Recitation Notes

1 Big O, Omega, and Theta

1.1 Definitions

Big O: A function f(n) is in the set O(g(n)) if for some constant c, and all $n \geq n_0$:f(n) $\leq cg(n)$

In other words, f(n) will not grow faster than g(n) as the size of the input (n) gets large. We can view g(n) as an upper bound for f(n). With large input, we want to look at the shape of the graph denoting the amount of work the function does as the size of the input increases.

Big Omega: A function f(n) is in the set Ω g(n) if for some constant c and all $n \geq n_0$: $f(n) \geq cg(n)$.

In other words, g(n) is a lower bound on f(n).

Big Theta: A function f(n) is in the set Θ g(n) if for some constant c and all $n\geq$ n_0 : sf(n) $\geq cg(n)$ AND $f(n) \leq cg(n)$ so $f(n) = cg(n)$. This is the tightest bound for a function f(n).

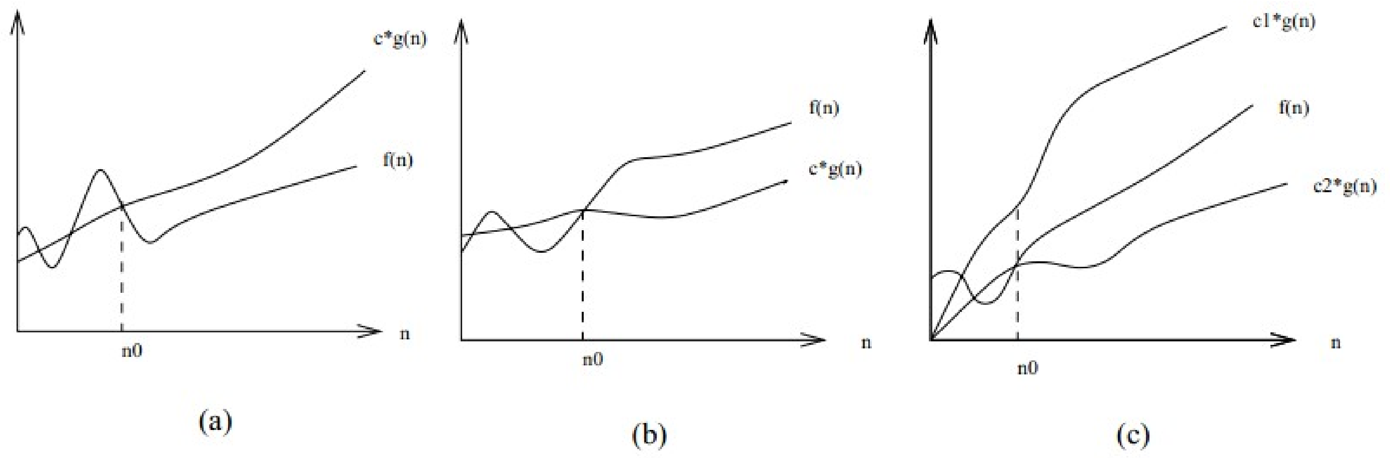

Figure 2.3: Illustrating the big (a) O, (b) Ω, and (c) Θ notations

A note on little-oh and little omega. You can think of these as stricter bounds of the same form as their big counterparts. A function f(n) is in the set o(g(n)) if for some constant c, and all $n \geq n_0$,f(n) <cg(n)

Similarly, a function f(n) is in the set ωg(n) if or some constant c, and all $n \geq n_0$,f(n) >cg(n) .

Question: why didn't we mention little-theta?

1.2 Proving

Here is an example of a statement we want to prove:

$T(n) = n^3 + 20n + 1$ is in the set $O(n^3)$.

We can ask ourselves whether we can find some constant c and n_0 such that $T(n) \leq cn^3$ for some $n \geq n_0$. Simplifying the inequality allows us to understand possible values c can take on to satisfy it.

Dividing both sides by n^3, we are left with $1 + \frac{20}{n^2} + \frac{1}{n^3} \leq c$.

Can we choose a constant c and n_0 to satisfy the Big-Oh definition?

Yes. If we choose $n_0 = 1$ and $c = 22$, for all $n \geq n_0$, the inequality holds.

Question: What would happen if we increased n_0 ?

We could have a smaller factor c and still satisfy the inequality.

1.3 Disproving

Let's take the same recurrence relation $T(n) = n^3 + 20n + 1$ and show that it does not belong in the set $O(n^2)$

Intuitively, we can understand that no choice of c and n_0 will make a function $f(n^3)$ grow at the same rate or more slowly than $f(n^2)$. Let's show it formally.

Using our strategy from before, we divide both sides of the inequality $n^3 + 20n + 1 \leq cn^2$ by n^2, giving us the result $n + \frac{20}{n} + \frac{1}{n^2} \leq c$.

Can we choose a constant c such that this inequality holds?

No. for any c we choose, there will be a larger n that will violate the inequality.

