Data Structures and Algorithms

Lecture #1

Topics : Recursion, Time Complexity

Course Logistics

1. There will be a couple of programming assignments which is to be held on a2oj and spoj.Make sure that you have registered on these sites and linked your account. The details of the assignment can be found on the slack channel (in the pinned post). In case you haven't been added to the workspace yet, use this link to register.

2. If you need any clarifications on the programming assignments or if you have some doubts regarding the lectures, you can ask on the slack channel.

3. The lecture notes would be uploaded regularly on Github. You can find the link on slack (under the pinned post in the channel).

4. If you have time, try to code the practice problems too. If not, you can just write the pseudocode and move on for now. In any case, make sure that you are solving the assignments regularly otherwise you'd just be wasting your time (and mine). If you just want to attend the lectures and not put in any effort from your side, then you're better off auditing ESO207 or any other online course. It doesn't matter how bad or good you performed in your last course. All that matters is the effort that you are willing to put in at this moment.

Recursion

"To understand recursion, you must first understand recursion"

Key Steps

1. **Function Definition** : Identify the precise definition and function signature. *Be a bit creative.*
2. **Trust the Magic Box** : Break it into sub problems. Assume that the answers to the sub problems are **absolutely** correct.
3. **Responsibility** : Ensure that you return the correct answer to your caller. If not, your sub problems would *misbehave*
4. **Base Cases** : Handle the base cases (by looking at the function definition).
5. **Unravelling the Magic / Tracing the Recursion** : This is the key to understanding how recursion works. Start out by drawing the recursion tree and simulate the process manually until you are convinced that there is no magic involved.
6. **Debugging** : Test your code on smaller cases and prove the result via induction. $n = 0, 1, 2, 3, 4$ suffices in a general setting. Never assume that your code will work for bigger test cases but fail for smaller ones. [The Bigger answers depend on smaller ones]

Example 1 : Subset Sum Problem

Problem Statement : You are given an array $A[1, ...N]$, of n *positive* integers (not necessarily distinct). You are also given a number k. You need to find out whether there exists a subset of the n elements with sum equal to k.

1. Note that you are not asked to print any (or all) subsets.
2. You are also not required to print the total number of such subsets. You just need to return *true* or *false* according to the constraints.

Solution := First, let us try to find out the number of subsets possible out of n elements. In our subset, we can choose x elements, where $x = 0, 1, 2, ...N$. Hence, the number of choices are

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n}$$

If you know a little bit of combinatorics, you'll realize that the sum is nothing but 2^n. Let's look at it from a different perspective, i.e let's focus on the objects instead of the boxes. For each element, we have 2 choices, either we can include it in our final set or we can ignore it. Hence, the total number of ways is $2 * 2 * \cdots$ (n times) which is nothing but 2^n. The recursive solution to the problem hinges at this observation. We'll try to simulate the choices of taking an element or leaving it.

1. **Function Definition** := The most important part of recursion is the precise definition. What do we want our function to do? One choice is to convert the given problem statement into a definition. But, it's not always useful . (We'll see later why). So, let us say that function **subset_sum(arr,n,k)** returns true if *There is a subset of the first n elements which sums upto k.* Notice the **first n elements** in the definition. Clearly, our answer remains the same even after modifying the definition a bit.

2. **Trust the Magic Box** := What if I told you that there is a magical box which would give you the correct result that you intend from the function for any reduced input size. However, *There is a catch.* **The magic box would only work correctly if after using it you return the correct answer from your side**. You are allowed to use the box any number of times.

   Let us understand what do we mean by reduced input size. The parameters of the function are *Arr, n, k.* It is easy to see that the array is an invariant, i.e, it is not modified with time. However, we could change *n, k* if we wanted to. Hence, these are counted as input parameters.

   Let us simulate the choices for the last element. We have 2 choices, either we can take it in our final set, or leave it. Suppose we decide to take it. It means that we need to form the remaining sum from the remaining elements. Going by the definition, we need to find out whether there exists a subset of the first $n-1$ elements with sum as $n - lastElement$. This is nothing but **subset_sum(arr, n-1 ,k - lastElement)**. Notice that input size has reduced and we can assume that the magical box gives us the correct answer for this query.

   Now, we simulate the second choice. We ignore the last element. This means that we need to form the entire sum from the remaining elements. This is nothing but **subset_sum(arr, n-1 ,k)**. Again, notice that the input size has reduced and hence we can get the correct answer for our query.

   Did we miss any choice for the last element? No.

