Chapter 15

Algorithms

This chapter covers how to analyze the running time of algorithms.

15.1 Introduction

The techniques we've developed earlier in this course can be applied to analyze how much time a computer algorithm requires, as a function of the size of its input(s). We will see a range of simple algorithms illustrating a variety of running times. Three methods will be used to analyze the running times: nested loops, resource consumption, and recursive definitions.

We will figure out only the big-O running time for each algorithm, i.e. ignoring multiplicative constants and behavior on small inputs. This will allow us to examine the overall design of the algorithms without excessive complexity. Being able to cut corners so as to get a quick overview is a critical skill when you encounter more complex algorithms in later computer science classes.

15.2 Basic data structures

Many simple algorithms need to store a sequence of objects $a_1, \ldots, a_n$. In pseudocode, the starting index for a sequence is sometimes 0 and sometimes

1, and you often need to examine the last subscript to find out the length. Sequences can be stored using either an array or a linked list. The choice sometimes affects the algorithm analysis, because these two implementation methods have slightly different features.

An array provides constant-time access to any element. So you can quickly access elements in any order you choose and the access time does not depend on the length of the list. However, the length of an array is fixed when the array is built. Changing the array length takes time proportional to the length of the array, i.e. $O(n)$. Adding or deleting objects in the middle of the array requires pushing other objects sideways. This can also take $O(n)$ time. Two-dimensional arrays are similar, except that you need to supply two subscripts e.g. $a_{x,y}$.

In a linked list, each object points to the next object in the list. An algorithm has direct access only to the elements at the ends of the list.[1] Objects in the middle of the list can only be accessed by walking element-by-element from one end, which can take $O(n)$ time. However, the length of the list is flexible and objects can added to, or removed from, the ends of the list in constant time. Once you are at a position in the middle of a list, objects can be added or deleted at that position in constant time.

A linked list starts with its *head* and ends with its *tail*. For example, suppose our list is $L = (1, 7, 3, 4, 7, 19)$. Then $head(L)$ is 1 and $tail(L)$ is 19. The function *pop* removes and returns the value at the head of a list. I.e. pop(L) will return 1 leave the list L containing $(7, 3, 4, 7, 19)$.

For some algorithms, the big-O performance does not depend on whether arrays or linked lists are used. This happens when the number of objects is fixed and the objects are accessed in sequential order. However, remember that a big-O analysis ignores multiplicative constants. All other things being equal, array-based implementations tend to have smaller constants and therefore run faster.

[1]Strictly speaking, this works only for certain types of linked lists. See a data structures text for all the gory details.

15.3 Nested loops

Algorithms based on nested loops are the easiest to analyze. Suppose that we have a set of 2D points and we would like to find the pair that are closest together. Our code might look as in Figure 15.1, assuming that the function dist computes the distance between two 2D points.

To analyze this code in big-O terms, first notice that the start-up code in lines 1-4 and the ending code in line 12 takes the same amount of time regardless of the input size n. So we'll say that it takes "constant time" or $O(1)$ time. The block of code inside both loops (lines 7-11) takes a constant time to execute once. So the big-O running time of this algorithm is entirely determined by how many times the loops run.

The outer loop runs n times. The inner loop runs n times during each iteration of the outer loop. So the block of code inside both loops executes $O(n^2)$ times.

```
01  closestpair(p_1, ..., p_n) : array of 2D points)
02          best1 = p_1
03          best2 = p_2
04          bestdist = dist(p_1,p_2)
05          for i = 1 to n
06                  for j = 1 to n
07                          newdist = dist(p_i,p_j)
08                          if (i ≠ j and newdist < bestdist)
09                                  best1 = p_i
10                                  best2 = p_j
11                                  bestdist = newdist
12          return (best1, best2)
```

Figure 15.1: Finding the closest pair using nested loops.

This code examines each pair of 2D points twice, once in each order. We could avoid this extra work by having the inner loop (j) run only from $i+1$ to n. In this case, the code inside both loops will execute $\sum_{i=1}^{n}(n-i)$ times. This is equal to $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$. This is still $O(n^2)$: our optimization improved the constants but not the big-O running time. Improving the big-

O running time—$O(n \log n)$ is possible–requires restructuring the algorithm and involves some geometrical detail beyond the scope of this chapter.

