Introduction To Algorithms

Text Notes and Summary

Original text by: T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein

Chapter 2 Getting Started

2.1 Insertion Sort

Sorting problem: Given a sequence of n numbers $(a_1, a_2, \ldots, a_n)$, produce a premutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

We present a first solution to the sorting problem, the *insertion sort* algorithm, as a subroutine which takes as parameter an array $A[1..n]$ containing a sequence of length n to be sorted. When the procedure INSERTION-SORT is finished, it rearranges the elements within A so that they are sorted.

INSERTION-SORT(A)

```
1  for j = 2 to A.length do
2      key = A[j]
       // Insert A[j] into the sorted sequence A[1..j − 1]
3      i = j − 1
4      while i > 0 and A[i] > key do
5          A[i + 1] = A[i]
6          i = i − 1
7      end while
8      A[i + 1] = key
9  end for
```

Loop Invariants and Correctness: In order to prove for correctness of some algorithms, we can use a *loop invariant*, which is a property of the state of the algorithm which holds during all iterations of the loop. We must show three things in order to prove that a loop invariant holds:

- **Initialization:** It is true prior to the first iteratin of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, we show that the loop invariant holds prior to every iteration of the loop. The third property helps us use the loop invariant to prove correctness (typically using the loop invariant along the condition which made the loop to terminate).

We can now give a loop invariant of the given INSERTION-SORT algorihtm: *At the start of each iteration of the* ***for*** *loop of lines 1 - 8 the subarray* $A[1..j-1]$ *consists of the elements originally in* $A[1..j-1]$*, but in sorted order.*

Let us see the loop invariant holds and see how it can prove correctness of the sorting algorithm.

- *Initialization:* Before the first iteration, at $j = 2$, the subarray is just $A[1]$ which trivially contains the elements of $A[1]$ in sorted order.
- *Maintenance:* Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$. Which leaves the subarray $A[1..j]$ consisting in elements originally in $A[1..j]$ but in sorted order. Incrementing j for the next iteration of the loop preserves the loop invariant. (A more formal proof would require to prove another loop invariant for this **while** loop.)
- *Termination:* The loop terminates when $j > A.length = n$. Because each iteration increments j by one, we must have $j = n + 1$ at that time. Since the loop invariant holds at the termination time, we see that the array $A[1..n]$ consists of the elements originally in $A[1..n]$ but in sorted order. Hence the entire array A is sorted at the end, hence the algorithm is correct.

2.2 Analyzing Algorithms

Analyzing an algorithm means to predict the resource that the alorithm requires. Such resources may be memory, communication bandwidth, computer hardware, or computational time. Generally, by analyzing several algorihtms for a problem, we can identify a most efficient one (or discard inferior candidate algorithms). We will mostly study the computational time taken by algorithms in this book.

Model: Before we can analyze an algorithm, we must have a *model* of the implementation technology that we will use, including a model for the resources of that technology and their costs. Unless explicited, assume a generic one processor, random-access machine (RAM). That is, instructions executed one after another with no concurrent operations, where each instruction taking constant time (being the set of instroctions in the RAM model the commonly found in real computers: load, store, copy, arithmetic, conditional branch, subroutine call and return.

2.2.1 Analysis of Insertion Sort

The time taken by the INSERTION-SORT procedure depends on the input size, and also depending on how are arranged the item inside the array for two inputs of the same size. In general, running time will be a function of input size.

The notion of *input size* depends on the problem: it may be the number of items in the input (as in the array size in a sorting problem), or the total number of bits to represent input (like the problem of integer multiplication). Sometimes, it is more appropiate to describe the size of input with two parameters (or more), such as in a graph problem with number of edges and number of vertices.

The *running time* of an algorithm on a particular input is the time it takes to execute each primitive operation times the number of executions of those operations. We may assume that a constant amount of time is required to execute each line of pseudocode. That is, each execution of the ith line takes c_i time, where c_i is a constant.

Line Number	Cost	# of Executions
1	c_1	n
2	c_2	$n-1$
3	c_3	$n-1$
4	c_4	$\sum_{j=2}^{n} t_j$
5	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6	c_6	$\sum_{j=2}^{n} (t_j - 1)$
8	c_8	$n-1$

We analyzed above the INSERTION-SORT procedure time cost of each statement and the number of times each statement is executed. For each $j = 2, 3, \ldots, n$, where $n =$

A.length, denote t_j be the number of times the **while** loop in line 5 is executed for that value of j. Then, the execution time of the algorithm is the sum of all the cost of each line times its number of executions:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

Even for inputs of the same size, an algorithm's running time may depend on which input of that size is given. In this case, the *best case* occurs when the array is already sorted, since for each j we find that $A[i] \leq key$ in line 5 when $i = j-1$ initially, thus having $t_j = 1$. Which yields a running time:

$$\begin{aligned} T_{\text{best}}(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8). \end{aligned}$$

We thus can express this running time as $T_{\text{best}}(n) = an+b$ for some constants a, b; thus a linear function of n.

On the other side, if the array is in reverse order, we get the *worst case* of $t_j = j$ for each value of j. And using that $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$, and that $\sum_{j=2}^{n} (j-1) = \frac{n(n+1)}{2} - 1$, we can rearrange the terms as before to get $T_{\text{worst}}(n) = an^2 + bn + c$, for some constants a, b, c: a quadratic function of n.

We could also compute an *average-case* running time for a random input, and compute the expected value of the running time.

2.2.2 Order of Growth

It is the rate of growth or order of growth of the running time that interests us. We therefore consider only the most significative term as the size of input increases. Thus, the worst case running time of Insertion-Sort has a rate of growth of n^2. We sa that it has a worst case running time of $\Theta(n^2)$. We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth.

2.3 Designing algorithms

For insertion sort, we used an *incremental* approach: having sorted $A[1..j-1]$ we then insert $A[j]$ in the proper place, yielding the sorted subarray $A[1..j]$. We examine a different approach now:

2.3.1 Divide and Conquer

Many algorithms are *recursive* in sturcture: to solve a given problem, they call themselves recursively one or more times to deal with a closely related subproblem. These typically follow a *divide-and-conquer* approach:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** he subproblems by soving them recursively. If the subproblem sizes are small enough, solve the problem in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

When an algorithm contains a recursive call to itelf, we can describe its running time by a *recurrence*, which describes the overall running time on a problem of size n in terms of the running time of smaller inputs.

In divide and conquer algorithms we can describe the running time as a recurrence. As said, if the problem size is small enough, say $n \leq c$ for some constant c. the straightforward solution takes constant time, write as $\Theta(1)$. Suppose that the division of the problem yields a subproblems, with each one being $1/b$ size of the original. And define $D(n)$ and $C(n)$ be the times to divide the subproblems and combine them, respectively, then we have the running time as:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

The Merge-Sort algorithm follows this divide-and-conquer approach:

- Divide the n-element array to be sorted into two subarrays of $n/2$ elements each.
- Sort the two subarrays recursivelly using merge sort.
- Merge the two sorted arrays to produce the sorted array.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since it is an already sorted 1-element array.

