BRAC University

Algorithms

Lecture 2.1

Rubayat Khan, Lecturer
9/16/2015

In this lecture we will look at some more equations of n.

```
methodA(){
        i = 1;
        s = 1;
        while (s<=n){
                i++;
                s = s + i;
        }
}
```

s	1	3	6	10	....	n
i	1	2	3	4	....	k

We need to find an equation in terms of n. When i = 2, s= 3 [sum of the first 2 terms], when i = 4, s = 10 [sum of first 4 terms], therefore when i = k, s = sum of the first k terms and that is k(k+1)/2.

$n = k(k+1)/2$
$n = (k^2 + k)/2$
$k = \sqrt{n}$
The best and the worst case is $\sqrt{n}$

Bubble sort:
So far we have dealt with algorithms with one or no loops. This one has 2 nested loops. Calculation the cost of such algorithm could be a little complex.

```
for i=1 to n
        for j = i+1 to n
                if A[j]<A[i]
                swap A[i] with A[j]
```

i	j
1	n times
2	n-1 times
3	n-2 times
n	1 time

The total cost is n + (n-1) + (n-2) + ... 1 [summation the inner loop]. We simplify saying that the inner loop runs n times.
$n(n+1)/2$.

Now let us see which line runs how many times.

for i=1 to n	n+1 time (one for the last check)
for j = i+1 to n	n x n times
if A[j]<A[i]	n(n-1)
swap A[i] with A[j]	0 in best case n(n-1) in worst

Total cost (in worst case): $n+1 + n^2 + n^2 - n + n^2 - n = O(n^2)$
Total cost (in best case):): $n+1 + n^2 + n^2 - n + 0 = O(n^2)$
In conclusion, bubble sort has the same best and worst case.

Selection Sort:

```
for i=1 to n
    min = i
    for j=i+1 to n
        if (A[j]<A[min])
            min=j

    swap (A[i], A[min])
```

Selection sort has the same worst and best case, n^2, yet selection sort is preferred over bubble. WHY?

Let us look at some more pseudocodes and deduce the time complexity.

```
for (i=1 ; i<=n, i*2){
}
```

i	1	2	4	8	....	n
execution	1	2	3	4	k-1	k

$2^{(k-1)} = n$
$k-1 = \log_2^n$
$k = \log_2^n + 1$
Worst and best case both $\log_2^n$.

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

```
for (j=n ; j>=1; j=j/2){
}
```

j	n	...	20	10	5	2	1
Loop runs	k	...	5	4	3	2	1

Notice the values of j form a geometric series as the common ratio between any 2 consecutive terms is equal. We need to find the value of k^th term.

$ar^{k-1} = n$

$1x2^{k-1} = n$

$k-1 = \log_2^n$

$k = \log_2^n + 1$

```
for (i=1 ; i<=n, i*2)
        for (j=n ; j>=1; j=j/2){
        }
}
```

So far we have come across a number of sorting algorithms - bubble, selection and insertion. Why have scientists spent years developing them ? What was it like before these were invented? Simpler, how did you sort items before you learnt these algorithms?

For example you have n numbers to sort without knowing the numbers. The method you approach would be like this, [remember you are blind folded]

Step 1: arrange the numbers arbitrarily
Step 2: Open the fold and check if they are sorted
if true your are done else repeat steps 1 and 2 until you sort them.

This is the most basic algorithm, knows as brute force or exhaustive search, where we exhaust ourselves by trying out all the possible arrangements in the world and then picking the right one. The output we get by the exhaustive search (all possible configurations) is called the **natural search space**. There is no smartness used in here hence it is very much inefficient. Brute force does not exist only in sorting but in everything. As we go along this course we will study how development of algorithms has helped immensely to leave the brute force approach behind. Let us focus on sorting for now. If I ask you to sort 3, 2,1 these 3 numbers in the exhaustive way, you would list all the possible arrangements.

3	2	1
2	3	1
2	1	3
3	1	2
1	2	3
1	3	2

6 possible arrangements for 3 numbers. What for 4? 24 arrangements. How do we know that it will be 24? The answer is **permutation**. 3! = 6, 4! = 24. Now let us prove brute force is inefficient. So far we saw there is nothing worse than n^2. For n = 3, n^2 gives 9 and n! = 6. For n = 4, n^2 = 16 and n! = 24. As n increases n! grows way quicker than n^2. Thus if we plot the 2 curves we will see that at one point n! upper bounds n^2 a big time.

Let us look at a different scenario. We have set {1, 2, 3}. What are all the possible **combinations**? {} {1} {2} {3} {1, 2} {1, 3} {2, 3} {1, 2, 3}. There are 8 possible combinations, which is 2^3. What would be the natural search space of a set containing k elements? Answer is 2^k.
The natural search space is huge and coming back to our question why we need efficient algorithms is to shrink the natural search space.

So far we have seen a number of functions growing with the increasing size of n. If we sort them from the fastest (slowest growth rate) to the slowest (quickest growth rate) complexity we get: $O(1)$, $O(\log_2 n)$, $O(\sqrt{n})$, $O(n)$, $O(n \log_2 n)$, $O(n^k)$, $O(2^n)$, $O(n!)$