CS 161 Winter 2020 Section 1

January 9-10, 2020

Asymptotic Analysis

Asymptotic Analysis Definitions

Definition 1 (Big-Oh notation)**.** Let f, g be functions from the positive integers to the non-negative reals. Then we say that:

$f = O(g)$ if there exist constants $c > 0$ and n_0 such that for all $n > n_0$,

$$f(n) \leq c \cdot g(n).$$

$f = \Omega(g)$ if there exist constants $c > 0$ and n_0 such that for all $n > n_0$,

$$f(n) \geq c \cdot g(n).$$

$f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

Asymptotic Analysis Problems

1. For each of the following functions, prove whether $f = O(g)$, $f = \Omega(g)$, or $f = \Theta(g)$. For example, by specifying some explicit constants n_0 and $c > 0$ such that the definition of Big-Oh, Big-Omega, or Big-Theta is satisfied.

(a)	$f(n) = n\log(n^3)$	$g(n) = n\log n$
(b)	$f(n) = 2^{2n}$	$g(n) = 3^n$
(c)	$f(n) = \sum_{i=1}^{n} \log i$	$g(n) = n\log n$

(a) $f(n) \in \Theta(g(n))$. Since $f(n) = n\log(n^3) = 3n\log n$. To prove big O, choose any n_0 and any c ≥ 3 (for example $c = 4$), then we see $f(n) = 3n\log n \leq 4n\log n = cg(n) \quad \forall n \geq n_0$. To prove big Omega, choose any $0 <$ c ≤ 3 (for example $c = 2$, then $f(n) = 3n\log n \geq 2n\log n = cg(n) \quad \forall n \geq n_0$ for any n_0 of your choice.

(b) $f(n) \in \Omega(g(n))$. Notice $f(n) = 2^{2n} = 4^n$. Choose $c = 1, n_0 = 1$, then $f(n) = 4^n \geq 1 \times 3^n = cg(n) \quad \forall n \geq n_0$. To disprove Big O, use contradiction. Assume for the sake of contradiction

that f=O(g). Therefore there exists some c,n_0 such that $\forall n \geq n_0$:

$$4^n \leq c3^n$$
$$n \log 4 \leq \log c + n \log 3$$
$$n \leq \frac{\log c}{\log 4 - \log 3}$$

Now for any c and n_0, lets consider a large value n > $n_0 + \frac{\log c}{\log 4 - \log 3}$. We have reached a contradiction since we showed we must have $n \leq \frac{\log c}{\log 4 - \log 3}$, so $f \neq O(g)$

(c) Inspecting the summation we notice

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + ... + \log n$$
$$\sum_{i=1}^{n} \log i \leq \log n + \log n + \log n + ... + \log n$$
$$\sum_{i=1}^{n} \log i \leq n \log n$$

So for any choice of c,n_0 such as c=1 and n_0=1, $\sum_{i=1}^{n} \log i \leq n \log n$, which proves Big-Oh.

In order to prove Big-Omega, inspect the summation again. (In the first step we omit terms < log(n/2) and replace larger terms with log(n/2)).

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + ... + \log(\frac{n}{2}) + ... + \log n$$
$$\sum_{i=1}^{n} \log i \geq \log(\frac{n}{2}) + \log(\frac{n}{2}) + ... + \log(\frac{n}{2})$$
$$\sum_{i=1}^{n} \log i \geq \frac{n}{2} \log(\frac{n}{2}) = \frac{n}{2}(\log(n) - \log(2))$$

So to prove $\sum_{i=1}^{n} \log i = \Omega(nlogn)$ we need for all $n > n_0$:

$$\sum_{i=1}^{n} \log i \geq \frac{n}{2}(\log(n) - \log(2)) \geq cn \log n$$
$$(1 - 2c) \log n \geq 1$$

which holds if we pick $c_0 = \frac{1}{4}, n_0 = 4$

2. Give an example of f, g such that f is not in $O(g)$ and g is not in $O(f)$.

There are many such examples. Here is one:

$$f(n) = n.$$

$$g(n) = \begin{cases} 1 & \text{if } n \text{ is odd} \\ n^2 & \text{if } n \text{ is even} \end{cases}.$$

Induction

Snowball Fight

On a flat ice sheet, an *odd* number of penguins are standing such that their pairwise distances to each other are all different. At the strike of dawn, each penguin throws a snowball at the penguin closest to them. Show that there is always some penguin that doesn't get hit by a snowball.

