# Software 2 Theory Lecture 3 Sorting and Algorithm Complexity

Algorithms take up different amounts of resources, signified by ‘complexity’ and ‘the Big-O notation’

Some algorithms are better than others, or may run better in certain conditions, but this can be hard to measure, as size of inputs vary, which is why to evaluate the effectiveness and efficiency of an algorithm (performance being measured in terms of space [memory/hardware resources] and time), a (time) complexity is calculated: function

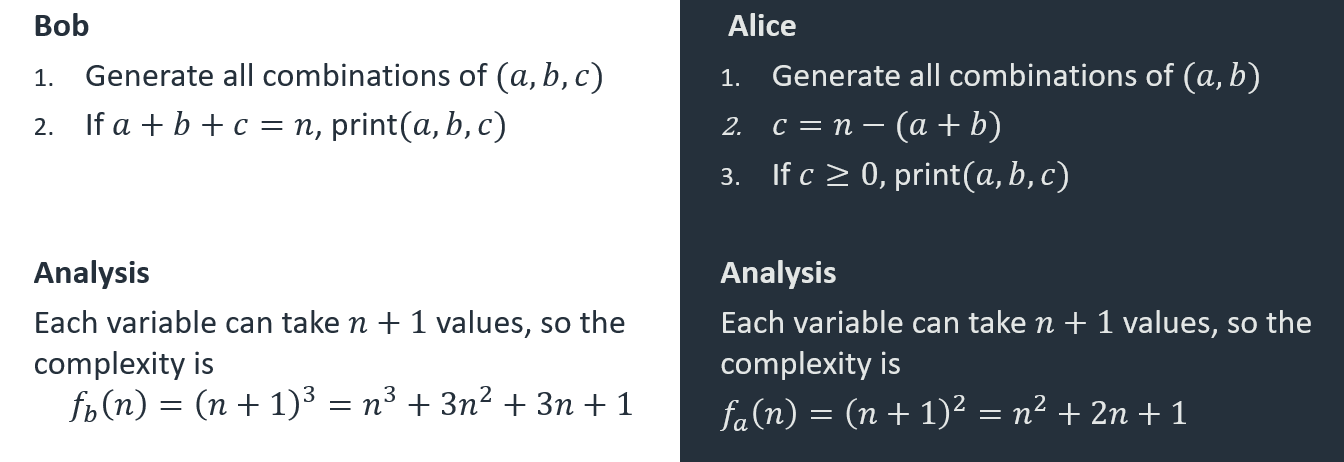
If you were to run the same code on different devices, in a non-standardised test environment, algorithms will complete in differing times – not properly evaluating the algorithms.

O(1) – constant time

* Adding to an array
* Stacks (push/pop)
* Queues (enqueue/dequeue)
* Delete a node from a linked list

O(n) – Linear time

* Searching for a non-existent item in a list
* Counting the number of items in a linked list



From analysis, the time taken by Bob’s algorithm grows more quickly than that for Alice’s (since n3 grows more quickly than n2), the smaller terms (3n2 + 3n + 1) are negligible to this total time, so can be ignored. The leading term is important here.

## Definition of Ω and Θ

We can define 𝑓∈Ω(𝑔) if there exist positive integers 𝑐 and 𝑛0 such that𝑓(𝑛)≥𝑐𝑔(𝑛) for all 𝑛≥𝑛0.

Interpretation: f grows no slower than g

Finally, if 𝑓∈𝑂(𝑔) and 𝑓∈Ω(𝑔) then 𝑓∈Θ(𝑔) and 𝑔∈Θ(𝑓), 𝑓 and 𝑔 grow at the same rate (signifying average case and the best-case scenario).

### Notation Warning

It has been traditional to write “𝑓=𝑂(𝑔)“ (e.g. Skiena Ch. 2)

This is an abuse of notation, both incorrect and confusing  
It implies that you could write “𝑂(𝑔)=𝑓“ and “𝑓=𝑂(𝑔), ℎ=𝑂(𝑔)⇒𝑓=ℎ”

𝑂(𝑔) is a set of functions, so we should write 𝑓∈𝑂(𝑔) and 𝑂(𝑓)=𝑂(𝑔) means the sets are the same