2 Divide and Conquer

2.1 Definition

Divide and Conquer is an algorithmic technique based around taking a hard problem and breaking it down into multiple, smaller subproblems that are each easier to solve

There are three steps to this technique:

1. Divide - Break problem into smaller subproblems
2. Conquer - Two cases:
   (a) If the subproblem is "large" - Solve using recursion
   (b) If the subproblem is "small" - Solve directly (recursive base case)
3. Combine - Merge the solutions from the subproblems together to get the runtime of the recurrence

2.2 Runtime

$T(n) = D(n) + aT(\frac{n}{b}) + C(n)$

$T(n)$ is the runtime of the whole algorithm on a problem of size n.

$D(n)$ is the runtime of dividing the problem into subproblems.

$aT(\frac{n}{b})$ is the runtime of solving a subproblems of size $\frac{n}{b}$ recursively.

$C(n)$ is the runtime of combining the subproblem solutions into the solution for $T(n)$

2.3 Recurrence Example

MergeSort:

- Divide -
- Conquer -
- Combine -

Runtime Analysis:

- Divide -
- Conquer -
- Combine -

Recurrence:

3 Recurrences

Recurrence = Expression for complexity in terms of itself
examples:

- $T(n) = \sqrt{T(n-1)} + n$
- $S(n) = 2S(\frac{n(n-1)}{n+1}) + \log n$

Hard to tell what exactly the runtime or space complexity is.
We want to prove that these functions T and S are Θ of well-known, closed-form functions.

3.1 Master Method

1. Look for the form $T(n) = aT(\frac{n}{b}) + f(n)$.
   All divide and conquer follow this form. Proof:

2. Compare $f(n)$ to $n^{\log_b a}$.
   - **Case 1:** $f(n) \in O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$
     implies $T(n) \in \Theta(n^{\log_b a})$.

     Example: Binary Tree Traversal $T(n) = 2T(\frac{n}{2}) + 1$

     $f(n) =$ $a =$ $b =$ $\log_b a =$

- **Case 2:** $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$
  implies $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$.

  Example: Mergesort $T(n) = 2T(\frac{n}{2}) + n$

  $f(n) =$ $a =$ $b =$ $\log_b a =$

  Example: $T(n) = 2T(\frac{n}{2}) + n \log n$

  $f(n) =$ $a =$ $b =$ $\log_b a =$

- **Case 3:** $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$

  implies *nothing.*

  unless $af(\frac{n}{b}) \leq kf(n)$ for some $k < 1$ and sufficiently large n.

  called the *regularity condition.*

  Then it implies: $T(n) \in \Theta(f(n))$

  Example: Quick Select $T(n) = T(\frac{n}{2}) + n$

  $f(n) =$ $a =$ $b =$ $\log_b a =$

  Example: $T(n) = 2T(\frac{n}{2}) + 15n^3$

  $f(n) =$ $a =$ $b =$ $\log_b a =$

3.2 Substitution

$T(n) = 2T(\sqrt{n}) + \log_2 n$

3.3 Induction

A proof by induction requires that every step must be justified; just like a regular proof. However, it employs a neat trick: to prove a statement about an arbitrary number n, first prove it is true when n = 1 (the base case) and then, assuming it is true for n = k, demonstrate it is true for n=k+1.

The concept is that if you want to show that someone can climb to the nth floor of a fire escape, you need only show that you can climb the ladder up to the fire escape (n=1) and then show that you know how to climb the stairs from any level of the fire escape (n=k) to the next level (n=k+1). Thus, you have proved that they can reach the first flight and, given an arbitrary flight, they can climb up to the next flight. Therefore, they can climb from $1 \rightarrow 2$, $2 \rightarrow 3$, $3 \rightarrow 4$, ... , $n - 1 \rightarrow n$.

If you've done proof by induction before you may have been asked to assume the n-1 case and show the n case, or assume the n case and show the n+1 case.

Check out this source link for more background

4 Proving Correctness

4.1 Merge-Sort

Claim: Given a list L of n elements, merge-sort correctly sorts the elements L

Proof.
Proof by Induction.

Base Case
The base case of merge sort is when $n = 1$. When $n = 1$, the array has one element. Any array with one element is already sorted, thus merge sort is correct in the base case.

When $n = k$
Given an array of length $|L|$, assume that that when $n < |L|$ we can call merge sort on the array of length n and merge sort will return the correct output.

When $n = k + 1$
Merge sort will divide the array of length $n = k + 1$ into two sub-arrays of size $|\frac{n}{2}|$. We know that merge sort will correctly sort these two sub-arrays because their size $n < k + 1$. Now, we need to confirm that the merge step will correctly merge the two sub-arrays. We know that each sub-array is sorted such that the smallest element is at the 0^{th} spot and the largest element is at the $n - 1^{th}$ spot. The merge element takes the minimum element of each of the sub-arrays and compares them. It takes the smaller of the two elements and puts it in the final array and deletes it from its original array. It repeats this step until both of the sub-arrays are empty.

Thus, we have proved, by induction, that merge-sort will sort any list L.

$\square$