Algorithms and Complexity

Emanuele Viola

May 5, 2019

Note: This text is in a preliminary state.
Please send all bug reports and comments to *(my five-letter last name)@ccs.neu.edu*

Fonts:
x x `x`

`x`
`Terminology: Power time,`
`Bad terminology: P vs. NP`

Chapter 0

Introduction

What is an algorithm? An algorithm for a problem is a simple, step-by-step, procedure that solves the problem on every input.

0.1 The model (or, what did you exactly mean by "simple" before?)

The details of the model are not too important if you don't care about polynomial differences in running times, such as the difference between $O(n)$ and $O(n^2)$. But they matter if you do. The goal of the model is to capture what you can do on an actual computer. The processor in the computer has some *registers,* typically of 64 bits. You can perform various operations like basic arithmetic in constant time. At the same time, we are doing asymptotic analysis: we have to think of larger and larger inputs to understand how the resources required to solve the problem scale. We can't keep the registers fixed at 64 bits. We also can't make them unrestricted. For example if you are working with an array of n entries, where n can be a trillion, you don't want to think of having registers with a trillion bits, because it isn't realistic.

A good way to define the model then is to introduce the parameter w which is the bit-length of the registers. You can perform all standard operations between w-bit registers in constant time, including direct addressing the memory. An algorithm in this model gives rise to executable code once we fix w to a particular value like $w = 64$.

Typically, we do not specify what w is. Your algorithm should work for any choice of w. If a problem involves numbers, for example if your input is a list of numbers, you should think that each number in the input may have w bits. Your running time is expressed in terms of the number of operations among w-bit registers.

Note that you cannot assume that the input numbers are small. Several times, we will show that under the additional assumption that the input numbers are small we can develop a faster algorithm, faster than what we can do for arbitrary w-bit numbers. A glaring example of this is Radix Sort.

0.2 The great schism of algorithms

TBD

0.3 What is a proof?

Many of the algorithms that we see are non-trivial, and we must argue that they are correct by providing a *proof.* What is a proof? Similarly to an algorithm, a proof is *a short sequence of simple claims that explain why something is true in every possible case.* What do we mean by "simple"? Here lies a major difference between algorithms and proofs. Algorithms are written for computers. Profos are written for the human

brain. Computers are very good at following a long list of steps, humans are not. A proof should extract the few salient claims that suffice to convince ourselves. And that is why I added "short" before sequence. Just like algorithms should work on every possible input, a proof should work in every possible case. We will see several examples.

0.4 Pyramids

Both algorithms and proofs are like pyramids. Let's start with algorithms. You can write the same alg. using a high-level language, the top of the pyramid, or a low-level one like processor instructions, the base of the pyramid. Your understanding is complete when you can move freely up and down the pyramid, expanding each high-level instruction into a lower-level instruction as you please. Typically you want to present things as some middle level of the pyramid.

Proofs are just like that. What is at the bottom? The basic axioms of mathematics.

Chapter 1

Some basic algorithms

In this chapter we see some basic algorithms.

1.1 Computing the maximum of an array

As our first example we consider computing the maximum of an array $A[1..n]$ of integers. This can be solved as follows.

```
Computing maximum of A[1..n] :
m = A[1];
for (i = 2, i ≤ n, i++)
  m = max{m, A[i]}
return m
```

For the proof of correctness we can notice that right before each execution of $i++$, the variable m contains the maximum of $A[1..i]$.

The running time is clearly $O(n)$.

It is worth pointing out that this works in the general model described in Section 0.1. We did not specify the magnitude of the integers in A. These are arbitrary integers with w bits. Our algorithm works for any choice of w. Its running time is $O(n)$, using that the `max` operation with w-bit integers takes constant time.

1.2 The sorting problem

We now move to a less trivial example. In the *sorting problem* we are given as input an array of numbers $A[1..n]$ and we want to compute the *sorted array* $B[1..n]$ with the property that $B[i] \leq B[i+1]$. Sorting is a basic operation which shows up in countless algorithms. Often when you look at data you need it sorted. For example you sort emails by date and so on. It is also useful pedagogically as it allows us to describe several algorithmic techniques with a simple problem. Finally, sorting has an unexpected application to *complexity*, described at the end of this book, arguably justifying the importance of sorting from a problem-independent viewpoint.

