Methods for solving recurrences

Analyzing the complexity of mergesort

The *merge* function

Consider the following implementation:

```
 1  int* merge(int *v1, int n1, int *v2, int n2)
 2  {
 3      int* r = malloc((n1+n2)*sizeof(int));
 4      int i=0, j=0, k=0;
 5
 6      while (i<n1 && j<n2)
 7      {
 8          if (v1[i] > v2[j])
 9              r[k++] = v2[j++];
10          else
11              r[k++] = v1[i++];
12      }
13
14      while (i<n1)
15          r[k++] = v1[i++];
16
17      while (j<n2)
18          r[k++] = v2[j++]
19
20      return r;
21  }
```

We can easily identify the following asymptotic bounds for each line:

- lines 3 and 5 $\rightarrow O(n1 + n2)$
- line 13 $\rightarrow O(n1)$
- line 16 $\rightarrow O(n2)$
- it can be said that every other line runs in constant time $O(1)$

Summing these up, we get a bound for the entire function:

$$\text{merge} = O(n1 + n2) + O(n1) + O(n2) = O(n1 + n2)$$

Another bound can be found by using our intuition; each element from the two vectors must be read at least once, and thus we have: merge $= \Omega(n1 + n2)$. When the two bounds are combined, the final result is: merge $= \Theta(n1 + n2)$.

The *mergesort* function

The implementation is give below:

```
1  int* mergesort(int *v, int n)
2  {
3      if (n == 1)
4          return v;
5
6      v1 = malloc(n/2*sizeof(int));
7      v2 = malloc((n - n/2)*sizeof(int));
8
9      int i = 0;
10
11     while (i < n/2)
12         v1[i] = v[i];
13
14     while (i < n)
15         v2[n/2 - i] = v[i];
16
17     v1 = mergesort(v, n/2);
18     v2 = mergesort(v, n - n/2);
19
20     return merge(v1, n/2, v2, n - n/2);
21 }
```

We can try once more to assign a bound to each line:

- lines 6 and 7 $\rightarrow O(n/2) = O(n)$
- lines 11 and 14 $\rightarrow \Theta(n/2) = \Theta(n)$
- line 17 $\rightarrow T(\lfloor n/2 \rfloor)$; by this we mean the complexity of the same function, but with a different input size (in this case, n divided by 2 and rounded down)
- line 18 $\rightarrow T(\lceil n/2 \rceil)$
- line 20 $\rightarrow \Theta(n/2 + n - n/2) = \Theta(n)$
- other lines $\rightarrow O(1)$

Before adding them up, we must take into consideration line 3. If $n = 1$, the whole function runs in $O(1)$ because the *return* statement is executed. Otherwise, we get:

$$T(n) = O(n) + \Theta(n) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$$

This can further change into:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$$

For all algorithms that we are going to encounter, $T(n)$ is considered to be $= O(1)$, for sufficiently small values of n. Because of this, the distinction made before is no longer necessary, and we rely exclusively on the recurrence relation above to describe the running time of the function.

Another simplifying assumption is that $\lfloor n/2 \rfloor = \lceil n/2 \rceil = n/2$. For large values of n (which are usually the ones of interest in complexity analysis), we consider that the addition or subtraction of a single unit does not influence the result. Thus, the previous expression becomes:

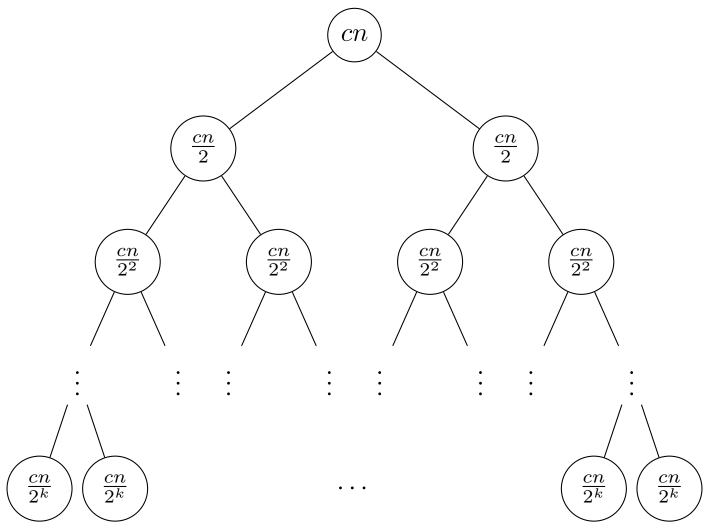
$$T(n) = T(n/2) + T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n)$$

Finally, we replace the $\Theta(n)$ above (which denotes a set of functions) with a single function from $\Theta(n)$, written as cn, where $c \in \mathbb{R}$ is a constant. There are a number of reasons for this. First of all, the running time of that particular part of the recurrence relation must be described by a function, and not by a whole set of functions. Secondly, by not doing this, we risk missing the fact that the running time of that part gradually decreases as the recurrence unfolds. For example $T(n/2) = 2T(n/4) + \Theta(n/2) = 2T(n/4) + \Theta(n)$. By replacing $\Theta(n)$ with cn, we get $T(n/2) = 2T(n/4) + cn/2$.

The iteration method

This method is helpful in finding an informed estimation about the bounds that can be applied to the running time of an algorithm. It does not also establish a definite proof about our findings, but it is of great help as a starting point.

