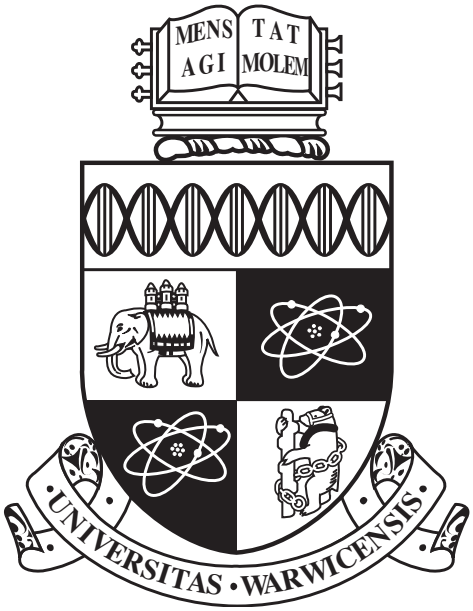
University of Warwick
Department of Computer Science

CS126

Design of Information Structures



Cem Yilmaz
January 22, 2022

Contents

1 Analysis of Algorithms

1.1 Classification of a good algorithm

A good algorithm is one that optimises the following:

- Time
- Memory
- Network bandwidth
- Energy consumption

However, the main focus in this module will be the running time, in particular, this $O(x)$

1.2 Running Time

Running time of an algorithm typical grows with the input size. However, for different inputs of the same size the running time of an algorithm can vary. Then, for an input of fixed size n, we have different running times. We can categorise them as the following:

- Average case - the typical running time an algorithm requires and is often very difficult to determine
- Best case - what is the minimum running time of the algorithm and is generally not useful
- Worst case - upper bound on the running time, for any possible input and is more standard to analyse. Our focus is generally this.

1.3 Finding the running time

1.3.1 Experimental Analysis

The first method to use is experimental analysis. For this, we use computer and run simulations. For this, we write a program implementing the algorithm and run the program with inputs of varying size and composition, noting the time needed. However, there are limitations:

- It is necessary to implement the algorithm. Sometimes this can be impossible.
- Need to make sure that we have considered all kinds of inputs. This sometimes cannot be possible and otherwise it would not be indicative of running time
- Different algorithms may run different in different systems due to different features. It can be especially worse for different hardware.

1.3.2 Theoretical Analysis

The second method is theoretical analysis. For this, we use pen and paper. We use a high-level description of the algorithm instead of an implementation. We then characterise running time as a function of the input size n which takes in account all possible inputs. This would indeed allow us to evaluate the speed of an algorithm independent of the hardware or software environment. A good way to do high level description is to use pseudo-code. We also assume that the algorithm runs in an idealised machine. We assume simple memory hierarchy that is unbounded, infinite precision in arithmetic operations etc.

1.4 Random Access Machine (RAM) Model

It is a simple mode of computation with a singular CPU. It only executes a single program. It also has a bank of memory cells where each cell can hold arbitrarily large positive integers. Every cell gets assigned an ID that allows us to access the information in some *unit time*. In CPU, as learned from CS132, the CPU has the program stored inside. It is also connected to a program counter and registers $R_0, R_1, \ldots$ along with memory cells. It can do basic operations between two numbers stored in the registers, which include but are not limited to:

- Addition

- Comparison
- Fetch an element from the memory
- Write an element to the memory

However, one thing that categorises and defines RAM as what it is is the fact that *we can access any memory cell in unit times*.

1.5 Primitive Operations

Primitive operations are basic computations that are performed by an algorithm. These take constant time in the RAM model and we count the primitive operations that happen. The assumption is that the number of primitive operations are proportional to the actual running time. Some examples of primitive operations include evaluating an expression, assigning a value, calling a method, indexing into an array etc.

Example 1.1. Operations in a code

```
1       public static double arrayMax(double[] data) {
2     int n = data.length;
3     double currentMax = data[0];
4     for (int j = 1; j < n; j++)
5       if (data[j] > currentMax)
6         currentMax=data[j];
7     return currentMax;
8   }
```

Listing 1: Operations

In particular,

- Step 3 has 2 operations
- Step 4 has 2 operations
- Step 5 has $2n$ operations
- Step 6 has $2n-2$ operations
- Step 7 has $2n-2$ operations and finally
- Step 8 has 1 operation.

