the steepest step would lead to the peak always.

Greedy algorithms tend to be very efficient and can be implemented in a relatively straightforward fashion. Many a times in O(n) complexity as there would be a single choice at every point. However, most attempts at creating a correct greedy algorithm fail unless a precise proof of the algorithm's correctness is first demonstrated. When a greedy strategy fails to produce optimal results on all inputs, we instead refer to it as a heuristic instead of an algorithm. Heuristics can be useful when speed is more important than exact results (for example, when "good enough" results are sufficient).

7.1 Event Scheduling Problem

The first problem we'll look at that can be solved with a greedy algorithm is the event scheduling problem. We are given a set of events that have a start time and finish time, and we need to produce a subset of these events such that no events intersect each other (that is, having overlapping times), and that we have the maximum number of events scheduled as possible.

Here is a formal statement of the problem:

> *Input*: *events*: a set of intervals (s_i, f_i) where s_i is the start time, and f_i is the finish time.
>
> *Solution*: A subset *S* of *Events*.
>
> *Constraint*: No events can intersect (start time exclusive). That is, for all intervals $i = (s_i, f_i), j = (s_j, f_j)$ where $s_i < s_j$ it holds that $f_i \leq s_j$.
>
> *Objective*: Maximize the number of scheduled events, i.e. maximize the size of the set *S*.

We first begin with a backtracking solution to the problem:

// event-schedule -- schedule as many non-conflicting events as possible function **event-schedule**(*events* array of *s*[1..*n*], *j*[1..*n*]): set if *n* == 0: return ∅ fi if *n* == 1: return {*events*[1]} fi let *event* := *events*[1] let *S1* := union(**event-schedule**(*events* - set of conflicting events), *event*) let *S2* := **event-schedule**(*events* - {*event*}) if *S1*.size() >= *S2*.size(): return *S1* else return *S2* fi end

The above algorithm will faithfully find the largest set of non-conflicting events. It brushes aside details of how the set

> *events* - set of conflicting events

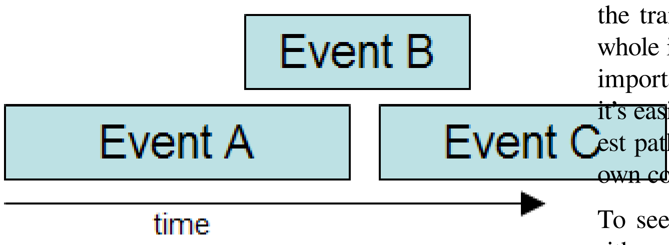
is computed, but it would require $O(n)$ time. Because the algorithm makes two recursive calls on itself, each with an argument of size $n-1$, and because removing conflicts takes linear time, a recurrence for the time this algorithm takes is:

$$T(n) = 2 \cdot T(n-1) + O(n)$$

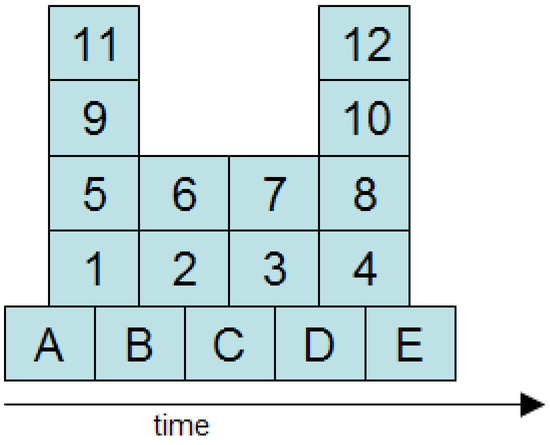
which is $O(2^n)$.

But suppose instead of picking just the first element in the array we used some other criterion. The aim is to just pick the "right" one so that we wouldn't need two recursive calls. First, let's consider the greedy strategy of picking the shortest events first, until we can add no more events without conflicts. The idea here is that the shortest events would likely interfere less than other events.

There are scenarios were picking the shortest event first produces the optimal result. However, here's a scenario where that strategy is sub-optimal:



Above, the optimal solution is to pick event A and C, instead of just B alone. Perhaps instead of the shortest event we should pick the events that have the least number of conflicts. This strategy seems more direct, but it fails in this scenario:



Above, we can maximize the number of events by picking A, B, C, D, and E. However, the events with the least conflicts are 6, 2 and 7, 3. But picking one of 6, 2 and one of 7, 3 means that we cannot pick B, C and D, which includes three events instead of just two.

7.1.1 = Longest Path solution to critical path scheduling of jobs

Construction with dependency constraints but concurrency can use critical path determination to find minimum time feasible, which is equivalent to a longest path in a directed acyclic graph problem. By using relaxation and breath first search , the shortest path can be the longest path by negating weights(time constraint) , finding solution, then restoring the positive weights. (Relaxation is determining the parent with least accumulated weight for each adjacent node being scheduled to be visited)

7.2 Dijkstra's Shortest Path Algorithm

With two (high-level, pseudocode) transformations, Dijsktra's algorithm can be derived from the much less efficient backtracking algorithm. The trick here is to prove the transformations maintain correctness, but that's the whole insight into Dijkstra's algorithm anyway. [TODO: important to note the paradox that to solve this problem it's easier to solve a more-general version. That is, shortest path from s to all nodes, not just to t. Worthy of its own colored box.]

To see the workings of Dijkstra's Shortest Path Algorithm, take an example:

There is a start and end node, with 2 paths between them ; one path has cost 30 on first hop, then 10 on last hop to the target node, with total cost 40. Another path cost 10 on first hop, 10 on second hop, and 40 on last hop, with total cost 60.

