0.1 Worst and best case analysis

Last time we gave the formal definitions of O, Ω, and Θ. Today I will elaborate a little bit more on how these relate to algorithms, and also how it relates to whether something is the worst case time complexity or the best case time complexity.

It is easy to think big-O complexity means the same thing as "worst case time complexity" and big-Ω complexity means the same thing as "best case time complexity." Actually, those two things are completely separate.

Worst case vs. best case: Your algorithm has different runtimes on different inputs, and a lot of times we want to know how fast the algorithm runs on an input of a certain size. Worst case runtime means that you are feeding the worst possible input (of that size) into your algorithm. Best case runtime means that you are feeding the best possible input into your algorithm. For an input of size n, perhaps the worst case runtime is $T(n) = 2n^2 + 5$, and the best case runtime is $3n$.

Big-O vs. Big-Omega: This refers to a way of bounding complicated functions by a simpler function, so it is easier to work with them. For example, the worst case runtime $T(n) = 2n^2 + 5$ can be bounded from below by a constant times n^2, so we say that $T(n)$ is $\Omega(n^2)$. It can also be bounded from above by a constant times n^2 (for sufficiently large values of n), so we say that $T(n)$ is $O(n^2)$ as well.

Similarly, the best case runtime $T(n) = 3n$ can be bounded both below and above by a constant times n, so we say that the best case runtime is $O(n)$, $\Omega(n)$, and $\Theta(n)$.

You can even bound things that are not runtimes of algorithms. For example, the function $f(x) = x^5 + 4x^4 + 3x^3 + 2x \log x$ is $\Theta(x^5)$, even though f is not the runtime of an algorithm, and x does not correspond to an input size.

This is really what I should have said in class, but instead I ended up saying something about a clock. In retrospect it was not a very good example.

0.2 Divide and conquer algorithms

Correctness proofs follow design patterns – if you see one correctness proof, it will help you write similar correctness proofs. Algorithms also follow design patterns. You have your "incremental algorithms," where you build the solution one step at a time. (Insertion sort was a good example of this, where we started with only the first element being sorted, and then incrementally added to the sorted part of the array on every iteration.) We also have greedy algorithms, and dynamic programming algorithms (we'll explain what those are in a few weeks). And today we'll be covering a broad class of algorithms called divide-and-conquer algorithms.

Divide and conquer algorithms have two steps:

1. Break the problem up into a series of smaller problems that are exactly like the original problem

2. Combine the answers to the subproblems to produce an answer to the larger problem.

Before we go into full-blown divide and conquer mode, first I want to show you a recursive algorithm for raising a number to an integer power. Then I will show you merge sort, which is a classic example of a divide and conquer algorithm.

0.3 Exponentiation via repeated squaring

Suppose we want to find 3^n for some nonnegative integer n. The naive way to do it is using a for loop:

```
answer = 1
for i = 1 to n:
    answer = answer * 3
return answer
```

This runs in $O(n)$ time, since the body of the for loop takes $O(1)$ time to execute, and the for loop test is executed $n+1$ times. (Like the algorithm in the previous lecture, this is $n+1$ instead of n because the for loop test also gets executed on the iteration where we break out of the loop.)

However, we can get a faster algorithm if we write this recursively. By that I mean we can write a function that calls itself.

```
Exponentiator(n):
    if n = 0:
        return 1
    else if n mod 2 = 0: (i.e. n is even)
        x = Exponentiator(n / 2)
        return x * x
    else: (if n is odd)
        x = Exponentiator((n - 1) / 2)
        return 3 * x * x
```

Note that it is important to separate this calculation into even and odd cases so that the input to Exponentiator is always an integer. If this algorithm were extended to fractional inputs, it might never terminate.

0.3.1 Correctness

We can prove the correctness of this using strong induction.

Base case: Suppose $n = 0$. Then Exponentiator returns 1, which is correct, since $3^0 = 1$.

Inductive step: Suppose Exponentiator(k) produces correct output for all integers $k < n$ (this is the inductive hypothesis). Then we must prove that Exponentiator(n) produces correct output, i.e. that it returns 3^n.

Suppose n is even. Then the algorithm returns x^2, where $x = \text{Exponentiator}(n/2)$. By the inductive hypothesis, $x = 3^{n/2}$. So the algorithm returns $(3^{n/2})^2 = 3^n$.

Suppose n is odd. Then $x = \text{Exponentiator}((n-1)/2) = 3^{(n-1)/2}$. The algorithm returns $3x^2 = 3 \cdot (3^{(n-1)/2})^2 = 3 \cdot 3^{n-1} = 3^n$.

