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:

\[ f(n) = O(g(n)) \text{ if there exist constants } c > 0 \text{ and } n_0 \text{ such that for all } n > n_0, \]
\[ f(n) \leq c \cdot g(n). \]

\[ f(n) = \Omega(g(n)) \text{ if there exist constants } c > 0 \text{ and } n_0 \text{ such that for all } n > n_0, \]
\[ f(n) \geq c \cdot g(n). \]

\[ f(n) = \Theta(g(n)) \text{ if } f = O(g) \text{ and } f = \Omega(g). \]

You will use “big-oh notation” A LOT in class. Additionally, you may occasionally run into “little-oh notation”:

**Definition 2** (Little-oh notation). Let \( f, g \) be functions from the positive integers to the non-negative reals. Then we say that:

\[ f(n) = o(g(n)) \text{ if for every constant } c > 0 \text{ there exist a constant } n_0 \text{ such that for all } n > n_0, \]
\[ f(n) < c \cdot g(n). \]

\[ f(n) = \omega(g(n)) \text{ if for every constant } c > 0 \text{ there exist a constant } n_0 \text{ such that for all } n > n_0, \]
\[ f(n) > c \cdot g(n). \]

Asymptotic Analysis Problems

1. For each of the following functions, prove whether \( f(n) = O(g(n)) \), \( f(n) = \Omega(g(n)) \), or \( f(n) = \Theta(g(n)) \).

For example, you can prove by specifying some explicit constants \( n_0 \) and \( c > 0 \) such that the definition of big-oh or big-omega is satisfied. **Bonus: prove little-oh and little-omega (where applicable).**

\[ (a) \quad f(n) = n \log(n^3) \quad g(n) = n \log n \]
\[ (b) \quad f(n) = 2^{2n} \quad g(n) = 3^n \]
\[ (c) \quad f(n) = \sum_{i=1}^{n} \log i \quad g(n) = n \log n \]
(a) \( f(n) \in \Theta(g(n)) \). Since \( f(n) = n \log(n^3) = 3n \log n \). To prove big-ooh, chose any \( c \) above 3 (for example \( c = 4 \)), then \( f(n) = 3n \log n \leq 4n \log n = cg(n) \ \forall n \geq n_0 \) of any \( n_0 \) of your choice. To prove big Omega, chose any \( c \) below 3 (for example \( c = 2 \), then \( f(n) = 3n \log n \geq 2n \log n = cg(n) \ \forall n \geq n_0 \) of any \( n_0 \) of your choice.

(b) \( f(n) \in \Omega(g(n)) \). Since \( f(n) = 2^{2n} = 4^n \). Chose \( c = 1, n_0 = 1 \), then \( f(n) = 4^n \geq 1 \times 3^n = cg(n) \ \forall n \geq n_0 \). To disprove Big O, use contradiction.

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

So pick any \( c \) and \( n > \frac{\log c}{\log 4 - \log 3} \) We have a contradiction.

(c) Inspect summation

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

So we have proven Big-Oh

In order to prove Big-Omega, inspect summation again

\[
\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + ... + \log \left(\frac{n}{2}\right) + ... + \log n \\
\sum_{i=1}^{n} \log i \geq \log \left(\frac{n}{2}\right) + \log \left(\frac{n}{2}\right) + ... + \log \left(\frac{n}{2}\right) \\
\sum_{i=1}^{n} \log i \geq \frac{n}{2} \log \left(\frac{n}{2}\right) = \frac{n}{2} (\log(n) - \log(2))
\]

So

\[
\frac{n}{2} (\log(n) - \log(2)) \geq cn \log n
\]

\[
(1 - 2c) \log n \geq 1
\]

Now we can pick \( c_0 = \frac{1}{4} \), \( n_0 = 4 \)

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

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

How NOT to prove claims by induction

In this class you’ll prove a lot of claims, many of them by induction. You will also prove some wrong claims, and catching those mistakes will be an important skill!

The following are examples of a false proof where an obviously untrue claim has been ‘proven’ using strong induction with some minor error or detail left out. Your task is to investigate the ‘proofs’ and identify the mistakes made.

1. **Fake Claim 1.** For every nonnegative integer \( n \), \( 2^n = 1 \).
   
   Inductive Hypothesis: for all integers \( n \) such that \( 0 \leq n \leq k \), \( 2^n = 1 \).

   Base Case: For \( n = 0 \), \( 2^0 = 1 \).