The start node is given distance zero so it can be at the front of a shortest distance queue, all the other nodes are given infinity or a large number e.g. 32767 .

This makes the start node the first current node in the queue.

With each iteration, the current node is the first node of a shortest path queue. It looks at all nodes adjacent to the current node;

For the case of the start node, in the first path it will find a node of distance 30, and in the second path, an adjacent node of distance 10. The current nodes distance , which is zero at the beginning, is added to distances of the adjacent nodes, and the distances from the start node of each node are updated , so the nodes will be 30+0 = 30 in the 1st path , and 10+0=10 in the 2nd path.

Importantly, also updated is a previous pointer attribute for each node, so each node will point back to the current node, which is the start node for these two nodes.

Each node's priority is updated in the priority queue using the new distance.

That ends one iteration. The current node was removed from the queue before examining its adjacent nodes.

In the next iteration, the front of the queue will be the node in the second path of distance 10, and it has only one adjacent node of distance 10, and that adjacent node will distance will be updated from 32767 to 10 (the current node distance) + 10 (the distance from the current node) = 20.

In the next iteration, the second path node of cost 20 will be examined, and it has one adjacent hop of 40 to the target node, and the target nodes distance is updated from 32767 to 20 + 40 = 60 . The target node has its priority updated.

In the next iteration, the shortest path node will be the first path node of cost 30, and the target node has not been yet removed from the queue. It is also adjacent to the target node, with the total distance cost of 30 + 10 = 40.

Since 40 is less than 60, the previous calculated distance of the target node, the target node distance is updated to 40, and the previous pointer of the target node is updated to the node on the first path.

In the final iteration, the shortest path node is the target node, and the loop exits.

Looking at the previous pointers starting with the target node, a shortest path can be reverse constructed as a list to the start node.

Given the above example, what kind of data structures are needed for the nodes and the algorithm ?

# author , copyright under GFDL class Node : def __init__(self, label, distance = 32767): # a bug in constructor, uses a shared map initializer # , adjacency_distance_map = {}): self.label = label self.adjacent = {} # this is an adjacency map, with keys nodes, and values the adjacent distance self.distance = distance # this is the updated distance from the start node, used as the node's priority # default distance is 32767 self.shortest_previous = None #this the last shortest distance adjacent node # the logic is that the last adjacent distance added is recorded , for any distances of the same node added def add_adjacent(self, local_distance, node): self.adjacent[node]=local_distance print "adjacency to ", self.label, " of ", self.adjacent[node], " to ", \ node.label def get_adjacent(self) : return self.adjacent.iteritems() def update_shortest(self, node): new_distance = node.adjacent[self] + node.distance #DEBUG print "for node ", node.label, " updating ", self.label, \ " with distance ", node.distance , \ " and adjacent distance ", node.adjacent[self] updated = False # node's adjacency map gives the adjacent distance for this node # the new distance for the path to this (self)node is the adjacent distance plus the other node's distance if new_distance < self.distance : # if it is the shortest distance then record the distance, and make the previous node that node self.distance = new_distance self.shortest_previous= node updated = True return updated MAX_IN_PQ = 100000 class PQ: def __init__(self , sign = −1): self.q = [None] * MAX_IN_PQ # make the array preallocated self.sign = sign # a negative sign is a minimum priority queue self.end = 1 # this is the next slot of the array (self.q) to be used , self.map = {} def insert(self, priority, data): self.q[self.end] = (priority, data) # sift up after insert p = self.end self.end = self.end + 1 self.sift_up(p) def sift_up(self, p): # p is the current node's position # q[p][0] is the priority, q[p][1] is the item or node # while the parent exists (p >= 1) , and parent's priority is less than the current node's priority while p / 2 != 0 and self.q[p/2][0]*self.sign < self.q[p][0]*self.sign: # swap the parent and the current node, and make the current node's position the parent's position tmp = self.q[p] self.q[p] = self.q[p/2] self.q[p/2] = tmp self.map[self.q[p][1]] = p p = p/2 # this map's the node to the position in the priority queue self.map[self.q[p][1]] = p return p def remove_top(self): if self.end == 1 : return (−1, None) (priority, node) = self.q[1] # put the end of the heap at the top of the heap, and sift it down to adjust the heap # after the heap's top has been removed. this takes log2(N) time, where N iis the size of the heap. self.q[1] = self.q[self.end-1] self.end = self.end - 1 self.sift_down(1) return (priority, node) def sift_down(self, p): while 1: l = p * 2 # if the left child's position is more than the size of the heap, # then left and right children don't exist if (l > self.end) : break r= l + 1 # the selected child node should have the greatest priority t = l if r < self.end and self.q[r][0]*self.sign > self.q[l][0]*self.sign : t = r print "checking for sift down of ", self.q[p][1].label, self.q[p][0], " vs child ", self.q[t][1].label, self.q[t][0] # if the selected child with the greatest priority has a higher priority than the current node if self.q[t] [0] * self. sign > self.q [p] [0] * self.sign : # swap the current node with that child, and update the mapping of the child node to its new position tmp = self. q [t] self. q [t] = self.q [p] self. q [p] = tmp self.map [tmp [1]] = p p = t else: break # end the swap if the greatest priority child has a lesser priority than the current node # after the sift down, update the new position of the current node. self.map [self.q[p][1]] = p return p def update_priority(self, priority, data) : p = self. map[data] print "priority prior update", p, "for priority", priority, " previous priority", self.q[p][0] if p is None : return −1 self.q[p] = (priority, self.q[p][1]) p = self.sift_up(p) p = self.sift_down(p) print "updated ", self.q[p][1].label , p, "priority now ", self.q[p][0] return p class NoPathToTargetNode (BaseException): pass def test_1() : st = Node('start', 0) p1a = Node('p1a') p1b = Node('p1b') p2a = Node('p2a') p2b = Node('p2b') p2c = Node('p2c') p2d = Node('p2d') targ = Node('target') st.add_adjacent (30, p1a) #st.add_adjacent (10, p2a) st.add_adjacent (20, p2a) #p1a.add_adjacent(10, targ) p1a.add_adjacent(40, targ) p1a.add_adjacent(10, p1b) p1b.add_adjacent(10, targ) # testing alternative #p1b.add_adjacent(20, targ) p2a.add_adjacent(10, p2b) p2b.add_adjacent(5,p2c) p2c.add_adjacent(5,p2d) #p2d.add_adjacent(5,targ) #chooses the alternate path p2d.add_adjacent(15,targ) pq = PQ() # st.distance is 0, but the other's have default starting distance 32767 pq.insert(st.distance, st) pq.insert(p1a.distance, p1a) pq.insert(p2a.distance, p2a) pq.insert(p2b.distance, p2b) pq.insert(targ.distance, targ) pq.insert(p2c.distance, p2c) pq.insert(p2d.distance, p2d) pq.insert(p1b.distance, p1b) node = None while node != targ : (pr, node) = pq.remove_top() #debug print "node ", node.label, " removed from top " if node is None: print "target node not in queue" raise elif pr == 32767: print "max distance encountered so no further nodes updated. No path to target node." raise NoPathToTargetNode # update the distance to the start node using this node's distance to all of the nodes adjacent to it,

