Tutorial 1:

The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency (performance of the algorithm) and also allows us to compare the relative performance of alternative algorithms.

Note: Algorithm is a strategy to solve a problem. So when we talked about the performance of an algorithm, we're talking about how good that strategy is for coming at the solution of the problem that we're thinking about.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the ***asymptotic*** efficiency of algorithms.

That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

ORDER COST DEFINITIONS

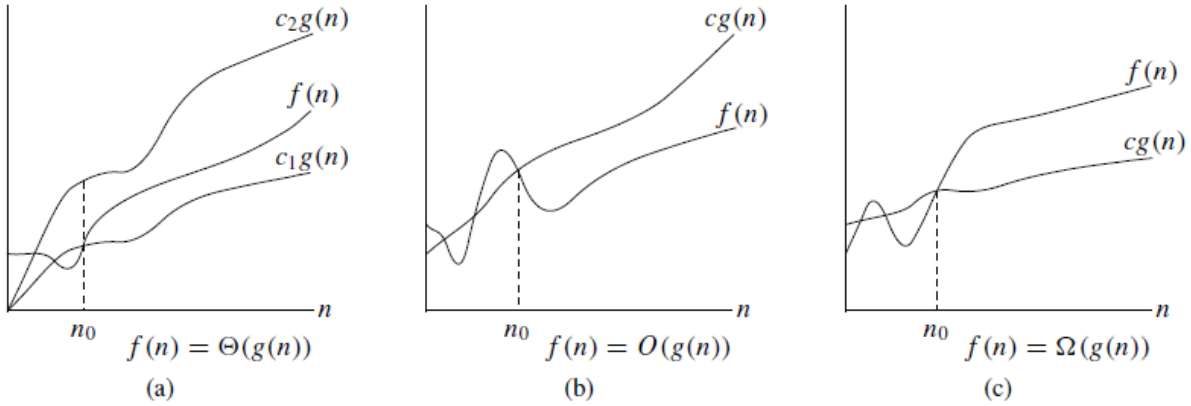
Intuitions:

- $f(n) \in O(g(n))$ if f's cost is no more than g's.
- $f(n) \in \Omega(g(n))$ if f's cost is no less than g's.
- $f(n) \in \Theta(g(n))$ if f has the same cost as g.

Formalisations:

- $f(n) \in O(g(n))$ iff $\exists c \overset{\infty}{\forall} n \;\; f(n) \leq c \cdot g(n)$
- $f(n) \in \Omega(g(n))$ iff $\exists c \overset{\infty}{\forall} n \;\; f(n) \geq c \cdot g(n)$
- $f(n) \in \Theta(g(n))$ iff $\exists c, d \overset{\infty}{\forall} n \;\; c \cdot g(n) \leq f(n) \leq d \cdot g(n)$

Where c and d must be positive.



Big O notation

Big O notation captures the rate of growth of two functions.

f(n) = O(g(n))
f(n) and g(n) grow in same way as their input grows up to constants

formally : there are constant N and c so that each n>N
$f(n) \leq C\ g(n)$

Examples: Write O(n) notation for f(n) = 3n+3

Ans: We can keep the part of the function that is the fastest growing. So for example, we have 3n term and 3.Now, 3n grows as n grows but 3 does not. And so the dominant term of those 2 pieces of the sum is the 3n. Also we can drop constant as 3n is same as n in this big O sense. .

3n+3 = O(3n) = O(n)

Example: Consider the function $g(n)=100+n^2+2^n$. Which of the following is true?

1. g(n)=O(1)
2. $g(n)=O(n^2)$
3. $g(n)=O(2^n)$

Ans: 3. $g(n)=O(2^n)$ as among all the options the fastest growing term is the exponential function 2^n

Example: $f(n)=4\log_2(n)+3n\log_2(n)+n$ Write O(n) notation

$f(n)=O(n\log_2(n))$

Here the product $n\log_2(n)$ is the fastest growing term because each of n and $\log_2(n)$ grows as n grows.

Try to write O(n) of these functions:

$f(n) = 3n^2+4$, $f(n) = 4n^2+2n+10$

Solution to tutorial:

Using the definitions of O, Ω, and Θ, show that:

1. $n^{\frac{5}{2}} \in \Omega(n^2)$
2. $n^k \in O(k^n)$ for all constants $k > 1$
3. $\log(n!) \in \Theta(n \log n)$

1. **$n^{5/2} \in \Omega(n^2)$;**

As we know the definition Ω is $f(n) \geq C\, g(n)$ so

$n^{2.5} \geq C.n^2$

$n^2 . n^{0.5} \geq C.n^2$

$\sqrt{n} \geq C$

2. **$n^k \in O(k^n)$ when k>1**

As we know the definition O is $f(n) \leq C\, g(n)$ so

$\log(n^k) \leq \log(k^n)$

$k\log(n) \leq n\log(k)$

$2.\log(n) \leq n\log(2)$ if k=2

$2.\log(n) \leq n.1$

$\log(n) \leq n$ if we drop the constant.