We proceed using induction. (Note: the following induction proof structure is what we would expect for a homework or exam induction proof problem).

- **Inductive hypothesis.** In a group of $2n + 1$ penguins, there is some penguin that doesn't get hit by a snowball. (Note: it's important to see here that our "variable" that we're performing induction on is the n in the $2n + 1$ expression).
- **Base case.** We let $n = 0$. In a group with $2(0) + 1 = 1$ penguins, we want to show that there is some penguin that doesn't get hit by a snowball. Clearly, the single penguin in this group is not going to be hit by any snowball.
- **Inductive step.** Let $n = k$, where $k \geq 0$ and is an integer. We assume that our inductive hypothesis holds for k, so in a group of $2k + 1$ penguins, we know that there is always some penguin that doesn't get hit by a snowball. We want to show that in a group of $2(k + 1) + 1 = 2k + 3$ penguins, there is still going to be some penguin that doesn't get hit by a snowball.

  Consider the two penguins that are closest to each other (i.e. the two penguins with the minimum pairwise distance). We know that these two penguins must throw snowballs at each other at dawn. Aside from these two penguins, we're left with the remaining $2k + 1$ penguins, and none of these $2k + 1$ penguins would be hit by any of the original two penguins we considered (since those two throw at each other). By our inductive hypothesis, we know that among the remaining $2k + 1$ penguins, some penguin must not be hit. Thus, considering all penguins together, we know that with $2k + 3$ penguins, there must be some penguin doesn't get hit by a snowball.
- **Conclusion.** By induction, we conclude that the inductive hypothesis holds for all integers $n \geq 0$, i.e. for any odd number of penguins, there will always be some penguin that doesn't get hit by a snowball.

Divide and Conquer

Maximum Sum Subarray

Given an array of integers $A[1..n]$, find a contiguous subarray $A[i, ..j]$ with the maximum possible sum. The entries of the array might be positive or negative, and assume there is at least one element in the array.

1. What is a brute force solution, and what would be its runtime?
2. The maximum sum subarray may lie entirely in the first half of the array, or it may lie entirely in the second half. What is the third and only other possible case?

3. Using the above, apply divide and conquer to arrive at a more efficient algorithm.
   (a) What is the algorithm?
   (b) Prove its correctness (Hint: use strong induction).
   (c) What is the runtime?
4. Advanced (Take Home) - Can you do even better using other non-recursive methods? ($O(n)$ is possible)

1. The brute force approach involves summing up all possible $O(n^2)$ subarrays and finding the max amongst them for a total run time of $O(n^3)$. We can optimize this by pre-computing the running sums for the array so that we can find the sum of each subarray in $O(1)$ giving us a total run time of $O(n^2)$
2. The maximum sum subarray can also overlap both halves; in other words it passes through the middle element.
3. (a) Here's an English description of the algorithm:
   - If length is one, we return the array if the element is positive, and otherwise return [ ].
   - Next, divide the array in two, `L` and `R`, and recurse on each half to find the max-sum subarray of `L` and the max-sum subarray of `R`. Call these `maxLeft` and `maxRight`.
   - The best subarray of the third type will be the concatenation best subarray that *ends* at the midpoint and the best subarray that *starts* at the midpoint. Call this `maxAcross`.
   - Once we have these three subarray candidates, we return the max-sum subarray among these three types as our final answer.

   (b) We proceed using proof by strong induction.
   - **Inductive Hypothesis.** Given an array of n integers, our algorithm will return a contiguous subarray with the maximum possible sum.
   - **Base Case.** Let $n = 1$, which means we have an array with one integer. Our algorithm will return a contiguous array of maximum possible sum. If the single element is positive, the entire array will be returned. If the single element is non-positive, then the empty subarray has the maximum possible sum (0). Thus, the base case is satisfied.
   - **Inductive Step.** We assume our inductive hypothesis is true for any n where $1 \leq n < k$. We want to show that for an array with k integers, our algorithm will still return a contiguous subarray with the maximum possible sum.