and update its priority if # a shorter distance was found for an adjacent node (.update_shortest(..) returns true). # this is the greedy part of the dijsktra's algorithm, always greedy for the shortest distance using the priority queue. for adj_node , dist in node.get_adjacent(): #debug print "updating adjacency from ", node.label, " to ", adj_node.label if adj_node.update_shortest(node): pq.update_priority(adj_node.distance, adj_node) print "node and targ ", node, targ , node <> targ print "length of path", targ.distance print " shortest path" #create a reverse list from the target node, through the shortes path nodes to the start node node = targ path = [] while node <> None : path.append(node) node = node. shortest_previous for node in reversed(path): # new iterator version of list.reverse() print node.label if __name__ == "__main__": test_1()

7.3 Minimum spanning tree

Greedily looking for the minimum weight edges ; this could be achieved with sorting edges into a list in ascending weight. Two well known algorithms are Prim's Algorithm and Kruskal's Algorithm. Kruskal selects the next minimum weight edge that has the condition that no cycle is formed in the resulting updated graph. Prim's algorithm selects a minimum edge that has the condition that only one edge is connected to the tree. For both the algorithms, it looks that most work will be done verifying an examined edge fits the primary condition. In Kruskal's, a search and mark technique would have to be done on the candidate edge. This will result in a search of any connected edges already selected, and if a marked edge is encountered, than a cycle has been formed. In Prim's algorithm, the candidate edge would be compared to the list of currently selected edges, which could be keyed on vertex number in a symbol table, and if both end vertexes are found, then the candidate edge is rejected.

7.4 Maximum Flow in weighted graphs

In a flow graph, edges have a forward capacity , a direction, and a flow quantity in the direction and less than or equal to the forward capacity. Residual capacity is capacity-flow in the direction of the edge, and flow in the other direction.

Maxflow in Ford Fulkerson method requires a step to search for a viable path from a source to a sink vertex, with non-zero residual capacities at each step of the path. Then the minimum residual capacity determines the maximum flow for this path. Multiple iterations of searches using BFS can be done (the Edmond-Karp algorithm), until the sink vertex is not marked when the last node is off the queue or stack. All marked nodes in the last iteration are said to be in the minimum cut.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

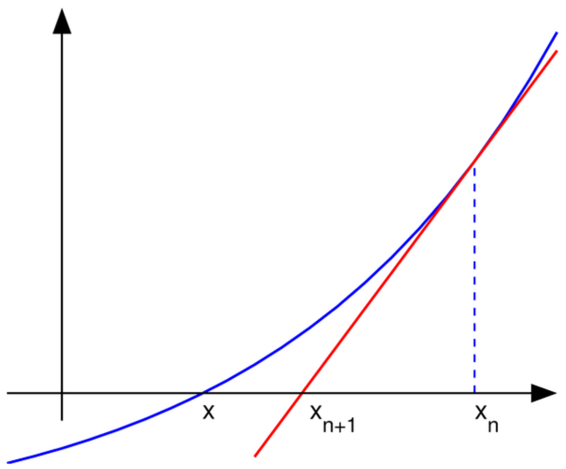
Chapter 8

Hill Climbing

Hill climbing is a technique for certain classes of optimization problems. The idea is to start with a sub-optimal solution to a problem (i.e., *start at the base of a hill*) and then repeatedly improve the solution (*walk up the hill*) until some condition is maximized (*the top of the hill is reached*).

One of the most popular hill-climbing problems is the network flow problem. Although network flow may sound somewhat specific it is important because it has high expressive power: for example, many algorithmic problems encountered in practice can actually be considered special cases of network flow. After covering a simple example of the hill-climbing approach for a numerical problem we cover network flow and then present examples of applications of network flow.