1.3 Bubble sort

Bubble sort is a simple sorting algorithm which is a good choice when the data to be sorted is small, say $n = 100$.

```
BubbleSort(a[1..n]) :
for (i = n; i > 1; i--)
  for (j = 1; j < i; j++)
    if (a[j] > a[j + 1]) swap a[j] and a[j + 1];
```

Why does this work? We can argue as follows. Fix i and let $a'[1], \ldots, a'[n]$ be the array at the start of inner loop. Note at the end of the loop: $a'[i] = \max_{k \le i} a'[k]$ and the positions $k > i$ are not touched. Since the outer loop is from n down to 1, the array is sorted.

How long does it take? The number of operations is $T(n) = (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$. This quadratic time is infeasible when the data gets large, and it begs the question of whether we can sort faster.

1.4 Counting sort

Now we come to a brilliant sorting algorithm which shows the power of direct addressing. We are going to show how to sort in *linear time,* under the assumption that the numbers are small. Specifically we introduce a parameter k and we assume that the input array consists of integers from 0 to k. The philosophy behind counting sort is this: suppose x is an input element. Where should x go in the output? It should go to a location whose index equals the number of input elements that less than x. CountingSort computes these indices in an auxiliary array $C[0..k]$; it then uses them to place the output elements directly in the right output location.

```
Countingsort (A[1..n]) {
// Initializes C to 0
for (i=0; k ; i++)
  C[i] = 0;
// Set C[i] = number of elements = i
for (i=1; n ; i++)
  C[ A[ i ] ] =C[ A[ i ] ]+1;
// Set C[i] = number of elements ≤ i.
for (i=1; k ; i++)
  C[i] = C[i]+C[i-1] ;
for (i=n; 1 ; i - -) {
  B[ C[ A[ i ] ] ] = A[ i ] ; //Place A[i] at right location
  C[ A[ i ] ] = C[ A[ i ] ]-1; //Decrease for equal elements
}
```

Counting sort evidently runs in time $O(n+k)$ and space $O(n+k)$.

The proof of correctness is essentially provided by the comments. At the end of the second for loop the array C contains at position i the number of elements in the input array equal to i. In the next for loop we keep a running sum to modify C so that now at position i we have the number of elements at most i. In the final for loop we place element $A[i]$ into position $C[A[i]]$ in the output array. We start from the last input element and then go down, so we decrease $C[A[i]]$ at each iteration. This also preserves the relative order of equal elements. This feature may sound a little odd. To make sense of it you should think that the elements carry *additional information* in addition to their value. For example, you are sorting emails based on date. Preserving the relative order is critical in a number of applications; in particular it is exploited by the next algorithm.

1.5 Radixsort

Radixsort is an extension of Countingsort that works for larger numbers. We can sort in linear time integers in the range $0..n^c$ for any fixed c. Its description is simple. We think of the keys as numbers written *in base* k. Then we sort them one digit at the time, starting with the *least significant*.

```
Radixsort(A[1..n])
  for i that goes from least significant digit to most
    use counting sort algorithm to sort array A on digit i
```

Radix sort is a great algorithm. However, to apply it we have think of keys as digits. This makes the algorithm not too versatile. In general it is preferable to be able to sort anything that can be compared. The use can simply specify the comparison function, and the algorithm works. BubbleSort works in this setting. This begs the question of whether there are comparison-based sorters which run faster than BubbleSort. The answer is "yes" and we use it to introduce a powerful paradigm.

Chapter 2

Divide and conquer

2.1 Odd-even mergesort

In this section we present an efficient sorting algorithm which enjoys the following property: *the only way in which the input is accessed is via* Compare-Exchange *operations.* Compare-Exchange takes two indexes i and j and swaps $A[i]$ and $A[j]$ if they are in the wrong order. It has the following code:

```
Compare-Exchange(Array A[0..(n − 1)] and indexes i and j with i < j):
if A[i] > A[j]
 swap A[i] and A[j]
```

Why care about this property? It makes the comparisons *independent from the data,* and this allows us to implement the algorithm with a network – a *sorting network* – of fixed Compare-Exchange operations. This gives an efficient hardware implementation, which surprisingly we will also use later to argue that certain problems are hard.

We call an algorithm with this property *oblivious.* Of the sorting algorithms seen earlier, exactly one is oblivious. Mergesort is not, because the merge operations performs comparisons which depend on the outcome of previous ones. Similarly, Countingsort accesses the input using direct accessing, which is not allowed. But Bubblesort is oblivious! Recall however that it takes quadratic time – not something that we would call efficient regarding sorting. Now we will see an oblivious algorithm which achieves $O(n \log^2 n)$ time. The algorithm has the same structure as Mergesort, except that the problematic merge subroutine is replaced with an oblivious subroutine, which we now present.

