CS 213 - Data Structures and Algorithms

Vishal Neeli

1 Recursion

1.1 Design principle

1. Reduce the problem of size n to a size n-1
2. Figure out the base case (usually n=0 or 1)
3. Terminate recursion at the base case. Make sure that every n reaches the base case.

1.2 Parameterization

- It is consumes extra memory if pass a new array (part of old array copied into this) for recursion. Instead we should pass the indices as the parameters.
- In creating recursive methods, it is often useful to define additional functions(or methods) to facilitate recursion.

1.3 Tail Recursion

- Recursion has to maintain function calls on stack that makes it more expensive than iterative methods.
- To reduce this extra usage of resources, we can write a algorithm in a tail recursive way i.e, the function should directly return the function call (without any other operations on the result returned by the function call).

2 Algorithm analysis

We primarily compare running time in this course. To analyse algorithms, runtime should be machine independent for which we use 'RAM' model of computation.

2.1 RAM model

- Generic single processor model
- Computer supports simple constant time instructions
  - Arithmetic $(+, -, \times, /, floor, ..)$
  - Data movement (load, store, copy)
  - Control (branch, function call)
- We assume that the cost (runtime) of all simple instructions is 1
- Sequential execution - No concurrent execution
- Flat memory model and accessing a memory costs 1 unit.

2.2 Asymptotic Notation

- Θ notation: Given functions g, we define

  $\Theta(g(n)) = \{f(n) : \exists$ positive constants c_1, c_2, n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$

  g(n) is a asymptotic tight bound for f(n).

- O notation: Given functions g, we define

  $$O(g(n)) = \{f(n) : \exists \text{ positive constants } c, n_0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0\}$$

  g(n) is a asymptotic upper bound for f(n).

- Ω notation: Given functions g, we define

  $$\Omega(g(n)) = \{f(n) : \exists \text{ positive constants } c, n_0 \text{ such that } cg(n) \leq f(n), \forall n \geq n_0\}$$

  g(n) is a asymptotic lower bound for f(n).

3 Divide and Conquer

- Divide and conquer algorithm is recursive in nature. Basically, we divide a problem into smaller problems (untill we hit the base case) and combine them to get the solution for larger problem.
- For the time complexity of a divide and conquer algorithm, we can write a recurrence relation

  $$T(n) = aT(\frac{n}{b}) + C(n) + D(n)$$

  where a = no of subproblems a problem is divided, b = factor by which n is reduced, C(n) and D(n) are the time taken in combine and divide step.

4 Recurrence Relation

The recurrence relations of type $T(n) = aT(\frac{n}{b}) + f(n)$ can be solved using the following methods:

- Plug and chuck : Keep on substituting in the value of $T(\frac{n}{b})$ in terms of smaller n untill we see a pattern. Pattern can be used to easily get the solution
- Substitution method : (This is more like check for your solution) In this method, we first guess the solution and prove that using induction, i.e, substitute the solution for $T(\frac{n}{b})$ and check whether it you can arrive at the proposed relation for $T(n)$
- Recursion-tree method: We draw the recursion tree for this method. In the recursion tree, leaves contain the cost for the base case and each node contains the cost for the merge and divide process. We sum all of them to get a solution for the recurrence relation.

  Theorem (The master theorem)**.** *Let $a \geq 1$, $b > 1$ and $f(n)$ be a function, then let $T(n)$ be defined on non-negative integers, then the recurrence relation*

  $$T(n) = aT(\frac{n}{b}) + f(n)$$

  is bound assymptotically in three cases.(If possible, let $f(n) = \Theta(n^d)$)

  $$\begin{aligned} T(n) &= \sum_{k=0}^{log_b n} a^k f(\frac{n}{b^k}) \\ &= \sum_{k=0}^{log_b n} a^k (\frac{n}{b^k})^d \\ &= \begin{cases} O(n) & \text{if } a < b^d \text{ or } f(n) = \Omega(n^{log_b a + \epsilon}) \\ O(n^d log n) & \text{if } a = b^d \text{ or } f(n) = \Theta(n^{log_b a}) \\ O(n^l og_b a) & \text{if } a > b^d \text{ or } f(n) = O(n^{log_b a - \epsilon}) \end{cases} \end{aligned}$$

  - In the first case ($a < b^d$ or $f(n) = \Omega(n^{log_b a + \epsilon})$), the work done at the first node dominates the time taken. Hence, total time taken = time taken at first node. Also this means that as the depth increases, the work done at each level decreases.
  - In the second case ($a = b^d$ or $f(n) = \Theta(n^{log_b a})$), the work done at all levels is equal. Hence, the total time taken = time at initial node $\times$ no of levels.
  - In the third case ($a > b^d$ or $f(n) = O(n^{log_b a - \epsilon})$), the work done at the deepest level dominates the time taken. Hence, total time taken = time taken at the lowest level

- Solving the recurrence relation $T_n = T_{n-1} + c_n$:

  $$\begin{aligned} a_n T_n &= b_n T_{n-1} + c_n \\ \text{Select } \sigma_n \text{ such that } \sigma_n b_n &= \sigma_{n-1} a_{n-1} \\ \sigma_n a_n T_n &= \sigma_n b_n T_{n-1} + \sigma_n c_n \\ \sigma_n a_n T_n &= \sigma_{n-1} b_{n-1} T_{n-1} \\ \sigma_n a_n T_n &= \sigma_0 a_0 T_0 + \sum_{k=1}^{n} c_n \sigma_n \\ T_n &= \frac{a_0 \sigma_0 T_0 + \sum_{k=1}^{n} c_n \sigma_n}{\sigma_n a_n} \end{aligned}$$