8.1 Newton's Root Finding Method



An illustration of Newton's method: The zero of the f(x) *function is at* x. *We see that the guess* x_{n+1} *is a better guess than* x_n *because it is closer to* x. *(*from Wikipedia*)*

Newton's Root Finding Method is a three-centuries-old algorithm for finding numerical approximations to roots of a function (that is a point x where the function $f(x)$ becomes zero), starting from an initial guess. You need to know the function $f(x)$ and its first derivative $f'(x)$ for this algorithm. The idea is the following: In the vicinity of the initial guess x_0 we can form the Taylor expansion of the function

$$f(x) = f(x_0 + \epsilon) \approx f(x_0) + \epsilon f'(x_0) + \frac{\epsilon^2}{2} f''(x_0) + \ldots$$

which gives a good approximation to the function near x_0 . Taking only the first two terms on the right hand side, setting them equal to zero, and solving for ϵ , we obtain

$$\epsilon = -\frac{f(x_0)}{f'(x_0)}$$

which we can use to construct a better solution

$$x_1 = x_0 + \epsilon = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

This new solution can be the starting point for applying the same procedure again. Thus, in general a better approximation can be constructed by repeatedly applying

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

As shown in the illustration, this is nothing else but the construction of the zero from the tangent at the initial guessing point. In general, Newton's root finding method converges quadratically, except when the first derivative of the solution $f'(x) = 0$ vanishes at the root.

Coming back to the "Hill climbing" analogy, we could apply Newton's root finding method not to the function $f(x)$, but to its first derivative $f'(x)$, that is look for x such that $f'(x) = 0$. This would give the extremal positions of the function, its maxima and minima. Starting Newton's method close enough to a maximum this way, we climb the hill.

Example application of Newton's method

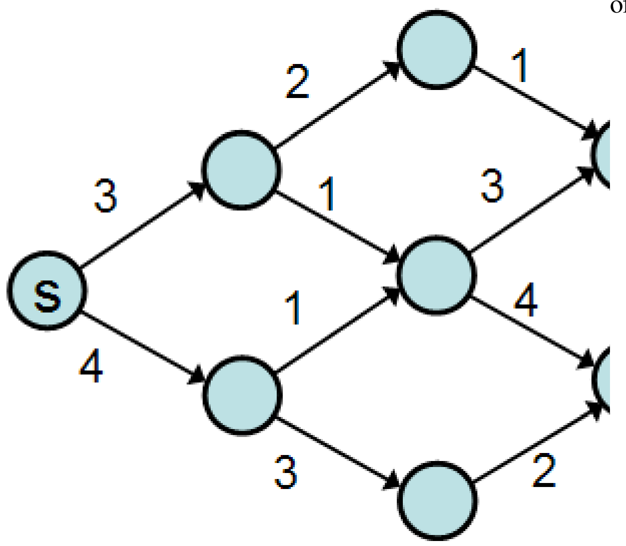
The net present value function is a function of time, an interest rate, and a series of cash flows. A related function is Internal Rate of Return. The formula for each period is (CF_i x (1+ i/100) ^{t} , and this will give a polynomial function which is the total cash flow, and equals zero when the interest rate equals the IRR. In using Newton's method, x is the interest rate, and y is the total cash flow, and the method will use the derivative function of the polynomial to find the slope of the graph at a given interest rate (x-value), which will give the x_{n+1} , or a better interest rate to try in the next iteration to find the target x where y (the total returns) is zero.

Instead of regarding continuous functions, the hill-climbing method can also be applied to discrete networks.

8.2 Network Flow

Suppose you have a directed graph (possibly with cycles) with one vertex labeled as the source and another vertex labeled as the destination or the "sink". The source vertex only has edges coming out of it, with no edges going into it. Similarly, the destination vertex only has edges going into it, with no edges coming out of it. We can assume that the graph fully connected with no dead-ends; i.e., for every vertex (except the source and the sink), there is at least one edge going into the vertex and one edge going out of it.

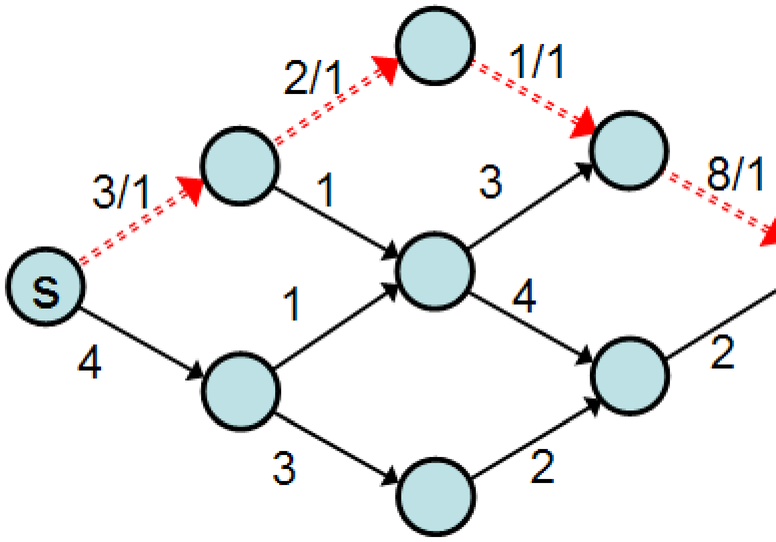
We assign a "capacity" to each edge, and initially we'll consider only integral-valued capacities. The following graph meets our requirements, where "s" is the source and "t" is the destination:



