Algorithm Analysis – Running Time

CS 111 – Fall 2015

Nov 12 Lecture

Example 1: Sequential Search

```
for (int i=0; i < arr.length; i++) {
    if (target == arr[i]) {
        return i;
    }
}
```

Analysis must relate count to array (input) length, say n

What operations to count?

Loop mechanics

- Initializing i
- Checking i against array length
- Incrementing i

Algorithm characteristic

Comparing target against array item

Operation	Time per	How many (success)	How many (failure)
Initializing i	1	1	1
Checking i against array length	1	Best: 1 Worst: n	Best: n+1 Worst: n+1
Incrementing i	1	Best: 0 Worst: n-1	Best: n Worst: n
Comparing target against array item	1	Best: 1 Worst: n	Best: n+1 Worst: n+1

Example 1: Sequential/Liner Search (Success)

Operation	Time per	How many
Initializing i	1	1
Checking i against array length	1	Best: 1 Worst: n
Incrementing i	1	Best: 0 Worst: n-1
Comparing target against array entry	1	Best: 1 Worst: n

Loop mechanics total time:
Best: 1*1 + 1*1 + 1*0 = 2
Worst: 1*1 + 1*n + 1*(n-1) = 2n-1

Algorithmic, total time:
Best: 1*1 = 1
Worst: 1*n = n

Total Running Time

Best case

$2 + 1 = 3$

Worst case

$1 + 2n-1 + n = 3n$

Example 2: Partial Sums – Version 1

```
for (int j=0; j < output.length;
      j++) {
  for (int i=0; i <= j; i++) {
      output[j] += input[i];
  }
}
```

↑

Algorithm characteristic
Adding to array item

Output length = Input length = n

Operation	**Time per**	**How many**
Initializing j	1	1
Checking j against array length	1	n+1
Incrementing j	1	n
Initializing i	1	n
Checking i against array length	1	2+3+…(n+1)
Incrementing i	1	1+2+…n
Adding to array item	1	1+2+…n

Sidebar: Sum of series of integers

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) + n = \frac{n*(n+1)}{2}$$

Which means…..

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) = \frac{(n-1)*((n-1)+1)}{2}$$

$$= \frac{n*(n-1)}{2}$$

and…

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) + n + (n+1) = \frac{(n+1)*((n+1)+1)}{2}$$

$$= \frac{(n+1)*(n+2)}{2}$$

Sidebar: Sum of series of integers

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) + n = \frac{n*(n+1)}{2}$$

Which means.....

$$2 + 3 + \ldots + (n-2) + (n-1) + n + (n+1)$$

$$= (1 + 2 + 3 + \ldots + (n-2) + (n-1) + n + (n+1)) - 1$$

$$= \frac{(n+1)*(n+2)}{2} - 1$$

So…

Output length = Input length = n

Operation	**Time per**	**How many**
Initializing j	1	1
Checking j against array length	1	n+1
Incrementing j	1	n
Initializing i	1	n
Checking i against array length	1	2+3+…(n+1) = (n+1)*(n+2)/2 - 1
Incrementing i	1	1+2+…n = n*(n+1)/2
Adding to array item	1	1+2+…n = n*(n+1)/2

Total Running Time:

$$1*1 + 1*(n+1) + 1*n + 1*n + 1*[(n+1)*(n+2)/2 - 1] + 1*[n*(n+1)/2] + 1*[n*(n+1)/2]$$

$$= (3n + 2) + [n^2/2 + 3n/2 + 1 - 1] + [n^2/2 + n/2] + [n^2/2 + n/2]$$

$$= \mathbf{3n^2/2 + 5n/2 + 2}$$

Example 3: Partial Sums – Version 2

This is not included in the table since it's an outlier. We'll add 1 unit of time for it at the end.

```
output[0] = input[0];

for (int j=1; j < output.length;
      j++) {
    output[j] = output[j-1] +
                input[j];
}
```

Algorithm characteristic

Adding to array item

Output length = Input length = n

Operation	Time per	How many
Initializing j	1	1
Checking j against array length	1	n
Incrementing j	1	n-1
Adding to array item	1	n-1

Total Running Time:

```
   1 + n + (n-1) + (n-1) + 1 (outlier)
 = 3n
```

Comparing Running Times

Program	Running Time Units
Sequential Search (worst case)	`3n`
Partial Sums Version 1	`3n`^{2}`/2 + 5n/2 + 2`
Partial Sums Version 2	`3n`

In a different class because the dominant (highest degree) term is of the order n^2,
while the other two have dominant term of the order n

We boil running times down to the Order of Complexity, called "Big Oh", so we can classify according to order – $O(n^2)$ runs much slower than $O(n)$, *for large enough n (so that edge effects are taken out of the equation)*

Converting to Big O

Program	Running Time Units	Big Oh
Sequential Search (worst case)	$3n$	$O(n)$
Partial Sums Version 1	$3n^2/2 + 5n/2 + 2$	$O(n^2)$
Partial Sums Version 2	$3n$	$O(n)$

From function in input size (e.g. n) to Big O:

1. Ignore all but highest order term
2. Ignore any multiplying constant in highest order term

Algorithm vs Program

More often than not, we will need to analyze the running time of an algorithm BEFORE we even implement it (we may need to explore choices, analyze each, pick best, THEN implement)

Which means we will count ONLY operations that are characteristic of the algorithm (ignoring looping mechanism)

ALGORITHM	Running Time Units	Big Oh
Sequential Search (worst case)	`n`	`O(n)`
Partial Sums Version 1	$n^2/2 + n/2$	$O(n^2)$
Partial Sums Version 2	`n`	`O(n)`

Ignoring looping mechanism does NOT change the Big O (makes sense because looping mechanism is just supporting scaffolding to execute the algorithm, so it cannot supersede the algorithm's running time.)