15.4 Merging two lists

When code contains a while loop, rather than a for loop, it can be less obvious how many times the loop will run. For example, suppose we have two sorted lists, $a_1, \ldots, a_p$ and $b_1, \ldots, b_q$. We can merge them very efficiently into a combined sorted list. To do this, we make a new third empty list to contain our merged output. Then we examine the first elements of the two input lists and move the smaller value onto our output list. We keep looking at the first elements of both lists until one list is empty. We then copy over the rest of the non-empty list. Figure 15.2 shows pseudocode for this algorithm.

```
01  merge(L1,L2: sorted lists of real numbers)
02          O = emptylist
03          while (L1 is not empty or L2 is not empty)
04                  if (L1 is empty)
05                          move head(L2) to the tail of O
06                  else if (L2 is empty)
07                          move head(L1) to the tail of O
08                  else if (head(L1) <= head(L2))
09                          move head(L1) to the tail of O
10                  else move head(L2) to the tail of O
11          return O
```

Figure 15.2: Merging two lists

For merge, a good measure of the size of the input is the length of the output array n, which is equal to the sum of the lengths of the two input arrays $(p + q)$. The merge function has one big while loop. Since the operations within the loop (lines 4-10) all take constant time, we just need to figure out how many times the loop runs, as a function of n.

A good way to analyze while loops is to track a resource that has a known size and is consumed as the loop runs. In this case, each time the loop runs, we move one number from L_1 or L_2 onto the output list O. We have n

numbers to move, after while the loop halts. So the loop must run n times. So merge takes $O(n)$ (aka linear) time. This is called a *resource consumption* analysis.

15.5 A reachability algorithm

The function in Figure 15.3 determines whether one node t in a graph is reachable from another node s, i.e. whether there is a path connecting s and t. To do this, we start from s and explore outwards by following the edges of the graph. When we visit a node, we place a marker on it so that we can avoid examining it again (and thereby getting the code into an infinite loop). The temporary list M contains nodes that we have reached, but whose neighbors have not yet been explored.[2]

```
01 reachable(G: a graph; s,t: nodes in G)
02         if s = t return true
03         Unmark all nodes of G.
04         M = emptylist
05         Mark node s and add it to M
06         while (M is not empty)
07                 p = pop(M)
08                 for every node q that is a neighbor of p in G
09                         if q = t return true
10                         else if q is not marked, mark q and add it to M
11         return false
```

Figure 15.3: Can node s be reached from node t?

We can use a resource consumption argument to analyze the while loop in this code. Suppose that G has n nodes and m edges. No node is put onto the list M more than once. So the loop in lines 6-10 runs no more than n times.

[2]Line 5 is deliberately vague about which end of M the nodes are added to and, therefore, vague about the order in which nodes are explored.

Line 8 starts at a node q and find all its neighbors p.[3] So it traces all the edges involving q. During the whole run of the code, a graph edge might get traced twice, once in each direction. There are m edges in the graph. So lines 9-10 cannot run more than $2m$ times.

In total, this algorithm needs $O(n+m)$ time. This is an interesting case because neither of the two terms n or m dominates the other. It is true that the number of edges m is no $O(n^2)$ and thus the connected component algorithm is $O(n^2)$. However, in most applications, relatively few of these potential edges are actually present. So the $O(n+m)$ bound is more helpful.

Notice that there is a wide variety of graphs with n nodes and m edges. Our analysis was based on the kind of graph that wold causes the algorithm to run for the longest time, i.e. a graph in which the algorithm reaches every node and traverses every edge, reaching t last. This is called a *worst-case* analysis. On some input graphs, our code might run much more quickly, e.g. if we encounter t early in the search or if much of the graph is not connected to s. Unless the author explicitly indicates otherwise, big-O algorithm analyses are normally understood to be worst-case.

15.6 Binary search

We will now look at a strategy for algorithm design called "divide and conquer," in which a larger problem is solved by dividing it into several (usually two) smaller problems. For example, the code in Figure 15.4 uses a technique called binary search to calculate the square root of its input. For simplicity, we'll assume that the input n is quite large, so that we only need the answer to the nearest integer.

This code operates by defining a range of integers in which the answer must live, initially between 1 and n. Each recursive call tests whether the midpoint of the range is higher or lower than the desired answer, and selects the appropriate half of the range for further exploration. The helper function squarerootrec returns when it has found $\lfloor\sqrt(n)\rfloor$. The function then checks whether this value, or the next higher integer, is the better approximation.

[3] We assume that the internal computer representation of the graph stores a list of neighbors for each node.

```
01 squareroot(n: positive integer)
02         p = squarerootrec(n, 1, n)
03         if (n − p^2 ≤ (p + 1)^2 − n)
05                 return p
05         else return p + 1


11 squarerootrec(n, bottom, top: positive integers)
12         if (bottom = top) return bottom
13         middle = floor((bottom+top)/2)
14         if (middle^2 == n)
15                 return middle
16         else if (middle^2 ≤ n)
17                 return squarerootrec(n,middle,top)
18         else
19                 return squarerootrec(n,bottom,middle)
```

Figure 15.4: Binary search for $\sqrt{n}$

This isn't the fastest way to find a square root,[4] but this simple method generalizes well to situations in which we have only a weak model of the function we are trying to optimize. This method requires only that you can test whether a candidate value is too low or too high. Suppose, for example, that you are tuning a guitar string. Many amateur players can tell if the current tuning is too low or too high, but have a poor model of how far to turn the tuning knob. Binary search would be a good strategy for optimizing the tuning.

