Analysis of Algorithms

An Introduction

Produced by: Dr. Siobhán Drohan
Ms. Mairead Meagher

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Analysis of Algorithms

- What is an efficient algorithm?
- Examples of costs.
- Recurrence Relations.
- Looking at comparative numbers.

Analysis of Algorithms

- What does it mean to be a ‘good’ algorithm?
  - Quick to run ?
  - Needs low memory requirements?

- Both!

- We look at how quick algorithms run, on large data.

Complexity of Algorithms

- The complexity of an algorithm underlying a program is usually defined in terms of the program input.
- Let **n** be a measure of the quantity of input data for an algorithm. **T(n) - time complexity** - is the time needed by an algorithm to complete execution as a function of size of input **n**.

Complexity of Algorithms

Let us look at the following piece of code

```
for (int i = 0; i<10; i++)
    for (int j = 0; j<8; j++)
        System.out.println("hi");
```

How many times is "hi" written to the screen.

Answer: 80.

Complexity of Algorithms

- If we replaced the 8 and 10 by n:

```
for (int i = 0; i<n; i++)
    for (int j = 0; j<n; j++)
        System.out.println("hi");
```

- Then we say that the cost of the loop is n^2.

Complexity of Algorithms

- We calculated the cost of the loop on the previous slides as n^2.
- When discussing algorithms, we call this $O(n^2)$.
- This is called the 'big O' notation (short for 'order of'). In general this means that the cost is **'roughly'** n^2.
- If the true cost is, for instance, $5 * n^2$ or $\frac{1}{2} n^2$, we still call it $O(n^2)$. This is because what we are interested is the highest power that the algorithm takes up. If the full cost was, for instance $1000*n + n^2$, we would still call it $O(n^2)$.

Cost of Sorting Algorithms

Recall Bubble Sort

6 5 3 1 8 7 2 4

```
for(int j = n - 1; j >= 0; j--)
{
    for(int i = 0; i < j; i++)
    {
       if (array[i] > array [i+1])
          swap(array, i, i+1);
    }
}
```

We will pass through the array n-1 times, where n is the number of elements in the array (outer loop).

In the example above:

- First pass makes n – 1 comparisons → i.e. 7
- Second pass makes n – 2 comparisons → i.e. 6
- Third pass makes n – 3 comparisons → i.e. 5
- :
- Seventh pass makes n – 7 comparisons → i.e.1

Calculating the Cost of Bubble Sort

So, the cost of the algorithm is $7 + 6 + 5 + 4 + 3 + 2 + 1$.
Which is the same as $(n - 1) + (n - 2) + (n - 3) + \ldots + (n - 7)$
Which is the same as $n(n+1)/2$

We will pass through the array n-1 times, where n is the number of elements in the array (outer loop).
In the example above:

- First pass makes n – 1 comparisons → i.e. 7
- Second pass makes n – 2 comparisons → i.e. 6
- Third pass makes n – 3 comparisons → i.e. 5
- :
- Seventh pass makes n – 7 comparisons → i.e.1

Cost of Simple Sorts

- So Bubble Sort is $n(n+1)/2$
- According to our 'big O' notation (drop constants and just leave highest power) we call this $O(n^2)$

- Selection Sort – similarly this is $O(n^2)$.

- Insertion sort is also $O(n^2)$.

- In fact, most simple sorts are $O(n^2)$.

Cost of sorting algorithms

- Even though all three sorts are $O(n^2)$, insertion sort and selection sort are more efficient than bubblesort.

- Bubblesort is very inefficient because there are so many swaps during each pass. And (in the basic version) the fact that the list may be sorted early, the algorithm ignores this.

Some other sorting algorithms

Algorithm	Data Structure	Time Complexity: Best Case	Time Complexity: Average Case	Time Complexity: Worst Case
Bubble sort	Array	O(n)	$O(n^2)$	$O(n^2)$
Selection sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	Array	O(n)	$O(n^2)$	$O(n^2)$
Merge sort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Heap sort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Smooth sort	Array	O(n)	O(n log(n))	O(n log(n))
Quick Sort	Array	O(n log(n))	O(n log(n))	$O(n^2)$

Cost of Searching Algorithms

Binary Search

- Algorithm (array already sorted)
  - Pick middle value, check is it the ‘lookingFor’ (if it is, terminate)
  - If ‘lookingFor’ less than the middle value, search through list up to this value (i.e. top half of the list)
  - If ‘lookingFor’ greater than the middle value, search through list after this value (i.e. bottom half of the list)

Binary Search - Cost

- Best case:
  - the ‘lookingFor’ value is in the middle of the array.

- Worst case:
  - ‘lookingFor’ is not in the array at all. In this case:
    - We start with the full array (n).
    - On the second pass, we only use half the array (n/2)+1.
    - On the third pass, we only use half the array (n/2/2)+1.
    - And so on.
    - This pattern evaluates to O(log n) (this is great from a cost perspective).

Binary Search - Cost

- Average case:
  - 'lookingFor' is in the array, but not in the middle on iteration one.
  - The processing for this case is similar to the worst case above:
    - We start with the full array (n).
    - If not found, on the second pass, use half the array (n/2)+1.
    - If not found, on the third pass, use half the array (n/2/2)+1 +1 which is the same as (n/4)+1+1.
    - And so on.
    - This pattern also evaluates to O(log n) (this is great from a cost perspective).

Binary Search - Cost

- T (n) is the time necessary to process n.
- Recall the formula for Binary Search from the previous slide:
  - If the value is not at the midpoint, half the array → T(n) = T (n/2) + 1
  - If not found, half the array again → T(n) = T(n/4) + 1 + 1
  - And so on.

- This evaluates to a "Recurrence relation" of:

  $T(n) = T(n/2^k) + 1 + 1 + 1$ (k times)

- So, what value is k?
- Using substitution etc., we arrive at a big O value for T(n) = O(log n)

Recurrence relations

Recurrence	Algorithm	Big-Oh Solution
$T(n) = T(n/2) + O(1)$	Binary Search	*O(log n)*
$T(n) = T(n-1) + O(1)$	Sequential Search	*O(n)*
$T(n) = T(n-1) + O(n)$	Selection Sort (other n^2 sorts)	*O(n^2)*
$T(n) = 2\ T(n/2) + O(n)$	Mergesort (average case Quicksort)	*O(n log n)*

Relative values of numbers

	constant	*logarithmic*	*linear*		*quadratic*	*cubic*
n	**O(1)**	**O(log N)**	**O(N)**	**O(N log N)**	**$O(N^2)$**	**$O(N^3)$**
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1,024	1	10	1,024	10,240	1,048,576	1,073,741,824
1,048,576	1	20	1,048,576	20,971,520	10^{12}	10^{16}