And from the answer we can see that since n grows faster than log(n) for all k>1

3. **$\log (n!) \in \Theta(n \log n)$**

$\log (n!) \in \Theta(n \log n)$

$\log(n!) = \log(1) + \log(2) + ... + \log(n-1) + \log(n)$

You can get the upper bound by

$\log(1) + \log(2) + ... + \log(n) <= \log(n) + \log(n) + ... + \log(n)$
$= n*\log(n)$

And you can get the lower bound by doing a similar thing after throwing away the first half of the sum:

$\log(1) + ... + \log(n/2) + ... + \log(n) >= \log(n/2) + ... + \log(n)$
$>= \log(n/2) + ... + \log(n/2)$
$= n/2 * \log(n/2)$

Rank the following in order of increasing order cost:

1. $f(n) = n^{2.5}$
2. $f(n) = \sqrt{2n}$
3. $f(n) = n + 10$
4. $f(n) = 10^n$
5. $f(n) = 100^n$
6. $f(n) = n^2 \log n$

Answer:

$f_2(n) < f_3(n) < f_6(n) < f_1(n) < f_4(n) < f_5(n)$

Explanation: $f_4(n)$, $f_5(n)$ because these functions are exponential will grow faster. $f_4(n) < f_5(n)$ $10<100$. Other four function are polynomial and will grow slower than exponential. $f_2(n)$ will be slowest out of the polynomial because it has the smallest degree. $f_6(n)$ will be bounded by $f_1(n)$ because $f_6 = n^2 \log(n)$ and $f1 = n^2\sqrt{n}$ and $\log(n) = O(\sqrt{n})$.

QUESTION 3 - INSERTION SORT I

```
def insertion_sort(A):
  n = len(A)
  for j from 2 to n:
    x = A[j]
    i = j-1
    while i > 0 and A[i] > x:
      A[i+1] = A[i]
      i = i-1
    A[i+1] = x
```

Let's finish the correctness proof done in class.

The (outer) loop invariant was:

1. $A[1..j-1]$ is permutation of $A_0[1..j-1]$.
2. $A[1..j-1]$ is sorted.
3. $A[j..n]$ is the same as $A_0[j..n]$.

Goal: prove parts 1 and 2 for the *inductive case*.

Let's say that A is the current state of the list, and A' is its state after the next iteration.

1. Assume $A[1..j-1]$ is a permutation of $A_0[1..j-1]$.
   Show that $A'[1..j]$ is a permutation of $A_0[1..j]$.
2. Assume $A[1..j]$ is sorted.
   Show that $A'[1..j]$ is sorted.

Please check previous year solutions:

Proofs from Tutorial 1

The main event this week was finishing the proof of correctness for insertion sort that you started in class. As a reminder, this really comes down to proving the *loop invariant*, which is a statement that is true after every iteration of the loop. Ours was:

1. $A[1\,..\,j{-}1]$[1] is permutation of $A_0[1\,..\,j{-}1]$.
2. $A[1\,..\,j{-}1]$ is sorted.
3. $A[j\,..\,N]$ is the same as $A_0[j\,..\,N]$.

These things tend to be proven inductively, so we need to prove a *base case* and *inductive case* for each of the three conditions. I'll assume you're comfortable with induction, by the way, read up on it if not, you'll be using it a lot in the course.

For consistency, let's change the notation we used in tutorial a bit. Let A_j be the state of the input array A after the j-th iteration of the loop. This means that A_0 is the original state of the array. This has one particularly nice consequence, at time j, the sub-array $A_j[0\,..\,j]$ is the *sorted section* of the array.

In class, you proved the base case for all three of the invariants, as well as the inductive case for the part 3. Our job is to prove the inductive case for parts 1 and 2. We really only finished the proof of part 1, so that's what I'll focus on here. Hence, our job is as follows:

> Inductively assume that $A_j[1\,..\,j]$ is a permutation of $A_0[1\,..\,j]$, and prove that $A_{j+1}[1\,..\,j+1]$ is a permutation of $A_0[1\,..\,j+1]$.