The Merge-Sort array relies on the auxiliary procedure Merge(A, p, q, r), where A is an array, p, q and r are indices into the array such that $p \leq q < r$. The procedure assumes the subarrays $A[p..q]$ and $A[q+1..r]$ are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$. This procedure can be easily implemented to have a runtime of $\Theta(n)$, where $n = r - p + 1$.

The merge sort algorithm is then:

```
Merge-Sort(A, p, r)
1  if p < r then
2      q = ⌊(p + r)/2⌋
3      Merge-Sort(A, p, q)
4      Merge-Sort(A, q + 1, r)
5      Merge(A, p, q, r)
6  end if
```

And to sort the required sequence A we call the algorithm procedure as Merge-Sort$(A, 1, A.length)$. Assuming that n is a power of 2, we can calculate the running time of the algorithm can be calculated as.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$$

In later chapters we will solve the recurrence using the Master Theorem.

Chapter 3 Growth Of Functions

3.1 Asymptotic notation

When studying the running-time of algorithms, we are interested in the asymptotic behaviour of a time-cost function taking values in the set of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$.

Definition. Let g be a function defined in $\mathbb{N}$. We denote $\Theta(g(n))$ the set of functions defined by:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in \mathbb{R}^+, \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n),\ \forall n \ge n_0\}.$$

If f is another function taking values in $\mathbb{N}$, we denote that f is in this set as "$f(n) = \Theta(g(n))$". We say that g is an *asymptotically tight bound* for f.

Note how the definition of $\Theta(g(n))$ requires that every member f of the set to be *asymptotically nonnegative* ($f(n) \ge 0$ for sufficiently large n). Consequently g itself must be asymptitically nonnegative, or $\Theta(g(n))$ is the empty set.

Definition. Let g be a function defined in $\mathbb{N}$. We denote $O(g(n))$ the set of functions defined by:

$$O(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \le f(n) \le cg(n),\ \forall n \ge n_0\}.$$

If f is another function taking values in $\mathbb{N}$, we denote that f is in this set as "$f(n) = O(g(n))$". We say that g is an *asymptotic upper bound* for f.

Definition. Let g be a function defined in $\mathbb{N}$. We denote $\Omega(g(n))$ the set of functions defined by:

$$\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \le cg(n) \le f(n),\ \forall n \ge n_0\}.$$

If f is another function taking values in $\mathbb{N}$, we denote that f is in this set as "$f(n) = \Omega(g(n))$". We say that g is an *asymptotic lower bound* for f.

Note from the above definitions that for any function g, we have:

$$\Theta(g(n)) \subseteq \Omega(g(n)), \text{ and } \Theta(g(n)) \subseteq O(g(n)).$$

Theorem 3.1 *For any two functions f and g, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Asymptotic notation in equations and inequalities. When using asymptotic notation in formulas, we interpret it as standing for some anonymous function that we do not to care. For example, $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, for some function f in the set $\Theta(n)$ (In this case $f(n) = 3n + 1$ which is indeed $\Theta(n)$).

Using asymptotic notation in this manner we can help eliminate inessential detail and clutter in an equation. For example in merge sort recurrence: $T(n) = 2T(n/2) + \Theta(n)$. If we are interested only in the asymptotic behavoir of $T(n)$, there is no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$.

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. In some cases, asymptotic notation appears on the left-hand side of an equation, like: $2n^2 + \Theta(n) = \Theta(n^2)$. We interpret it using the rule: *No matter how the anonymous functions are chosen on the left hand of the equals sign, there is a way to choose the anonymous function on the right of the equal sign to make the equation valid.* Thus on our example, $\forall f \in \Theta(n)$, $\exists g \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n), \forall n$.

Asymptotically tight bounds. The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. For example $2n^2 + 3 = O(n^2)$ is asymptotically tight, but $n = O(n^2)$ isn't. We use o-notation to denote an upper bound that is not asymptotically tight.

Definition. We denote $o(g(n))$ the set of functions defined by:

$$o(g(n)) = \{f(n) : \forall c > 0,\ \exists n_0 \in \mathbb{N} \text{ such that } 0 \le f(n) < cg(n),\ \forall n \ge n_0\}.$$

That is equivalent to say that if $f(n) = o(g(n))$ if, and only if, $\lim_{n\to\infty} f(n)/g(n) = 0$.

By analogy we have lower bound that are not asymptotically tight.

Definition. We denote $\omega(g(n))$ the set of functions defined by:

$$\omega(g(n)) = \{f(n) : \forall c > 0,\ \exists n_0 \in \mathbb{N} \text{ such that } 0 \le cg(f < f(n),\ \forall n \ge n_0\}.$$

That is equivalent to say that if $f(n) = \omega(g(n))$ if, and only if, $\lim_{n\to\infty} f(n)/g(n) = \infty$.

3.1.1 Comparing Functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following assume f, g be asymptotically positive functions.

Symmetry:

i) $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$.

Reflexivity:

i) $f(n) = \Theta(f(n))$.

ii) $f(n) = O(f(n))$.

iii) $f(n) = \Omega(f(n))$.

Transitivity:

i) $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$.

ii) $f(n) = O(g(n))$ and $g(n) = O(h(n)) \implies f(n) = O(h(n))$.

iii) $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n))$.

iv) $f(n) = o(g(n))$ and $g(n) = o(h(n)) \implies f(n) = o(h(n))$.

v) $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n)) \implies f(n) = \omega(h(n))$.

Transpose Symmetry:

i) $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$.

ii) $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$.

We have an analogy between asymptotic comparisons of two functions and the comparisons of two real numbers. For example $f(n) = O(g(n))$ is like $a \leq b$; also $\Omega(\cdot)$ like $\geq$, $\Theta(\cdot)$ like $=$, $o(\cdot)$ like $<$, $\omega(\cdot)$ like $>$.

Remark. If we define the relation $f(n) \sim g(n)$ if, and only if, $f(n) = \Theta(g(n))$, then this relation is an equivalence relation.

Chapter 4 Divide And Conquer

In divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solveing them recursively. If the subproblem sizes are small enough, just solve them i a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough to solve recursively, we call that the *recursive case*. Once the problems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the *base case*. Sometimes, in addition to ubproblems that are smaller instances of the same problem, we have to solve subproblems that are NOT quite the same as the original problem, and cconsider solvim that problem part of the combine step.

4.3 The substitution method for solving recurrences

The *substitution method* for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use the Principle of Mathematical Induction to find the constants and show that the solution works.

