Asymptotic Analysis

Asymptotic Analysis Definitions

[Big-O 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 $n \geq n_0$,

\[ f(n) \leq c \cdot g(n). \]

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

\[ f(n) \geq c \cdot g(n). \]

$f = \Theta(g)$ 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-O, Big-Omega, or Big-Theta is satisfied.

\begin{align*}
(a) & \quad f(n) = n \log(n^3) \quad g(n) = n \log n \\
(b) & \quad f(n) = 2^{2n} \quad g(n) = 3^n
\end{align*}

(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 $c \geq 3$ (for example $c = 4$), then we see $f(n) = 3n \log n \leq 4n \log n = cg(n) \ \forall n \geq n_0$. To prove big Omega, choose any $0 < c \leq 3$ (for example $c = 2$, then $f(n) = 3n \log n \geq 2n \log n = cg(n) \ \forall n \geq n_0$ 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 $f(n) = 4^n \geq 1 \times 3^n = cg(n) \ \forall n \geq n_0$. 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 $n_0$, let's 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 $f \neq O(g)$.

**Master Theorem**

Recall the Master theorem from lecture:

**Theorem 0.1** Given a recurrence $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ with $a \geq 1$, and $b > 1$ and $T(1) = \Theta(1)$, then

$$T(n) = \begin{cases} 
O(n^d \log n) & \text{if } a = b^d 
\end{cases}$$

$$T(n) = \begin{cases} 
O(n^d) & \text{if } a < b^d 
\end{cases}$$

$$T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases}$$

What is the Big-O runtime for algorithms with the following recurrence relations?

1. $T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n^2)$
2. $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$

1. Using the Master Theorem, $a = 3, b = 2$, and $d = 2$. Since $a = 3 < b^d = 4$, we fall into the second case. So, the runtime is $O(n^d) = O(n^2)$.

2. Using the Master Theorem, $a = 4, b = 2$, and $d = 1$. Since $a = 4 > b^d = 2$, we fall into the third case. So the runtime is $O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)$.

**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 $2(0) + 1 = 1$ penguin, 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) Rough Pseudocode:

```python
def maxSubarray(A):
    # Base Case
    if length(A) == 1 and A[0] > 0:
        return A
    else if length(A) == 1 and A[0] <= 0:
        return []
    # Recursive case
    L = maxSubarray(A[ : n//2])
    R = maxSubarray(A[n//2 : ])

    Center_left = Maximum subarray that ends at index n/2
    Center_right = Maximum subarray that starts at index n/2
    C = Concatenate(Center_left,Center_right)

    return which of C, R, or L that has the maximum sum
```

(c) 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 \( \text{maxLeft} \) and \( \text{maxRight} \) must be the contiguous subarray with the max possible sum from their respective halves of the original array. The third candidate, \( \text{maxAcross} \), 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.

(d) 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 \( (O(n) \text{ work per level and } \log n + 1 \text{ levels total } = O(n \log n) \text{ total runtime}) \).
BONUS (optional extra practice)

1. Prove whether \( f = O(g) \), \( f = \Omega(g) \), or \( f = \Theta(g) \).

\[
f(n) = \sum_{i=1}^{n} \log i \quad g(n) = n \log n
\]

2. What is the Big-O runtime for algorithms with the following recurrence relation?

\[
T(n) = 2T(\sqrt{n}) + O(\log n)
\]

1. 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-O.

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 \( \sum_{i=1}^{n} \log i = \Omega(n \log n) \) 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. In order to solve this question, we must use a substitution trick. Define \( k = \log n \), so \( n = 2^k \), and \( \sqrt{n} = 2^\frac{k}{2} \). So the recurrence relation is:

\[
T(2^k) = 2T(2^\frac{k}{2}) + O(k)
\]
Next, let \( S(k) = T(2^k) \) so \( S\left(\frac{k}{2}\right) = T(2^{\frac{k}{2}}) \):

\[
S(k) = 2S\left(\frac{k}{2}\right) + O(k)
\]

Using master theorem, we get

\[
S(k) = O(k^d \log k) = O(k \log k) = O(\log n \log(\log n))
\]