Algorithms

And data structures

When we analyse an algorithm, we want to know

- Execution time
- Memory use

Typically we want to know how space and time requirements increase with the size of the input. Ideally we would like our algorithm's space requirements to be a constant factor of the input size and the execution time to be independent of the input size.

When we analyse the execution time of an algorithm we can determine the total time requirements by considering two factors

- Time taken to execute each statement
- Frequency of execution of each statement.

By multiplying the two items together for each statement and summing we get the total cost of executing a given piece of code. While the former is determined by the compiler, the OS and the machine the latter is a function of the code or algorithm itself. By focussing on the latter we can determine the general execution time characteristic of an algorithm irrespective of the language it is implements in or the machine it runs on. In cases where the frequency of execution give us a complex expression such as $n^3 - 2n^2 + 4n - 3$ While it possible to calculate such an expression in many practical situations it is not worth the effort. The multiplicative constants and lower order terms are insignificant compared to the input size. If the input size is sufficiently large, we can focus on the **order of growth** of the running time. When we study the asymptotic efficiency of an algorithm, we are looking at how the running time increases with the input size in the limit as the input size increases.

$$\Theta(g(n)) = \begin{Bmatrix} f(n): there\ exists\ positive\ constants\ c_1, c_2, n_0 such\ that \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) for\ all\ n\ \geq n_0 \end{Bmatrix}$$

$g(n)$ is an asymototically tight bound for $f(n)$

we ignore the lower order terms using the following notation

$$g(n) \sim f(n)$$

Which means

$$lim_{N \to \infty} \frac{f(n)}{g(n)} = 1$$

We talk about the order of growth of the algorithm as the input size N grows. The following table shows some of the most important order of growth functions.

ORDER OF GROWTH EXAMPLES

Name	Function	Code	Descrip	Example
Constant	1	`int a =10;`	*statement*	*assignment*
Logarithmic	$logN$		*halfing*	*binary search*
Linear	N	`for (int i = 0; i < 10; i++)` `a+=i;`	*loop*	*summing*
Linearithmic	$NlogN$		*divide and conquer*	*sorting*
Quadratic	N^2		*double loop*	*pair checking*
Cubic	N^3		*Triple loop*	*triple checking*
Exponential	2^N			

All these functions, except for the exponential case can be described by the following expression

$$g(n) \sim an^b(logn)^c$$

Where a, b and c are constants. Typically, we do not state the base of the logarithm as a logarithm in one base can be converted to a logarithm in another base using a constant. So, we can absorb this with the constant a in our previous expression,

$$log_b x = log_a x \times log_b a$$

Analysis of Algorithms

Iterative Algorithm (Insertion Sort)

```
public override void Sort<T>(T[] a)
{
	if (a == null || a.Length == 1) return;

	for (int j = 1; j < a.Length; j++)                    c1n
	{
		T key = a[j];                                 c2(n − 1)
		int i = j - 1;                                c3(n − 1)


		while (i >= 0 && Less(key, a[i]))             c4 ∑_{j=1}^{j=n−1} tj
		{
			a[i + 1] = a[i];                      c5 ∑_{j=1}^{j=n−1}(tj − 1)
			i-                                    c6 ∑_{j=1}^{j=n−1}(tj − 1)
		}

		a[i + 1] = key;                               c7(n − 1)

	}
}
```

A general expression for the running time is then given by as follows. Note that the test conditions on the loops execute one more time than the body of the loop as they will execute on one iteration when the test fails.

$$c_1 n + c_2(n-1)c_2 + (n-1) + c_4 \sum_{j=1}^{j=n-1} t_j + c_5 \sum_{j=1}^{j=n-1}(t_j - 1) + c_6 \sum_{j=1}^{j=n-1}(t_j - 1) + c_7(n-1)$$

In the worst case scenario $t_j = j + 1$ and our expression becomes

$$c_1 n + c_2(n-1)c_2 + (n-1) + c_4 \sum_{j=1}^{j=n-1}(j+1) + c_5 \sum_{j=1}^{j=n-1} j + c_6 \sum_{j=1}^{j=n-1} j + c_7(n-1)$$