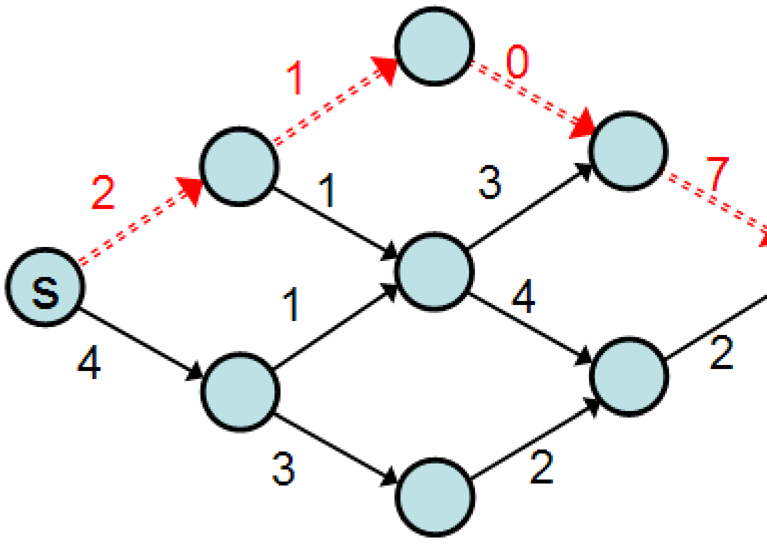
We'd like now to imagine that we have some series of inputs arriving at the source that we want to carry on the edges over to the sink. The number of units we can send on an edge at a time must be less than or equal to the edge's capacity. You can think of the vertices as cities and the edges as roads between the cities and we want to send as many cars from the source city to the destination city as possible. The constraint is that we cannot send more cars down a road than its capacity can handle.

The goal of network flow is to send as much traffic from s to t as each street can bear.

To organize the traffic routes, we can build a list of different paths from city s to city t . Each path has a carrying capacity equal to the smallest capacity value for any edge on the path; for example, consider the following path p :



Even though the final edge of p has a capacity of 8, that edge only has one car traveling on it because the edge before it only has a capacity of 1 (thus, that edge is at full capacity). After using this path, we can compute the **residual graph** by subtracting 1 from the capacity of each edge:



(We subtracted 1 from the capacity of each edge in p because 1 was the carrying capacity of p .) We can say that path p has a flow of 1. Formally, a **flow** is an assignment $f(e)$ of values to the set of edges in the graph $G = (V, E)$ such that:

$$\forall e \in E : f(e) \in \mathbb{R}$$

$$\forall (u,v) \in E : f((u,v)) = -f((v,u))$$

$$\forall u \in V, u \neq s,t : \sum_{v \in V} f(u,v) = 0$$

$$\forall e \in E : f(e) \leq c(e)$$

Where s is the source node and t is the sink node, and $c(e) \geq 0$ is the capacity of edge e . We define the value of a flow f to be:

$$\text{Value}(f) = \sum_{v \in V} f((s,v))$$

The goal of network flow is to find an f such that Value(f) is maximal. To be maximal means that there is no other flow assignment that obeys the constraints 1-4 that would have a higher value. The traffic example can describe what the four flow constraints mean:

1. $\forall e \in E : f(e) \in \mathbb{R}$. This rule simply defines a flow to be a function from edges in the graph to real numbers. The function is defined for every edge in the graph. You could also consider the “function” to simply be a mapping: Every edge can be an index into an array and the value of the array at an edge is the value of the flow function at that edge.

2. $\forall (u,v) \in E : f((u,v)) = -f((v,u))$. This rule says that if there is some traffic flowing from node *u* to node *v* then there should be considered negative that amount flowing from *v* to *u*. For example, if two cars are flowing from city *u* to city *v*, then negative two cars are going in the other direction. Similarly, if three cars are going from city *u* to city *v* and two cars are going city *v* to city *u* then the net effect is the same as if one car was going from city *u* to city *v* and no cars are going from city *v* to city *u*.

3. $\forall u \in V, u \neq s,t : \sum_{v \in V} f(u,v) = 0$. This rule says that the net flow (except for the source and the destination) should be neutral. That is, you won't ever have more cars going into a city than you would have coming out of the city. New cars can only come from the source, and cars can only be stored in the destination. Similarly, whatever flows out of *s* must eventually flow into *t*. Note that if a city has three cars coming into it, it could send two cars to one city and the remaining car to a different city. Also, a city might have cars coming into it from multiple sources (although all are ultimately from city *s*).

4. $\forall e \in E : f(e) \leq c(e)$.

8.3 The Ford-Fulkerson Algorithm

The following algorithm computes the maximal flow for a given graph with non-negative capacities. What the algorithm does can be easy to understand, but it’s non-trivial to show that it terminates and provides an optimal solution.

function **net-flow**(graph (*V*, *E*), node *s*, node *t*, cost *c*): flow initialize *f*(*e*) := 0 for all *e* in *E* loop while not *done* for all *e* in *E*: *// compute residual capacities* let *cf*(*e*) := *c*(*e*) - *f*(*e*) repeat let *Gf* := (*V*, {*e* : *e* in *E* and *cf*(*e*) > 0}) find a path *p* from *s* to *t* in *Gf // e.g., use depth first search* if no path *p* exists: signal *done* let *path-capacities* := map(*p*, *cf*) *// a path is a set of edges* let *m* := min-val-of(*path-capacities*) *// smallest residual capacity of p* for all (*u*, *v*) in *p*: *// maintain flow constraints f*((*u*, *v*)) := *f*((*u*, *v*)) + *m f*((*v*, *u*)) := *f*((*v*, *u*)) - *m* repeat repeat end

The Ford-Fulkerson algorithm uses repeated calls to Breadth-First Search (use a queue to schedule the children of a node to become the current node). Breadth-First Search increments the length of each path +1 so that the first path to get to the destination, the shortest path, will be the first off the queue. This is in contrast with using a Stack, which is Depth-First Search, and will come up with *any* path to the target, with the “descendants” of current node examined, but not necessarily the shortest.