We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example consider the recurrence given by: $T(n) = 2T(\lfloor n/2 \rfloor) + n$. We guess that the solution is $T(n) = O(n \log n)$. So, we claim that $T(n) \leq cn \log n$, for some $c > 0$. Proving the claim would then imply that $T(n) = O(n \log n)$. Using induction, we start by assuming that the bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$. And substituting into the recurrence yields:

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2(c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \qquad \text{(H.IND.)} \\ &\leq cn \log(n/2) + n \\ &= cn \log n + (1 - c \log 2)n \leq cn \log n, \end{aligned}$$

where the last inequality holds iff $c \geq 1/\log 2$. To complete the proof we need to check that our solution holds for a base case. That is, we need to show in our case, that we can choose c great enough so that the bound holds for the base case too.

This requeriment may lead to problems sometimes. Assume for the sake of the argument that $T(1) = 1$, then the bound $T(n) \leq cn \log n$ yields to $T(1) \leq c \cdot 1 \cdot \log 1 = 0$!! We can overcome this by taking advantage of asymptotic definition, requiring us that the bound holds for $n \geq n_0$, for some $n_0 \in \mathbb{N}$.

Subtleties In almost all cases in which the recurrence has constants or lower-order terms, it will be necessary to have additional terms in the upper bound to cancel out the constants or lower-order terms. Without the right additional terms, the inductive case of the proof will get stuck in the middle, or generate an impossible constraint; this is a signal to go back to your upper bound and determine what else needs to be added to it that will allow the proof to proceed without causing the bound to change in asymptotic terms.

Consider for example, the following recurrence:

$$\begin{cases} T(1) = 1, \\ T(n) = 2T(n-1) + c_1. \end{cases}$$

Where $c_1 > 0$ is a constant. Iterating manually on the recurrence ($T(n) = 2 \cdot (2 \cdots (2 \cdot (2 \cdot 1 + c_1) + c_1) \cdots) + c_1$), we can guess that the solution would be $O(2^n)$.

We will guess an upper bound of $k2^n - b$, where b is some constant. We include the b in anticipation of having to deal with the constant c_1 that appears in the recurrence relation, and because it does no harm. In the process of proving this bound by induction, we will generate a set of constraints on k and b, and if b turns out to be unnecessary, we will be able to set it to whatever we want at the end.

Our property, then, is $T(n) \leq k2^n - b$, for some two constants k and b. Note that this property logically implies that $T(n)$ is $O(2^n)$. Let's prove the claim by induction.

The base case $n = 1$ is: $T(1) = 1 \leq k2^1 - b = 2k - b$. This is true as long as $k \geq (b+1)/2$. The inductive case: We assume our property is true for $n - 1$. We now want to show that it is true for n.

$$\begin{aligned} T(n) &= 2T(n-1) + c1 \\ &\leq 2(k2n - 1 - b) + c1 \qquad \text{(H.IND)} \\ &= k2n - 2b + c1 \\ &\leq k2n - b \end{aligned}$$

This last inequality holds as long as $b \geq c1$. So we end up with two constraints that need to be satisfied for this proof to work, and we can satisfy them simply by letting $b = c_1$ and $k = (b+1)/2$, which is always possible, as the definition of $O(\cdot)$ allows us to choose any constant. Therefore, we have proved that our property is true, and so $T(n)$ is $O(2^n)$.

Had we not added the lower term $-b$ in the upper bound of $T(n) \leq k2^n - b$, we could have not proven the bound by induction. Indeed, in the first inequality in the hypothesis of induction we would've got $T(n) \leq 2k^n + c_1$, which is $\not\leq k2^n$

Changing Variables Sometimes, a little algebraic manipulation can make an unknown recurrence similar to a known one. Consider for example:

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n.$$

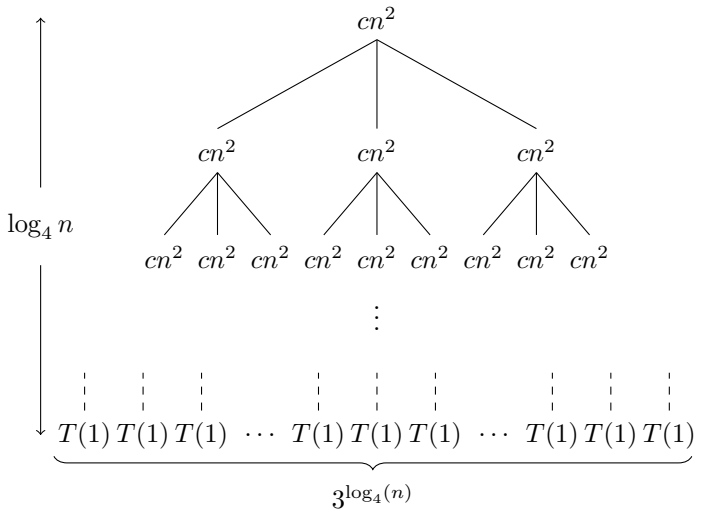
Then takign $m = \log n$, yields $T(2^m) = 2T(2^{m/2}) + m$. Renaming $S(m) = T(2^m)$, to produce the new recurrence:

$$S(m) = 2S(m/2) + m.$$

We already know $S(m) = O(m \log m)$, thus $T(n) = T(2^m) = S(m) = O(m \log m) = O(\log(n) \cdot \log(\log(n)))$.

4.4 The Recursion-Tree Method for Solving Recurrences

Drawing out a recursion tree serves as a straightforward way to devise a good guess. In a *recursion tree*, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. For example consider the recursion tree for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We start by finding an upper bound on the recurrence, and focusing only on asymptotic behavior, assume $4 \mid n$ and we write $T(n) = 3T(n/4) + cn^2$, for some $c > 0$. The recursion tree for the recurrence would be as follows:



The subproblem size for a node at depth i is easily seen to be $n/4^i$. Hence, the problem hits a leaf $n = 1$ when $n/4^i = 1$, that is, $i = \log_4 n$. Thus the problem has $\log_4 n + 1$ levels.

To determine the cost at each level of the three, see that the number of nodes at level i is 3^i. Since we we reduce by 4 the size of problem each time we go down, the cost of each node at level i is $c(n/4^i)^2$. Adding for all nodes on level i, we get the total cost for level i of the tree is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. The bottom level has $3^{\log_4(n)} = n^{\log_4 3}$ nodes, each contributing a cost of $T(1)$. Thus a total cost of $n^{\log_4 3} T(1) = \Theta(n^{\log_4 3})$ (assuming $T(1)$ is constant). Adding up all the costs to determine the cost of the entire tree:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4(n)-1} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2). \end{aligned}$$

Thus, we have derived a guess (not a proof: this was quite ... informal). This recurrence is indeed $\Theta(n^2)$, which can be easily seen. By the definition of the recurrence it is trivially $\Omega(n^2)$. The bound $O(n^2)$ can be proven by substitution easily.

4.5 The Master Theorem

The master method provides a way of solving recurrences of the form $T(n) = aT(n/b) + f(n)$. This recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b. Here $f(n)$ would represent the cost of dividing the problem and combining the results of the subproblems.