Algorithm Odd-Even-Merge(A) merges the two already sorted halves $[a_0, a_1, \ldots, a_{n/2-1}]$ and $[a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}]$ of the sequence $A = [a_0, a_1, \ldots, a_{n-1}]$, resulting in a sorted output sequence. It works in a remarkable and mysterious way. First it merges the *odd* subsequence of the entire array A, then the *even*, and finally it makes $O(n)$ Compare-Exchange-Operations. Throughout, we assume that n is a power of 2.

```
Odd-Even-Merge(A = [a_0, ..., a_(n−1)]):
if n = 2
 Compare-Exchange(A, 0, 1)
else {
 Odd-Even-Merge([a_0, a_2, ..., a_(n−2)], n/2) //the even subsequence
 Odd-Even-Merge([a_1, a_3, ..., a_(n−1)], n/2) //the odd subsequence
 for i ∈ {1, 3, 5, 7, ..., n − 3}
  Compare-Exchange(A, i, i + 1)
}
```

We shall now argue that this algorithm is correct.

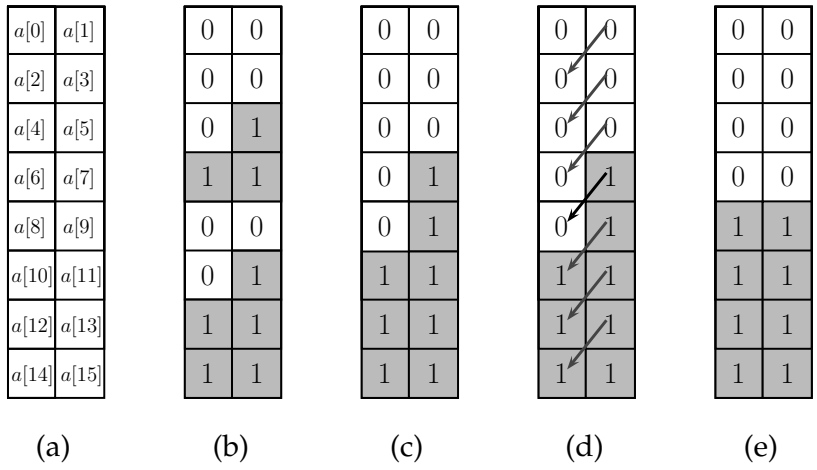


Figure 2.1: Odd-Even-Merge illustration

Lemma 1. *If* $[a_0, a_1, \ldots, a_{n/2-1}]$ *and* $[a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}]$ *are sorted, then Odd-Even-Mergesort*$([a_0, a_1, \ldots, a_{n-1}])$ *outputs a sorted array.*

Proof. To prove this lemma we invoke the so-called "0-1 principle." This principle says that it suffices to prove the lemma when each a_i is either 0 or 1, assuming that the algorithm only accesses the input via Compare-Exchange operations. For completeness we sketch a proof of this principle in this paragraph. Let $A = [a_0, \ldots, a_{n-1}]$ be an input to Odd-Even-Merge, and let $B = [b_0, \ldots, b_{n-1}]$ be the output sequence produced by the algorithm. If the algorithm fails to correctly sort A, then consider the smallest index k such that $b_k > b_{k+1}$. Define a function f such that $f(c) = 1$ if $c \geq b_k$ and $f(c) = 0$ otherwise. For an array $X = [X_0, X_1, \ldots, X_{n-1}]$ let $f(X)$ be the sequence $[f(X_0), f(X_1), \ldots, f(X_{n-1})]$ obtained by applying f to each element of X. Observe that $f(B)$ is not sorted. However it is easy to verify that f commutes with any Compare-Exchange operation applied to any sequence X, i.e.,

$$f(\text{Compare-Exchange}(X, i, j)) = \text{Compare-Exchange}(f(X), i, j).$$

Because Odd-Even-Merge is just a sequence of Compare-Exchange, we have that

$$f(B) = f(\text{Odd-Even-Merge}(A)) = \text{Odd-Even-Merge}(f(A))$$

and so the algorithm fails to correctly merge the 0-1 sequence $f(A)$. It only remains to notice that $f(A)$ is a valid input for Odd-Even-Merge. This is indeed the case because if a sequence X is sorted then $f(X)$ is also sorted.

We now prove the lemma by induction on n, based on the recursive definition of Odd-Even-Merge. Refer to Figure 2.1.