- In one search, a path from source to target is found. All nodes are made unmarked at the beginning of a new search. Seen nodes are “marked” , and not searched again if encountered again. Eventually, all reachable nodes will have been scheduled on the queue , and no more unmarked nodes can be reached off the last the node on the queue.

- During the search, nodes scheduled have the finding “edge” (consisting of the current node , the found node, a current flow, and a total capacity in the direction first to second node), recorded.

- This allows finding a reverse path from the target node once reached, to the start node. Once a path is found, the edges are examined to find the edge with the minimum remaining capacity, and this becomes the flow that will result along this path , and this quantity is removed from the remaining capacity of each edge along the path. At the “bottleneck” edge with the minimum remaining capacity, no more flow will be possible, in the forward direction, but still possible in the backward direction.

- This process of BFS for a path to the target node, filling up the path to the bottleneck edge’s residual capacity, is repeated, until BFS cannot find a path to the target node (the node is not reached because all sequences of edges leading to the target have had their bottleneck edges filled). Hence memory of the side effects of previous paths found, is recorded in the flows of the edges, and affect the results of future searches.

- An important property of maximal flow is that flow can occur in the backward direction of an edge, and the residual capacity in the backward direction is the current flow in the foward direction. Normally, the residual capacity in the forward direction of an edge is the initial capacity less forward flow. Intuitively, this allows more options for maximizing flow as earlier augmenting paths block off shorter paths.
- On termination, the algorithm will retain the marked and unmarked states of the results of the last BFS.
- the minimum cut state is the two sets of marked and unmarked nodes formed from the last unsuccessful BFS starting from the start node, and not marking the target the node. The start node belongs to one side of the cut, and the target node belongs to the other. Arbitrarily, being "in Cut" means being on the start side, or being a marked node. Recall how are a node comes to be marked, given an edge with a flow and a residual capacity.

Example application of Ford-Fulkerson maximum flow/ minimum cut

An example of application of Ford-Fulkerson is in baseball season elimination. The question is whether the team can possibly win the whole season by exceeding some combination of wins of the other teams.

The idea is that a flow graph is set up with teams not being able to exceed the number of total wins which a target team can maximally win for the entire season. There are game nodes whose edges represent the number of remaining matches between two teams, and each game node outflows to two team nodes, via edges that will not limit forward flow; team nodes receive edges from all games they participate. Then outflow edges with win limiting capacity flow to the virtual target node. In a maximal flow state where the target node's total wins will exceed some combination of wins of the other teams, the penultimate depth-first search will cutoff the start node from the rest of the graph, because no flow will be possible to any of the game nodes, as a result of the penultimate depth-first search (recall what happens to the flow , in the second part of the algorithm after finding the path). This is because in seeking the maximal flow of each path, the game edges' capacities will be maximally drained by the win-limit edges further along the path, and any residual game capacity means there are more games to be played that will make at least one team overtake the target teams' maximal wins. If a team node is in the minimum cut, then there is an edge with residual capacity leading to the team, which means what , given the previous statements? What do the set of teams found in a minimum cut represent (hint: consider the game node edge) ?

Example Maximum bipartite matching (intern matching)

This matching problem doesn't include preference weightings. A set of companies offers jobs which are made into one big set , and interns apply to companies for specific jobs. The applications are edges with a weight of 1. To convert the bipartite matching problem to a maximum flow problem, virtual vertexes s and t are created , which have weighted 1 edges from s to all interns, and from all jobs to t. Then the Ford-Fulkerson algorithm is used to sequentially saturate 1 capacity edges from the graph, by augmenting found paths. It may happen that a intermediate state is reached where left over interns and jobs are unmatched, but backtracking along reverse edges which have residual capacity = forward flow = 1, along longer paths that occur later during breadth-first search, will negate previous suboptimal augmenting paths, and provide further search options for matching, and will terminate only when maximal flow or maximum matching is reached.

Chapter 9

Unweighted Graph Algorithms

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A* Please edit and omit unweighted in title

9.1 Representation of Graph

9.1.1 Adjacency Matrix

Double plus good.

9.1.2 Adjacency List

HeLLO , difference?!

9.1.3 Comparison

the list might work better with level 1 cache with adjacency objects (which node, visited, inPath, pathWeight, fromWhere).

9.2 Depth First Search

9.2.1 Pseudocode

dfs(vertex w) if w has already been marked visited return mark w as visited for each adjacent vertex v dfs(v)

Non recursive DFS is more difficult. It requires that each node keep memory of the last child visited, as it descends the current child. One implementation uses a indexed array of iterators, so on visiting a node, the node's number is an index into an array that stores the iterator for the nodes child. Then the first iterator value is pushed onto the job stack. Peek not pop is used to examine the current top of the stack, and pop is only invoked when the iterator for the peeked node is exhausted.