From the properties of series we know that

$$\sum_{j=1}^{j=n-1}(j+1) = 2 + \cdots n = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=1}^{j=n-1}(j) = 1 + 2 + \cdots n - 1 = \frac{n(n+1)}{2} - n$$

Sorting

We will consider Sorting different sorting algorithms in turn. Before we do the following table shows when we might want to use each one

Appropriate Sorting Algorithm

Description	Algorithm
Small number of elements	Insertion Sort
The collection is already mainly sorted	Insertion sort
Easy to code	Insertion sort / Selection sort
Want very good average case behaviour	Quick sort
The algorithm must be stable	Merge sort
Need good performance in worst case	Heap sort

> **Stable Sort**
>
> A stable sort keeps elements with the same key in the same order relative to each other in the sorted output

Insertion Sort

Insertion sort works much like one would sort ones hand in bridge.

```
public override void Sort<T>(T[] a)
{
	if (a == null || a.Length == 1) return;

	for (int i = 1; i < a.Length; i++)
	{
		for (int j = i; j > 0 && Less(a[j], a[j - 1]); j--)
		{
			Switch(a, j - 1, j);
		}
	}
}
```

```
public override void Sort<T>(T[] a)
{
	if (a == null || a.Length == 1) return;                          c1

	for (int i = 1; i < a.Length; i++)                               c2(n)
	{
		for (int j = i; j > 0 && Less(a[j], a[j - 1]); j--)      c3 ∑_{j=1}^{j=n-1} t_j
		{
			Switch(a, j - 1, j);                             c4 ∑_{j=1}^{j=n-1} t_j

		}
	}
}
```

WORST CASE

In the worst case the number of comparisons is given by $\frac{n(n-1)}{2}$ The following diagram shows why. We have $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$

i	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	F	E	D	C	B	A
2	E	F	D	C	B	A
3	D	E	F	C	B	A
4	C	D	E	F	B	A
5	B	C	D	E	F	A

BEST CASE

In the best case we have $n - 1$ comparisons

i	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	A	B	C	D	E	F
2	A	B	C	D	E	F
3	A	B	C	D	E	F
4	A	B	C	D	E	F
5	A	B	C	D	E	F

AVERAGE CASE

On the average case we will need to do

In the worst case insertion requires the following number of comparisons. First we remind ourselves of the result that the sum of the first n integers is given by

SUM OF FIRST N INTEGERS

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

Base Case

Show the hypothesis hold for n = 0

$$S(0) = \frac{1(0)}{2} = 0$$

Inductive hypothesis

Asuume P(k) hold for some unspecified value of k

$$P(k) 1 + 2 + \cdots + k - 1 + k = \frac{k(k+1)}{2}$$

Show that if the hypothesis holds for k it holds for k+1. We need to show that

$$(1 + 2 + \cdots + k - 1 + k) + (k+1) = \frac{(k+1)\big((k+1)+1\big)}{2}$$

Using the inductive hypothesis the left hand side can be written as

$$\frac{k(k+1)}{2} + (k+1)$$

Re-arraging this we get

$$\frac{k(k+1) + 2(k+1)}{2}$$

Factoring out on the numerator

$$\frac{(k+1)(k+2)}{2} = \frac{(k+1)\big((k+1)+1\big)}{2} = rhs$$

SUM OF FIRST N-1 INTEGERS

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

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

Let

$$S = 1 + 2 + \cdots + n - 1 + n$$

$$2S = 1 + 2 + \cdots + n - 1 + n + 1 + 2 + \cdots + n - 1 + n$$

$$2S = 1 + 2 + \cdots + n - 1 + n +$$

$$n + n - 1 + \cdots + 2 + 1$$

$$2S = n(n+1)$$

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

Proof by induction

$$S(1) = \frac{1(2)}{1} = 1$$

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

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