The base case $n = 2$ is clear. Assume that Odd-Even-Merge correctly merges any two sorted 0-1 sequences of size $n/2$. We view an input sequence of n elements as an $n/2 \times 2$ matrix, with the left column corresponding to elements at the even-indexed positions $0, 2, \ldots, n-2$ and the right column corresponding to elements at the odd-indexed positions $1, 3, \ldots, n-1$ (Figure 2.1(a)).

Since the upper half of the matrix is sorted by assumption, the right column in the upper half has the same number or exactly one more 1 than the left column in the upper half. The same is true for the lower half (Figure 2.1(b)). Because each (length-$(n/4)$) column in each half of the matrix is also individually sorted by assumption, the induction hypothesis guarantees that after the two calls to Odd-Even-Merge both the left and right (length-$(n/2)$) columns are sorted (Figure 2.1(c)).

At this point only one of 3 cases arises:

1) The odd and even subsequences have the same number of 1s.
2) The odd subsequence has a single 1 more than the even subsequence.
3) The odd subsequence has two 1s more than the even subsequence.

In the first two cases, the sequence is already sorted. In the third case, the Compare-Exchange operations (Figure 2.1(d)) yield a sorted sequence (Figure 2.1(e)). □

Given Odd-Even-Merge, we can sort by the following algorithm which has the same structure as Mergesort

```
Oblivious-Mergesort(A = [a_0, ..., a_(n-1)]):
if n ≥ 2 {
 Oblivious-Mergesort([a_0, a_1, ..., a_n/2-1])
 Oblivious-Mergesort([a_n/2, a_n/2+1, ..., a_n-1])
 Odd-Even-Merge([a_0, a_1, ..., a_n-1])
}
```

It only remains to argue efficiency. Let $S_M(n)$ denote the number of Compare-Exchange operations for Odd-Even-Merge for an input sequence of length n. We have the recurrence

$$S_M(n) = 2 \cdot S_M(n/2) + n/2 - 1,$$

which yields $S_M(n) = O(n \cdot \log n)$.

Finally, let $S(n)$ denote the number of calls to Compare-Exchange for Oblivious-Mergesort with an input sequence of length n. Then we have the recurrence $S(n) = 2 \cdot S(n/2) + (n \cdot \log n)$, which yields $S(n) = O(n \cdot \log^2 n)$.

2.2 Fast Walsh-Hadamard transform

Let $x_1, x_2, \ldots, x_v$ be a collection of v variables, and consider a *multilinear polynomial* p over those variables. This is simply an expression like this, in the case $v = 2$:

$$p(x_1, x_2) = 4 + 17x_1 - 3x_2 + 9x_1x_2.$$

It is a sum of numbers multiplied by any product of variables. In general, p has the form

$$p(x_1, x_2, \ldots, x_v) = \sum_{m \in \{0,1\}^v} c_m x_m$$

where c_m is a number and $x_m := \prod_{i:m_i=1} x_i$ is the product of the variables corresponding to m. The c_m are called the *coefficients* of the polynomial.

We are interested in evaluating such a polynomial over every ± 1 assignment for the variables. There are 2^v such assignments. For example, in the case above we have

- $p(1,1) = 4 + 17 - 3 + 9 = 27$
- $p(-1,1) = 4 - 17 - 3 - 9 = -25$
- $p(1,-1) = 4 + 17 + 3 - 9 = 15$
- $p(-1,-1) = 4 - 17 + 3 + 9 = -1$

Here the values for the variables are listed in order, written in binary with 1 corresponding to 0 and -1 corresponding to 1. So the first value is $x_1 = x_2 = 1$ which corresponds to the binary 00 for the number 0. The second value is $x_1 = 1$ and $x_2 = 0$, which corresponds to the binary 01 for the number 1, the third value is $x_1 = -1$ and $x_2 = 1$ which corresponds to the binary 10 for the number 2, and so on.

How quickly can we compute this in terms of the number of variables? Let $n = 2^v$. The naive algorithm simply evaluates the polynomial from scratch on each of the n assignments. Each evaluation takes time $O(n)$, resulting in an $O(n^2)$ time algorithm. We shall now present a divide-and-conquer algorithm for this task which runs in time $O(n \log n)$.

Define the *Hadamard* matrix H_v as follows:

$$H_0 = [1]$$
$$H_v = \begin{bmatrix} H_{v-1} & H_{v-1} \\ H_{v-1} & -H_{v-1} \end{bmatrix}.$$

In particular,

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \; H_2 = \begin{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & \begin{bmatrix} -1 & -1 \\ -1 & +1 \end{bmatrix} \end{bmatrix}$$

Note that H_v is a $2^v \times 2^v$ matrix.