To analyze how long binary search takes, first notice that the start-up and clean-up work in the main squareroot function takes only constant time. So we can basically ignore its contribution. The function squarerootrec makes one recursive call to itself and otherwise does a constant amount of work. The base case requires only a constant amount of work. So if the running time of squarerootrec is $T(n)$, we can write the following recursive definition for $T(n)$, where c and d are constants.

[4] A standard faster approach is Newton's method.

- $T(1) = c$
- $T(n) = T(n/2) + d$

If we unroll this definition k times, we get $T(n) = T(\frac{n}{2^k}) + kd$. Assuming that n is a power of 2, we hit the base case when $k = \log n$. So $T(n) = c + d \log n$, which is $O(\log n)$.

15.7 Mergesort

Mergesort takes an input linked list of numbers and returns a new linked list containing the sorted numbers. This is somewhat different from bubble and insertion sort, which rearrange the values within a single array (and don't return anything).

Mergesort divides its big input list (length n) into two smaller lists of length $n/2$. Lists are divided up repeatedly until we have a large number of very short lists, of length 1 or 2 (depending on the preferences of the code writer). A length-1 list is necessarily sorted. A length 2 list can be sorted in constant time. Then, we take all these small sorted lists and merge them together in pairs, gradually building up longer and longer sorted lists until we have one sorted list containing all of our original input numbers. Figure 15.5 shows the resulting pseudocode.

```
01 mergesort(L = a_1, a_2, ..., a_n: list of real numbers)
02         if (n = 1) then return L
03         else
04                 m = ⌊n/2⌋
05                 L_1 = (a_1, a_2, ..., a_m)
06                 L_2 = (a_{m+1}, a_{m+2}, ..., a_n)
07                 return merge(mergesort(L_1),mergesort(L_2))
```

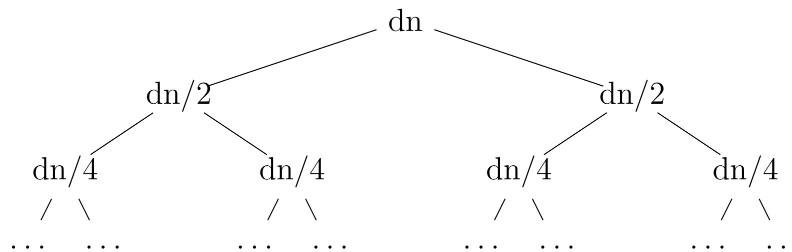
Figure 15.5: Sorting a list using mergesort

On an input of length n, mergesort makes two recursive calls to itself. It also does $O(n)$ work dividing the list in half, because it must walk, element by element, from the head of the list down to the middle position. And it

does $O(n)$ work merging the two results. So if the running time of mergesort is $T(n)$, we can write the following recursive definition for $T(n)$, where c and d are constants.

- $T(1) = c$
- $T(n) = 2T(n/2) + dn$

This recursive definition has the following recursion tree:



The tree has $O(\log n)$ non-leaf levels and the work at each level sums up to dn. So the work from the non-leaf nodes sums up to $O(n \log n)$. In addition, there are n leaf nodes (aka base cases for the recursive function), each of which involves c work. So the total running time is $O(n \log n) + cn$ which is just $O(n \log n)$.

15.8 Tower of Hanoi

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. It consists of three pegs and a set of k disks of graduated size that fit on them. The disks start out in order on one peg. You are allowed to move only a single disk at a time. The goal is to rebuild the ordered tower on another peg without ever placing a disk on top of a smaller disk.

The best way to understand the solution is recursively. Suppose that we know how to move k disks from one peg to another peg, using a third temporary-storage peg. To move $k+1$ disks from peg A to peg B using a third peg C, we first move the top k disks from A to C using B as temporary storage. Then we move the biggest disk from A to B. Then we move the other k disks from C to B, using A as temporary storage. So our recursive solver would have pseudocode as in Figure 15.6.

```
01  hanoi(A,B,C: pegs, d_1, d_2 ... d_n: disks)
02          if (n = 1) move d_1 = d_n from A to B.
03          else
04                  hanoi(A,C,B,d_1, d_2, ... d_{n-1})
05                  move d_n from A to B.
06                  hanoi(C,B,A,d_1, d_2, ... d_{n-1})
```

Figure 15.6: Solving the Towers of Hanoi problem

The function hanoi breaks up a problem of size n into two problems of size $n-1$. Alert! Warning bells! This can't be good: the sub-problems aren't much smaller than the original problem!

