Interview Question from before

The Question: Design an algorithm to return the largest sum of contiguous integers in an array of ints.

Example: if the input is \((-10, 2, 3, -2, 0, 5, -15)\), the largest sum is 8, which we get from \((2, 3, -2, 0, 5)\).

A Naive Algorithm

MaxSeq1 (int arr[], int n)
int max = 0;
for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
        int sum = 0;
        for (int k = i; k <= j; k++)
            sum += arr[k];
        if (sum > max)
            max = sum;
return max;
Naive Algorithm

- Analysis from last time showed this takes $O(n^3)$ steps
- Worst case and best case is the same
- Can we do better?

A Better Algorithm

MaxSeq2 (int arr[], int n)
int max = 0;
for (int i = 1; i <= n; i++)
    int sum = 0;
    for (int j = i; j <= n; j++)
        sum += arr[j];
        if (sum > max)
            max = sum; //and store i and j if desired
return max;

Analysis of MaxSeq2

- Let $f(n)$ be the number of operations this algorithm performs on an array of size $n$. Then:

$$f(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} 1$$

(1)

$$= \sum_{i=1}^{n} (n - i + 1)$$

(2)

$$= \sum_{i=1}^{n} i$$

(3)

$$= (n + 1)(n/2)$$

(4)

$$= O(n^2)$$

(5)

Challenge

- MaxSeq2 is much better than MaxSeq1 ($O(n^2)$ vs $O(n^3)$)
- But it’s still not great, can you do better?
Beyond Big-O

- Both MaxSeq1 and MaxSeq2 have same best case and worst case behavior.
- We can say more about them than big-O time.
- I.e. We can say that each has run time approx "=" to some amount; We can also say that MaxSeq2 is approx "<" MaxSeq1.
- \( O \) is an asymptotic analogue to "\( \leq \)", but we’d also like analogues to "\(<\", \( \geq \)" and "\( >\)."

Formal Defns

- \( O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \} \)
- \( \Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \} \)
- \( \Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \} \)

Formal Defns (II)

- \( o(g(n)) = \{ f(n) : \text{for any positive constant } c > 0 \text{ there exists } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \} \)
- \( \omega(g(n)) = \{ f(n) : \text{for any positive constant } c > 0 \text{ there exists } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0 \} \)

 Relatives of big-O

When would you use each of these? Examples:

- \( O \) "\( \leq \)" This algorithm is \( O(n^2) \) (i.e. worst case is \( \Theta(n^2) \))
- \( \Theta \) "\( \equiv \)" This algorithm is \( \Theta(n) \) (best and worst case are \( \Theta(n) \))
- \( \Omega \) "\( \geq \)" Any comparison-based algorithm for sorting is \( \Omega(n \log n) \)
- \( o \) "\( < \)" Can you write an algorithm for sorting that is \( o(n^2) \)?
- \( \omega \) "\( > \)" This algorithm is not linear, it can take time \( \omega(n) \)
Rule of Thumb

- Let \( f(n) \), \( g(n) \) be two functions of \( n \)
- Let \( f_1(n) \), be the fastest growing term of \( f(n) \), stripped of its coefficient.
- Let \( g_1(n) \), be the fastest growing term of \( g(n) \), stripped of its coefficient.

Then we can say:

- If \( f_1(n) \leq g_1(n) \) then \( f(n) = O(g(n)) \)
- If \( f_1(n) \geq g_1(n) \) then \( f(n) = \Omega(g(n)) \)
- If \( f_1(n) = g_1(n) \) then \( f(n) = \Theta(g(n)) \)
- If \( f_1(n) < g_1(n) \) then \( f(n) = o(g(n)) \)
- If \( f_1(n) > g_1(n) \) then \( f(n) = \omega(g(n)) \)

Useful Facts

Consider some functions \( f(n) \) and \( g(n) \) that are asymptotically nonnegative

- \( f(n) \) is \( o(g(n)) \) if \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \)
- \( f(n) \) is \( \omega(g(n)) \) if \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \)
- Note that \( f(n) \) is \( \omega(g(n)) \) if and only if \( g(n) \) is \( o(f(n)) \)

L’Hopital’s Rule

- Let \( f(n) \) and \( g(n) \) be differentiable functions that both go to infinity. Then by L’Hopital:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}
\]

More Examples

The following are all true statements:

- \( \sum_{i=1}^{n} i^2 \) is \( O(n^3) \), \( \Omega(n^3) \) and \( \Theta(n^3) \)
- \( \log n \) is \( o(\sqrt{n}) \)
- \( \log n \) is \( o(\log^2 n) \)
- \( 10,000n^2 + 25n \) is \( \Theta(n^2) \)
Problems

True or False? (Justify your answer)

- $n^3 + 4$ is $\omega(n^2)$
- $n \log n^3$ is $\Theta(n \log n)$
- $\log^3 5n^2$ is $\Theta(\log n)$
- $10^{-10} n^2 + n$ is $\Theta(n)$
- $n \log n$ is $\Omega(n)$
- $n^3 + 4$ is $o(n^4)$

Proofs

- To prove an asymptotic relationship between two functions $f(n)$ and $g(n)$ takes more effort
- To do this, we need to start with the formal definition of the relationship we are trying to establish
- In particular, we will need to show that there exist constants which satisfy the appropriate definitions

Example

Let $f(n) = 10 \log^2 n + \log n$, $g(n) = \log^2 n$. Let’s show that $f(n) = \Theta(g(n))$.