Theorem 4.1 ***(Master Theorem).*** *Let $a \geq 1$ and $b \geq 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n),$$

where we interpred n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then T(n) has the following asymptotic bounds:

1. *If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$.*
2. *If $f(n) = \Theta(n^{\log_b(a)}$, then $T(n) = \Theta(n^{\log_b(a)} \log n)$.*
3. *If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

As an example consider the recurrence given by $T(n) = 9T(n/3) + n$. For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the theorem and conclude that $T(n) = \Theta(n^2)$.

Chapter 5 Probabilistic Analysis and Randomized Algorithms

5.1 The hiring problem

Suppose we need to hire an office assistant. You interview a person and decide to hire that person or not, and we must pay the employment agency a fee for interviewing the person (c_i). To actually hire the applicant we must pay a higher cost (c_h), since we must fire the current office assistant. So after each interview, if the new applicant is better than the current office assistant, we fire the office assistant and hire the new applicant. We wish to estimate what the price of this strategy will be.

The following HIRE-ASSISTANT, expresses this strategy for hiring among n applicants.

```
HIRE-ASSISTANT(n)
1  best = 0    // Actual best candidate.
2  for i = 1 to n do
3       interview candidate i
4       if candidate i is better than candidate best then
5            best = i
6            hire candidate i
7       end if
8  end for
```

The cost model for this problem differs from the model described in previous chapters. We focus not on the running time, but on the costs incurred by the interviewing and hiring. Interviewing has a low cost c_i, and hiring an expensive cost c_h. Let m be the number of applicants hired during the strategy, the total cost associated with this algorithm is $O(c_i n + c_h m)$. No matter how many people we hire, we always interview n candidates and thus always incur the cost $c_i n$ associated with interviewing. We therefore concentrate on analyzing $c_h m$, the hiring cost. Here the quantity m varies with each run of the algorithm.

In the worst case, candidates are in increasing order of quality, thus we hire all n candidates. However, candidates do not always come in this order, nor we know or control the order they come in. Thus it is natural to ask what we expect to happen in an average case.

Probabilistic analysis *Probabilistic analysis* is the use of probability in the analysis of problems. Most commonly we use it to analyze the running time of algorithms. In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about the sitribution of the input. When we analyze our algorithm, computing an *average-case running time*, where we take the average over the distribution of the possible inputs (we take the expected value).

For the hiring problem we can assume that the applicants come in a random order, and that between any two candidates we can decide which one is better (that is there is a *total ordering*). Thus we can rank each applicant with an integer from 1 to n, denoting $rank(i)$ the rank of applicant i (where the highest rank yields the best applicant). So we have that the list of ranks $(rank(1), rank(2), \ldots, rank(n))$ is a permutation of the list $(1, 2, \ldots, n)$. Assumin random ordering in the applicants, we can say that the list of ranks is equally likely to be one of the $n!$ permutations of $[n]$ (the ranks form a *uniform random permutation*).

Randomized Algorithms In the hiring problem, we assumed that the applicants come in order. Insted we can change a bit the model and say we have a list of n candidates, and on each day we choose randomly which candidate to interview. So we have gained control of the process and enforced a random order.

More generally, we call an algorithm *randomized* if its behavior is determined not only by its input but also by values produced by a *random number generator*. Assume we have a random number generator RANDOM(a, b) which models a discrete uniform random variable $U(a, b)$. When analyzing the running time of a randomized algorithm we take the expected value of the running time over the distribution of values returned by the random number generator.

5.2 Indicator random variables

Indicator rrandom variables provide a convenient method for converting between probabilities and expectations. Suppose we are given a sample space S and an event $A \subseteq S$. Then the *indicator random variable* $\mathbb{1}_A$ associated with event A is definde as

$$\mathbb{1}_A(\omega) = \begin{cases} 1 & \text{if } w \in A, \\ 0 & \text{otherwise.} \end{cases}$$

Lemma. *Given a sample space S and an event $A \subseteq S$, let $\mathbb{1}_A$ be the indicator variable of the event A. Then $\mathbb{E}[\mathbb{1}_A] = \mathbb{P}(A)$.*

As an example, return to the hiring problem. Let X be te random variable whose value equals he number of times we hire a new office assistant. Now let X_i be the indicator random variable associated with the event in which candidate i is hired. Thus, we have

$$X_i = \begin{cases} 1 & \text{if candidate } i \text{ is hired,} \\ 0 & \text{otherwise.} \end{cases}$$

Then the key step is to realize that $X = \sum_{i=1}^{n} X_i$. Now to calculate $\mathbb{P}(\text{candidate } i \text{ is hired})$, we have in our strategy that candidate i is hired iff it is better than all the prior $1, \ldots, i-1$ candidates. Because we have assumed that the candidates arrive in a random order, the first i candidates are in random order. So any of the first i candidates could be the best-qualified so far. Thus candidate i has a probability of $1/i$ of being better qualified than the candidates 1 through $i-1$. Hence $\mathbb{E}[X_i] = 1/i$. Now we can compute $\mathbb{E}[X]$.

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1).$$

Thus, although we interview n people, we actually only hire aroung $\ln n$ of them on average. So we can conclude that if the candidates are presented in random order, the HIRE-ASSISTANT algorithm has an average-case total hiring cost of $O(c_h \ln n)$.

5.3 Randomized algorithms

In the previous section we showed how knowing the distribution on the inputs can help us analyze the average-case behavior of an algorithm. Many times, we do not have such

knowledge, thus precluding the average-case analysis. As mentioned, we may be able to use a randomized algorithm. In these cases we can use a randomized algorithm and force a random distribution.

In the hiring problem, we assumed a random order on inputs. We can instead, impose a distribution on the inputs. For example, before running the alorithm, randomly permute the candidates in order to enforce the property that every permutation is equally likely. Although we have modified the algorithm, we still expect to hire a new office assistant approximately $\ln n$ times. But now we expect this to be the case for any input, rather than for inputs drawn from a particular distribution. For our hiring strategy, to produced our randomized algorithm we can do the following.

```
RANDOMIZED-HIRE-ASSISTANT(n)
 1  randomly permute the list of candidates
 2  best = 0    // Actual best candidate.
 3  for i = 1 to n do
 4       interview candidate i
 5       if candidate i is better than candidate best then
 6            best = i
 7            hire candidate i
 8       end if
 9  end for
```

With this simple change we have created a randomized algorithm whose performance matches the obtained by assuming that the candidates were presented ina random order. The expected hiring cost of the procedure RANDOMIZED-HIRE-ASSISTANT is $O(c_h \ln n)$.

5.3.1 Randomly permuting arrays

Many randomized algorihtms randomize the input by permuting an array. We study two algorithms for doing this. Suppose we are given an array A of n elements, without loss of generality, containing elements 1 through n. Our goal is to produce a random permutation of the array.

