ALGORITHMS

by Wikibooks contributors

From **Wikibooks**,
the open-content textbooks collection

Principal authors: Macneil Shonle (C) · Matthew Wilson (C) · Martin Krischik (C) · Jyasskin (C) · Gkhan (C) · WhirlWind (C)

The current version of this Wikibook may be found at:
http://en.wikibooks.org/wiki/Algorithms

Contents

0 Overview

Preamble

This book is about the creation and analysis of efficient algorithms. After introducing some necessary mathematical background this book covers:

- the **divide and conquer** technique;
- the use of **randomization** in algorithms;
- the general, but typically inefficient, **backtracking** technique;
- **dynamic programming** as an efficient optimization for some backtracking algorithms;
- **greedy algorithms** as an optimization of other kinds of backtracking algorithms; and
- **hill-climbing** techniques, including network flow.

The goal of the book is to show you how you can methodically apply different techniques to your own algorithms to make them more efficient. While this book mostly highlights general techniques, some well-known algorithms are also looked at in depth. This book is written so it can be read from "cover to cover" in the length of a semester, where sections marked with a $\star$ may be skipped.

This book is a tutorial on techniques and is not a reference. For references we highly recommend the tomes by [Knuth] and [CLRS]. Additionally, sometimes the best insights come from the primary sources themselves (e.g. [Hoare]).

Why a Wikibook on Algorithms?

A wikibook is an undertaking similar to an open-source software project: A contributor creates content for the project to help others, for personal enrichment, or to accomplish something for the contributor's own work (e.g., lecture preparation).

An open book, just like an open program, requires time to complete, but it can benefit greatly from even modest contributions from readers. For example you can fix "bugs" in the text (where the bug might be typographic, expository, technical, aesthetic or otherwise) in order to make a better book. If you find an opportunity to fix a bug, simply click on "edit", make your changes, and click on save. Other contributors may review your changes to be sure they are appropriate for the book. If you are unsure, you can visit the discussion page and ask there. Use common sense.

If you would like to make bigger contributions, you can take a look at the sections or chapters that are too short or otherwise need more work and start writing! Be sure to skim the rest of the book first in order to avoid duplication of content. Additionally, you should read the Guidelines for Contributors page for consistency tips and advice.

This book is intentionally kept narrow-in-focus in order to make contributions easier (because then the end-goal is clearer). This book is part two of a series of three computer science textbooks on algorithms, starting with *Data Structures* and ending with *Advanced Data Structures and Algorithms*. If you would like to contribute a topic not already listed in any of the three books try putting it in the *Advanced* book, which is more eclectic in nature. Or, if you think the topic is fundamental, you can go

to either the Algorithms discussion page or the Data Structures discussion page and make a proposal.

Additionally, implementations of the algorithms (in either Ada, C, Python, Java, or Scheme) as an appendix are welcome.

1 Introduction

live version • discussion • edit lesson • comment • report an error • ask a question

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.

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.

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, 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).

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 or 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.

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.

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.

2 Mathematical Background

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

Mathematical Definitions

A **set** is a collection of objects, containing at most one of each object. An object can be *anything*, including another set. The fundamental operation on a set is "member of", written "$\in$". Two sets are equal if and only if they contain the same members. The following are all sets:

{1,2}
: The set containing the integers "1" and "2".

$\mathbb{N}$
: The set of all natural numbers. We define $\mathbb{N}$ such that $0 \in \mathbb{N}$, but $\mathbb{N}$ is sometimes defined such that $0 \notin \mathbb{N}$.

$\mathbb{Z}$
: The set of all positive and negative integers and zero.

$\mathbb{Q}$
: The set of all rational numbers (i.e. well-defined ratios of two integers).

$\mathbb{R}$
: The set of all real numbers, rational numbers and numbers like π and *e* with infinite decimal expansions.

$\emptyset = \{\}$
: The set that contains nothing, known as the **empty set**.

We can also describe a new set using a *set comprehension* of the form

$$\{i \in S : P(i)\}$$

where *S* is an already-defined set and *P* is an arbitrary predicate (something that is either true or false of everything it is applied to). You get the new set by going through all of the elements *i* in *S* (whose elements you already know), and picking out the ones for which *P*(*i*) is true. For example,

$$\{i \in \mathbb{Z} : i \bmod 2 = 0\}$$

is the set of all even integers. Frequently in this book, we will leave out the $\in S$ and expect it to be implied from the context.