Now we ended up doing two completely different proofs with the two tutorials, so I'll write up both approaches here.

Monday's proof

Monday was a 'constructive' proof, in which we directly showed that $A_{j+1}[1\,..\,j+1]$ would be a permutation of $A_0[1\,..\,j+1]$.

As always, make the inductive assumption that $A_j[1\,..\,j]$ is a permutation. Let $x = A[j+1]$ be the element in the array that we're about to sort, and let i be the index that x[2] will be at *after* it's been shuffled to the correct place in A_{j+1}. The proof will proceed by showing that the sub-arrays $A_{j+1}[1\,..\,i{-}1]$, $A_{j+1}[i\,..\,i]$, and $A_{j+1}[i{+}1\,..\,j{+}1]$ combine to form a permutation of $A_0[1\,..\,j{+}1]$.

Consider $A_{j+1}[1\,..\,i{-}1]$, called B_1 for convenience. The inner `while` loop of the pseudocode will never execute on these elements by nature of its condition. Therefore, this sub-array is unchanged by this iteration of the loop, and is equal to $A_j[1\,..\,i{-}1]$.

Consider $A_{j+1}[i{+}1\,..\,j{+}1]$, called B_2 for convenience. These are exactly the elements that the inner `while` loop has shifted right one place in the array. Therefore, this is equal to $A_j[i\,..\,j]$.

Now observe that, which concatenated, B_1 and B_2 exactly form $A_j[1\,..\,j]$ due to the logic above. By our inductive assumption, $A_j[1\,..\,j]$ is a permutation of $A_0[1\,..\,j]$. Finally, note that[3] :

$$A_{j+1}[1\,..\,j+1] = B_1 + [x] + B_2$$

And that we defined $x = A[j+1]$. Hence, $A_{j+1}[1\,..\,j{+}1]$ is a permutation of $A_j[1\,..\,j]$ with $A[j+1]$ included in it somewhere. Therefore, the sub-array $A_{j+1}[1\,..\,j+1]$ is a permutation of $A_0[1\,..\,j+1]$.

[1] Remember arrays are 1-indexed in this course.

[2] I'm going to assume that all elements are distinct in the array, so even if x has value 1 and some other element has value 1, they are *not* equal.

[3] Where + represents array concatenation.

Tuesday's proof

Tuesday was a proof by contradiction, in which we show that if we assume it is *not* a permutation, we get a logically inconsistent result (and so the assumption must be false).

First, make the inductive assumption that the loop invariant holds for $A_j[1\,..\,j]$. Now, suppose that $A_{j+1}[1\,..\,j{+}1]$ is *not* a permutation of $A_0[1\,..\,j{+}1]$. Therefore, there exists some element $y \in A_{j+1}[1\,..\,j{+}1]$ such that $y \notin A_0[1\,..\,j{+}1]$.

The algorithm introduces no *new* elements into the array, and so y must be some element of A_0. We will break into three cases and show each leads to a contradiction.

- Suppose that $y \in A_j[1\,..\,j]$. Then this would make $A_j[1\,..\,j]$ not a permutation of $A_0[1\,..\,j]$, which breaks our inductive assumption. Contradiction.
- Suppose that $y = A_j[j{+}1]$, the element that we will be sorted this iteration. By part 3 of the loop invariant and our inductive assumption, $A_j[j{+}1]$ has not been modified by the algorithm, so $A_j[j{+}1] = A_0[j{+}1]$ (note the subscript). But we required (end of second paragraph) that $y \notin A_0[1\,..\,j{+}1]$. Contradiction.
- Suppose that $y \in A_j[j{+}2\,..\,N]$ (where N is the length of the array). By the proof of part 3 of the loop invariant for *this* iteration, $A_{j+1}[j{+}2\,..\,N]$ cannot have been changed, so $y \in A_{j+1}[j{+}2\,..\,N]$. Contradiction.

Therefore, our assumption is false, so $A_{j+1}[1\,..\,j{+}1]$ is a permutation of $A_0[1\,..\,j{+}1]$.

Discussion

Well, that was fairly involved. These proofs are quite verbose, realistically you could get away with a bit less rigour than what I've done here. That isn't to say this sort of formality is undesirable, it's what you should be aiming for in some situations (assignments) but not in others (exams).

As a general thing, proof by contradiction should be your first attempt if you've got no idea how to prove something. If it works they tend to come out fairly cleanly like above, the Tuesday proof is a lot simpler than the Monday one. If it doesn't work it often gives you enough understanding to try another method.

Feel free to point out errors, suggest improvements, or ask questions about this.