In the first method, we create an array P of n random elements, then sort A as in the order of P.

```
PERMUTE-BY-SORTING(A)
 1  n = A.length
 2  let P[1..n] be a new array
 3  for i = 1 to n do
 4       P[i] = RANDOM(1, n^3)
 5  end for
 6  sort A, using P as sort keys
```

In line 4 we take a random number between 1 and n^3. With this range we make it likely that all the priorities in P are unique (it is easily seen that the probability that all entries are unique is at least $1 - 1/n$, and the algorithm can be implemented even if there are repetitions).Assume nevertheless, that priorities in P are unique.

The costly step comes in line 6 for sorting the array A. We will see this comparison sort can be done in $\Omega(n \log n)$ time. After this sort, if $P[i]$ is the jth smallest priority, then $A[i]$ lies in position j of the output. In this manner we obtain a permutation of A.

Proposition. *Procedure PERMUTE-BY-SORTING produces a uniform random permutation of the input, assuming that all the priorities are distinct.*

A better method for generating a random permutation is to permute the given array in place. The following procedure does so in $O(n)$ time.

```
PERMUTE-IN-PLACE(A)
 1  n = A.length
 2  for i = 1 to n do
 3       swap A[i] with A[RANDOM(i, n)]
 4  end for
```

We shall use a loop invariant to show that the procedure produces a uniform random permutation.

Proposition. *Procedure PERMUTE-IN-PLACE produces a uniform random permutation of the input. Proof:* We will use the following loop invariant: *Just prior to the ith iteration of the loop in lines 2 and 3, fo each possible $(i-1)$-permutation of the n elements, the subarray $A[1..i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.*

We need to see the invariant holds prior to every loop iteration. And use the loop invariant to show correctness at loop termination.

Initialization: Before the first loop, at $i = 1$, the loop invariant says for each 0-permutation, the subarray $A[1..0] = \varnothing$, contains this 0-permutation with probability $n!/n! = 1$. Which holds trivially.

Maintenance: Assume that just before the ith iteration, each possible $(i-1)$-permutation appears in the subarray $A[1..i-1]$ with probability $(n-i+1)!/n!$. We shall see, that after the ith iteration, each possible i-permutation appears in the subarray $A[1..i]$ with probability $(n-i)!/n!$.

Consider any i-permutation $(x_1, x_2, \ldots, x_i)$, which is an $(i-1)$-permutation $(x_1, x_2, \ldots, x_{i-1})$, followed by an x_i placed in $A[i]$ by the algorithm. Let E_1 denote the event in which the first $i-1$ iterations have created the particular $(i-1)$-permutation $(x_1, \ldots, x_{i-1})$ in $A[1..i-1]$; which by the loop invariant has $\mathbb{P}(E_1) = (n-i+1)!/n!$. Let E_2 be the event that the ith iteration puts x_i in $A[i]$. The i-permutation $(x1, \ldots, x_i)$ appears in $A[1..i]$, precisely when both E_1 and E_2 occur, that is the event $E_1 \cap E_2$. Now we have $\mathbb{P}(E_1 \cap E_2) = \mathbb{P}(E_2|E_1)\,\mathbb{P}(E_1)$.

The probability $\mathbb{P}(E_2|E_1)$ equals $1/(n-i+1)$, which is choosing randomly x_i from the array $A[i..n]$. Thus we have

$$\mathbb{P}(E_1 \cap E_2) = \frac{1}{n-i+1}\frac{(n-i+1)!}{n!} = \frac{(n-i)!}{n!}.$$

Termination: At termination, $i = n+1$, hence the subarray $A[1..n]$ is a given n-permutation with probability $(n-(n+1)+1)!/n! = 0!/n! = 1/n!$.

Thus, RANDOMIZE-IN-PLACE produces a uniform random permutation. ■

Chapter 6 Heapsort

6.1 Heaps

Definition. A *(binary) heap* data structure is an array object that we can view as a nearly complete binary tree (that is lowest level may be filled up to a point). Each node of the tree corresponds to an element of the array. An array A that represents a heap is an object with an $A.length$ attribute, which counts the number of elements in the array; and an $A.heapsize$ attribute, which represents how many elements in the heap are stored within array A; where it is satisfied $0 \leq A.heapsize \leq A.length$.

The root of the tree is $A[1]$, and given an index i of a node we can compute the indexes of the parent and left and right child nodes.

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

Definition. There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a *heap property.* In a *max-heap*, the *max-heap property* is that for every node other than the root, $A[\text{PARENT}(i)] \geq A[i]$. Similarly a *min-heap*, the *min-heap property* is that for every node other than the root, $A[\text{PARENT}(i)] \leq A[i]$.

Viewing the heap as a tree, we have that the *heigth* of a node in a heap is te number of edges on the longest simple downwad path from the node to a leaf, and the *height of the heap* is the height of its root.

Proposition. *Given a heap of n elements, we have:*

1. *The heap has $\lfloor \log_2 n \rfloor$ elements.*
2. *With the array representation of the heap, the leaves are indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$; thus having $\lceil n/2 \rceil$ of them.*

6.1.1 Zero indexed arrays

We discuss throughout the notes arrays with indexes starting at 1. However, it is customary to work with zero indexed arrays. An easy way to see how to transform formulas from one set of indexes is as follows. Let $\mathcal{J}$ be the set of indexes starting at 0 and J those starting at 1. Then given a formula (mapping) f, for 1-based indexes ($f : \mathcal{I} \to \mathcal{I}$), we want to find how the mapping traduces in 0-based indexes (i.e. some $g : \mathcal{J} \to \mathcal{J}$). Since we have a bijection $\phi : \mathcal{J} \to \mathcal{I}$, $\phi(j) = j + 1$, we can give the following commutative diagram.

$$\begin{array}{ccc} \mathcal{J} & \xrightarrow{\quad g \quad} & \mathcal{J} \\ {\scriptstyle\phi}\downarrow & & \uparrow{\scriptstyle\phi^{-1}} \\ \mathcal{I} & \xrightarrow[\quad f \quad]{} & \mathcal{I} \end{array}$$

Hence, we can write $g = \phi^{-1} \circ f \circ \phi$. For example, the zero based index of the parent node in a heap can be computed as:

PARENT(j)
1 **return** $\lfloor (j+1)/2 \rfloor - 1$ // Zero based index

6.2 Maintaining the heap property

In order to maintain the max-heap property we call the MAX-HEAPIFY procedure. Given an array A and an index i such that the subtrees rooted at LEFT(i) and RIGHT(i) satisfy the max-heap property, but $A[i]$ might be smaller than its children, the routine "floats down" $A[i]$ so that the binary tree rooted at index i is a max-heap.

