Justifying relations

For justifying asymptotic relations, we need to argue using the definition. Using calculus is NOT an option, because the relationship between the calculus theorems and the definitions of asymptotic relations is not immediately clear. If you have some unexpicable urge to use calculus, make sure that you start by proving the correspondence between asymptotics and whatever mathematical machinery you employ. In the general case, we just write up the definition and find some suitable values for the constants. We can, if possible, use the generally accepted fact that "polynomials grow faster than polylogarithms, and exponentials grows faster than polynomials". BUT, you need to make sure that you in fact deal with a polylogaritm or a polynomial and not some mix between the two.

Another fact you can use is that if you can factor the expression into two expressions that are factorwise dominating. As an example of this techique, assume we want to show that $n^2 = \mathcal{O}(n^3 \lg n)$. Since $n^2 = n^2 \cdot 1$, n^2 is dominated by n^3 and 1 is dominated by $\lg n$, we get immediately the desired result.

In the same manner, we can prove that $n^3 \lg n = \mathcal{O}(n^4)$ by saying that since $n^4 = n^3 \cdot n$, $n^3 = \mathcal{O}(n^3)$ and $\lg n = \mathcal{O}(n)$ since polynomials grow faster than polylogarithms, also this relation holds.

Analysing algorithms

In general, we have some standard ways of deriving the runtime of algorithms. The first step is to decide what operation are constants. The second step is to decide how many times there operations are repeated. There are two patterns that can be handled very easily if spotted.

1. If the algoritm has loops that are repeated some number of times that is some simple function of the size of the input, then you simply multiply the runtime of the loop body with this simple function to get the runtime of the loop. One such example is the MERGE function, where the merging loop will execute a number of times that is linear in the size of the input. Since the loop body is constant, the overall runtime of the loop is $\mathcal{O}(n)$.

2. If the algorithm contains calls to itself, the runtime can be expressed as a recurrence. This recurrence can then be bounded using a number of different methods. One such method is described below, under "summing recursive trees".

If none of these patterns applies, the way to approach the problem is to in some way sum the total number of times that the constant operations are executed. If one for example has a loop where the number of times it is executed is not immediately obvious from the context, try to derive an expression for each time the loop is executed, and then sum that expression over the inputs.

Expected analysis

Question

Give the expected running time for the following algorithm, assuming the input alphabet is $\{0, 1\}$.

```
FOO(A)
1  n ← size(A)
2  for i ← 1 to ⌊n/2⌋
3        do if not A[i] = A[n − i + 1]
4              then return False
5  return True
```

Answer

As is immediate from the pseudocode, the algorithm contains one loop which is doing constant work. Thus, the expected running time is proportional to the number of times the loop will start to execute, and this is what we need to find out.

Assume an input array A of size n. Let's call the number of times the loop executes k, meaning we want to find $E[k]$, the expected value of k. To start with, what is the possible values of k? Obviously, the loop has to start executing at least once, so the smallest value of k is 1. At most, the loop will execute $\lfloor \frac{n}{2} \rfloor$ times, which means that $k \in \{1, 2, \ldots, \lfloor \frac{n}{2} \rfloor\} = U$.

By the definition of expected value, we have

$$E[k] = \sum_{i \in U} p_i i$$

where p_i is the probability that k will be equal to i.

In order to compute the expected value, we apparently need an expression for p_i. To find this, we start by looking at the condition for exiting or continuing in the loop. In order to evaluate to true, this condition requires that $A[i]$ and $A[n - i + 1]$ are equal. As the input alphabet is binary, the probability of this is 1/2. Therefore we get $p_1 = 1/2$, $p_2 = (1/2)^2$ and so on, meaning that $p_i = (1/2)^i$. The only exception is $p_{\lfloor \frac{n}{2} \rfloor}$, the probability of running the loop the maximum number of times. In this case the loop will stop after this iteration regardless of what the outcome of the test on line 3 is, either because if finding a mismatch (returning false), or because the for loop is done (which means going on and returning true from line 5). Therefore $p_{\lfloor \frac{n}{2} \rfloor} = (1/2)^{\lfloor \frac{n}{2} \rfloor} + (1/2)^{\lfloor \frac{n}{2} \rfloor}$, which makes the sum of all probabilities appropriately equal to 1.

