CS 361, Lecture 4

Jared Saia
University of New Mexico

Today's Outline

- Interview question improvement
- Formal definition of big-O analysis
- Intro to $\Theta, \Omega, o, \omega$
- Intro to correctness proofs

Important Log Facts

- $\log^x f = (\log f)^x$
- $\log xy = \log(xy)$ (multiplication binds the tightest)

Examples:

- $\log^2 n^5 = (\log n^5)^2 = 25(\log n)^2 = 25 \log^2 n$
- $\log 5n = \log 5 + \log n$

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 be the number of operations this algorithm performs. The:

$$
\begin{aligned}
f &= \sum_{i=1}^{n}\sum_{j=i}^{n} 1 && (1)\\
&= \sum_{i=1}^{n}(n-i+1) && (2)\\
&= \sum_{i=1}^{n} i && (3)\\
&= (n+1)(n/2) && (4)\\
&= O(n^2) && (5)
\end{aligned}
$$

Challenge

- MaxSeq2 is much better than MaxSeq1 ($O(n^2)$ vs $O(n^3)$)
- But it's still not great, can you do better?
- We'll come back to this when we do recurrences

Beyond Big-O

- Both MaxSeq1 and MaxSeq2 have same best case and worst case behavior
- In a sense, we can say more about them than big-O time
- I.e. we can say more than that their run time is approx "$\leq$" some amount
- Want a way of saying "assymptotically equal to"
- In general, want assymptotic analogues of $\leq$, $\geq$, $=$, etc.

Formal Defn of Big-O

- Before we go beyond big-O, what precisely does it *mean*?
- It has a precise, mathematical definition:
- *A function $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$*

Example

- Let's try to show that $f(n) = 10n + 100$ is $O(g(n))$ where $g(n) = n$
- We need to give constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$
- In other words, we need constants c and n_0 such that $10n + 100 \leq cn$ for all $n \geq n_0$

Example

- We can solve for appropriate constants:

$$10n + 100 \leq cn \qquad (6)$$
$$10 + 100/n \leq c \qquad (7)$$

- So if $n > 1$, then c need be greater than 110.
- In other words, for all $n > 1$, $10n + 100 \leq 110n$
- So $10n + 100$ is $O(n)$

Another Example

- Let's try to show that $f(n) = n^2 + 100n$ is $O(g(n))$ where $g(n) = n^2$
- We need to give constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$
- In other words, we need constants c and n_0 such that $n^2 + 100n \leq cn^2$ for all $n \geq n_0$

Another Example

- We can solve for appropriate constants:

$$n^2 + 100n \leq cn^2 \qquad (8)$$
$$1 + 100/n \leq c \qquad (9)$$

- So if $n > 1$, then c need be greater than 101.
- In other words, for all $n > 1$, $n^2 + 100n \leq 101n^2$

Relatives of big-O

Following are some of the relatives of big-O:

O	"$\leq$"
Θ	"$=$"
Ω	"$\geq$"
o	"$<$"
ω	"$>$"

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 $\Theta(n^2)$)
Θ	"$=$"	This algorithm is $\Theta(n)$ (best and worst case are $\Theta(n)$)
Ω	"$\geq$"	Any algorithm for sorting is worst case $\Omega(n \log n)$
o	"$<$"	Can you write an algorithm for sorting that is $o(n^2)$?
ω	"$>$"	This algorithm is not linear, it can take time $\omega(n)$

Formal Defns

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

Formal Defns (II)

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

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

Examples

- Worst case time of linear search is $\Theta(n)$
- Worst case time of linear search is $\Omega(\log n)$
- Best case time of linear search is $\Theta(1)$
- Worst case time of binary search is $o(n)$
- Worst case time of binary search is $\Theta(\log n)$

More Examples

The following are all true statements:

- From last lecture, $\sum_{i=1}^{n} i^2$ 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 $\Theta(n^2)$

In Class Exercise

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 $o(n^4)$

(These equations represent run times of algorithms)

Asymptotic Analysis - Take Away

- In studying behavior of algorithms, we'll 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)

Correctness of Algorithms

- Another important aspect of algorithms is their correctness
- An algorithm by definition *always* gives the right answer to the problem
- A procedure which doesn't always give the right answer is a *heuristic*
- All things being equal, we prefer an algorithm to a heuristic
- How do we prove an algorithm is really correct?

Loop Invariants

Must useful tool is loop invariants. Three things must be shown about a loop invariant

- **Initialization:** Invariant is true before first iteration of loop
- **Maintenance:** If invariant is true before iteration i, it is also true before iteration $i+1$ (for any i)
- **Termination:** When the loop terminates, the invariant gives a property which can be used to show the algorithm is correct

Example Loop Invariant

- We'll prove the correctness of a simple algorithm which solves the following interview question:
- *Find the middle of a linked list, while only going through the list once*
- The basic idea is to keep two pointers into the list, one of the pointers moves twice as fast as the other
- (Call the head of the list the 0-th elem, and the tail of the list the $(n-1)$-st element, assume that $n-1$ is an even number)

Example Algorithm

```
  GetMiddle (List l){
    pSlow = pFast = l;
    while ((pFast->next)&&(pFast->next->next)){
      pFast = pFast->next->next
      pSlow = pSlow->next
    }
    return pSlow
}
```


Example Loop Invariant

- *Invariant: At the start of the i-th iteration of the while loop, pSlow points to the i-th element in the list and pFast points to the $2i$-th element*
- **Initialization:** True when $i = 0$ since both pointers are at the head
- **Maintenance:** if pSlow, pFast are at positions i and $2i$ respectively before i-th iteration, they will be at positions $i+1$, $2(i+1)$ respectively before the $i+1$-st iteration
- **Termination:** When the loop terminates, pFast is at element $n-1$. Then by the loop invariant, pSlow is at element $(n-1)/2$. Thus pSlow points to the middle of the list

Challenge

- Figure out how to use a similar idea to determine if there is a loop in a linked list *without marking nodes!*

Todo

- Read Chapter 4 (Recurrences) in text