The code for $arrayMax$ has $6n+1$ as the worst case and $4n+3$ as the best case. Let $T(n)$ be the running time of arrayMax. Then,

$$a(4n+3) \leq T(n) \leq a(6n+1)$$

Where a is the time to execute a primitive operation.

1.6 Sorting algorithms

Indeed, the time complexity can be seen clearly in sorting algorithms for times.

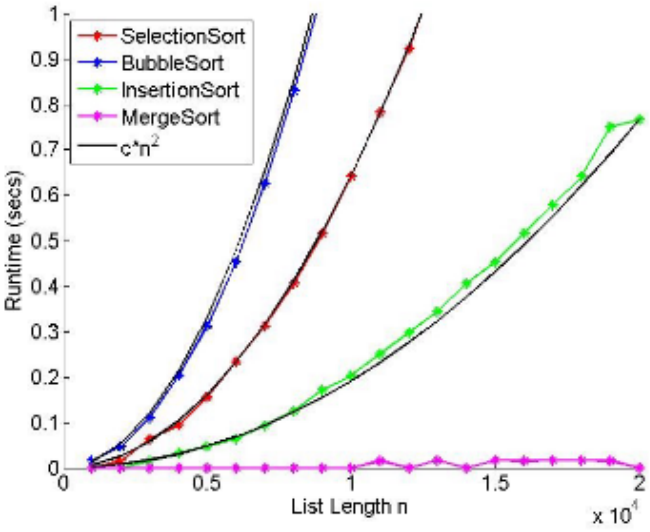


Figure 1: Complexity of sorting algorithms

2 Asymptotic Notation

2.1 Big-O notation

Definition 2.1. Big O

For two functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 i.e. $n_0, c \geq 1$ and such that

$$f(n) \leq cg(n) \tag{1}$$

for any $n \geq n_0$ and $n \geq n_0$.

Example 2.1. Example 1

$2n + 10$ is in $O(n)$.

$$2n + 10 \leq cn \tag{2}$$
$$(c - 2)n \geq 10 \tag{3}$$
$$n \geq \frac{10}{c - 2} \tag{4}$$

We pick $c = 3$ and $n_0 = 10$.

Example 2.2. Example 2

n^2 is not $O(n)$

$$n^2 \leq cn \tag{5}$$
$$n \leq c \tag{6}$$

c is a constant and thus this inequality cannot be satisfied.

Example 2.3. Example 3

$7n - 2$ is in $O(n)$

$$7n - 2 \leq cn \tag{7}$$

Pick $c = 7$ and $n_0 = 1$.

Example 2.4. Example 4

Is is true that $t > 0$, $(1 + n)^t$ is in $O(n^t)$

$$(1 + n)^t = \sum_{i=0}^{t} \binom{t}{i} n^i \tag{8}$$

However, t is the biggest number therefore it is $O(n^t)$

Thus, the Big-O notation gives an upper bound on the growth rate of a function as n grows towards infinity. We can use the Big-O notation to rank functions according to their growth rate.

2.1.1 General Rules

- Drop lower order terms
- Drop constant terms
- Use the smallest possible class of functions

3 Relatives of Big-O

3.1 Big Omega

Definition 3.1. Big Omega

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $c > 0$ and an integer constant $n_0 \geq 1$ such that for $n \geq n_0$

$$f(n) \geq cg(n) \tag{9}$$

3.2 Big Theta

Definition 3.2. Big Theta

$f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and ann integer constant $n_0 \geq 1$ such that for $n \geq n_0$

$$c'g(n) < f(n) \leq c''g(n) \tag{10}$$

3.3 Summary

- Big-O is asymptotically less than or equal to $g(n)$
- Big-Ω is asymptotically greater than or equal to $g(n)$
- Big-Θ is asymptotically sandwiched between $g(n)$ differing by constant