- We want positive constants $c_1, c_2$ and $n_0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$
- In other words, we want $c_1, c_2$ and $n_0$ such that:
  \[0 \leq c_1 \log^2 n \leq 10 \log^2 n + \log n \leq c_2 \log^2 n\]
  Dividing by $\log^2 n$, we get:
  \[0 \leq c_1 \leq 10 + 1/\log n \leq c_2\]
- If we choose $c_1 = 1$, $c_2 = 11$ and $n_0 = 2$, then the above inequality will hold for all $n \geq n_0$

Example 2

Let $f(n) = \log^a n$, $g(n) = n^b$ for any constants $a$ and $b > 0$. Let’s show that $f(n) = o(g(n))$.

- For any positive constant $c$, we want to show there is an $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.
- In other words, we want to show that there is $n_0 > 0$ such that
  \[0 \leq \log^a n \leq cn^b\]
  Dividing by $n^b$, we get:
  \[0 \leq \frac{\log^a n}{n^b} \leq c\]
- We know that $\lim_{n \to \infty} \frac{\log^a n}{n^b} = 0$ (by L’Hospital) so for any constant $c$, there must be a $n_0$ such that the above inequality is satisfied for all $n \geq n_0$. 
Example 3

Let \( f(n) \) be asymptotically positive and let \( g(n) = 10 \cdot f(n) \).
Let's show that \( f(n) = \Theta(g(n)) \)

- We must show that there are positive constants \( c_1, c_2 \) and \( n_0 \)
  such that:
  \[
  c_1 \cdot 10f(n) \leq f(n) \leq c_2 10f(n)
  \]
  Dividing through by \( f(n) \), we have
  \[
  c_1 10 \leq 1 \leq c_2 10
  \]
- If we choose \( c_1 = c_2 = 1/10 \) and \( n_0 = 1 \), then the above
  inequality is satisfied for all \( n \geq n_0 \)

In-Class Exercise

Let \( f(n) \) be an asymptotically positive function and let \( g(n) = f(n) \log n \).
Show that \( f(n) = o(g(n)) \)

- Write down exactly what needs to be shown to prove that
  \( f(n) = o(g(n)) \)
- Now solve for \( n_0 \) as a function of \( c \) in the above statement

Asymptotic Analysis - Take Away

- In studying behavior of algorithms, we’ll be more concerned with rate of growth than with constants
- \( O, \Theta, \Omega, o, \omega \) give us a way to talk about rates of growth
- Asymptotic analysis is an extremely useful way to compare run times of algorithms
- However, empirical analysis is also important (you’ll be studying this in your project)

Recurrence Relations

- Getting the run times of recursive algorithms can be challenging
- Recall our algorithm for binary search
- Let \( T(n) \) be the run time of this algorithm on an array of size \( n \)
- Then we can write \( T(1) = 1 \), \( T(n) = T(n/2) + 1 \)
Alg: Binary Search

```c
bool BinarySearch (int arr[], int s, int e, int key){
    if (e-s<=0) return false;
    int mid = (e-s)/2;
    if (arr[key]==arr[mid]){ return true;
    }else if (key < arr[mid]){ return BinarySearch (arr,s,mid,key);}
    else{ return BinarySearch (arr,mid,e,key) }
}
```

Recurrence Relations

“Oh how should I not lust after eternity and after the nuptial ring of rings, the ring of recurrence” - Friedrich Nietzsche, Thus Spoke Zarathustra

- $T(n) = T(n/2) + 1$ is an example of a recurrence relation
- A Recurrence Relation is any equation for a function $T(n)$, where $T$ appears on both the left and right sides of the equation.
- We always want to “solve” these recurrence relation by getting an equation for $T(n)$, where $T$ appears on just the left side of the equation

A Side Note

- The running time of an algorithm on a constant size input is always $\Theta(1)$
- Thus for convenience, we usually omit statements of the boundary conditions and just assume $T(n)$ is constant when $n$ is a constant.
- Example: Instead of saying “If $n = 1$, $T(n) = \theta(1)$, and if $T(n) = 2 \cdot T(n/2) + \Theta(n)$”, we just say “$T(n) = 2 \cdot T(n/2) + \Theta(n)$”

Todo

- Start hw2
- Sign up for the class mailing list (on class web page)
- Read Chapter 3 and 4 in the text