9.2.2 Properties

9.2.3 Classification of Edge

Tree Edge

Backward Edge

Forward Edge

Cross Edge

IT is good techniques from :Yogesh Jakhar

9.3 Breadth First Search

9.3.1 Pseudocode

bfs (x):

q insert x; while (q not empty) y = remove head q visit y mark y for each z adjacent y q add tail z

9.3.2 Example

9.3.3 Correctness

9.3.4 Analysis

9.3.5 Usage

A breadth first search can be used to explore a database schema, in an attempt to turn it into an xml schema. This is done by naming the root table, and then doing a referential breadth first search . The search is both done on the refering and referred ends, so if another table refers to to the current node being searched, than that table has a one-to-many relationship in the xml schema, otherwise it is many-to-one.

9.4 Classical Graph Problems

9.4.1 Directed graph cycle detection

In a directed graph, check is *acyclic* by having a second marker on before dfs recursive call and off after, and checking for second marker before original mark check in dfs adjacency traversal. If second marker present, then cycle exists.

9.4.2 Topological Sort of Directed Graph

1. check for cycle as in previous section.
2. dfs the acyclic graph. Reverse postorder by storing on stack instead of queue after dfs calls on adjacent nodes.

The last node on stack must be there from first dfs call, and removing the last node exposes the second node which can only have been reached by last node. Use induction to show topological order.

9.4.3 Strongly Connected Components in Directed Graphs

1. Strong connected components have cycles within and must by acyclic between components (the kernel directed acyclic graph).
2. Difference between dfs reverse postorder of original graph vs the same on reverse graph is that first node is least dependent in latter. Thus all non strong connected nodes will be removed first by dfs on the original graph in the latter's order, and then dfs will remove only strongly connected nodes by marking , one SC component at each iteration over reverse postorder of reverse graph , visiting unmarked nodes only. Each outgoing edge from a SC component being traversed will go to an already marked node due to reverse postorder on reverse graph.

9.4.4 Articulation Vertex

9.4.5 Bridge

9.4.6 Diameter

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

Chapter 10

Distance approximations

Calculating distances is common in spatial and other search algorithms, as well as in computer game physics engines. However, the common Euclidean distance requires calculating **square roots**, which is often a relatively heavy operation on a CPU.

10.1 You don't need a square root to *compare* distances

Given (x1, y1) and (x2, y2), which is closer to the origin by Euclidean distance? You might be tempted to calculate the two Euclidean distances, and compare them:

d1 = sqrt(x1^2 + y1^2) d2 = sqrt(x2^2 + y2^2) return d1 > d2

But those square roots are often heavy to compute, and what's more, *you don't need to compute them at all.* Do this instead:

dd1 = x1^2 + y1^2 dd2 = x2^2 + y2^2 return dd1 > dd2

The result is exactly the same (because the positive square root is a *strictly monotonic* function). This only works for *comparing* distances though, not for calculating individual values, which is sometimes what you need. So we look at approximations.

10.2 Approximations of Euclidean distance

10.2.1 Taxicab/Manhattan/L1

The w:taxicab distance is one of the simplest to compute, so use it when you're very tight on resources:

Given two points (x1, y1) and (x2, y2),

$dx = |x1 - x2|$ (w:absolute value)

$dy = |y1 - y2|$

$d = dx + dy$ (taxicab distance)

Note that you can also use it as a "first pass" since it's **never lower** than the Euclidean distance. You could check if data points are within a particular bounding box, as a first pass for checking if they are within the bounding sphere that you're really interested in. In fact, if you take this idea further, you end up with an efficient spatial data structure such as a w:Kd-tree.

However, be warned that taxicab distance is **not w:isotropic** - if you're in a Euclidean space, taxicab distances change a lot depending on which way your "grid" is aligned. This can lead to big discrepancies if you use it as a drop-in replacement for Euclidean distance. Octagonal distance approximations help to knock some of the problematic corners off, giving better isotropy:

10.2.2 Octagonal

A fast approximation of 2D distance based on an octagonal boundary can be computed as follows.

Given two points (p_x, p_y) and (q_x, q_y) , Let $dx = |p_x - q_x|$ (w:absolute value) and $dy = |p_y - q_y|$. If $dy > dx$, approximated distance is $0.41dx + 0.941246dy$.

> Some years ago I developed a similar distance approximation algorithm using three terms, instead of just 2, which is much more accurate, and because it uses power of 2 denominators for the coefficients can be implemented without using division hardware. The formula is: *1007/1024 max(*|x|,|y|*) + 441/1024 min(*|x|,|y|*) - if (max(*|x|.|y|*)<16min(*|x|,|y|*), 40/1024 max(*|x|,|y|*), 0).* Also it is possible to implement a distance approximation without using either multiplication or division when you have very limited hardware: *(((max << 8) + (max << 3) - (max << 4) - (max << 1) + (min << 7) - (min << 5) + (min << 3) - (min << 1)) >> 8).* This is just like the 2 coefficient min max algorithm presented earlier, but with the coefficients 123/128 and 51/128. I have an article about it at http://web.oroboro.com:90/rafael/docserv.php/index/programming/article/distance
> --Rafael

(Apparently that article has moved to http://www.flipcode.com/archives/Fast_Approximate_Distance_Functions.shtml ?)

(Source for this section)

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

Chapter 11

Appendix A: Ada Implementation

11.1 Introduction

Welcome to the Ada implementations of the Algorithms Wikibook. For those who are new to Ada Programming a few notes:

- All examples are fully functional with all the needed input and output operations. However, only the code needed to outline the algorithms at hand is copied into the text - the full samples are available via the download links. (Note: It can take up to 48 hours until the cvs is updated).
- We seldom use predefined types in the sample code but define special types suitable for the algorithms at hand.
- Ada allows for default function parameters; however, we always fill in and name all parameters, so the reader can see which options are available.
- We seldom use shortcuts - like using the attributes Image or Value for String <=> Integer conversions.

