Recitation 2

1 Topics

- Recurrences
- Binary Search
- Python Costs
- Document Distance

2 Recurrences

It is often the case that we write algorithms that divide the given problem into subproblems, solve these subproblems, then combine these solutions to arrive at our final solution. This paradigm is called *Divide and Conquer*. It is convenient to analyze the complexity of such algorithms using recurrences.

2.1 Structure of a Recurrence

In short, a recurrence is a function that is defined in terms of itself. Recurrences have two types of terms: those related to the work required to solve subproblems and those related to the work to divide the problem into subproblems and to later combine these solutions. For example, a common recurrence is

$$T(n) = 2T(\frac{n}{2}) + \Theta(n).$$

What this is saying is that to solve a problem of size n, we must solve 2 subproblems of size $\frac{n}{2}$ and we can divide the problem in half and later combine the solutions in linear time.

2.2 Solving Recurrences

To solve a recurrence means to relate the asymptotic complexity of the given recurrence to a simple, closed form function. In other words, we want to remove the $T(...)$ terms from the right hand side. For example the solution to the above recurrence is

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n\log(n)).$$

We can solve this using the recursion tree method.

Tree Method

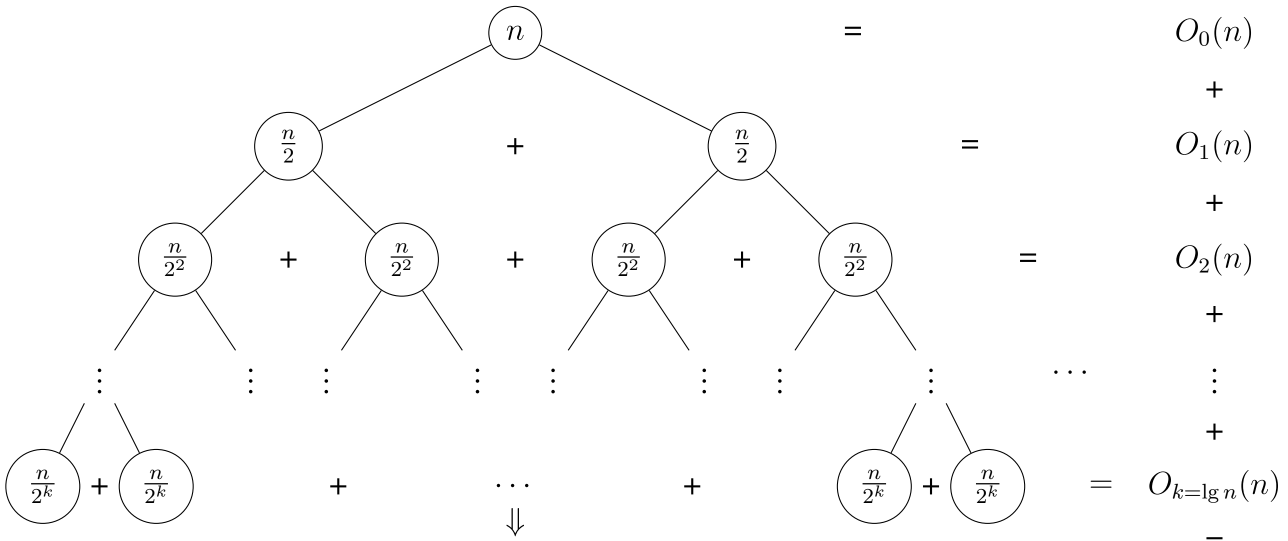
One way to solve recurrences is to draw a recursion tree where each node in the tree represents a subproblem and the value at each node represents the amount of work spent at each subproblem. The root node represents the original problem. In a recursion tree, every node that is not a leaf has a children, representing the number of subproblems it is splitting into. To figure out how much work is being spent at each subproblem, first find the size of the subproblem with the help of b, then substitute the size of the subproblem in the recurrence formula $T(n)$, then take the value of $f(n)$ as the amount of work spent at that subproblem. Long story short, a node with a problem size of x, the node will have a children each contributing $f(x/b)$ amount of work.

The work at the leaves is $T(1)$, since at that point we have divided the original problem up until it can no longer be further divided. Note that this means that the work contributed by the leaves are $O(1)$.

Once we have our tree, the total runtime can be calculated by summing up the work contributed by all of the nodes. We can do this by summing up the work at each level of the tree, then summing up the levels of the tree.

Example: $T(n) = 2T(n/2) + O(n)$ **Recursion Tree**

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


n
$=$
$O_0(n)$
$+$
$\frac{n}{2}$ + $\frac{n}{2}$
$=$
$O_1(n)$
$+$
$\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$
$=$
$O_2(n)$
$+$
$\vdots$
$+$
$\frac{n}{2^k}$ + $\frac{n}{2^k}$ + $\cdots$ + $\frac{n}{2^k}$ + $\frac{n}{2^k}$
$=$
$O_{k=\lg n}(n)$
$=$
$\Downarrow$

$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) \quad = \quad O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \quad \Leftrightarrow \quad O(n \cdot \log n)$$

This recursion tree is considered somewhat balanced. The work done at each level stays consistent (in this case, it is $O(n)$ at each level) so the total work done can be calculated by multiplying the work at each level by the number of levels, hence $O(n \log n)$ running time for merge sort. However, there are two other cases that may happen. If the work at each level geometrically decreases as we go down the tree, then the work done at the root node (i.e. $f(n)$) will dominate the runtime. If instead the work at each level geometrically increases as we go down the tree, then the work done at the bottom level (i.e. number of leaves $\times f(1)$ or O(number of leaves) or $O(n^{\log_b a})$) will dominate the runtime.