   Inductive Step: Suppose the inductive hypothesis holds for \( k \); we will show that it is also true \( k + 1 \), i.e. \( 2^{k+1} = 1 \). We have
   \[
   2^{k+1} = \frac{2^{2k}}{2^{k-1}} = \frac{2^k \cdot 2^k}{2^{k-1}} = \frac{1 \cdot 1}{1} \quad \text{(by strong induction hypothesis)}
   \]
   Conclusion: By strong induction, the claim follows

   The error in this proof occurs in the inductive step. In order for the inductive hypothesis to apply to the denominator, the exponent, \( k - 1 \), must be a nonnegative integer. This requires the implicit assumption that \( k \geq 1 \). The inductive step must hold for \( k = 0 \), so the assumption that \( k \geq 1 \) is invalid and the inductive step fails.

2. **Fake Claim 2.** For every positive integer \( n \),
   \[
   \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \ldots + \frac{1}{n \text{ terms}} = \frac{3}{2} - \frac{1}{n}. \quad (1)
   \]

   Inductive Hypothesis: (1) holds for \( n = k \)

   Base Case: For \( n = 1 \),
   \[
   \frac{1}{1 \cdot 2} = 1/2 = \frac{3}{2} - \frac{1}{1}.
   \]
Inductive Step: Suppose the inductive hypothesis holds for $n = k$; we will show that it is also true $n = k + 1$. We have

$$\left(\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \cdots + \frac{1}{(k-1) \cdot k}\right) + \frac{1}{k \cdot (k+1)} = \frac{3}{2} - \frac{1}{k} + \frac{1}{k \cdot (k+1)}$$

(by weak induction hypothesis)

$$= \frac{3}{2} - \frac{1}{k} - \frac{1}{k+1}$$

$$= \frac{3}{2} - \frac{1}{k+1}.$$

Conclusion: By weak induction, the claim follows

The first part of the long derivation of the inductive step is wrong — the summation in the parentheses only contains $k - 1$ summands!

Binary Search

Given a sorted array $A$, prove that binary search as defined below correctly returns the index of the element $x$ in $A$ if it appears in $A$ (or None otherwise).

```python
def binarySearch(x, A, l, r):
    if l > r:
        return None
    m = (l + r) / 2
    if x == A[m]:
        return m
    if x < A[m]:
        return binarySearch(x, A, l, m-1)
    else:
        return binarySearch(x, A, m+1, r)
```

The first call to the algorithm is: `binarySearch(x, A, 0, len(A) - 1)`.

We prove the following claim: for any $l, r$, if $x$ appears in $A$ between index $l$ and index $r$ (including $l$ and $r$), then calling `binarySearch(x, A, l, r)` will return the index of $x$ in $A$ (and None otherwise). (We say that an element appears between $l$ and $r$ when there is an index $i$ such that $l \leq i \leq r$ and $A[i] = x$.)

For a given $l, r$, define $n$ to be the number of elements between indices $l$ and $r$: if $l \leq r$, $n = r - l + 1$, and otherwise $n = 0$. We prove the claim by induction on the number of elements between indices $l$ and $r$, that is, by induction on $n$.

- **Base case.** Whenever $n = 0$, by definition $l > r$. In that case, there are no elements between indices $l$ and $r$, and the algorithm correctly returns None.

- **Inductive hypothesis.** For any $l', r'$ such that there are at most $n - 1$ elements between indices $l'$ and $r'$, calling `binarySearch(x, A, l, r)` will return the index of $x$ in $A$ if it is between $l'$ and $r'$ (and None otherwise).

- **Inductive step.** In each call to `binarySearch`, we have three cases:
  - If $x = A[m]$, then the algorithm correctly returns $m$. 

– If \( x < A[m] \), then we know that \( x \in A[l..r] \) if and only if it is in \( A[l..m - 1] \) since the array is sorted. By the inductive hypothesis, we know \( \text{binarySearch}(x, A, l, m - 1) \) returns the index of \( x \) correctly, or None if it does not appear in the array.

– If \( x > A[m] \), then we similarly know that \( x \in A[l..r] \) if and only if it is in \( A[m + 1..r] \). By the inductive hypothesis, we know \( \text{binarySearch}(x, A, m + 1, r) \) returns the index of \( x \) correctly, or None if it does not appear in the array.

We can now conclude the proof. If \( x \) appears in \( A \), it must appear between the indices 0 and \( \text{len}(A) - 1 \). By the claim above, binarySearch will return the index of \( x \) if it appears in \( A \) and None otherwise.