     Since the left and right subarrays have length less than k, we can apply our inductive hypothesis, so we know that `maxLeft` and `maxRight` must be the contiguous subarray with the max possible sum from their respective halves of the original array. The third candidate, `maxAcross`, is defined as the maximum-sum contiguous subarray that crosses the midpoint. Thus, our three candidate subarrays are each the best subarrays of their respective types. Ultimately, given an array of size k, our algorithm returns the candidate with the maximum possible sum.
   - **Conclusion.** By strong induction, we conclude that the inductive hypothesis holds for any array with size $n \geq 1$, i.e. given an array of n integers, our algorithm will return a contiguous subarray with the maximum possible sum.

   (c) The runtime is $O(n \log n)$. We can examine the recursion tree and notice that we branch into 2 half-sized subproblems at each step, i.e. at level t, we have 2^t subproblems with size $\frac{n}{2^t}$. The work done per level is the number of subproblems at a level multiplied with the work done per subproblem, and in this case, our analysis resembles the MergeSort runtime

analysis shown in Lecture 2 ($O(n)$ work per level and $\log_n + 1$ levels total $= O(n \log n)$ total runtime).

(Optional Bonus Problem) Toom-Cook Multiplication

In this problem we will explore ideas that help improve the $O(n^{1.58})$ Karatsuba multiplication algorithm.

Suppose we want to compute the product of two n-digit numbers, x and y. In Karatsuba multiplication, we split each number into two $n/2$-digit numbers, but this time we will split them into three $n/3$-digit numbers. In particular, suppose that we write

$$x = a_2 \cdot 10^{2n/3} + a_1 \cdot 10^{n/3} + a_0 \qquad \text{and} \qquad y = b_2 \cdot 10^{2n/3} + b_1 \cdot 10^{n/3} + b_0.$$

Then

$$\begin{aligned} xy &= (a_2 \cdot 10^{2n/3} + a_1 \cdot 10^{n/3} + a_0)(b_2 \cdot 10^{2n/3} + b_1 \cdot 10^{n/3} + b_0) \\ &= (a_2 b_2) \cdot 10^{4n/3} + (a_2 b_1 + a_1 b_2) 10^n + (a_2 b_0 + a_1 b_1 + a_0 b_2) 10^{2n/3} + (a_1 b_0 + a_0 b_1) 10^{n/3} + (a_0 b_0) \end{aligned}$$

Thus, we can create a recursive multiplication algorithm if we can compute the five coefficients above using some $n/3$-digit multiplications. Given this,

(a) Find a way to compute the coefficients using nine $n/3$-digit multiplications. What is the runtime of the corresponding algorithm?

(b) Find a way to compute the coefficients using seven $n/3$-digit multiplications. What is the runtime of the corresponding algorithm?

(c) How many $n/3$-digit multiplications can you afford if you want to beat Karatsuba?

(d) **(Out of scope of this class.)** Find a way to compute the coefficients using six $n/3$-digit multiplications. What is the runtime of the corresponding algorithm?

(e) **(Even more out of scope of this class.)** Find a way to compute the coefficients using five $n/3$-digit multiplications. What is the runtime of the corresponding algorithm?

(a) We can compute each of the $a_i b_j$ terms individually, and this gives us nine multiplications in total.

We can figure out the runtime by looking at the recursion tree. At level ℓ, the problems are of size $n/3^\ell$, and there are 9^ℓ problems at that level. This means that when $\ell = \log_3 n$, there are $9^{\log_3 n}$ problems of size 1. This means that the overall runtime is $O(9^{\log_3 n}) = O(n^{\log_3 9}) = O(n^2)$.

(b) In the spirit of Karatsuba if we compute $(a_2 + a_1 + a_0)(b_2 + b_1 + b_0)$, we can get away with not computing one of the 5 coefficients. We choose not to compute the most painful one – the coefficient of $10^{2n/3}$. Explicitly, we compute

$$(a_2 + a_1 + a_0)(b_2 + b_1 + b_0), a_2 b_2, a_2 b_1, a_1 b_2, a_1 b_0, a_0 b_1, a_0 b_0.$$

The analysis is the same as part (a) with 9 replaced by 7. This gives us the runtime $O(n^{\log_3 7}) = O(n^{1.77})$.