The iteration method consists of (partially) drawing the recursion tree. Each node represents the cost of a single sub-problem, and the number of its children is equal to the number of sub-problems that are directly created from that point. For the *mergesort* function, we would get something like the following tree:



How tall is this tree ? The height is equal to $k + 1$, because we have the root level and then k additional levels. How large is the value of k? A leaf must represent a problem that cannot be subdivided any longer; thus we can say that $\frac{n}{2^k} = 1$. This leads to $n = 2^k$, which means that $k = \log n$ (we omit writing the base of the logarithm). Thus, the above tree for *mergesort* is $\log n + 1$ levels tall. How many leaves can be found on the last level ? The number of nodes doubles from one level to the next, starting with the root node. There are 2^i nodes on level i (assuming the root node is on level 0), so we have $2^{\log n} = n$ leaves.

In order to get an estimate for the run time of *mergesort* we have to sum the contents of each node. A useful observation is that on every level i, there are 2^i nodes, each one containing the expression $\frac{cn}{2^i}$. The sum for every level i is thus $\frac{cn}{2^i} \cdot 2^i = cn$. Since there are $1 + \log n$ levels, the global sum is $cn(1 + \log n) = cn \log n + cn = \Theta(n \log n)$.

The substitution method

The previous method gave us the intuition that *mergesort* $= \Theta(n \log n)$, but we would also like a formal proof, just to be sure. This can be obtained with the substitution method, which requires a "guess" for $T(n)$ (in our case, we got from the previous method that $T(n) = \Theta(n \log n)$). With this starting point, we rely on induction to prove the truth of the statement $P(n) \equiv c_1 n \log n \leq T(n) \leq c_2 n \log n$, for all $n > n_0$, $n \in \mathbb{N}$, where n_0, c_1 and c_2 are chosen beforehand, as per the definition of Θ. There are two steps in the induction process:

- we prove that, for any given $n \in \mathbb{N}$, if P holds for all natural values smaller than n (induction hypothesis), then P also holds for n (the induction step).

- we prove that P holds for the base case (sometime there are multiple base cases).

Let's start with the induction step. We presume that P holds for all values smaller than n, and try to prove that $P(n)$ also holds based on this assumption. An interesting value smaller than n is $n/2$. If $P(n/2)$ holds, it means that $\exists n_0 \in \mathbb{N}$ and $c_1, c_2 \in \mathbb{R}^+$ such that $\forall n \geq n_0$:

$$c_1 n/2 \log(n/2) \leq T(n/2) \leq c_2 n/2 \log(n/2)$$

Now, we want to substitute (hence "substitution method") these inequalities into the recurrence $T(n) = 2T(n/2) + cn$, and this basically means that we multiply both ends by 2 and add cn afterwards. By doing this, we get:

$$2c_1 n/2 \log(n/2) + cn \leq 2T(n/2) + cn \leq 2c_2 n/2 \log(n/2) + cn$$

$$c_1 n \log(n/2) + cn \leq T(n) \leq c_2 n \log(n/2) + cn$$

$$c_1 n(\log n - \log 2) + cn \leq T(n) \leq c_2 n(\log n - \log 2) + cn$$

$$c_1 n \log n - c_1 n + cn \leq T(n) \leq c_2 n \log n - c_2 n + cn$$

Remember that the goal of this step is to prove that $P(n)$ holds; in other words we have to prove that $c_1 n \log n \leq T(n) \leq c_2 n \log n$. For this to be true, starting from where we just left, we must have $cn - c_1 n \geq 0$ and $cn - c_2 n \leq 0$. This means that $c_1 \leq c$ and $c_2 \geq c$. Since c_1 can be chosen arbitrarily small and c_2 can be chosen arbitrarily large, the induction step has been successfully proven.

Why do we need to show that P holds for at least one base case ? Well, the induction step shows us that we can prove $P(n)$ by relying on that fact that P holds for smaller values. However, if we follow this reasoning, at some point we are going to find some m that is the smallest value (or the smallest interesting value). In this case (base case), we cannot rely on the induction step to prove $P(m)$, as "all values smaller than m" doesn't make any sense.

What's the base case for our recurrence relation ? If we consider 0 to be the base case, the expression $\log(0)$ will show up, which doesn't make sense. If 1 is the base case, then we're going to have $0 \leq T(n) \leq 0$, but this goes against the convention that $T(1) = O(1)$. Thus, our base case will be 2 ($n_0 = 2$). For 2, the inequalities become:

$$2c_1 \log 2 \leq T(2) \leq 2c_2 \log 2$$

$$2c_1 \leq 2T(1) + 2c \leq 2c_2$$

$$2c_1 \leq 2(c+1) \leq 2c_2$$

$$c_1 \leq c + 1 \leq c_2$$

As before, these can be easily satisfied with a proper choice of c, c_1 and c_2. It is important to note that we were able to choose the base case at will, because it represents the n_0 from the definition of Θ, which is existentially qualified.

Another very important thing is that the constants must not change when applying the substitution method. Here's an example of what this means and why it matters so much: we are going to prove that $T(n) = O(n)$, where $T(n) = 2T(n/2) + n$ (we omit the base case for brevity). For this problem, $P(n) \equiv T(n) \leq cn$. The induction hypothesis give us $T(n/2) \leq cn/2$. During substitution we get:

$$T(n) \leq 2cn/2 + n \leq cn + n \leq (c+1)n$$

While $c + 1$ is a constant, it is not the one we were looking form, and also $T(n) \leq (c+1)n$ does not imply that $T(n) \leq cn$. The induction step is complete only if we can prove the exact form of the inequality (in this case $T(n) \leq cn$).

The master method

The master method is basically a recipe for solving recurrence relations. We can use it to obtains answers in very little time, but the drawback is that it cannot be applied in every situation. It can be used for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function. There are three possible cases when the master theorem provides an answer:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.