MAX-HEAPIFY(A, i)
1 $largest = i$ // Current largest node
2 $l = \text{LEFT}(i)$
3 $r = \text{RIGHT}(i)$
4 **if** $l \leq A.heapsize$ **and** $A[l] > A[i]$ **then**
5 $largest = l$
6 **end if**
7 **if** $r \leq A.heapsize$ **and** $A[l] > A[largest]$ **then**
8 $largest = r$
9 **end if**
// Max-Heapify the modified child subtree
10 **if** $largest \neq i$ **then**
11 SWAP($A[i], A[largest]$)
12 MAX-HEAPIFY($A, largest$)
13 **end if**

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fic up the relationships amog the elements $A[i]$ and its childs, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i. It is easily seen that the child subtrees have at most $2n/3$ nodes (check for the worst case in which the heap tree is half-full). Thus the running time of the MAX-HEAPIFY is given by the recurrence:

$$T(n) = T(2n/3) + \Theta(1),$$

which applying the master theorem, we get that $T(n) = O(\log_2 n)$.

6.3 Building a heap

We can use the MAX-HEAPIFY procedure in a bottom-up manner to convert an arbitrary array $A[1..n]$ into a max-heap. procedure in a bottom-up manner to convert an arbitrary array $A[1..n]$ into a max-heap. Since each of the leaves in the array $A[\lfloor n/2 \rfloor + 1..n]$ is a 1-element max-heap, we can build a max-heap iterating upwards through all other nodes.

BUILD-MAX-HEAP(A)
1 $A.heapsize = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1 **do**
3 MAX-HEAPIFY(A, i)
4 **end for**

In order to prove its correctness, use the following loop invariant: *At the start of each iteration of the **for** loop, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.*

Initialization: Prior to the first initialization, the nodes $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ are leafs thus roots of trivial max-heaps.

Maintenance: The children of node i are numbered greater than i. By the loop invariant these are childs are roots of a max-heap. This satisfies the condition on which a call to MAX-HEAPIFY(A, i) would make at node i a max-heap root. Moreover the MAX-HEAPIFY call preserves the property that nodes $i+1, i+2, \ldots, n$ are all roots of max-heaps. Tecrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$, hence by loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In partiular, node 1 is.

To derive an upper bound for the running time of the BUILD-MAX-HEAP procedure, observe that an n-element heap has heaight $\lfloor \log_2 n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h. Since the time required by *Max-Heapify* when called on a node of height h is $O(log_2 n) = O(h)$, we can express the total cost of BUILD-MAX-HEAP as being bounded above by:

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right),$$

and using the fact that $\sum_{h=0}^{\infty} h/2^h = 2$, constant, we finally can write

$$T(n) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n).$$

Thus we can build a max-heap from an unordered array in linear time. Similarly we achieve this same bound to build a min-heap from an unordered array.

6.4 The heapsort algorithm

The heapsort algorithm takes as an input an unordered array $A[1..n]$. It starts by building an max-heap by calling BUILD-MAX-HEAP. Since the maximum element on A is at $A[1]$ we can put it into its final correct position by swapping it with $A[n]$. If we now discard element $A[n]$ from the heap by decrementing $A.heapsize$, the children of the new root are max-heaps, but not the new root. But we can restore the max-heap by caling MAX-HEAPIFY$(A, 1)$ which leaves a max-heap in $A[1..n-1]$. This proces is iterated for each node until obtaining a sorted array.

```
HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 for i = A.length downto 2 do
3     SWAP(A[1], A[i])
4     A.heapsize = A.heapsize − 1
5     MAX-HEAPIFY(A, 1)
6 end for
```

A more formal proof of correctness can be done by checking the loop invariant of Exercise 6.4-2. The HEAPSORT procedure takes time $O(n \log n)$, since the call to BUILD-MAX-HEAP takes $O(n)$ time, and we call $n-1$ times the procedure MAX-HEAPIFY which takes time $O(\log n)$.

6.5 Priority queues

Definition. A *priority queue* is a data structure for mainaining a set S of elements, each with an associated value called a *key*. A *max-priority queue* supports the following operations.

- INSERT(S, x) inserts the element x into S, that is $S = S \cup \{x\}$.
- MAXIMUM(S) returns the element of S with the largest key.
- EXTRACT-MAX(S) removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k) increases the value of element x's key to the new and greater value k.

Alternatively, a *min-priority queue* supports operations INSERT, MINIMUM, EXTRACT-MIN, DECREASE-KEY. We now study each of these procedures for max-priority queues.

```
HEAP-MAXIMUM(A)
1 return A[1]
```

This procedure HEAP-MAXIMUM implements MAXIMUM in $\Theta(1)$ time.

```
HEAP-EXTRACT-MAX(A)
1 if A.heapsize < 1 then
2     error "heap underflow"
3 end if
4 max = A[1]
5 A[1] = A[A.heapsize]
6 A.heapsize = A.heapsize − 1
7 MAX-HEAPIFY(A, 1)
8 return max
```

The running time of HEAP-ECTRACT-MAX is clearly $O(\log n)$, which comes from the call to MAX-HEAPIFY.

For the HEAP-INCREASE-KEY we increase the given item's key, and changing it wits its parents, untill its parent has a greater key. At all moments the node has childs which are max-heaps, and at termination, the node is the root of a max-heap tree. Thus it terminates with a max-heap.

```
HEAP-INCREASE-KEY(A, i, key)
1 if key < A[i] then
2     error "new key is smaller than current key"
3 end if
4 A[i] = key
5 while i > 1 and A[PARENT(i) < A[i] do
6     SWAP(A[i], A[PARENT(i))
7     i = PARENT(i)
8 end while
```

The running time of HEAP-INCREASE-KEY is $O(\log n)$ since the path from the last node updated and the initial element at i, has length $O(\log n)$.

The procedure MAX-HEAP-INSERT inserts a new key at the end of the heap (incrementing heap size) with value $-\infty$, thus maintaining the max-heap property, and calls HEAP-INCREASE-KEY to this new last node with the value we want to add. Clearly $O(\log n)$ time.

```
HEAP-INCREASE-MAX(A, i, key)
1 A.heapsize = A.heapsize + 1
2 A[A.heapsize] = −∞
3 HEAP-INCREASE-KEY(A, A.heapsize, key)
```

Chapter 7 Quicksort

7.1 Description of quicksort

Quicksort applies the divide-and-conquer for sorting a typical subarray $A[p..r]$

- **Divide:** Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is $\leq A[q]$, which in turn is, $\geq$ than any element in $A[q+1..r]$. Compute the index q as part of the partitioning procedure.
- **Conquer:** Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.
- **Combine:** Because the subarrays are already sorted, no need to combine them: the entire array $A[p..r]$ is now sorted.

The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
1  if p < r then
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
5  end if
```

The key to this algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1 do
4       if A[j] ≤ x then
5            i = i + 1
6            SWAP(A[i], A[j])
7       end if
8  end for
9  SWAP(A[i + 1], A[r])
10 return i + 1
```

We can prove the above PARTITION procedure with the following loop invariant: *At the beginning of each iteration of the loop of lines 3-7, for any array index k,*

1. *If $p \leq k \leq i$, then $A[k] \leq x$.*
2. *If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.*
3. *If $k = r$, then $A[k] = x$.*

Thus at termination of the **for** loop we have x at the last position of the array, and performing the last SWAP$(A[i+1], A[r])$, we partition $A[p..r]$ into three regions: $\leq x, x, > x$, hence a call to quicksort at the both left and right regions yields to sorting the array $A[p..r]$.