Now let c be an n-tuple of coefficients in a polynomial. In the above example,

$$c = (4, 17, -3, 9).$$

Now the crucial observation is that what we are trying to compute is precisely the matrix-vector product

$$H_v \cdot c.$$

For example,

$$\begin{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & \begin{bmatrix} -1 & -1 \\ -1 & +1 \end{bmatrix} \end{bmatrix} \cdot (4, 17, -3, 9) = (27, -25, 15, -1).$$

We can give a simple divide-and-conquer algorithm for computing $H_v \cdot c$. The main observation is that if we write $c = (c_0, c_1)$ where $|c_0| = |c_1| = |c|/2$, then

$$H_v \cdot c = \begin{bmatrix} H_{v-1}c_0 + H_{v-1}c_1 \\ H_{v-1}c_0 - H_{v-1}c_1 \end{bmatrix},$$

so we just need to compute recursively $H_{v-1}c_0$ and $H_{v-1}c_1$ and then we can obtain $H_v \cdot c$ via simple additions.

```
Fast-Walsh-Hadamard-Transform(c):
If c has length 1 output c and return.
Write c = (c_0, c_1) where |c_0| = |c_1| = |c|/2.
Let y_0 ←Fast-Walsh-Hadamard-Transform(H_{v-1}c_0)
Let y_1 ←Fast-Walsh-Hadamard-Transform(H_{v-1}c_1)
Output the vector (y_0 + y_1, y_0 - y_1).
```

The running time of this algorithm satisfies the same recurrence as Mergesort

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

which as we saw gives $T(n) = O(n \log n)$.

Chapter 3

Dynamic programming

Life can only be understood backwards; but it must be lived forwards. Soren Kierkegaard

3.1 Dynamic programming in economics

Dynamic programming powers many *economic models*. In this section I give an example from *personal finance*: An individual called Andrew needs to decide how much to save and how much to invest for the next L years of his life. He makes this decision only once at the beginning of each year, and he does not care what happens after L years – we can think that $L = 200$ and Andrew does not care what happens in the event that he lives more than 200 years. Consuming c dollars during a year gives Andrew a certain *utility*, a function of c. A popular choice for this function is $\log c$, which captures the intuitive fact that consuming \$20 is a lot better than consuming \$10, but consuming \$1000020 is not much better than consuming \$1000010, and also the fact that consuming nothing is not a good idea ($\log 0 = -\infty$). At the beginning of a year i, Andrew can choose to consume any amount c up to the total k that he has at the beginning of the year. The remaining $k - c$ he would invest at an annual rate of $1 + \rho$. The more he saves, the more money he would have to spend the next year. Also, every year Andrew earns W dollars. He also starts with a budget of W. The question is: how much money should Andrew consume each year to maximize the sum of the utilities through his lifetime?

Let us illustrate the problem via the simplest example. Suppose that $L = 2$, so Andrew is only making one decision: how much to consume this year. The second year is his last year, so he will consume everything. Suppose that $W = \$128$, and that $\rho = 1$, so that the interest rate is 2. If Andrew consumes everything the first year, he will get a total utility of

$$2 \cdot \log w = 14.$$

If he saves \$127 the first year, he will get a utility of $\log 1 + \log(128 + 2 \cdot 127) = 8.57....$ But if he on the other hand saves \$32 he gets the utility

$$\log(128 - 32) + \log(128 + 2 \cdot 32) = 14.169$$

which is higher than both.

Calculating the amounts that Andrew should save is complicated, and there is no closed form. So one uses an algorithm to compute them. Let us now describe this algorithm for a generic L. First we identify the subproblems for a dynamic programming solution. Define

$$C[k, i]$$

to be the amount Andrew should consume at the beginning of year i to maximixe his total utility for the years $i, i + 1, i + 2, \ldots, L$, assuming he has a budget of k. What is the range of k? We assume that all amounts are rounded to the nearest integer. Since Andrew starts with a budget of W, earns W every year,

and the annual interest rate is $(1+\rho)$, Andrew will never have more cash than $M := LW(1+\rho)^L$. You should think of $(1+\rho)^L$ as a fairly small number: if you earn a typical income and put it all into savings, you won't become a billionaire. Hence, k is an integer in the range 0 to M, and the number of subproblems is ML.

Also define

$$U[k, i]$$

to be the corresponding total utility for the same years.