You should be aware of the other notation though, as you may come across it

## Simple Big-O Orderings

O(1) Constant time

O(logan) Logarithmic a>1

O(n) Linear

O(nlogn) n log n

O(n2) Quadratic

O(en) Exponential

O(n!) Factorial

### Polynomials

If 𝑓(𝑛)=𝑎𝑘𝑛𝑘+𝑎(𝑘−1) 𝑛(𝑘−1)+…+𝑎0 then 𝑓∈𝑂(𝑛𝑘)  
The order is the largest power, the multiplying constant does not matter

Logs

log𝑎𝑛=log𝑛/log𝑎 =𝑐 log𝑛   
and since multiplying constants do not matter,   
log𝑎𝑛∈𝑂(log𝑛)

## Sorting

Sorting is a basic algorithmic building build, which can form the basis of performance gains for many other algorithms. It is one of the most thoroughly studied problems in CS with numerous different algorithms highlighting different considerations.

### Designing an algorithm

We start by designing an algorithm using a method called invariants.

**Idea:** Iterate and, at each operation, maintain some property of the data (the invariant). At the start, the input has the property, and at the end, the output has the property.

**Sorting invariant:** At iteration k, the list 0…k is sorted.

**At the start:** elements 0…0 are sorted (nothing is sorted)

**At the end:** elements 0…(n-1) are sorted (everything is sorted)

In the loop body, we put instructions to maintain the invariant.

**SelectionSort(l : list)**

**for k=0…l.length-1**

**i=IndexOfMin(l(k…end))**

**swap(l(i),l(k))**

**end for**

At the start of iteration k, we are sorted up to k-1. After iteration k, we should be sorted up to k.

Need to find the element in position k.

This will be the smallest element left.

Then move it into position.

Size of inputs: (Length of list) n=l.length

Type of input:

* Best case: list is already in order (unrealistic)
* Average case: maybe, but need to analyse all cases, need to know probability of different inputs
* Worst case: gives an upper bound on the run time, a very useful analysis

Worst case time complexity: A completely unsorted list  
(occasionally we will look at average time complexity for some algorithms)

To calculate the number of steps, we need a computational model for basic operations.

* Arithmetic operations take unit time
* Memory access takes unit time
* No limit to memory size of variation in speed
* Subroutines take the sum of their individual operations, no overheads

### 

### Designing a better algorithm

This time we’ll use a design method called divide-and-conquer.

Here the idea is the divide the list into two equal parts, and sort them separately. Then merge the results.

We do not expect sorting to be 𝑂(𝑛), so if we shorten the list, we gain a lot.

For example, if we have a quadratic time algorithm, then halving the input size gives a times 4 speedup

But we must be able to merge the results efficiently

We recursively call MergeSort.

**MergeSort(l : list)  
if l.length==1 then return l**

**m=l.length/2**

**l1=l(0…m-1)**

**l2=l(m…end)**

**MergeSort(l1)**

**MergeSort(l2)**

**return Merge(l1,l2)**

It sorts the first and second halves of the list.

They are combined in the Merge subroutine.

Since l1 and l2 are sorted, the smallest element overall must either be at the start of l1 or l2.

**Merge(l1 : list, l2 : list)**

**m=new list**

**while l1 and l2 are not empty**

**if l1(0)≤l2(0)**

**insert l1(0) in m**

**delete l1(0)**

**else**

**insert l2(0) in m**

**delete l2(0)**

**end if**

**end while**

**return m**

We remove this from the respective list and insert it in the merged list.

All operations in the iteration are unit time, so Merge is O(n).

So 𝑓(𝑛)=2𝑓(𝑛/2)+2𝑛+2

Make some simplifications

2𝑛+2≃2𝑛

List length is exactly a power of 2, 𝑛=2^𝑘. This avoids complications of unequal list lengths.

𝑓(2𝑘)=2𝑓(2(𝑘−1) )+2⋅2𝑘  
 =2(2𝑓(2(𝑘−2) )+2⋅2(𝑘−1) )+2⋅2𝑘  
 =22 𝑓(2(𝑘−2))+2⋅2𝑘+2⋅2𝑘

Continue this k times to get to 𝑓(20 )=𝑓(1)=1

𝑓(𝑛)=𝑓(2𝑘)=𝑘⋅2⋅2𝑘

Finally, use 𝑘=log2𝑛

𝑓(𝑛)=log2𝑛⋅2⋅𝑛

Clearly this is 𝑂(𝑛 log𝑛)