Text:

1. Recurrences

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

Consider the following recursive function:

```
void disp (int n)
{
        if(n>0)                          ----------------------------> 1
        {
                printf("%d", n);         ----------------------------> 1
                disp(n-1);               ----------------------------> T(n-1)
        }
}
```

Here, the running time complexity T(n) can be described in two parts.

i) *T(n) = 1* when *n=0*, i.e. the *if* condition would be executed once and the function would end.

ii) *T(n) = T(n-1) + 1*, this is the recursive call to the function itself and is dependent on the running time of the sub or recursive call *T(n-1)*. The constant *1* is added because of the *printf()* statement, which would be executed once in each call.

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

Similarly, the worst-case running time *T (n)* of the *merge sort* algorithm could be described by the recurrence

$$T(n) = \begin{cases} \theta(1) & n = 1 \\ 2T(n/2) + \theta(1) & n > 1 \end{cases}$$

whose solution was claimed to be *T(n) = (n log n)*.

There four basic methods for solving recurrences that is, for obtaining asymptotic bounds on the solution. In the substitution method, we guess a bound and then use mathematical induction to prove our guess correct. The iteration method converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence. The iteration tree method solves the recurrence graphically using the tree structure. The master method provides bounds for recurrences of the form

T (n) = aT (n - b) + (n)

T (n) = aT (n / b) + (n),

where $a > 1$, $b > 1$, and $f(n)$ is a given function; it requires memorization of three cases, but once you do that, determining asymptotic bounds for many simple recurrences is easy.

1.1. Substitution Method

A lot of things in this method reduce to induction. In the substitution method for solving recurrences we

- ✓ Guess the form of the solution.
- ✓ Use mathematical induction to find the constants and show that the solution works.

Let us see some examples to see how this method works

Example

Let us consider the following recurrence:

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

We guess that the solution is $T(n) = O(n \log n)$.

So we must prove that $T(n) \leq c\, n \log n$ for some constant c. (We will get to n_0 later, but for now let's try to prove the statement for all $n \geq 1$)

As our inductive hypothesis, we assume $T(n) \leq c\, n \log n$ for all positive numbers less than n.

Therefore, $T(n/2) \leq c\, n/2 \log(n/2))$, and

$$\begin{aligned} T(n) &\leq 2(c\, n/2 \log(n/2)) + n \\ &\leq c\, n \log(n/2) + n \\ &= c\, n \log n - cn \log 2 + n \\ &= c\, n \log n - cn + n \\ &\leq c\, n \log n \text{ (for } c \geq 1) \end{aligned}$$

Now we need to show the base case.

This is tricky, because if $T(n) \leq c\, n \log n$, then $T(1) \leq 0$, which is not a thing.

So we revise our induction so that we only prove the statement for $n \geq 2$, and the base cases of the induction proof (which is not the same as the base case of the recurrence!) are $n = 2$ and $n = 3$. (We are allowed to do this because asymptotic notation only requires us to prove our statement for $n \geq n_0$, and we can set $n_0 = 2$.) We choose $n = 2$ and $n = 3$ for our base cases because when we expand the recurrence formula, we will always go through either $n = 2$ or $n = 3$ before we hit the case where $n = 1$.

So proving the inductive step as above, plus proving the bound works for $n = 2$ and $n = 3$, suffices for our proof that the bound works for all $n > 1$. Plugging the numbers into the recurrence formula, we get

$T(2) = 2T(1) + 2 = 4$ and

$T(3) = 2T(1) + 3 = 5$.

So now we just need to choose a c that satisfies those constraints on $T(2)$ and $T(3)$. We can choose $c = 2$, because $4 \leq 2 \cdot 2\log 2$ and $5 \leq 2 \cdot 3\log 3$.

Therefore, we have shown that $T(n) \leq 2n \log n$ for all $n \geq 2$, so $T(n) = O(n \log n)$.

Warning: Using the substitution method, it is easy to prove a weaker bound than the one you're supposed to prove. For instance, if the runtime is $O(n)$, you might still be able to substitute $c.n^2$ into the recurrence and prove that the bound is $O(n^2)$. Which is technically true, but don't let it mislead you into thinking it's the best bound on the runtime. People often get burned by this on exams!

Consider the following recurrence relation, which shows up fairly frequently for some types of algorithms:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + C_1 & n > 1 \end{cases}$$

we can guess that this will be $O(2^n)$.

To use the substitution method to prove this bound, we now need to guess a closed-form upper bound based on this asymptotic bound. We will guess an upper bound of $k2^n - b$, where b is some constant.

We include the b in anticipation of having to deal with the constant C_1 that appears in the recurrence relation, and because it does no harm. In the process of proving this bound by induction, we will generate a set of constraints on k and b, and if b turns out to be unnecessary, we will be able to set it to whatever we want at the end.

Our property, then, is $T(n) \leq k2^n - b$, for some two constants k and b. Note that this property logically implies that $T(n)$ is $O(2^n)$, which can be verified with reference to the definition of O.

Base case: $n = 1$. $T(1) = 1 \leq k2^1 - b = 2k - b$. This is true as long as $k \geq (b + 1)/2$.

Inductive case: We assume our property is true for $n - 1$. We now want to show that it is true for n.

$$\begin{aligned} T(n) &= 2T(n-1) + C_1 \\ &\leq 2(k2^{n-1} - b) + C_1 \quad \text{(by IH)} \\ &= k2^n - 2b + C_1 \end{aligned}$$

$\leq k2^n - b$

This is true as long as $b \geq C_1$.

So we end up with two constraints that need to be satisfied for this proof to work, and we can satisfy them simply by letting $b = C_1$ and $k = (b + 1)/2$, which is always possible, as the definition of O allows us to choose any constant. Therefore, we have proved that our property is true, and so $T(n)$ is $O(2^n)$.