3. **Responsibility** := As mentioned earlier, after using the subproblems, it is your responsibility to return the correct answer from your side. It is not difficult to see that if the answer of either of the 2 queries is *true*, we can return *true* from our side, else we return *false.*

4. **Base Cases** := Moving on to the base cases, we notice that the variable inputs are n and k. And they tend to decrease at each subsequent call. Let's deal with k first. What if $k < 0$? First of all, let's ask ourselves if this case even possible? I mean, it is clearly mentioned that all the numbers are positive, then why would the user ask a negative query? The answer is that the usesr might not ask a negative query but you might? What does $k < 0$ signify? It simply means that at the previous choice, you took an element with larger value and hence you cannot proceed with it now. Hence, we return false.

   What if $k == 0$. According to the function definitons, it is clear that we can form a subset with sum 0, i.e the empty set. Hence, we return true.

   Now we can assume that $k > 0$.

   What about n. Can n be negative? No. What if $n == 0$. At this stage, we need to form the positive sum k with first 0 elements. Clearly, this is not possible. Hence, we return false.

5. **Unravelling the Magic / Tracing the Recursion** := If we notice carefully, we have solved the entire problem by actually following some simple rules. But where is that magic happening. For that, you need to draw the recursion tree taught in the class and simulate the recursion. [If time permits, I'll try to upload the recursion tree].

6. **Debugging** := Once you are convinced the recursion tree way, you won't find difficulties in debugging. Just convince yourself that it works for all inputs of size 1,2,3 and the rest would follow.

Algorithm 1 Find whether there exists a subset with sum k

Require: An array, an integer n, and an integer k

```
 1: function SUBSET_SUM($arr, N, K$)
 2:     if k <0 then
 3:         return False
 4:     if k == 0 then
 5:         return True
 6:     if n == 0 and k!=0 then
 7:         return False
 8:     $Last\_Element \leftarrow A[N]$                ▷ One Based Indexing
 9:     $Take\_it \leftarrow Subset\_Sum(arr, n-1, k - last\_element)$
10:     $Leave\_it \leftarrow Subset\_Sum(arr, n-1, k)$
11:     return $take\_it$ || $leave\_it$
```

Don't Let Recursion Fool You

Looking at the ease with which recursion solves your problem, you might be tempted to apply it in every question. However, it doesn't always work out. Remember, We need to be a bit creative in the function definition. Here's a tricky question which demonstrates this fact.

The Equal Partition Problem

Problem : Suppose you are given an array $A[1, \cdots N]$ and you need to determine whether you can partition A into 2 subsets such that the sum of both the subsets is equal. (Every element should be in exactly one of the subset). You just need to return *True* or *False*

Solution : As I said in the class, the simple recursive definition won't work here. Why? Simply because after using the magical box (the sub-problems), you aren't able to fulfill your responsibility. Hence, the subproblems would misbehave. This is because knowing the answer for the first $(n-1)$ elements via recursion does not give us a way to calculate the answer for the first n elements. Clearly, we need to think of something else.

Notice that if we play around with the function definition, the above function can be converted to a known instance of an easy problem. First, let us find out the sum of all the elements and call this *Total_Sum*. If the total sum is odd, then there is no way we can partition it into 2 subsets with equal sum (as it would violate the condiion that sum is odd). So, if the sum is odd, we can simply return false.

Now, if the total sum is even, we find half of the total sum and call it *Half_Sum*. As you can see, if we can somehow find a subset with sum as *Half_Sum*, then the sum of the elements which are not in the set would be *Total_Sum* - *Half_Sum* = *Half_Sum*. Hence, we can clearly see that if there is a subset with sum as *Half_Sum*, we can easily partition it into 2 sets of equal sum. All that remains is to find this subset's existence. And that is what the **Subset_Sum** problem does. Hence, we have arrived at a simple solution with very less effort from our side.