An **ordered n-tuple**, often called just an **n-tuple**, is an ordered group of *n* objects, possibly not all of the same type. It is written "$\langle a, b, c, \ldots \rangle$". The same object can appear more than once in a given n-tuple. Two n-tuples are considered equal when they have exactly the same objects in exactly the same positions. The most common n-tuple is the "ordered pair," written "$\langle a, b \rangle$".

We can compare sets based on their elements. The fundamental comparisons are true when one set contains all of the members of another set. In the following definitions, *A* and *B*, and *C* are sets.

Comparison name	Notation	Mathematical definition	English definition
Subset	$A \subseteq B$	$x \in A \Rightarrow x \in B$	All members of *A* are also members of *B*. For any set *A*, $\emptyset \subseteq A$ and $A \subseteq A$.
Proper subset	$A \subset B$	$A \subseteq B \wedge A \neq B$	This comparison is used when we want to exclude the set itself.
Superset	$A \supseteq B$	$B \subseteq A$	
Proper superset	$A \supset B$	$B \subset A$	

There are also many operations that apply to sets.

Operation name	Notation	Mathematical definition	English definition
Union	$A \cup B$	$\{x : x \in A \vee x \in B\}$	The set of things that are in either *A* or *B*.
Intersection	$A \cap B$	$\{x : x \in A \wedge x \in B\}$	The set of things that are in both *A* and *B*.
Set difference	$A \setminus B$ or sometimes *A* - *B*	$\{x : x \in A \wedge x \notin B\}$	The set of things in *A* but not *B*.
Complement	A^{-1}, A^C, or A'	$\{x \in U : x \notin A\}$	The set of things not in *A*. This is only well-defined with respect to some implied universal set *U*.
Power set	$\wp(A)$ or 2^A	$\{S : S \subset A\}$	The set of subsets of A. For any set *A*, $\emptyset \in \wp(A)$ and $A \in \wp(A)$.
Cartesian product	$A \times B$	$\{\langle a, b \rangle : a \in A \wedge b \in B\}$	The set of every pairing of an element from *A* with an element from *B*.

We can extend the Cartesian product definition to include n-ary products, defined as

$$A_1 \times \ldots \times A_n = \{\langle a_1, \ldots, a_n \rangle : a_i \in A_i \quad \forall 1 \leq i \leq n\}.$$

Using this extension, we can also define Cartesian exponentiation:

$$A^n = \underbrace{A \times \ldots \times A}_{\text{n times}}.$$

A **relation** *R* on a set *S* is a subset of $S \times S$. We often write xRy to mean $\langle x, y \rangle \in R$. Some common relations are: "is taller than", "knows about", "is derived from", "lives near", and "<". We can classify relations by what objects they relate:

A relation *R* on *S* is	when (for all *a*, *b*, and *c* in *S*)
reflexive	*aRa*
irreflexive	$\neg aRa$
symmetric	if *aRb* then *bRa*
antisymmetric	if *aRb* and *bRa*, then *a* = *b*
asymmetric	if *aRb* then $\neg bRa$
transitive	if *aRb* and *bRc* then *aRc*
intransitive	if *aRb* and *bRc* then $\neg aRc$
a weak partial order	it is reflexive, antisymmetric, and transitive
a weak total order	it is a weak partial order and either *aRb* or *bRa*
a strict partial order	it is irreflexive, asymmetric, and transitive
a strict total order	it is a strict partial order and either *aRb*, *bRa*, or *a* = *b*
well founded	there are no infinite descending chains in *S*

We usually use the symbol "≤" for a weak order and "<" for a strict order. The set of integers, the set of floating-point numbers, and the set of strings can all have a total ordering defined on them.

Note that none of the pairs of reflexive and irreflexive, symmetric and asymmetric, or transitive and intransitive are opposites of each other. For example, the relation $\{\langle 1, 1 \rangle, \langle 1, 2 \rangle\}$ is neither reflexive nor irreflexive.