3 Fibonacci*

The Fibonacci Sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... The next number is found by adding up the two numbers before it. I.e. 3 is found by adding the two numbers before it (1+2). Here we will explore 3 different algorithms for computing the n^{th} Fibonacci number and analyze their time complexity. We denote the n^{th} Fibonacci number as F_n. Code for the following 3 algorithms is in recitation1.py which is available on the Stellar site under recitation materials.

3.1 Naive Recursion*

By definition, $F_n = F_{n-1} + F_{n-2}$. As this is the "naive" algorithm, let's not try to be too clever and instead simply write an algorithm using only this definition! See recitation1.py for python code for this algorithm.

Now to analyze the runtime. Formally this algorithm can be analyzed by solving the recurrence, $T(n) = T(n-1) + T(n-2) + \Theta(1)$. This is a tough recursion to solve! Let us separately find an upper and lower bound instead of a Θ relation.

It is clear that the recurrence $T(n) = 2T(n-1) + \Theta(1)$ is strictly greater than our original, so let us use it to find an upper bound. Each recursive call results in two child recursive calls until the base case is reached. Therefore, there will be $\Theta(2^i)$ recursive calls made at the i^{th} level of recursion. Since, the subproblem size only decreases by one on each call, there will be $\Theta(n)$ levels of recursion before the base case is reached. Therefore this recurrence solves to be $\Theta(2^n)$ and we can conclude that our algorithm is $O(2^n)$

The recurrence $T(n) = 2T(n-2) + \Theta(1)$ is strictly less than our original. Using similar logic as above we can see that this recurrence solves to $\Theta(2^{\frac{n}{2}})$ and we conclude that our algorithm is $\Omega(2^{\frac{n}{2}})$.

Challenge Problem: Find a tight asymptotic bound to this algorithms runtime. Hint: Draw a tree diagraming recursive calls and look for the pattern!

3.2 Memoized Recursion (Advanced!)*

It's often the case that we can improve the efficiency of algorithms by exploiting natural "structures" present in the problem. Notice in the naive algorithm that we often compute the same thing multiple times! This occurs because we have overlapping subproblems. For example, both F_{n-1} and F_{n-2} depend on the solution to F_{n-3}. We can take advantage of this structure by memoizing (storing) the solutions to subproblems as we go. Therefore instead of recalculating them we can simply look them up! Look in recitation1.py for python code.

This improved algorithm has a time complexity of $\Theta(n)$. This can be seen from the fact that we in total solve for $\Theta(n)$ F_is, each of which take only constant time to compute.

3.3 Repeated Squaring of Matrix

Take a moment to think back to the recursive squaring algorithm from lecture. In a similar fashion, we can compute the n^{th} Fibonacci number in logarithmic time by repeatedly squaring the matrix $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. Look in recitation1.py for python code for this algorithm.

In fact, $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$. To give a rough proof of why this is the case, let us use induction on n. Our claim is trivially true in the base case $n = 1$. Now assuming that our claim holds for this matrix to the n^{th} power, we must show that our claim is also true for this matrix to the $(n+1)^{th}$ power.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n+1} + F_n & F_n + F_{n-1} \\ F_{n+1} & F_n \end{bmatrix} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix}$$

Success!

The runtime analysis for this algorithm is identical to that for modular exponentiation using repeated squaring.

4 Binary Search

4.1 Motivation and Algorithm Description

Many of you may have already encountered binary search, if not in class then in your everyday lives. Consider the following game where your opponent is thinking of an integer number between 1 and 100. How would you, in the fewest number of guesses, figure out which number she was thinking of? You'd likely start by asking if the number is greater than or less than 50. If it were greater, you might continue this strategy, asking if the number is greater than or less than 75 and splitting all subsequent intervals until you narrow it down to a single value. With an ever so slight alteration in the problem statement, this is binary search.

Binary search is defined as follows: Given a sorted array, find the index of a target element or, if the element is not present in the array, return False.

The fundamental idea behind binary search is to split the array in two at each step. For each iteration of the algorithm, we ask if or target value is greater than or less than the middle element in our array. If the target is equal to the middle, we have found it and know its index. If the target is smaller than the midpoint, we recurse on the part of the array that ends at index middle-1, asking how the subarray's midpoint compares to the target. We do a similar process for the case where the target is larger than the midpoint. A python implementation of the algorithm follows.