This shows that Exponentiator produces correct output for all integers $n \geq 0$. (Note that this induction proof wouldn't work if n could be any real number.)

0.3.2 Runtime

Let $T(n)$ be the runtime of Exponentiator on input n. Then we can express the runtime as a recurrence relation:

$$T(n) = \begin{cases} c_1 & \text{for } n = 0 \\ T(n/2) + c_2 & \text{for } n \text{ even} \\ T((n-1)/2) + c_3 & \text{for } n \text{ odd} \end{cases}$$

Observe that if the pseudocode had said "return Exponentiator(n/2) * Exponentiator(n/2)" instead of "x = Exponentiator(n/2); return x * x", the runtime would be substantially worse, since we would have to make two calls to Exponentiator every time.

0.4 Merge sort

Merge sort is a divide and conquer algorithm for sorting arrays. To sort an array, first you split it into two arrays of roughly equal size. Then sort each of those arrays using merge sort, and merge the two sorted arrays.

Example: Suppose you are sorting $A = [5, 2, 4, 7, 1, 3, 2, 6]$. We divide this into the arrays $[5, 2, 4, 7]$ and $[1, 3, 2, 6]$, and MergeSort each of those arrays. To sort $[5, 2, 4, 7]$, we divide it into the arrays $[5, 2]$ and $[4, 7]$, and MergeSort each of those arrays. To sort $[5, 2]$, we divide it into the arrays $[5]$ and $[2]$. At this point we have reached the base case, so MergeSorting the array $[5]$ just returns the array $[5]$, and MergeSorting the array $[2]$ just returns the array $[2]$. But now we need to merge together $[5]$ and $[2]$, which gives us $[2, 5]$. Merging together $[2, 5]$ and $[4, 7]$ gives us $[2, 4, 5, 7]$. Finally, merging together $[2, 4, 5, 7]$ and $[1, 2, 3, 6]$ gives us $[1, 2, 2, 3, 4, 5, 6, 7]$.

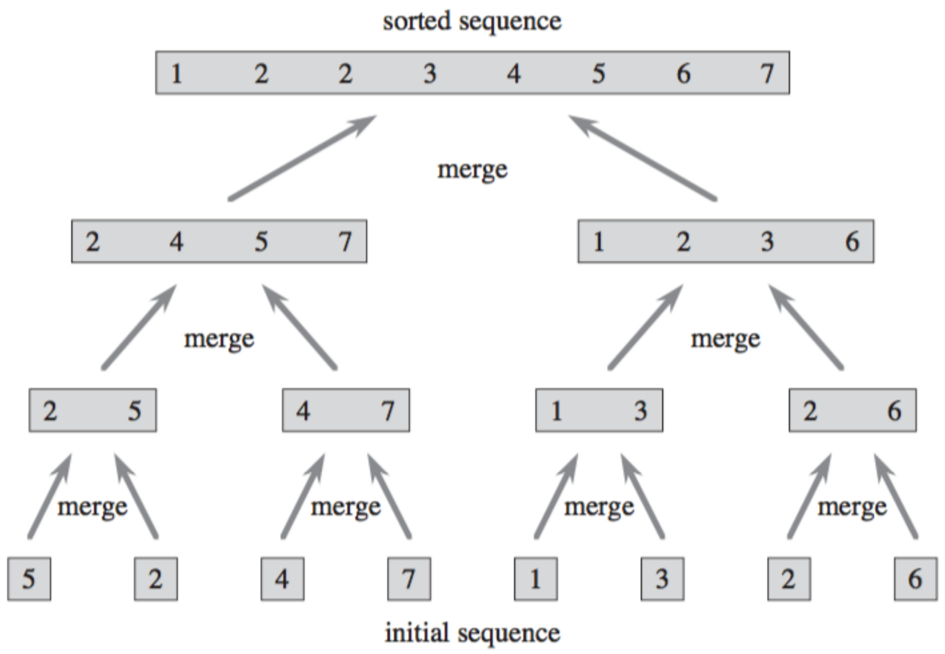

Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Here is the pseudocode for Merge Sort. It's not the same code that's in the book (actually it's from last quarter's lecture notes). The book version is on page 31 and 34, and the main difference is that the book version is more formal, and modifies the existing array instead of creating a new one.

```
MergeSort(A):
    n = length(A)
    if n <= 1:
        return A
    L = MergeSort(A[1 .. floor(n / 2)])
    R = MergeSort(A[(floor(n/2) + 1) .. n])
    return Merge(L, R)
```

Now here's how to do a merge. Assume you are given two sorted arrays, L and R, and you want to produce a sorted array that has all the elements in both L and R.