A **sequence** is an ordered list, ordered multiset, or an ordered array of items of the same type. The following variables are all sequences:

```
let A := array {"a", "tree", "that", "is", "inflexible", "will", "snap"}
let B := list {1, 4, 1, 5, 9, 2, 6, 5, 3, 5}
let C := array {}
```

Here, *A* is an array of strings, *B* is a list of integers, and *C* is an empty array. Note that *B* contains the values "1" twice and "5" three times.

A **subsequence** of a sequence is a new sequence formed by deleting some elements from the old sequence and leaving the remaining elements in the same relative order. For example,

```
let A_T_words:= array {"tree", "that"}
```

```
let B_evens := list {4, 2, 6}
let B_odds := list {1, 1, 5, 9, 5, 3, 5}
let B_primes := list {5, 2, 5, 3, 5}
let B_empty := list {}
```

Note that the **empty sequence** (*C* and *B_empty*, above) is a subsequence of every sequence. Also note that a sequence can have many different subsequences.

A **median** of a set of values is the value that separates the highest half of the values from the lowest half. To find the median, arrange all the observations from lowest value to highest value and pick the middle one. If there are an even number of values, the median can be defined as the mean of the two middle values.

The **closed form** of a function is a description of the function in mathematical terms that use a bounded number of well-known operations. For example, expressions using only addition, subtraction, multiplication, division, exponentiation, negation, absolute-value, and the factorial function would be considered closed form. Specifically not allowed is when the number of operations depends upon the values of the function's variable. For example, here is a non-closed form version of the function sum(*n*), the sum of the first *n* positive integers:

$$\mathrm{sum}(n) = \sum_{i=1}^{n} i = 1 + 2 + \cdots + (n-1) + n.$$

And, here is the closed-form version of that same function:

$$\mathrm{sum}(n) = \frac{n(n+1)}{2}.$$

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 the its size. For example, suppose an algorithm takes as input 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:

$f(n) = n^3 - 12n^2 + 20n + 110$
$g(n) = n^3 + n^2 + 5n + 5$

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; both 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.

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) = O(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

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

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)).

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.

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.

Substitution method

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

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.

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

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

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 innerloop take that long, and a tighter bound is O(n+m).]

3 Divide and Conquer

live version • discussion • edit lesson • comment • report an error • ask a question

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:

Divide and Conquer Methodology

1. Given a problem, identify a small number of significantly smaller subproblems of the same type
2. Solve each subproblem recursively (the smallest possible size of a subproblem is a base-case)
3. Combine these solutions into a solution for the main problem

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

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 $1 \leq i < j \leq n$ then $a[i] \leq a[j]$

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 lead 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 := 0

  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
```

This merge sort algorithm can be turned into an iterative algorithm by starting with all pairs, then all fours, et cetera... 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
       let a1[1..unit] := result[i..i+unit]
       let a2[1..unit] := result[i+unit..i+unit*2]
       result[i..i+unit*2] := merge(a1,a2)
     repeat
   repeat

   return result
end
```

This works because the array starts out as a set of chunks length 1 which are "sorted." Each iteration through the array (using counting variable *i*) doubles the size of sorted chunks by merging adjacent chunks into sorter larger versions. The current size of sorted chunks in the algorithm is represented by the *unit* variable.

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 $T(n)$:

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

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.

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. 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.

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 2*n*. 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.

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 $(P_x \cdot P_y)(1), (P_x \cdot P_y)(0), (P_x \cdot P_y)(-1)$.

Base Conversion

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=23 (and, of course, 16=24.) 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 = akNk+ak-1Nk-1+...+a1N1+a0

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

M = (akak-1...a0)N If we rewrite (1) as

(2) M = a0+N*(a1+N*(a2+N*...))

the algorithm for obtaining coefficients ai becomes more obvious. For example, a0=M modulo n and a1=(M/N) modulo n, and so on.

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.

Iterative Implementation

Not all programming languages (Basic is one example) 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.

Closest Pair: A Divide-and-Conquer Approach

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 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.

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 algoritm wouldn't save us anything in terms of efficiency. Thankfully, we can make another life saving observation at this point. For any particualr 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 (I won't let myself get away with that scot-free, click here to see the proof). Well this is simply fantastic news, because 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.