Now we can finally apply Kierkegaard quotation at the beginning of the chapter: we shall start from the last year. Specifically, what is $C[k, L]$? A moment's thought reveals that

$$C[k, L] = k,$$

because at the last year there is no point in saving anything.

Correspondingly,

$$U[k, L] = \log k.$$

Now for the recursion. Following Kierkegaard, we are going to "understand" life backwards. Given that we know how much to save at year $i+1$, how much should we save at year i? We should pick the value c that maximizes the utility at this year, $\log(c)$, plus the utility for the next years. After this year, Andrew will have $(k-c)(1+\rho)$ cash, given by how much he was able to invest this year times the interest, plus a wage of W. This yields the recursion:

$$U[k, i] := \max_{c:0\le c\le M} \log(c) + U[(k-c)(1+\rho) + W, i+1].$$

And the value for the consumption is the c that gives the maximum:

$$C[k, i] := \arg \max_{c:0\le c\le M} \log(c) + U[(k-c)(1+\rho) + W, i+1].$$

Each recursion step takes time $O(M)$, and the number of subproblems is $O(LM)$. Hence we can solve this problem in time $O(LM^2)$.

3.1.1 Advanced features of the model

Naturally, economic models are far more complex. For example, the quantities involved can be from larger domains, and to keep the number of subproblems small one has to discretize those quantities. This discretization involves another layer of techniques which works hand in hand with dynamic programming. We won't discuss it now (cf. Section 0.2). We just mention one way in which the model can be made more realistic and complex. In reality, the interest rate is not fixed, so we can model it better as the following random process. Independently every year, it is $(1+\rho)$ with some probability q, and $(1-\rho)$ with probability $1-q$. Now Andrew wants to maximize the *expected* sum of utilities. With this modification, the recursion becomes

$$U[k, i] := \max_c \log(c) + qU[(k-c)(1+\rho) + W, i+1] + (1-q)U[(k-c)(1-\rho) + W, i+1],$$

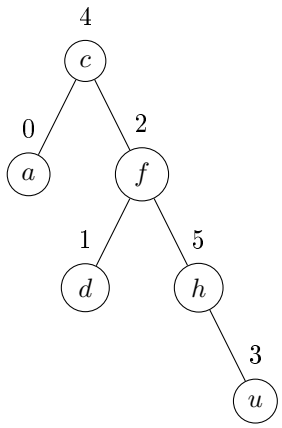
and similarly for $C[k, i]$.

Chapter 4

Data structures

4.1 How to represent a binary tree

A binary tree can be represented as 4 arrays and 2 variables. The arrays specify the parent, left child, right child, and key value for each node. The two variables specify the root and the number of nodes in the tree. A picture is worth a thousand more words – see Figure 4.1.



Index	0	1	2	3	4	5
Key	a	d	f	u	c	h
Parent	4	2	4	5	NULL	2
LeftChild	NULL	NULL	1	NULL	0	NULL
RightChild	NULL	NULL	5	NULL	2	3

Root = 4

NumNodes = 6

Figure 4.1: A binary tree and its representation

Note that the indexes of the nodes need not correspond to the location of the node in the tree. For example in the above example the root has index 4 and his children have index 0 and 2. Although when constructing a tree from scratch it is natural to have the index correspond to the location of the node in the array – this correspondence is lost once we perform operations on the tree, as we shall see shortly.

We also need a way to indicate that a node has no parent or children. The way we handle this is to set the corresponding entry in the array to NULL. You can think of NULL as being -1, or a number larger than the size of the tree. A different approach is to replace NULL with the index of the node itself, which is slightly more economical.