Anyway, hanoi breaks up a problem of size n into two problems of size $n-1$. Other than the two recursive calls, it does only a constant amount of work. So the running time $T(n)$ for the function hanoi would be given by the recursive definition (where c and d are constants):

- $T(1) = c$
- $T(n) = 2T(n-1) + d$

If we unroll this definition, we get

$$
\begin{aligned}
T(n) &= 2T(n-1) + d \\
&= 2 \cdot 2(T(n-2) + d) + d \\
&= 2 \cdot 2(2(T(n-3) + d) + d) + d \\
&= 2^3 T(n-3) + 2^2 d + 2d + d \\
&= 2^k T(n-k) + d \sum_{i=0}^{k-1} 2^i
\end{aligned}
$$

We'll hit the base case when $k = n - 1$. So

$$
\begin{aligned}
T(n) &= 2^k T(n-k) + d \sum_{i=0}^{k-1} 2^i \\
&= 2^{n-1} c + d \sum_{i=0}^{n-2} 2^i \\
&= 2^{n-1} c + d(2^{n-1} - 1) \\
&= 2^{n-1} c + 2^{n-1} d - d \\
&= O(2^n)
\end{aligned}
$$

15.9 Multiplying big integers

Suppose we want to multiply two integers. Back in the Bad Old Days of slow computers, we would need to multiply moderate-sized integers digit-by-digit. These days, we typically have a CPU that can multiply two moderate-sized numbers (e.g. 16-bit, 32-bit) as a single operation. But some applications (e.g. in cryptography) involve multiplying very, very large integers. Each very long integer must then be broken up into a sequence of 16-bit or 32-bit integers.

Let's suppose that we are breaking up each integer into individual digits, and that we're working in base 2. The recursive multiplication algorithm divides each n-digit input number into two $n/2$-digit halves. Specifically,

suppose that our input numbers are x and y and they each have $2m$ digits. We can then divide them up as

$$x = x_1 2^m + x_0$$
$$y = y_1 2^m + y_0$$

If we multiply x by y in the obvious way, we get

$$xy = A2^{2m} + B2^m + C$$

where $A = x_1y_1$, $B = x_0y_1 + x_1y_0$, and $C = x_0y_0$. Set up this way, computing xy requires multiplying four numbers with half the number of digits.

In this computation, the operations other than multiplication are fast, i.e. $O(m) = O(n)$. Adding two numbers can be done in one sweep through the digits, right to left. Multiplying by 2^m is also fast, because it just requires left-shifting the bits in the numbers. Or, if you aren't very familiar with binary yet, notice that multiplying by 10^m just requires adding m zeros to the end of the number. Left-shifting in binary is similar.

So, the running time of this naive method has the recursive definition:

- $T(1) = c$
- $T(n) = 4T(n/2) + O(n)$

The closed form for $T(n)$ is $O(n^2)$ (e.g. use unrolling).

The trick to speeding up this algorithm is to rewrite our algebra for computing B as follows

$$B = (x_1 + x_0)(y_1 + y_0) - A - C$$

This means we can compute B with only one multiplication rather that two. So, if we use this formula for B, the running time of multiplication has the recursive definition

- $P(1) = c$
- $P(n) = 3P(n/2) + O(n)$

It's not obvious that we've gained anything substantial, but we have. If we build a recursion tree for P, we discover that the kth level of the tree contains 3^k problems, each involving $n\frac{1}{2^k}$ work. So each non-leaf level requires $n(\frac{3}{2})^k$ work. The sum of the non-leaf work is dominated by the bottom non-leaf level.

The tree height is $\log_2(n)$, so the bottom non-leaf level is at $\log_2(n) - 1$. This level requires $n(\frac{3}{2})^{\log_2 n}$ work. If you mess with this expression a bit, using facts about logarithms, you find that it's $O(n^{\log_2 3})$ which is approximately $O(n^{1.585})$.

The number of leaves is $3^{log_2 n}$ and constant work is done at each leaf. Using log identities, we can show that this expression is also $O(n^{\log_2 3})$.

So this trick, due to Anatolii Karatsuba, has improved our algorithm's speed from $O(n^2)$ to $O(n^{1.585})$ with essentially no change in the constants. If $n = 2^{10} = 1024$, then the naive algorithm requires $(2^{10})^2 = 1,048,576$ multiplications, whereas Katatsuba's method requires $3^{10} = 59,049$ multiplications. So this is a noticable improvement and the difference will widen as n increases.

There are actually other integer multiplication algorithms with even faster running times, e.g. Schoöhage-Strassen's method takes $O(n \log n \log \log n)$ time. But these methods are more involved.