2 Recurrence Relations

2.1 Introduction

Many algorithms that you will encounter in this class (and upper level algorithms courses) possess runtimes, denoted by $T(n)$, which can be naturally modelled using recurrence relations.

So, what is a recurrence relation? It is a mathematical equation which is defined in terms of itself (just like a recursive function is defined in terms of itself meaning that the function makes a call to itself in its body, usually on a smaller input). Let us explore this further using an example.

The computational analogue of a mathematical function $f(n) = f(n-1) + n$ which calculates the sum of the first n natural numbers, namely $\sum_{i=1}^{n} i$, can be written in the C programming language as follows.

```c
int sum(int n)
{
    if (n == 0)
    {
        return 0;
    }
    else
    {
        return n + sum(n - 1);
    }
}
```

2.2 Essence of a Recurrence

Now, we have been saying that $T(n)$ denotes the runtime of an algorithm. But we can say a bit more than that about this important piece of notation. Essentially, $T(n)$ represents the number of $\Theta(1)$ operations our algorithm will perform on an input of size n.

For example, if we have a recurrence relation of the form $T(n) = T(n-1) + \Theta(1)$, it means that the algorithm reduces the size of the input it has to work on by 1, denoted by $T(n-1)$, by performing a constant amount of work (meaning that the work done is independent of the size of the input), denoted by $\Theta(1)$. The algorithm keeps on reducing the size of the input it is working on until it reaches the base case of the recurrence (meaning that the input to the algorithm is small enough such that the problem can be solved trivially without the need for any further recursive calls). For example, the recurrence relation for Binary Search can be written as follows,

$$T(n) = \begin{cases} T(n/2) + \Theta(1) & \text{, if } n > 1 \\ \Theta(1) & \text{, if } n = 1 \end{cases}$$

, because the algorithm reduces the size of the sorted array it is operating on by half each iteration (This is where the $T(n/2)$ comes from) by comparing the given key with the element in the middle of the sorted array (This is where the $\Theta(1)$ comes from). If the array has one element left, it performs a single $\Theta(1)$ time operation to determine whether the key is equal to this element or not which is what gives us the base case of the recurrence.

2.3 Solving Recurrences

So, how do you solve recurrences? The simplest method is known as the **Recursion Tree method** and that's all you need to know for now.

2.4 Problems

1. Solve the following recurrence relations.

   (a) $T(n) = T(n-1) + \Theta(1)$

   (b) $T(n) = T(n-1) + \Theta(n)$

   (c) $T(n) = T(n/2) + \Theta(1)$

   (d) $T(n) = 2T(n/2) + \Theta(n)$

   (e) $T(n) = 2T(n-1) + \Theta(1)$

   f) $T(n) = 2T(n/2) + \Theta(1)$