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 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. Sort the remaining points according to their y-coordinates
5. Scan the remaining points in the y order and compute the distances of each point to its five neighbors.
6. If any of these distances is less than d then update d.
7. Steps 2-6 define the merging process which must be repeated logn times because this is a divide and conquer algortithm:
8. Step 2 takes O(1) time
9. Step 3 takes O(n) time
10. Step 4 is a sort that takes O(nlogn) time
11. Step 5 takes O(n) time (as we saw in the previous section)
12. Step 6 takes O(1) time

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.

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.

4 Randomization

live version • discussion • edit lesson • comment • report an error • ask a question

Increasingly, **randomization** is being used to obtain algorithms that are fast in the average case and only perform poorly under highly improbable situations. Randomization is also used for approximation algorithms.

Ordered Statistics

Before covering randomized techniques, we'll start with a deterministic problem that leads to a problem that utilitizes 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?"

find-max

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

```
// find-max -- returns the maximum element, or -infinity if the array is empty
function find-max(array vals): element
  let result := −∞
  for-each v in vals:
    result := max(result, v)
  repeat

  return result
end
```

Here, we use the special value of negative infinity for when there are no elements in the array. When negative infinity is compared it is always less-than any value, thus

```
v == max(v, −∞)
```

for all *v*. You could write a similar routine to find the minimum element by calling the min function instead of the max function, and by beginning with the value of positive infinity (this is left as an exercise). In a programming language without infinity, you could use an additional flag that is set to true when the value should be interpreted as infinity instead and then check the flag before checking the value itself.

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

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 := −∞

  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* = 1.5*n*, 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.

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 nth-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 opitimzation 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 - 1
  let irand := random(m, n)   // returns any value from m to n
  let x := vals[irand]

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
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:

Randomization Guideline:
If your algorithm depends upon randomness, be sure you introduce the randomness yourself instead of depending upon the data to be random.

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.

Exercises

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

5 Backtracking

live version • discussion • edit lesson • comment • report an error • ask a question

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.

Backtracking Methodology

1. View picking a solution as a sequence of **choices**
2. For each choice, consider every **option** recursively
3. Return the best solution found

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.