The running time of the PARTITION on the subarray $A[p..r]$ is $\Theta(n)$, where $n = r - p + 1$.

7.2 Randomized version of quicksort

In order to not depend on the hypothesis that the input comes in a random distribution, we can change the quicksort algorithm to randomize it. We could explicitly permut the input (e.g. with RANDOMIZE-IN-PLACE procedure from section 5). A different approach is taken which yields to faster algorithm and simpler analysis: we will pick at random which element x will be taken to partition the array into the sections $\leq x, x, > x$. Thus we swap a random element from $A[p..r]$ with $A[r]$ and call the previous PARTITION procedure:

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  SWAP(A[i], A[r])
3  return PARTITION(A, p, r)
```

Thus the new quicksort calls RANDOMIZED-PARTITINO instead of PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r then
2       q = RANDOMIZED-PARTITION(A, p, r)
3       RANDOMIZED-QUICKSORT(A, p, q − 1)
4       RANDOMIZED-QUICKSORT(A, q + 1, r)
5  end if
```

7.3 Analysis of quicksort

7.3.1 Worst-case analysis

Using the substitutiton method we will see that the running time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time of the randomized algorithm. Then we have the recurrence:

$$T(n) = \max_{0 \leq q \leq n-1} \{T(q) + T(n-1-q)\} + \Theta(n),$$

where the parameter q ranges from 0 to $n-1$, depending on the obained partitions in RANDOMIZED-PARTITION. We guess that $T(n) \leq cn^2$, for some positive constant c. By the substitution method we get:

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} \{cq^2 + c(n-1-q)^2\} + \Theta(n) \\ &= c \cdot \max_{0 \leq q \leq n-1} \{q^2 + (n-1-q)^2\} + \Theta(n). \end{aligned}$$

The latter expression $q^2 + (n-1-q)^2$ is easily seen to achieve a maximum at either $q = 0$ or $q = n-1$. Hence:

$$\begin{aligned} T(n) &= c \cdot \max_{0 \leq q \leq n-1} \{q^2 + (n-1-q)^2\} + \Theta(n) \\ &\leq c(n-1)^2 + \Theta(n) = cn^2 - c(2n-1) + \Theta(n) \\ &\leq cn^2. \end{aligned}$$

Where the latter is tre for large enough c so that the term $c(2n-1)$ dominates over the $\Theta(n)$ term. Hence, $T(n) = O(n^2)$. A similar argument is made to prove that $T(n) = \Omega(n^2)$ by using substitution on $T(n) \geq cn^2$ for some $c > 0$. Thus the worst case-running time of quicksort is $\Theta(n^2)$.

7.3.2 Expected running time

The QUICKSORT and RANDOMIZED-QUICKSORT procedure differ only in how they select the pivot element. We can then couch our analysis of RANDOMIZED-QUICKSORT by discussing the QUICKSORT algorithm and PARTITION, with the assumption that the pivot elements are selected randomly from the subarray passed to RANDOMIZED-PARTITION. We will assume distinct numbers in the array.

The running time of QUICKSORT is dominated by the time spent in the PARTITION procedure. Each time the PARTITION procedure is called, it selects a pivot element, and this element is never again included in any further calls to QUICKSORT and PARTITION. Thus, there can be at most n calls to PARTITION. One call to PARTITION takes $\Theta(1)$ time, plus the time spent in the **for** loop to rearrange the subarray in lines 3-8. Each iteration of the **for** loop performs a comparison in line 4, comparing the pivot element to another element of the array A. If we can count the total number of times such comparison is executed, we can bound the total time spent in the **for** loop during the entire execution of QUICKSORT.

Lemma. *Let X be the total number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.*

Thus insted to analyze how many times is made in *each* call to PARTITION, we count the total number of such comparisons. To do so, we will begin to study when two elements of the array are compared. Let the elements of the array A be renamed as $z_1, z_2, \ldots, z_n$, where z_i is the smallest ith element. Also define $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ to be the set of elements between z_i and z_j, inclusive.

See first that some z_i and z_j are compared at most once, only when one of the two are taken as pivot by PARTITION and then that element is never again compared to any other elemens in the array.

The analysis use the indicator random variable $X_{ij} = \mathbb{1}\{z_i \text{ is compared to } z_j\}$, where we consider whether the comparison takes place at any time duing executino of the algorithm. Since each pair is compared at most once, we can characterize the total number of comparisons by $X = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}$, hence

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathbb{P}(z_i \text{ is compared to } z_j).\end{aligned}$$

Let us compute this last probability. First analyze when two elements are not compared. Since the elements are distinct, once a pivot x is chosen with $z_i < x < z_j$, then z_i and z_j will not be compared at any subsequent time. On the other hand, if z_i is the pivot before any other element in Z_{ij}, then z_i will be compared to each element in Z_{ij}, except for itself. The same happens with z_j and Z_{ij}. Thus z_i and z_j are compared if and only if the first element to be chosen as pivot from Z_{ij} is either z_i or z_j.

Whe can now compute the probability that this event occurs. Prior to the point at which any element of Z_{ij} is chosen as a pivot, the whole set Z_{ij} is together in the same partition. Therefore, any element of Z_{ij} is equally likely to be chosen as pivot. Because $|Z_{ij}| = j - i + 1$, and pivots are chosen randomly independently, the probability that any given element is the first one chosen as pivot is $1/(j - i + 1)$. Thus having:

$$\begin{aligned}&\mathbb{P}(z_i \text{ is compared to } z_j) = \\ &\quad= \mathbb{P}(z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}) \\ &\quad= \mathbb{P}(z_i \text{ is first pivot chosen from } Z_{ij}) \\ &\qquad+ \mathbb{P}(z_i \text{ is first pivot chosen from } Z_{ij}) \\ &\quad= \frac{2}{j-i+1}.\end{aligned}$$

Thus the total expectation is

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1} \\ &= \sum_{i=1}^{n-1}\sum_{k=1}^{n}\frac{2}{k} \\ &= \sum_{i=1}^{n-1}O(\log n) \\ &= O(n\log n).\end{aligned}$$

Thus we can conclude that, using the RANDOMIZED-PARTITION, the expected running time of quicksort is $O(n \log n)$ when values are distinct.

Chapter 8 Sorting in Linear Time

8.1 Lower bounds for sorting

So far we have been studying only *comparison sorts*: they use only on comparisons between the input elements to gain order informatino about an input sequence $(a_1, a_2, \ldots, a_n)$. We will assume, without loss of generality, that all the input elements are distinct.

So, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i > a_j$, $a_i \geq a_j$, to determine their relative order. Assuming distinct elements, all these operations yield identical information about relative order of a_i and a_j. Hence assume that all the comparisons are of the form $a_i \leq a_j$.