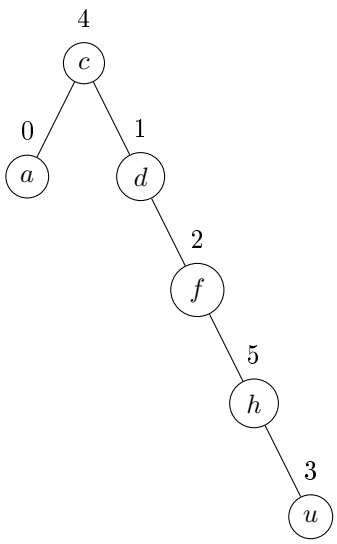
4.2 Inserting in a binary search tree

The following code inserts a node with key k in a binary search tree. We let MAXNODES be the maximum number of nodes we allow in our tree.

```
Insert(k):
//If there is no room, do nothing
if NumNodes >= MAXNODES
  return
//Otherwise puts a new node at the end of the arrays
Key[NumNodes] ← k
LeftChild[NumNodes] ← NULL
RightChild[NumNodes] ← NULL
//It remains to determine the parent
//If tree is empty then there is none
if NumNodes = 0
  Root ← 0
  Parent[NumNodes] ← NULL
  NumNodes++
  return
//Otherwise looks for the parent in the tree
x ← Root
forever
  //two ifs to check if x is parent, otherwise search
  if k ≤ Key[x] and LeftChild[x] = NULL
    LeftChild[x] ← NumNodes
    Parent[NumNodes] ← x
    NumNodes++
    return
  if k > Key[x] and RightChild[x] = NULL
    RightChild[x] ← NumNodes
    Parent[NumNodes] ← x
    NumNodes++
    return
  if k ≤ Key[x] and LeftChild[x] ≠ NULL
    x ← LeftChild[x]
  if k > Key[x] and RightChild[x] ≠ NULL
    x ← RightChild[x]
```

4.3 Rotations

```
Rotate-Right(i):
```



Index	0	1	2	3	4	5
Key	a	d	f	u	c	h
Parent	4	4	1	5	NULL	2
LeftChild	NULL	NULL	NULL	NULL	0	NULL
RightChild	NULL	2	5	NULL	1	3

Root = 4

NumNodes = 5

Figure 4.2: The effect of RotateRight(2) on the tree in Figure 4.1.

```
if i does not have a left child
  return
L ← LeftChild[i]
if i is the root
  Root ← L, Parent[L] ← NULL
If i is a right child of P
  RightChild[P] ← L, Parent[L] ← P
If i is a left child of P
  LeftChild[P] ← L, Parent[L] ← P
LR ← RightChild[L]
RightChild[L] ← i
Parent[i] ← L
LeftChild[i] ← LR
If LR ≠ NULL
  Parent[LR] ← i
```

The code for Rotate-Left is symmetric.

Let use illustrate the effect of Rotate-Right(2) on the tree in Figure 4.1. We are in the case in which the node $i = 2$ has a left child $L = 1$ but L does not have a right child, and i is the right child of $P = 4$. The effect of the operations is illustrated in Figure 4.2.

4.4 AA trees

There exist many different data structures that support Insert, Search, and Delete in logarithmic time and are based on trees. We now present *AA Trees,* which may be one of the simplest. An AA tree is a binary search tree where each node has a *level,* a value which behaves similarly but not exactly to the depth. Intuitively, only allowed path with nodes of the same level is a single left-right edge.

Definition 2. An *AA Tree* is a binary search tree where each node has a level, satisfying:
(1) The level of every leaf node is one.
(2) The level of every left child is exactly one less than that of its parent.
(3) The level of every right child is equal to or one less than that of its parent.
(4) The level of every right grandchild is strictly less than that of is grandparent.
(5) Every node of level greater than one has two children.

These rules also guarantee that an AA tree on n nodes has depth $O(\log n)$.

Claim 3. An AA Tree with n nodes has depth $O(logn)$.

Proof. Suppose the tree has depth d. The level of the root is at least $d/2$. Since every node of level > 1 has two children, the tree contains a full binary tree of depth at least $d/2 - 1$. Such a tree has at least $2^{d/2-1}$ nodes. □

To restructure an AA tree after an addition we follow this rule of thumb: First make sure that only left-right edges are within nodes of the same level, then worry about length of paths within same level. The first is handled by the function *Skew*, the second by *Split.*

```
Skew(x):
  if x has left-child with same level
    RotateRight(x)

Split(x):
  if the level of the right child of the right child of x is the same as the level of x,
  Level[RightChild[x]]++
  RotateLeft(x)
```

With these two in place, we can give the pseudocode for insertion:

```
AA-Insert(k):
  Insert k as in a binary search tree
  Let x be the node we just inserted
  while x ≠ NULL
    Skew(x)
    Split(x)
    x ← Parent[x]
```

Note that while you are calling the Skew and/or Split from a node y, the node may move. In particular, it may have a parent while it did not have one, which you will consider at the next iteration of the while loop.