Therefore we now get

$$E[k] = \sum_{i \in U} p_i i = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (1/2)^i i + (1/2)^{\lfloor \frac{n}{2} \rfloor} \left\lfloor \frac{n}{2} \right\rfloor = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \frac{i}{2^i} + (1/2)^{\lfloor \frac{n}{2} \rfloor} \left\lfloor \frac{n}{2} \right\rfloor \leq$$

$$\sum_{i=1}^{\infty} \frac{i}{2^i} + \frac{1}{2^{\frac{n-1}{2}}} \frac{n}{2} = \underbrace{\sum_{i=1}^{\infty} \frac{i}{2^i}}_{\leq 9/4} + \underbrace{\frac{n}{2^{\frac{n+1}{2}}}}_{\leq 1} \leq 13/4 = \mathcal{O}(1)$$

The bounding of the sum in the final step can be found in Appendix 1 in the course book, and is based on finding a convergent geometric sum that termwise dominates the sum. In this case we note that the ratio between consecutive terms is

$$\frac{i+1}{2^{i+1}} / \frac{i}{2^i} = \frac{i+1}{2^{i+1}} \cdot \frac{2^i}{i} = \frac{i+1}{2i} = \frac{i}{2i} + \frac{1}{2i} = \frac{1}{2} + \frac{1}{2i} \leq \frac{3}{4}$$

where the last inequalty holds if $i \geq 2$. Thus we can bound the sum with the geometric series with $\alpha = \frac{3}{4}$. The interesting thing is not really the exact value in this case, but rather that the series is bounded by a constant.

Remark Note that if the input alphabet is not binary but instead has k symbols, the effect will be that $p_i = (1/k)^i$ instead of $p_i = (1/2)^i$. This will make the analysis still constant, as the terms will be even smaller.

Summing recursive trees

Suppose that we want to prove that for

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T\left(\lfloor \frac{n}{2} \rfloor\right) + \frac{n^2}{3} & \text{if } n > 1 \end{cases}$$

we have $T(n) = \Omega(n \lg n)$

Lets start by unfolding the tree, and let's skip the floor function:

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n^2}{3} = 2 \cdot \left(2T\left(\frac{(n/2)}{2}\right) + \frac{(n/2)^2}{3}\right) + \frac{n^2}{3} =$$

$$= 4T\left(\frac{n}{4}\right) + \frac{n^2}{6} + \frac{n^2}{3} = 4 \cdot \left(2T\left(\frac{(n/4)}{2}\right) + \frac{(n/4)^2}{3}\right) + \frac{n^2}{6} + \frac{n^2}{3} =$$

$$= 8T\left(\frac{n}{8}\right) + \frac{n^2}{12} + \frac{n^2}{6} + \frac{n^2}{3} = \dots$$

Now we can see that we get a linear term once the $T(k)$ hits 1, and we have a sequence of n^2 terms. This can be more easily written as

$$T(n) = n + \sum_{i=0}^{\lg n} \frac{n^2}{3 \cdot 2^i} = n + \frac{n^2}{3} \sum_{i=0}^{\lg n} \frac{1}{2^i}$$

By noting that for all n we have

$$1 \leq \sum_{i=0}^{\lg n} \frac{1}{2^i} \leq \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2$$

we can conclude that $T(n) = \Theta(n^2)$, and therefore also $T(n) = \Omega(n \lg n)$

The main mistake to make here is the imprecise summing of the tree by saying "well, we have a quadratic term at every level and $\lg n$ levels, so the running time must be $n^2 \lg n$". What this argument fails to recognize is the fact that the size of the n^2 terms decreases. It is true that $n^2 \lg n$ is a conservative estimate of the upper bound. However, a conservative estimate like that gives us no information as to how the lower bound behaves. To do that we need to sum more exactly.