Algorithm 2 Find whether a set can be partitioned into 2 sets with equal sum

Require: An array, an integer n

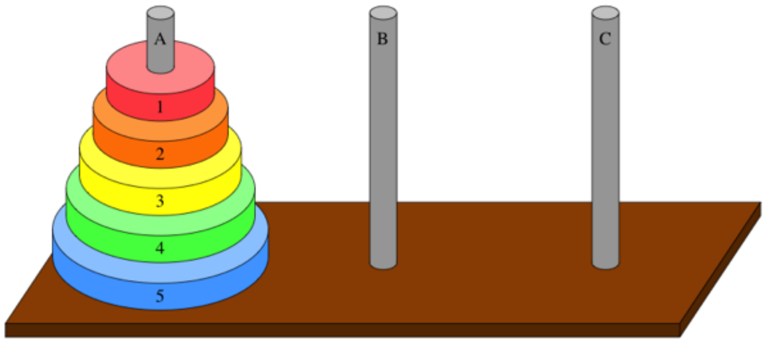
```
1: function EQUAL_PARTITION($arr, N$)
2:     $Total\_sum \leftarrow \sum a_i$
3:     if $Total\_Sum$ is odd then
4:         return False
5:     $Half\_sum \leftarrow \sum \frac{Total\_Sum}{2}$
6:     return $Subset\_Sum(arr, n, Half\_Sum)$
```

Example 2 : Tower of Hanoi

Problem Statement : You are given n discs lying on a *source* tower. You are also given 2 extra towers, namely *aux*, and *target*. The goal of this game is to move all the discs to the *target* tower using the *aux* tower as a buffer. The rules are as follows -

1. You can only move one disc at a time.
2. At any step, you cannot place a bigger disc on top of a smaller disc.

Note that the size of the disc is directly proportional to the number written on it. Hence, the biggest disc is numbered n and is lying at bottom of *source* initially while the smallest disc, with the number 1 is lying at the top of the *source* tower.



Here, A is the *source* tower, B is the *Aux* tower, while C is the *Target* tower
Develop a recursive method to solve to solve this problem.
Hint : Keep the biggest disk as it is and try to move the smaller disks via recursion to the auxillary tower. Then, move the biggest disk to its correct position and try to move all the disks on the auxillary tower to the target tower via recursion. Write proper pseudocode and understand its working.

Time Complexity

The Big Deal about log

You'll often encounter log showing up in a lot of major algorithms. Let us figure out what is the source of this intrusion. Let us start off with a big number, say, 2^{32}. If you know how fast exponential grows, by now you must've guessed that this number is pretty big. If you aren't able to visualize it, let's get a rough estimate of this number.

$$2^{32} = 2^{4*8} = 2^{4^8} = 16^8 < 10^8$$

So, as we can see, the number has at least 9 digits. Monetarily, it is bigger than *10 crores*, which is a pretty huge amount.

Now, what if we keep dividing the number by 2 until we hit 1. How many steps would that take? Exponentials grow fast, and hence on dividing they fall pretty fast too. Let us say we have performed k divisons to reach 1. The resulting number would be $n/2^k$

$$n/2^k = 1 \implies n = 2^k \implies \log_2 n = \log_2 2^k$$
$$k = log_2 n$$

So, we see that we have reduced the number to 1 in just 32 steps.

Now, let us try to find the value of $nlog(n)$. Clearly, it is $32 * 2^{32}$ Next, we try to find the value of n^2. It is 2^{64}. How much has it grown? Let us analyze the difference

$$2^{64} - 2^{32} = 2^3 2 2^3 2 - 1 \approx 2^{32} * 2^3 2 \approx 2^{64}$$

So, the growth is so fast, that the initial value has become negligible in comparison. They say that the difference between a million and a billion is roughly a billion. Now you can see why.

Finally, let us analyze 2^n. It is equal to $2^{2^{32}}$ which is pretty huge if you look at how fast exponentials grow.

This difference in the growth rate of functions is the main motivation to study different data structures and algorithms. Suppose that these values, $\log n, n, n^2, 2^n$ represents the number of hours that your computer takes to execute those algorithms. Now, you can visualize the vast difference in the runtime.

Note : Even if n was not a power of 2, the above analysis in the differnce would still be valid.

The Big O Notation

First, we introduce the O notation (pronounced as **Big Oh**). A function $f(n)$ is said to be $O(g(n))$ if

$$f(n) \leq cg(n) \quad \forall n > n_0$$

Although the formula might look quite obfuscated, the intuition is extremely simple. In short, we want $f(n)$ to be bounded above by a constant multiple of $g(n)$ for large n.

Examples

1. Is a quadratic equation $an^2 + bn + c = O(n^2)$?
   We notice that the growth of the lower order terms such as bn and c are shadowed by the leading coefficient. Hence, it is the highest order term that decides the behaviour in which the function grows for large n. So, we can simply drop the lower order terms.

   Now, we can easily see that we can bound an^2 by kn^2, where k is a positive constant greater than $|a|$. Hence, any quadratic function is $O(n^2)$.