4.1.1 Pseudocode (for reference)*

```
#This implementation returns the index of the found element
#If no such element is found, returns an index of -1

def binary_search(sorted_array,target):
    start = 0
    end = len(sorted_array)-1

    while start<=end:
        middle = (start+end)/2
        mid_value = sorted_array[middle]
        if mid_value == target:
            return middle
        else:
            if mid_value < target:
                start = middle + 1
            else:
                end = middle - 1

    #If nothing found, default to None
    return None
```

4.2 Timing Analysis

We can formulate binary search in a recursive way as follows. Suppose at some step we are performing binary search on an array of size n. At this step the algorithm will access the middle of the array and do a comparison between the middle number and the target value. Both of these operations are constant, that is $\Theta(1)$ time. We then continue this procedure on half the array with size $\frac{n}{2}$ until we reach the base case $T(1) = 1$ (checking a single value against the target takes constant time). Translating this into the language of a recursive formula, we obtain:

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

At each stage of the algorithm, we do $\Theta(1)$ work. Furthermore, since at each stage of the algorithm our input shrinks by a factor of 2, it takes $\Theta(\log n)$ steps for n to shrink to the base case of $T(1) = 1$. Thus, $\Theta(1)$ work at $\Theta(\log n)$ steps leads to a total running time of $\Theta(\log n)$.

4.3 Implementation Caveats*

Binary search in practice is simple to implement but deceptively tricky to get right. A commonly tossed-around factoid is that only 10% of programmers can actually write this simple procedure without bugs in edge-cases.

How do I handle recursion? How do I handle splitting odd-length arrays? When do I stop and ensure I don't accidentally skip a value? There are a lot of salient questions one has to keep track of to get the algorithm right. Pay extra close attention to how the provided algorithm handles setting its bounds. Also note that though we analyzed the time in terms of a recursion, the whole algorithm exists inside a single function call. This way we need not worry about python's stack limit or memory issues.

Furthermore, and more importantly, notice that **WE IMPLEMENTED THE "RECURSION" BY KEEPING TRACK OF JUST TWO POINTERS**, one for the beginning of the array and one for the end. We didn't recurse by calling something like $binary_search(sorted_array[0:n/2], 42)$, that is passing a slice of the array. Array slicing in python is essentially copying the array at the indices specified, taking $\Theta(n)$ work, and leading to a recursion of

$$T(n) = T(\frac{n}{2}) + \Theta(n) \tag{2}$$

We leave solving this recurrence to the reader as an exercise.

TL;DR - If using recursion on matrices, use pointers instead of array slicing and be careful about where you put your pointers.

5 Python Cost Model

All the information you need concerning the python cost model can be found on the stellar website under the python section of materials or at

`http://scripts.mit.edu/~6.006/fall08/wiki/index.php?title=Python_Cost_Model`

Some key takeaways from the page are . . .

- Addition, multiplication etc. scale linearly in the number of digits (or as the log n) of numbers, n,m. For our purposes in the course, such operations can be considered constant.
- Taking slices of lists is linear in the length of the input list, so if you're recursing inside of a pre-existing list, just keep track of the range of indices instead. ($O(n)$ vs $O(1)$ work)
- If you don't need to maintain an existing list, then for two lists, L and M, do L.extend(M) instead of L+M. The first takes O(M) time while the latter takes O(L+M) time.
- Don't delete from the beginning of a list. Every element has to be shifted, leading to this operation taking O(n) time. Consider using collections.deque if deleting from both ends is necessary.
- Dictionary access and assignment times are constant, so consider using them if you want to fetch unordered data quickly.

6 Document Distance

In this section, we analyze an algorithm for calculating the distance between two documents in terms of the angle between them. In other words, we try to see how much overlap these documents have in terms of their constituent words with no regard to their order. For example, [”Steve”,”is”,”pretty”,”cool”] is quite close to [”Steve”,”is”,”pretty”,”awesome”] but an exact match for [”is”,”steve”,”pretty”,”cool”]. This algorithm has applications in search engines, where we match documents based on the document distance between a query and a stored document.

In this section we walk through the algorithm line-by-line. See the ”Document Distance” handout for a full description of the algorithm and iterative improvements we make on it. The improvements lead to the following. Note that $\Theta(L)$ denotes time linear in the length of an array L.

1. $1 \rightarrow 2$ Adding two arrays, L, M lead to an $\Theta(L + M)$ time operation, which is particularly bad if we iteratively add a small M to the end of L this way. Using $L.extend(M)$ leads to a single $\Theta(M)$ operation.

2. $2 \rightarrow 3$ A major issue with our original dot product is we wasted a lot of time searching for matches of words between the two documents. If we first sort, then we can simply run through the lists in a $O(max(L, M))$ fashion as opposed to in $O(LM)$ time. Unfortunately, insertion sort is an $O(L^2)$ algorithm and leads to other inefficiencies. Instead, we could use the native Python sorted module which is an $O(L \log L)$ algorithm.

3. $3 \rightarrow 4$ When we are counting the number of word occurrences, we are iterating through the entire containing array to update a word’s count. That is, for each word update, we spend $\Theta(L)$ time just to find the word. Thus, if there are a total of n words in a document with L unique words, keeping track of the frequencies this way takes $\Theta(n * L)$ time. Using a dictionary, we can instead find a single word in $\Theta(1)$ time, and enumerate the frequency of every word in $\Theta(n)$ time.