Then you keep a pointer to the first element of each array. You know that the first element of the final array will either be the first element of L or the first element of R. So you choose the lowest one, and copy it to the result array. Then you increment the corresponding pointer so it points to the second element in that array instead of the first.

Now say we picked from array L in the first step. Then the second-lowest element in the result array either has to be the second element of L, or the first element of R. So once again, we choose the lowest one, and copy it to the result array, and increment the corresponding pointer. We keep doing this until we run out of elements in both the arrays.

```
Merge(L, R):
    m = length(L) + length(R)
    S = empty array of size m
    i = 1; j = 1
    for k = 1 to m:
        if L[i] <= R[j]:
            S[k] = L[i]
            i = i + 1
        else: (L[i] > R[j])
            S[k] = R[j]
            j = j + 1
    return S
```

The astute observer may ask what happens if one of the arrays gets depleted first. This code, as written, would result in a runtime exception, as we try to access an element that is outside the bounds of the array. In order to fix this, we add the "sentinel element" ∞ to the end of both arrays. ∞ will always be larger than any element that is actually in the array, so this will make us always pick from the other array after the first array has been depleted.

Example: Suppose we are merging $L = [2, 4, 5, 7]$ with $R = [1, 2, 3, 6, 9, 11]$. We start with $i = 1$ and $j = 1$. Observing that $R[1] < L[1]$, we let $S[1] = 1$ be the first element of the result array, and set $j = 2$. Now $L[1] \leq R[2]$, so we set $S[2] = 2$, and $i = 2$. Now $R[2] < L[2]$, so $S[3] = 2$, and $j = 3$, etc.

0.5 Runtime analysis

First we will analyze the runtime of the Merge subroutine. The first and third lines take constant time. Assume the second line, initializing the array S, takes time proportional to m. The body of the for loop takes a constant amount of time to execute, and the body gets executed m times. So the entire subroutine just takes $c_1 + c_2 m + c_3 + c_4 m$, which we will just write as $O(m)$.

Now we will analyze the runtime of MergeSort. To simplify our analysis, we will assume that the length of the array is a power of 2. Under those simplifying assumptions, the MergeSort runtime for an array of length n can be written as $T(n) = 2T(n/2) + \Theta(n)$ (for $n > 1$), and $T(n) = \Theta(1)$ (for $n = 1$). The $2T(n/2)$ term comes from the fact that we are making two sub-calls to MergeSort, each on arrays of length $n/2$. The $\Theta(n)$ is the time it takes to merge the arrays. There are also some constant terms arising from line 1 and line 2, but those constants get overwhelmed by the $\Theta(n)$ term, so we can drop them.

0.6 Proof of correctness

0.6.1 Proof of Merge

First we will prove the correctness of the Merge subroutine. The loop invariant we will use is

Loop invariant: At the start of each iteration k of the for loop, the nonempty part of S contains the $k-1$ smallest elements of L and R, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied to S.

To use this loop invariant, we must prove the three loop invariant conditions.

Initialization: The loop invariant holds prior to the first iteration of the loop. Here, $i = j = 1$, and S is completely empty. $L[1]$ is the smallest element of L, while $R[1]$ is the smallest element of R, so the initialization step holds.

Maintenance: To see that each iteration maintains the loop invariant, suppose without loss of generality that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied to S. The current nonempty part of S consists of the $k-1$ smallest elements, so after the loop is over and $L[i]$ is copied to S, the nonempty part of S will consist of the k smallest elements. Incrementing k (in the for loop update) and i reestablishes the loop invariant for the next iteration.

Termination: At termination, $k = m+1$. By the loop invariant, S contains the m smallest elements of L and R, in sorted order. This is the result that we wanted (i.e. the merging of the two sorted arrays to produce a new sorted array).

0.6.2 Proof of MergeSort

Here we use strong induction to prove that MergeSort correctly sorts all arrays A of size n.

Base case: Suppose $n = 1$, which means the array A has one element. Then MergeSort just returns A, which is sorted.

Inductive step: Suppose MergeSort correctly sorts all arrays of size less than n. Then the first half and second half of the array are both arrays of size less than n. So line 4 correctly sorts the first half of the array to produce L, and line 5 correctly sorts the second half of the array to produce R. Since L and R are now sorted, and we have already proven that the Merge subroutine is correct, MergeSort returns the sorted array.