Decision Tree. We can view comparison sorts abstractly in terms of *decision trees*, a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

In a decision tree, we annotate each internal node by $i : j$ for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence. We also annotate each leaf by a permutation $(\sigma(1), \sigma(2), \ldots, \sigma(n))$. The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each inernal node indicates a comparison $a_i \leq a_j$; the left subtree indicate the subsequent comparisons once we know that $a_i \leq a_j$, the right subtree the comparisons for the case $a_i > a_j$. When we arrive at a leaf, the algorithm has established an order $a_{\sigma(1)} \leq a_{\sigma(2)} \leq \cdots \leq a_{\sigma(n)}$.

Because any sorting algorithm can produce any of the $n!$ permutations, each one of those permutations must appear as one of the leaves of the decision tree for a comparison sort to be correct. Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort. Thus, considering only decision trees in which each permutation appears as a reachable leaf.

Lower bound for the worst case. The lenght of the longest simple path from the root of the decision tree to any of its reachable leaves, that is the height of the decision tree, represents the worst case number of comparisons that the corresponding sorting algorithm performs. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lowe bound on the running time of any comparison sort algorithm. The theorem establishes such lower bound.

Theorem 8.1 *Any comparison sort algorithm requires* $\Theta(n \log n)$ *comparisons on the worst case.*

Proof: It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have that $n! \leq l$. Since a binary tree of height h has at most 2^h leaves, we have $n! \leq l \leq 2^h$, which implies $h \leq \log(n!) = \Omega(n \log n)$. ■

Corollary. *Heapsort and merge sort are asymptotically optimal comparison sorts.*

Chapter 10 Elementary Data Structures

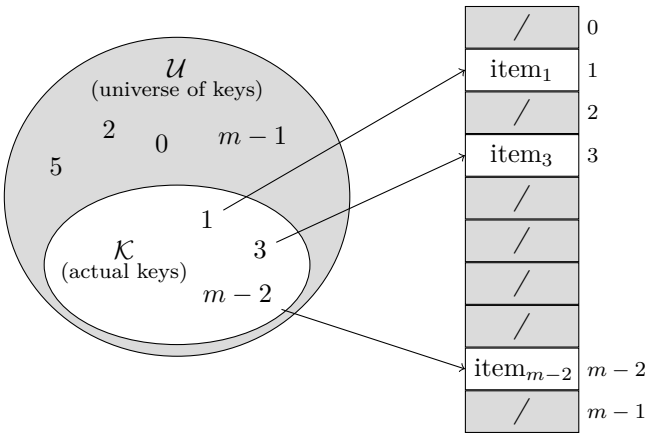
Chapter 11 Hash Tables

Many applications require the use of an abstract data type called a *dictionary*, which maintains a dynamic set of items, each with a distinct key. The operations supported by this dictionary are INSERT, DELETE and SEARCH (search for the item in the dictionary with the given key, if such item exists). These data structures are typically implemented using a very powerful data structure called *hash tables*, which allows to perform the mentioned operations in $\Theta(1)$ time.

A part from dictionaries, hash tables and their fundamental hash functions, have plenty of applications: databases, compiler variable table definitions, networking router lookup tables, substring matching, string commonalities, cryptographic hash functions ...

11.1 Direct-access tables

Suppose we need a dynamic set where each element has an integer key drawn from the universe $\mathcal{U} = \{0, 1, \ldots, m-1\}$. A simple and primitive approach is to use direct-access tables, which stores each element in an array indexed by the key of the element.



Where the actual keys $\mathcal{K}$, are the actual keys being used by the dictionary. This approach has two big drawbacks:

1. Keys may not be integers.
2. Big memory usage to keep array for ALL possible key values.

The first problem is solved with *prehashing*: all keys are finite and discrete, thus can be represented as a finite string of bits. The second problem will be solved with another approach in next section.

11.2 Hash Tables

When the set $\mathcal{K}$ of keys stored in adictionary is much smaller than the universe $\mathcal{U}$ of all possible keys, a hash table requires much less storage than a direct address table. Specifically, we can reduce the storage requirement to $\Theta(|\mathcal{K}|)$, while maintaining the $\Theta(1)$ time for search operations.

With hashing we define a *hash function*

$$h : \mathcal{U} \to \{0, 1, \ldots, m-1\},$$

which maps any key from the universe of keys, to a slot in a *hash table* $T[0\,..\,m-1]$. There is one hitch: since typically $|\mathcal{U}| > m$, by the pigeon-hole principle, there will different keys which will be mapped to the same slot, a situation called a *collision*.

Of course, the ideal situation is to have the least collisions at all, which will be achieved by a propper hash function h. We present now a method for resolving collisions called chaining. Later chapters present an alternative collisino resolution method called open addressing.

11.2.1 Chaining

In *chaining* we place all the elements that hash to the same slot, into the same linked list. We store also the key and the value in the list so that we can searh for a given key in each slot, and the list can be doubly-linked for ease of deletion of items.

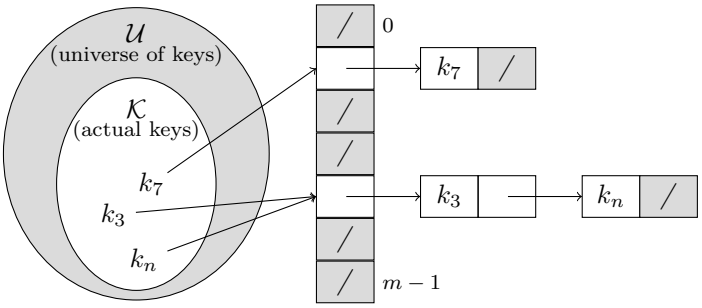
Thus the dictionary operations on a hash table T are easy to implement with chaining:

CHAINED-HASH-INSERT(T, x)
1 **if** there is no element with $x.key$ in list $T[h(x.key)]$ **then**
2 insert x at the head of the list
3 **else**
4 override such element with the new element x
5 **end if**

CHAINED-HASH-DELETE(T, x)
1 delete x from the list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)
1 search for an element with key k in list $T[h(k)]$

All these operations require searching in the list $T[h(k)]$ for some key k. Thus the worst case time of these operations is proportional to the length of the list $T[h(k)]$. Thus if $n = |\mathcal{K}|$, the worst-case behavior for hashing with chaining is then all n elements end up in the same linked list, and these operations take $\Theta(n)$ time! Hash tables however, have a very good average case (and empirically tested). We can make a first assumption for ease of analysis (however, this assumption ends up being false in reality).



Simple Uniform Hashing: Each key is equally likely to be linked to any slot of the table, independent of where other keys are linked.

Then, for $j = 0, 1, \ldots, m-1$, let us denote the length of the list T[j] by n_j, so that $n = n_0 + n_1 + \cdots + n_{m-1}$. Hence, the expected value of n_j is $\mathbb{E}[n_j] = n/m \equiv \alpha$, called the *load factor*.