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
```

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.)

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.

6 Dynamic Programming

live version • discussion • edit lesson • comment • report an error • ask a question

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.)

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 as the sixth Fibonacci number.

Note that every leaf of the call tree has the value 0 or 1. All of these ones are summed together to give the final answer, 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:

Memoization Methodology

1. Start with a backtracking algorithm
2. Look up the problem in a table; if there's a valid entry for it, return that value
3. Otherwise, compute the problem recursively, and then store the result in the table before returning the value

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 *n*th 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 $O(n)$ 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.

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:

$((M_1M_2)M_3)M_4$
$(M_1(M_2M_3))M_4$
$M_1((M_2M_3)M_4)$
$(M_1M_2)(M_3M_4)$
$M_1(M_2(M_3M_4))$

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). The time this function takes to multiply two matrices of size $a \times b$ and $b \times a$ 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 parantezise them in the two possible ways and see which way requires the least amount of multiplications. The two ways are

$((M_1M_2)M_3)$, and
$(M_1(M_2M_3))$.

To form the product in the first way requires 75000 scalar multiplications (5*100*100=50000 to form product (M_1M_2) and another 5*100*50=25000 for the last multiplications.) This might seem like alot, 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 imporant: imagine what would happen if we needed to multiply 50 matrices!

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

$$M_1 M_2 \dots M_n$$

splits the product at k:

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

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

$$(M_1 \dots M_k)$$
$$(M_{k+1} \dots M_n)$$

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 $M_m \dots M_n$. 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 $(M_m \dots M_k)(M_{k+1} \dots M_n)$) 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 := ∞

    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, unfortunatly, not a very good one. It spends mountains of time recomputing data and its running time is exponential.

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.

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 algorithm 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).

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 recurence 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:

Event B

Event A Event C

---------- time ---------->

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:

11 12

9 6 7 8

5 6 7 8

1 2 3 4

A B C D E

---------- time ---------->

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.

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.

8 Hill Climbing

live version • discussion • edit lesson • comment • report an error • ask a question

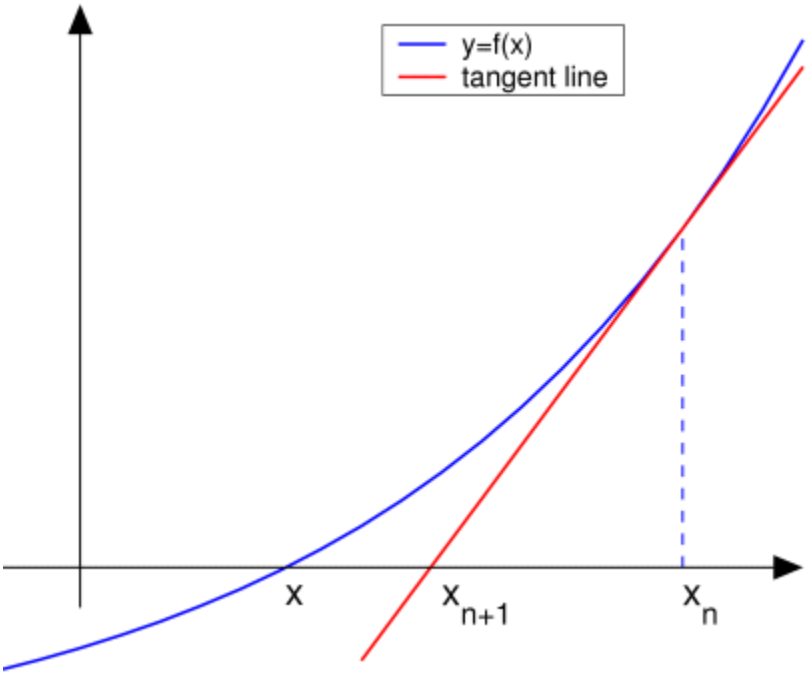
Hill climbing is a technique for certain classes of optimization problems. The idea is to start with a sub-optimal solution to a problem (i.e., *start at the base of a hill*) and then repeatedly improve the solution (*walk up the hill*) until some condition is maximized (*the top of the hill is reached*).

Hill-Climbing Methodology

1. Construct a sub-optimal solution that meets the constraints of the problem
2. Take the solution and make an improvement upon it
3. Repeatedly improve the solution until no more improvements are necessary/possible

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.

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 xn+1 is a better guess than xn 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

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

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.

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

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.

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.

Even though the final edge 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 can say that the path 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:

1. $\forall e \in E : f(e) \in \mathbb{R}$
2. $\forall (u,v) \in E : f((u,v)) = -f((v,u))$
3. $\forall u \in V, u \neq s,t : \sum_{v \in V} f(u,v) = 0$
4. $\forall e \in E : f(e) \leq c(e)$

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. $\forall e \in E : f(e) \leq c(e)$. This rule says that the flow must never be greater than the capacity allows.

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
```

9 Ada Implementation

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 then perhaps needed. However, we also hope it makes the code easier to understand

Introduction

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

To Lower

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

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

```
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 appoach be impossible with Ada? No, but it would be significantly more complex and slower.

Equal Ignore Case

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

```
--  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;
```

Dynamic Programming

Fibonacci numbers

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

Simple Implementation

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

```
...
```

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 exapmle. 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)

```
...
```

For this implementation we need a special cash 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 automaticly convert between the two. (the convertion 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 normaly do it a little different:

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

when you use a slightly larger array which also stores the elements 0 and 1 and initializes then 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)

```
  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.

