Algorithm Analysis Examples

Last gasp

Search

- Search Problem: Given an integer k and an array of integers A_0, A_1, A_2, A_3, A_4… A_{N-1} which are pre-sorted, find i such that $A_i = k$. (Return –1 if k is not in the list.)

- For example, {-32, 2, 3, 9, 45, 1002}. Given that k = 9, the program will return 3. i.e., the number 9 lives in the 3rd position.
  - Note: start counting positions from 0.

Brute Force Search

```
public int bruteForceSearch(int k, int[] array)
{
    for(int i=0; i<array.length; i++)
    {
        if(a[i] = = k)
        {
            return i;                    /*found it!*/
        }
    }
    return  –1;                          /*didn’t find, not in array*/
}
```

O(N), right?

Binary Search

1. Start in the middle of array.
2. If that is the correct number return.
3. If not, then check if the correct number is larger or smaller than the number in the current position.
4. Take correct half of the array and go to the middle.
5. Repeat.

Binary Search Example

1. Let's look for k = 54.
2. Start in middle of array

   11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83

3. Is 54 bigger than 41? Yes. So look in upper half of array.

   11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83

4. Is 54 bigger than 56? No. So take lower half of remaining array.

   11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83

Binary Search Example (cont.)

5. Is 54 bigger than 51? Yes, so take upper half of remaining array.

   11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83

6. And 51 is in the 9th position (starting from 0 … stupid array counting).

7. Note that we decreased the size of the search by roughly ½ each step.

So here's some code that will do this "binary search"…

Binary Search Code

```
public int binarySearch(int k, int[] array)
{
    int left = -1;
    int right = array.length;               /*left and right are the array bounds*/

    while(left+1 ! = right)                 /*stop when left and right meet */
    {
        int middle = (left+right)/2;        /*find the middle point*/

        if(k < array[middle])               /*in left half*/
            right = middle;                 /*new right is the old middle*/
        if(k == array[middle])              /*found it!*/
            return middle;                  /*new right is the old middle*/
        if(k > array[middle])               /*in right half*/
            left = middle;                  /*new left is the old middle*/
    }
    return  –1;                             /*didn’t find it.  Not in array*/
}
```

Binary Search Code (cont.)

- Ahhh, a “while” loop. So how many times does it iterate?
- Like “for” loops the Big-O answer is just the number of times through the loop times the most costly statement on the inside.
- Note: no recursion. (phew!)

Binary Search Code (cont. 2)

- With Big-O we are always looking for the worst case scenario.
- The worst case is that the array size has to be halved until we are down to an array size of 1 (just like the example).
- Example: Once through for size 32, then size 16, 8, 4, 2, 1.
- How many times through the loop?
- Just flip it around… 1, 2, 4, 8, 16, 32, …, 2^{i-1} where i is the number of times through the loop.

Binary Search Code (cont. 3)

- So the array size, $n = 2^{i-1}$.

  So $i = (\log(n)/\log(2)) + 1$.

- So the run time is $O(\log(n))$.
- Phew!
- And how does that compare to the BruteForceSearch algorithm which is $O(n)$?
- BinarySearch wins!

A Take Home Lesson

- If a loop is halved over and over or doubled over and over, it is O(log(N)).
  - Or $O(e^N)$ if it's a really bad algorithm.
    - e.g., recursive Fibonacci

- In fact, if a loop increases or decreases by a constant *multiplicative factor* each iteration, then it is O(log(N)).
  - Or $O(e^N)$ if it's a bad algorithm.

log(N) Example

```
for(int i = 1; i<n; i *= 37)
{
    total++;
}
```

Claim:

i increases by a factor of 37 each time, so takes log(N) time.

Proof:

$i = 1, 37, 37^2, 37^3, \ldots 37^{k-1}$ where 37^{k-1} is the last number that doesn't exceed n (k is the number of iterations). So $37^{k-1} \leq n$ which means $\log(37^{k-1}) \leq \log(n)$. Therefore, $k-1 \leq \log(n)/\log(37)$. So the *max* number of iterations is $k = (\log(n)/\log(37))+1$. Therefore the run time is $O(\log(n))$.

Linear Example

What about

```
for(int i = 0; i<n; i += 2)
{
    total++;
}
```

Increases by 2 each time, but **not** by a multiplicative factor of 2. So **not** log(n).

What is the run time? i = 0, 2, 4, 6, 8, … So this will run for n/2 iterations. So the run time is O(n) … throw away the constant!

Another Take Home Lesson

- When a loop increases or decreases by a *constant amount* each iteration, then its growth rate is O(N).

Example:

```
for(float x = 27.2; x > -n; x -= log(1.3))
{
    total++;
}
```

Don’t let the log fool you.
This is a constant!
log(1.3) = 0.1138

Just look At It…

- So just *glance* at this and tell me the run time.

```
for(int i = 1; i<n; i++)
{
    for(int j = 1; j<n; j*=2)
    {
        total++;
    }
}
```

And Again…

- Remember the binarySearch method is O(log(N)). So *glance* at the following and give me the run time.

```
for(int i = 1; i<n; i++)
{
    for(int j = 1; j<n; j+=2)
    {
        binarySearch(preSortedArray, j);
    }
}
```

yup… $N^2\log(N)$

And Yet Again…

- Figure this one out…

```
int counter = 1;
while(counter < n)
{
    binarySearch(preSortedArray, counter);
    counter *= 2;
}
```

hint: The while loop is just like a for loop (see next page). And it is growing by a *multiplicative factor* of 2.

So the while loop takes log(N) time and the binary search takes log(N) time so the run time is their product. $O((\log N)^2)$.

Convert While To For Loop

- If makes more sense, can *always* convert a while loop into a for loop.
  - The following are the same!

```
int j = 1;
while(j < n)
{
    …
    j *= 2;
}
```

```
for(int j = 1; j < n; j *= 2);
{
    …
}
```

How About This…

```
while(counter < n)
{
        if(n%2 == 0)
        {
              binarySearch(preSortedArray, counter);
        }
        else
        {
              bruteForceSearch (preSortedArray, counter);
        }
        counter += 1;
}
```

O(N) … (increases by *constant* amount)

O(log(N))

O(N)

Use Big-O Rule 1

Always have to go with the worst if/else case, so $O(N^2)$.

So Now

- So now you can glance at someone’s code and say “This is efficient.” Or, “That’s O(N^3). Very inefficient.”

- *Powerful stuff!!!*

- But a bit like politicians who say “This is bad” and then don’t offer any solutions.

- **Let’s move on to the rest of the course: offering better solutions!!!!**