(c) Clearly 7 wasn't enough. If we find a way that uses k multiplications, then the runtime will be $O(n^{\log_3 k})$. Thus, to beat Karatsuba, we need

$$\log_3 k < \log_2 3 \implies k < 3^{\log_2 3} = 5.70.$$

This means that if we want to get an improvement, we need to find a way that uses 5 multiplications.

(d) We improve incrementally by trying to find a better way to compute the two 2 term coefficients. The a_ib_j pairs in these terms only have the indexes 1 and 2 or 0 and 1, so this suggests that we may benefit from computing $(a_2 + a_1)(b_2 + b_1)$ and $(a_1 + a_0)(b_1 + b_0)$. Indeed, we can compute $a_2b_1 + a_1b_2$ if we have $(a_2 + a_1)(b_2 + b_1)$, a_1b_1, and a_2b_2. Similarly, we can compute $a_1b_0 + a_0b_1$ if we have $(a_1 + a_0)(b_1 + b_0)$, a_1b_1 and a_0b_0. Conveniently, they both need a_1b_1, and we need a_2b_2 and a_0b_0 anyway, so we can compute everything we need if we have

$$(a_2 + a_1 + a_0)(b_2 + b_1 + b_0), (a_2 + a_1)(b_2 + b_1), (a_1 + a_0)(b_1 + b_0), a_2b_2, a_1b_1, a_0b_0.$$

This gives us six $n/3$-digit multiplications. The runtime is is $O(n^{\log_3 6}) = O(n^{1.63})$. Even though this is *much* more complicated than Karatsuba, it's still worse.

(e) This exposition is just for those who are interested. By no means is anyone expected to know/fully understand this, but it is quite elegant because it uses one of the most underrated theorems of algebra: that any $k+1$ points of a degree k polynomial are enough to determine what the polynomial is.

First, note that we can view the product of xy as the following polynomial of degree 4 in the variable t, computed at $t = 10^{n/3}$:

$$(a_2 \cdot t^2 + a_1 \cdot t + a_0)(b_2 \cdot t^2 + b_1 \cdot t + b_0) =$$

$$= (a_2b_2) \cdot t^4 + (a_2b_1 + a_1b_2)t^3 + (a_2b_0 + a_1b_1 + a_0b_2)t^2 + (a_1b_0 + a_0b_1)t + (a_0b_0)$$

We know that we can determine polynomials of degree k by having $k+1$ points of the polynomial (two points determine a line, 3 points determine a parabola etc.). Hence, if we compute the polynomial at 5 points, we should be able to figure out the polynomial, and by extension, compute it at $t = 10^{n/3}$. Suppose we choose $t = 0, 1, -1, 2, -2$. We then compute:

$$a_0b_0, (a_2 + a_1 + a_0)(b_2 + b_1 + b_0), (a_2 - a_1 + a_0)(b_2 - b_1 + b_0),$$

$$(4a_2 + 2a_1 + a_0)(4b_2 + 2b_1 + b_0), (4a_2 - 2a_1 + a_0)(4b_2 - 2b_1 + b_0).$$

The formulas for retrieving the coefficients are a little complicated, but the fact that we can determine a degree 4 polynomial with 5 points means that it is possible. Note that instead of the last term we can actually compute just a_2b_2, since in some sense this corresponds to computing the polynomial at ∞. This makes the computations a little simpler.

A natural question is, how far can we take this? What if we split into k parts instead of 3? We might expect that instead we get polynomials of degree $2(k-1)$, and so we need $2k-1$ evaluation points, and so perhaps we get a runtime of $O(n^{\log_k(2k-1)})$. Since $\log_k(2k-1)$ goes to 1 as k goes to infinity, this seems great! However, As we increase k, the time we spend on additions and subtractions becomes more and more of a problem, and eventually can no longer be ignored by big-Oh. The actual best runtime of this kind of scheme was an open research question as late as 2005.