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

```
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.

10 History & Document Notes

Wikibook History

This book was created on 2004-01-31 and was developed on the Wikibooks project by the contributors listed in the next section. The latest version may be found at http://en.wikibooks.org/wiki/Algorithms.

PDF Information & History

This PDF was created on 2006-08-12 based on the 2006-08-11 version of the Algorithms Wikibook. A transparent copy of this document is available at Wikibooks:Algorithms/Printable version. An OpenDocument Text version of this PDF document and the template from which it was created is available upon request at Wikibooks:User talk:Hagindaz. A printer-friendly version of this document is available at Wikibooks:Image:Algorithms printable version.pdf.

Document Information

- Pages: 68
- Paragraphs: 1288
- Words: 22342
- Characters: 130088

11 Authors, Sources, & References

Principal Authors

- Macneil Shonle (Contributions) A large portion of my contributions here come from lectures made by [Impagliazzo] at UCSD. I like this project because it gives me a chance to explain algorithms in the way that I finally understood them.
- Matthew Wilson (Contributions) I typed in an outline after finishing a graduate algorithms course. M. Shonle has taken this textbook and really made it something great.
- Martin Krischik (Contributions) I supplied the Ada examples for the algorithms. You never know if an algorithm works until you have actually implemented it.
- Jyasskin (Contributions)
- Gkhan (Contributions)
- WhirlWind (Contributions)

All Authors number of contributions

Title page - Mshonle 121 · Anonymous 12 · Waxmop 8 · Krischik 5 · Robert Horning 3 · Gkhan 3 · Nikai 2 · Jguk 2 · Geocachernemesis 1 · WhirlWind 1 · Derek Ross 1 · Dysprosia 1 · ReneGabriels 1 · Alexs 1 · Mkn 1 · Andreas Ipp 1

Introduction - Mshonle 23 · Anonymous 13 · WhirlWind 4 · Krischik 3 · Sartak 3 · Jguk 2 · Mmartin 2 · Robert Horning 1 · Hagindaz 1 · DavidCary 1 · Nikai 1 · Jleedev 1 · Derek Ross 1

Mathematical Background - Mshonle 21 · Anonymous 8 · Jyasskin 8 · Gkhan 4 · WhirlWind 3 · Jguk 2 · Hagindaz 1 · R3m0t 1 · Nikai 1 · Derek Ross 1 · Robert Horning 1

Divide and Conquer - Mshonle 27 · Anonymous 11 · Krischik 2 · JustinWick 2 · Jguk 2 · WhirlWind 1 · Robert Horning 1 · Hagindaz 1

Randomization - Mshonle 16 · Anonymous 4 · Jguk 2 · R3m0t 1 · SudarshanP 1 · WhirlWind 1 · Gflores 1 · Robert Horning 1 · Hagindaz 1

Backtracking - Anonymous 13 · Mshonle 10 · Lynx7725 2 · Jguk 2 · Robert Horning 1 · Hagindaz 1

Dynamic Programming - Mshonle 13 · Anonymous 13 · Gkhan 7 · Krischik 3 · Jguk 2 · Kusti 1 · Chuckhoffmann 1 · Geocachernemesis 1 · Fry-kun 1 · Robert Horning 1 · Hagindaz 1

Greedy Algorithms - Mshonle 13 · Anonymous 7 · Jguk 2 · Robert Horning 1 · Hagindaz 1

Hill Climbing - Mshonle 16 · Andreas Ipp 3 · Jguk 2 · Anonymous 1 · Nikai 1 · Robert Horning 1 · Hagindaz 1

Ada Implementation - Mshonle 5 · Krischik 3 · Anonymous 2 · Liblamb 1 · Robert Horning 1 · Jguk 1 · Hagindaz 1

Sources

The following sources are used with permission from the original authors. Some of the sources have been edited (sometimes heavily) from the initial versions, and thus all mistakes are our own.

[Impagliazzo] Russell Impagliazzo. Lecture notes from algorithms courses 101 (Spring 2004; undergraduate), and 202 (Spring 2004, Fall 2004; graduate). University of California, San Diego. Used almost everywhere.

[Lippert] Eric Lippert. "Recursion and Dynamic Programming," from Fabulous Adventures In Coding. 21 July 2004. Used with permission. http://weblogs.asp.net/ericlippert/archive/2004/07/21/189974.aspx Used in the backtracking and dynamic programming chapters.

[Wikipedia] Wikipedia, the free encyclopedia. http://www.wikipedia.org. Quoted/appropriated pervasively.

References

The authors highly recommend the following reference materials.

[Aho] Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft. *Data Structures and Algorithms*. Addison Wesley, 1983.

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[Hoare] C. A. R. Hoare. "Algorithm 63: Partition", "Algorithm 64: Quicksort", "Algorithm 65: Find", from Communications of the ACM. Volume 4, Issue 7 (July 1961). Page 321. ISSN:0001-0782. http://doi.acm.org/10.1145/366622.366642.

[Knuth] Donald E. Knuth. *The Art of Computer Programming, Volumes 1-3*. Addison-Wesley Professional, 1998.

12 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding

them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the

Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents

or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the

Free Software Foundation.

External links

- GNU Free Documentation License (Wikipedia article on the license)
- Official GNU FDL webpage