All these rules make the code more elaborate than perhaps needed. However, we also hope it makes the code easier to understand

11.2 Chapter 1: Introduction

The following subprograms are implementations of the *Inventing an Algorithm* examples.

11.2.1 To Lower

The Ada example code does not append to the array as the algorithms. Instead we create an empty array of the desired length and then replace the characters inside.

File: to_lower_1.adb (view, plain text, download page, browse all)

function To_Lower (C : Character) return Character renames Ada.Characters.Handling.To_Lower; -- tolower - translates all alphabetic, uppercase characters -- in str to lowercase function To_Lower (Str : String) return String is Result : String (Str'Range); begin for C in Str'Range loop Result (C) := To_Lower (Str (C)); end loop; return Result; end To_Lower;

Would the append approach be impossible with Ada? No, but it would be significantly more complex and slower.

11.2.2 Equal Ignore Case

File: to_lower_2.adb (view, plain text, download page, browse all)

-- equal-ignore-case -- returns true if s or t are equal, -- ignoring case function Equal_Ignore_Case (S : String; T : String) return Boolean is O : constant Integer := S'First - T'First; begin if T'Length /= S'Length then return False; -- if they aren't the same length, they -- aren't equal else for I in S'Range loop if To_Lower (S (I)) /= To_Lower (T (I + O)) then return False; end if; end loop; end if; return True; end Equal_Ignore_Case;

11.3 Chapter 6: Dynamic Programming

11.3.1 Fibonacci numbers

The following codes are implementations of the Fibonacci-Numbers examples.

Simple Implementation

File: fibonacci_1.adb (view, plain text, download page, browse all)

...

To calculate Fibonacci numbers negative values are not needed so we define an integer type which starts at 0. With the integer type defined you can calculate up until Fib (87). Fib (88) will result in an Constraint_Error.

```
type Integer_Type is range 0 .. 999_999_999_999_999_999;
```

You might notice that there is not equivalence for the assert (n >= 0) from the original example. Ada will test the correctness of the parameter *before* the function is called.

```
function Fib (n : Integer_Type) return Integer_Type is begin if n = 0 then return 0; elsif n = 1 then return 1; else return Fib (n - 1) + Fib (n - 2); end if; end Fib; ...
```

Cached Implementation

File: fibonacci_2.adb (view, plain text, download page, browse all)

...

For this implementation we need a special cache type can also store a −1 as "not calculated" marker

```
type Cache_Type is range −1 .. 999_999_999_999_999_999;
```

The actual type for calculating the fibonacci numbers continues to start at 0. As it is a subtype of the cache type Ada will automatically convert between the two. (the conversion is - of course - checked for validity)

```
subtype Integer_Type is Cache_Type range 0 .. Cache_Type'Last;
```

In order to know how large the cache need to be we first read the actual value from the command line.

```
Value : constant Integer_Type := Integer_Type'Value (Ada.Command_Line.Argument (1));
```

The Cache array starts with element 2 since Fib (0) and Fib (1) are constants and ends with the value we want to calculate.

```
type Cache_Array is array (Integer_Type range 2 .. Value) of Cache_Type;
```

The Cache is initialized to the first valid value of the cache type — this is −1.

```
F : Cache_Array := (others => Cache_Type'First);
```

What follows is the actual algorithm.

```
function Fib (N : Integer_Type) return Integer_Type is begin if N = 0 or else N = 1 then return N; elsif F (N) /= Cache_Type'First then return F (N); else F (N) := Fib (N - 1) + Fib (N - 2); return F (N); end if; end Fib; ...
```

This implementation is faithful to the original from the Algorithms book. However, in Ada you would normally do it a little different:

File: fibonacci_3.adb (view, plain text, download page, browse all)

when you use a slightly larger array which also stores the elements 0 and 1 and initializes them to the correct values

```
type Cache_Array is array (Integer_Type range 0 .. Value) of Cache_Type; F : Cache_Array := (0 => 0, 1 => 1, others => Cache_Type'First);
```

and then you can remove the first if path.

```
if N = 0 or else N = 1 then return N; elsif F (N) /= Cache_Type'First then
```

This will save about 45% of the execution-time (measured on Linux i686) while needing only two more elements in the cache array.

Memory Optimized Implementation

This version looks just like the original in WikiCode.

File: fibonacci_4.adb (view, plain text, download page, browse all)

```
type Integer_Type is range 0 .. 999_999_999_999_999_999; function Fib (N : Integer_Type) return Integer_Type is U : Integer_Type := 0; V : Integer_Type := 1; begin for I in 2 .. N loop Calculate_Next : declare T : constant Integer_Type := U + V; begin U := V; V := T; end Calculate_Next; end loop; return V; end Fib;
```

No 64 bit integers

Your Ada compiler does not support 64 bit integer numbers? Then you could try to use decimal numbers instead. Using decimal numbers results in a slower program (takes about three times as long) but the result will be the same.

The following example shows you how to define a suitable decimal type. Do experiment with the digits and range parameters until you get the optimum out of your Ada compiler.

File: fibonacci_5.adb (view, plain text, download page, browse all)

```
type Integer_Type is delta 1.0 digits 18 range 0.0 .. 999_999_999_999_999_999.0;
```

You should know that floating point numbers are unsuitable for the calculation of fibonacci numbers. They will not report an error condition when the number calculated becomes too large — instead they will lose in precision which makes the result meaningless.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*

11.4 Text and image sources, contributors, and licenses

11.4.1 Text

11.4.2 Images

11.4.3 Content license