2. What about linear functions? Is it $O(n)$ or $O(n^2)$?
   First of all, any linear function, $an + b$ can be bounded above by kn where k is a positive function greater than $|a|$. So, it is definitely $O(n)$.
   However, the interesting part is that it is also $O(n^2)$ as the rate of growth of a linear function is always less than a quadratic function.
   Hence, any function which is $O(n^k)$ is also $O(n^{k+1})$.

Key Point := To check whether $f(n) = O(g(n))$, simply check whether $f(n)$ can be bounded above by a constant multiple of $g(n)$, as n approaches infinity. Hence, the name, **Asymptotic Upper Bound**

The Ω Notation

Next, we introduce the Ω notation. A function $f(n)$ is said to be $\Omega(g(n))$ if

$$f(n) \geq cg(n) \quad \forall n > n_0$$

In short, we want $f(n)$ to be bounded below by a constant multiple of $g(n)$ for large n.

Examples

1. Is a quadratic equation $an^2 + bn + c = \Omega(n^2)$?
   First, notice that n^2 dominates the lower order terms as n approaches infinity. Moreover, since $a > 0$, it implies that $an^2 \geq kn^2$, where k is constant greater than a. Hence, it is bounded below by n^2

2. What about linear functions? Is it $\Omega(n)$ or $\Omega(n^2)$?
   Any linear function $an + b$ is always greater than kn, where k is a constant greater than a.

   However, it is not $\Omega(n^2)$ as $n < n^2$ for $n > 1$

Consequently, any function which is $\Omega(n^k)$ is also $\Omega(n^{k-1})$.

Note := Please note that all the time complexity analysis of $f(x)$ is done on the assumption that $f(x) > 0$.

Key Point := To check whether $f(n) = \Omega(g(n))$, simply check whether $f(n)$ can be bounded below by a constant multiple of $g(n)$, as n approaches infinity. Hence, the name, **Asymptotic Lower Bound**

The Θ notation

Finally, we introduce the Θ notation. This is generally referred to as a **Tight Asymptotic Bound**. Stated mathematically, A function $f(n)$ is said to be $\Theta(g(n))$ if

$$ag(n) \leq f(n) \leq bg(n) \quad \forall n > n_0,$$

Here, a, b are some positive constants.

Simply stated, it means that the function $f(x)$ is sandwiched between constant multiples of $g(n)$ for sufficiently large n. Or in other words, it is bounded above and below by $g(n)$.

It is not to difficult to see that $f(n) = \Theta g(n)$ **iff** $f(n) = \Omega g(n)$ and $f(n) = O(g(n))$

A Slight Abuse of Notation := You will find the use of the O notation more frequent than the Ω and the Θ notation. It's true that we won't encounter Ω a lot. However, most of the analysis that you see in books / internet in terms of O notation is actually Θ notation. An example of this is that a linear function is $O(n^2)$ too, but we generally say that it $O(n)$. Hence, whenever you are required to find the time complexity of a program in Big Oh notation, the intended analysis is generally the Θ notation. We'll follow the same convention from now on.

Note := While analysing the time complexity of an algorithm, we are interested in the *worst case* complexity, i.e, we need an upper bound of the time complexity on all possible input.

Frequent Complexity Classes

Here are a few time complexity classes that you are going to encounter a lot.

1. *Constant* $:= O(1)$
2. *Logarithmic* $:= O(\log n)$
3. *Linear* $:= O(n)$
4. *?* $:= O(n \log n)$
5. *Quadratic, Cubic* $:= O(n^2), O(n^3)$
6. *Exponential* $:= O(2^n), O(a^n), O(na^n)$

Master Theorem

Suppose, we have a recurrence of the form

$$T(n) = aT(n/b) + \Theta(n^k \log^p n) \quad a \geq 1, b \geq 1, k \geq 0, p \epsilon \mathbb{R}$$

The solution to the above is

1. If $a > b^k$, then, $T(n) = \Theta(n^{\log_b a})$
2. If $a = b^k$
   (a) If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   (b) If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
   (c) If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$
3. If $a < b^k$
   (a) If $p > 0$, then $T(n) = \Theta(n^k \log^p n)$
   (b) If $p < 0$, then $T(n) = \Theta(n^k)$

Note := Although the *Master Theorem* encompasses almost all the recurrence relations that you'll see during the course (and afterwards), it's not necessary to memorize it. You should try to come up with the answer using the **Tree Method** or the **Pattern Method**. As you will see later, there are a couple of recurrence which keeps showing up at various places. After sometime, you can intuitively guess the solution without solving it (and verify).

Getting Acquainted with Time Complexity

Solve the following recurrences without using Master Theorem

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