For deletion we need another function, DecreaseLevel.

```
DecreaseLevel(x):
     If x has a child which is two levels below the level of x (this includes
       the case where x does not have a child but is at level at least 2)
       Decrease the level of x by one.
       If the right child of x had the same level of x, decrease the level of
       the right child of x by one too.
```

We can now give the pseudocode for deletion. First, let us consider removing a key which is at a leaf node x.

```
DeleteLeaf(x):
      Let y be the parent of x.
      Delete x.
      While y is not NULL
        Decrease level(y)
        Skew(y); Skew(y.right); Skew(y.right.right);
        Split(y); Split(y.right);
```

How to remove a key which is not at a leaf node. To remove a key k which is not at a leaf node, we do as follows. First, find a node x with key k. Go to the right child of x. Now keep going to the left child until you can't anymore, ending at node y. If y is a leaf then swap $Key[y]$ with $Key[x]$, and remove the leaf y as before. The only remaining case is when y is at level 1 and has a right child z but not a left child. In this case we set $Key[x] = Key[y]$, $Key[y] = Key[z]$, and finally we remove the leaf z.

Chapter 5

Interlude: what is a reduction?

A reduction is a name that you already do every day. The following dialogue explains the concept:

A: Tell me how you cook pasta.

B: What? Get the pot, put water

A: Stop! I want extremely precise, step-by-step instructions.

B: First get the pot ...

A: Stop! The pot is inside the drawer.

B: Ok, fine, I get it. Open the drawer. Get the pan. Put the pan on the cooktop. Turn on the heat. Wait until it boils. Get the pasta package. Get the scissors. Open the pasta package with the scissors. Pour the past in the water. Turn the heat to dim. Wait 1 minute. Get the wood spoon. Stir. Wait 4 minutes. Stir. Wait 5 minutes. Get the colander. Put the colander in the sink. Turn off the heat. Wear gloves. Grab the pot. Pour the pot inside the colander. Put the pot on the cooktop. Grab the colander. Lift it. Shake it. Put the colander down. Get a bowl. Put the bowl down. Get the colander. Pour the colander inside the bowl.

A: Good. Now do the same, but the pot is on the table.

B: Open the drawer. Put the pot insider the drawer. Close the drawer. Do like before.

In this story, B has *reduced* the problem of cooking pasta with a pot outside of the drawer to the problem of cooking pasta with a pot inside the drawer. Given the input, B does some *simple manipulation* (putting the pot in the drawer) and then *reuses* the solution to the other problem.

Another example was known to the babylonians millennia ago. They knew how to sum and divide by two, and had tables for *squaring*. To compute multiplication of two arbitrary numbers they reduced it squaring via the following formula:

$$x \cdot y = \frac{(x+y)^2 - x^2 - y^2}{2}.$$

Reductions take many forms. Try to think of a reduction you have done recently.

Chapter 6

Complexity

In the previous chapters we have seen many problems. We have given algorithms for these problems with various running times, such as $O(n \log n)$, n^2, n^3. Amazingly, for all we know each of those problems can be solved in linear time $O(n)$:

It is possible that every problem in this book has a linear-time algorithm!

Nobody can rule this out!

This is certainly one of the greatest intellectual changes of our time: prove that some problem cannot be solved in linear time. Scientists have tried very hard to solve this challenge, but so far they have failed. There are some justifications for their failures. The best one are the algorithms that we have seen themselves. They are clever, unexpected. They challenge our intuition of computation. Solving the challenge means putting a limit to this cleverness, ruling out any possible approach, however unexpected. In fact, we should perhaps take the cleverness at face value and accept that maybe these problems do have linear-time algorithms after all, we just haven't been clever enough yet.

So when faced with some problem X that they can't solve efficiently enough, short of proving that no efficient algorithm exists what people do is prove a *hardness reduction*. You consider a problem `HARD` that lots of smart people have tried very hard to solve efficiently, but couldn't. The reduction amounts to showing that *if you could solve problem X efficiently, then* `HARD` *could also be solved efficiently.*

Because you don't expect to be able to give an efficient algorithm for `HARD`, you also don't expect to be able to give an efficient algorithm for X. So you can move on. If your boss asked you to solve X, the reduction should be sufficient proof that you should be given a different task. If it is not sufficient proof, you should probably consider switching jobs.

Reductions can have any sort of parameters. For example, it was shown that if there is an algorithm that solves the Longest Common Subsequence problem in time $n^{1.999}$ than there is an efficient algorithm for a certain hard problem. This sort of explains why the algorithm we saw in Section ??? runs in time $\Omega(n^2)$.

For starters, we consider *polynomial-time reductions*. We show that if X can be solved in polynomial-time then so can `HARD`.

So we should start with some problem that people have tried very hard to solve in polynomial-time, but couldn't.

The 3SAT problem. TBD

We now give several reductions.