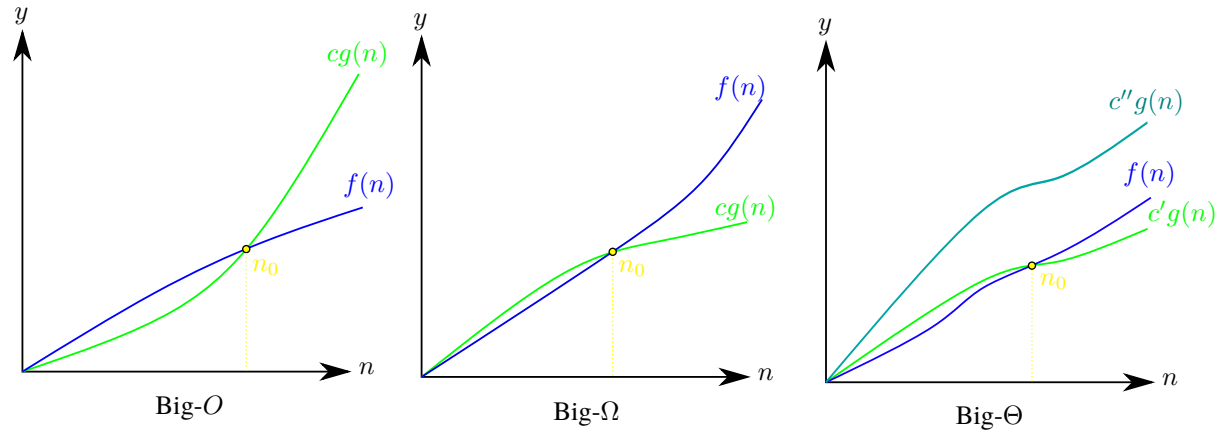


Figure 2: Summary of Big notations graphically

3.4 Examples

Example 3.1. Example 1

$5n^2$ is in $\Omega(n^2)$

$$5n^2 \geq cn^2, \text{ let } c = 5, n_0 = 1 \tag{11}$$

$5n^2$ is also $\Omega(n)$

$$5n^2 \geq cn, \text{ let } c = 1, n_0 = 1 \tag{12}$$

$5n^2$ is in $O(n^2)$

$$5n^2 < cn^2, \text{ let } c = 6, n_0 = 1 \tag{13}$$

Because $\Omega(n^2)$ and $O(n^2)$, we indeed can say that

$$\Theta(n^2) \tag{14}$$

4 Data Structures

4.1 Arrays

Definition 4.1. Array

An array is a sequences collection of variables of thesame type. Each variable, or cell, in an array has an index, which uniquely refers to the value stored in that cell.
A value stored in an array is often called an element.
The length of an array determines the maximum number of elements that can be stored.

4.1.1 Strengths

- We assume that we can access each cell k in $O(1)$ time.
- You can write or read once accessed.
- Access time is very fast $O(1)$.

4.1.2 Limitations

- We cannot change the length of an array

4.1.3 Declaring arrays in Java

Assignment to a literal form when initially declaring the array

```
elementType[] arrayName = { v0, v1, ..., vn-1}
```

Listing 2: Array

elementType : any Java base type, or class name
arrayName : any valid Java identifier
Remark : the initial values must be of the same type as the array.
We also use the new operator to declare arrays because it is not an instance of a class. That is,

```
new elementType[length]
```

Listing 3: Declaring array

The new operator returns a reference to the new array. This is assigned to the array variable measurements.

```
double [] measurements = new double [1000]
```

Listing 4: Example

4.1.4 Examples

A array can store primitive elements, such as characters. E.g.

Table 1: Primitive element array

S	A	M	P	L	E
0	1	2	3	4	5

...or pointer to references to objects. For example, the first cell can refer to the pointer of the word "Joseph" and the second can refer to the pointer of "Helen" etc.

4.1.5 Adding an entry

To add an entry e into array board at i we need to make room. We shift each $n - i \mapsto n - i + 1$. This is $O(n - i)$.

4.1.6 Concluding Remarks

- Read/write any element in $O(1)$ time
- The capacity does not change
- Shifting k elements requires $O(k)$ time
- Very easy to work with
- We are going to use arrays a lot when we implement

4.2 Linked Lists

4.2.1 Singly Linked List

Definition 4.2. Singly Linked List

A singly linked list is a conrete data of structure consisting fro a sequence of nodes, starting from a head pointer.

Each node stores element and a link to the next node. However, nodes do not have information on previous nodes. The first node is usually called head, and the last is called tail. However, if the node is tail, then the next link is NULL. However, to access k-th element in the node, we would then require to repeat getting node $k - 1$ times. This is called pointer hopping.

4.2.2 Time required for Singly Linked List

Our assumption is that CurrentNode.getNext() requires $O(1)$ time. In order to access k-th element, we need $O(k)$ time.

4.2.3 Insertion - Operations

To insert a head:

- Allocate a new node
- Insert a new element
- New node points to old head
- Update head to point new node