Today’s Outline

- Asymptotic Analysis review
- Why do we care?
- Logs
- An Interview Question

An Example

Consider the following (silly) algorithm:

Alg 1 (int n)
For i=1 to n
    For j=1 to i
        print "hi"

An Example (II)

First we write down the formula $f$ giving the number of basic operations the algorithm performs:

$$f = \sum_{i=1}^{n} i = (n + 1)n/2$$

Next we compute the big-O value for $f$:

$(n + 1)n/2$ is $O(n^2)$

We can then say that Alg1 takes $O(n^2)$ time. Or, for short, we just say Alg1 is $O(n^2)$

What is Asymptotic Analysis?

- Asymptotic notation is used to give a rough estimate of the rate of growth of a formula. The formula usually gives the run time of an algorithm
- Informally, $O$ notation is the leading (i.e. quickest growing) term of a formula with the coefficient stripped off
- $O$ is sort of a relaxed version of $\leq$
Examples from last class

Following are some formulas that represent the number of operations of some algorithm. Give the big-O notation for each.

- E.g. \( n, 10,000n - 2000, \) and \(.5n + 2 \) are all \( O(n) \)
- \( n + \log n, n - \sqrt{n} \) are \( O(n) \)
- \( n^2 + n + \log n, 10n^2 + n - \sqrt{n} \) are \( O(n^2) \)
- \( n \log n + 10n \) is \( O(n \log n) \)
- \( 10 \times \log^2 n \) is \( O(\log^2 n) \)
- \( n \sqrt{n} + n \log n + 10n \) is \( O(n \sqrt{n}) \)
- \( 10,000, 2^{50} \) and 4 are \( O(1) \)

Computing big-O of an Algorithm

Following is a shorter way to compute big-O for an algorithm:

- "Atomic operations" Constant time
- Consecutive statements Sum of times
- Conditionals Larger branch time plus test time
- Loops Sum of iterations
- Function Calls Time of function body
- Recursive Functions Solve Recurrence Relation

Alg: Binary Search

```cpp
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)}
}
```

Analysis of Binary Search

- Note that even in the worst case, the size of the array we search is being split in half in each call
- Thus if \( x \) is the number of recursive calls, and \( n \) is the original size of the array, \( n(1/2)^x = 1 \) in the worst case
- This implies that \( 2^x = n \)
- Taking log of both sides, we get \( x = \log n \)
- Since each invocation of the function takes \( O(1) \) time (minus the recursive calls), and the total number of invocations is at most \( \log n \), the running time is \( O(\log n) \)
- Much better than Linear Search

Alg: Linear Search

```cpp
bool LinearSearch (int arr[], int n, int key){
  for (int i=0;i<n;i++){
    if (arr[i]==key)
      return true;
  }
  return false;
}
```

Run Time?

Digression on Logs

Definition:

- \( \log_x y \) is by definition the value \( x \) such that \( x^z = y \)
- \( x^{\log_x y} = y \) by definition
Facts about exponents

Recall that:

- \((x^y)^z = x^{yz}\)
- \(x^{y+z} = x^y x^z\)

From these, we can derive some facts about logs

Facts about logs

To prove both equations, raise both sides to the power of 2, and use facts about exponents

- Fact 1: \(\log(xy) = \log x + \log y\)
- Fact 2: \(\log a^c = c \log a\)

Memorize these two facts

Incredibly useful fact about logs

- Fact 3: \(\log_c a = \log a / \log c\)

To prove this, consider the equation \(a = c^{\log_c a}\), take \(\log_2\) of both sides, and use Fact 2.  **Memorize this fact**

Log facts to memorize

- Fact 1: \(\log(xy) = \log x + \log y\)
- Fact 2: \(\log a^c = c \log a\)
- Fact 3: \(\log_c a = \log a / \log c\)

These facts are sufficient for all your logarithm needs. (You just need to figure out how to use them)

Logs and \(O\) notation

- Note that \(\log_8 n = \log n / \log 8\).
- Note that \(\log_{600} n^{200} = 200 \times \log n / \log 600\).
- Note that \(\log_{100000} 30^n n^2 = 20 \times \log n / \log 100000 + \log 30 / \log 100000\).
- Thus, \(\log_8 n, \log_{600} n^{200}, \text{and } \log_{100000} 30^n n^2\) are all \(O(\log n)\).
- In general, for any constants \(k_1\) and \(k_2\), \(\log_{k_1} n^{k_2} = k_2 \log n / \log k_1\), which is just \(O(\log n)\)

Take Away on Logs

- All log functions of form \(k_1 \log k_2 n^{k_3}\) for constants \(k_1, k_2\) and \(k_3\) are \(O(\log n)\).
- For this reason, we don’t really “care” about the base of the log function when we do asymptotic notation.
- Thus, binary search, ternary search and k-ary search all take \(O(\log n)\) time.
- Memorize the 3 log facts!
In Class Exercise

Simplify the following formulas, and then give the simplest possible \(O\) notation for each:

- \(\log 10 \times n^2\)
- \(\log^2 n^5\)
- \(\log \log n\)
- \(2^{\log_b n}\)

Does big-O really matter?

Let \(n = 100000\) and \(\Delta t = 1\mu s\)

<table>
<thead>
<tr>
<th>(\log n)</th>
<th>(1.2 \times 10^{-5}) seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sqrt{n})</td>
<td>(3.2 \times 10^{-4}) seconds</td>
</tr>
<tr>
<td>(n)</td>
<td>1 second</td>
</tr>
<tr>
<td>(n\log n)</td>
<td>1.2 seconds</td>
</tr>
<tr>
<td>(n\sqrt{n})</td>
<td>31.6 seconds</td>
</tr>
<tr>
<td>(n^2)</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>(n^3)</td>
<td>31.7 years</td>
</tr>
<tr>
<td>(2^n)</td>
<td>&gt; 1 century</td>
</tr>
</tbody>
</table>

(From Classic Data Structures in C++ by Timothy Budd)

Another Interview Question

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;

Analysis

- Need to count the total number of operations of MaxSeq1
- Might as well assume time to do the inner loop is 1 (since it’s a constant and therefore \(O(1)\))
- Let \(f\) be the number of ops, (recall \(\sum_{i=1}^x = (x/2)(x+1)\))

\[
f = \sum_{i=1}^n \sum_{j=1}^n (j - i) = \sum_{i=1}^n \sum_{j=i}^n (j - i)
\]

Analysis (II)

\[
f = \sum_{i=1}^n ((n - i)/2)(n - i + 1)
\]

\[
f = \sum_{i=1}^{n-1} (i/2)(i + 1)
\]

\[
f = 1/2 \sum_{i=1}^{n-1} (i^2 + i)
\]

\[
f = 1/2 \sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i
\]

\[
f = 1/2 \sum_{i=1}^{n-1} (O(n^3) + O(n^2))
\]

\[
f = O(n^3)
\]
Challenge

- MaxSeq1 is very slow
- This kind of algorithm won't impress an interviewer
- Can you do better?

Todo

- Finish Chapter 3 (Growth of Functions) in textbook