# Algorithms I

**Definition**

An **algorithm** is a *well defined* computation procedure that takes a set of values as input and produces a set of values as output.

Note: the term *well defined* is itself, not well defined.

**Definitions**

- **Problems** have specific inputs and outputs, input must be finite and not a stream of data.
- **Problem instances** is a specific set of inputs for a problem. A problem can have a Big-O but not a problem instance.
- A program is **correct** if for every input instance, it terminates with the correct output.

**Note**

- Randomised algorithms is a branch of incorrect algorithms.
- Some algorithms produce incorrect outputs with a probablility (e.g. quantum computing)
- Some algorithms loops infinitely for some inputs, but runs a lot faster that an algorithm that guarantees termination for cases where it terminates. It might be possible to determine whether it will terminate for a specific input before running it.
- Some algorithms gives an output within a margin of error (e.g. A* vs Dijkstra)

## Notation

### Arrays

- `A[1]` is the first item
- `A[1..n]` is an array of length `n`
- `A.length` is the number of items in the array

We write pseudocode that is

- Imparative
- Block structured
- Fixed form (indentation matters)
- Parameters are passed by values, objects are passed by pointers
- Loop induction (for loops) increments after the final loop

```
for i=1 to 10
  // do stuff
```

After this loop, consider `i=11`

## Sorting

Each **key** may have attached payloads.

### Insertion Sort

```
for j = 2 to A.length
  Key = A[j]
  i = j - 1
  while i > 0 && A[i] > Key
    A[i + 1] = A[i]
    i = i - 1
  A[i + 1] = Key
```

Use proof by induction for algorithms:

- **Initialisation**: find a property that is true at the start of the program

  $P$: at the start of each loop, $A[1...j-1]$ contains the $1...j-1$ items in sorted order.

  At the start of the first loop, that is just $[a_1]$, true.

  > **Note**
  > Define "the start of the loop" as: after assigning the value of $j$, but before running the first line of code in the loop.

- **Maintenance**: show that the property is maintained as the program is running.
- **Termination**: when the program terminates, show the output is correct.

  After the last loop, $A[1...A.\text{length}]$ would have been containing all the items $1...A.\text{length}$ in order.

  And then we can also show the program terminates as it only needs to complete the loop $A.\text{length}$ items.

> **Note**
> Which is the same as the following Hoare logic proof.
>
> Let $P, Q$ be pre and post-conditions, $B$ be body of the loop, $C$ be condition for the loop.
>
> Given:
> 1. $\{P\}\ B\ \{P\}$
> 2. $P \land \neg C \implies Q$
>
> Then $\{P\}$ while $C$ do $B$ $\{Q\}$

---

## Analysis

> **Definition**
> **Analysis** is about predicting the resources (CPU, memory, disk operations) for input instances we haven't ran our algorithm on.

| Input measurement | Description |
|---|---|
| $A$.length | Common for every day senarios, but may be incorrect if each item in array can have variable size (e.g. big integer) |
| no. of bits/bytes | Useful for algorithm that operates on some bit/byte value. |
| $2^{A.\text{length}}$ | Overestimates the size in most cases, but can be used for search lists. |

> **Definition**
> The **running time** of a program is the number of **basic operations**. (as they all cost 1)
>
> | Basic operation | Cost |
> |---|---|
> | Indexing an array $A[i]$ | 1 |
> | Arithmetic operation | 1 |
> | Comparisons | 1 |

| Basic operation | Cost |
|---|---|
| Assigment to variables | 1 |

One basic operation might not be equal to one clock cycle, if you change the cost of the basic operations, the running time changes.

**Note**
Comparisons (numbers) is usually done by subtracting one from another, then compare with 0.

## Orer of Growth

- $\Theta(g(n))$ is the **asymptotic tight bound** for $g(n)$

$$f(n) \in \Theta(g(n)) \Longrightarrow \exists c_1, c_2, n_0 \in \mathbb{R}^+ : (\forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n))$$

- $O(g(n))$ is the **asymptotic tight upper bound** for $g(n)$

$$f(n) \in O(g(n)) \Longrightarrow \exists c, n_0 \in \mathbb{R}^+ : (\forall n \geq n_0 : f(n) \leq cg(n))$$

- $\Omega(g(n))$ is the **asymptotic tight lower bound** for $g(n)$

$$f(n) \in \Omega(g(n)) \Longrightarrow \exists c, n_0 \in \mathbb{R}^+ : (\forall n \geq n_0 : cg(n) \leq f(n))$$

- $o(g(n))$ is the **asymptotic non-tight upper bound** for $g(n)$

$$f(n) \in o(g(n)) \Longrightarrow \forall c \in \mathbb{R}^+ : (\exists n_0 \in \mathbb{R}^+ : f(n) < cg(n))$$

- $\omega(g(n))$ is the **asymptotic non-tight lower bound** for $g(n)$

$$f(n) \in \omega(g(n)) \Longrightarrow \forall c \in \mathbb{R}^+ : (\exists n_0 \in \mathbb{R}^+ : cg(n) < f(n))$$

### Properties of Orders of Growth

$$\Theta(g(n)) \subseteq O(g(n))$$
$$\Theta(g(n)) \subseteq \Omega(g(n))$$

- **Transitive**: satisfied by all 5 orders

$$f(n) \in \Theta(g(n)) \land g(n) \in \Theta(h(n)) \Longrightarrow f(n) \in \Theta(h(n))$$

- **Reflexive**: satisfied by the tight bounds $\Theta, O, \Omega$

$$f(n) \in \Theta(f(n))$$

- **Symmetric**: satisfied by $\Theta$

$$f(n) \in \Theta(g(n)) \Longrightarrow g(n) \in \Theta(f(n))$$

### Analysis of Insertion Sort

```
for j = 2 to A.length          // ran (n-1)+1 times
  Key = A[j]                   // ran n-1 times
  i = j - 1                    // ran n-1 times
  while i > 0 && A[i] < Key    // ran sum_(j=2)^n t_j times
    A[i+1] = A[i]              // ran sum_(j=2)^n (t_j - 1) times
    i = i - 1                  // ran sum_(j=2)^n (t_j - 1) times

  A[i+1] = Key                 // ran n-1 times
```

Where $t_j$ is the number of times the while loop is tested on the $j$th cycle.

- Best case: $t_j = 1$ then $T(n) = pn + q$
- Worst case: $t_j = j$ then $T(n) = pn^2 + qn + r$
- Average case: the claim is that on average, half of the keys in $A[1...j - 1]$ will be less than $A[j]$

  $t_g = j/2$ gives $T(n) \in O(n^2)$

The worst case is useful because
- It gives the upper bound on resource
- Often the same as the average case

Insertion sort is an **incremental algorithm**: it builds up an output that satisfies some properties.

## Divide and Conquer

1. Split into 2 or more smaller subproblems.
2. call the same algorithm on each subproblem recursively.
3. Combine solutions to the subproblems to build the solution to the original problem.

> **Note**
> Recursion will terminate because the subproblem will get smaller and smaller.

### Merge Sort

```
// we are sorting A[p..r]
if p < r
  q = floor((p + r) / 2)
  MergeSort(A, p, q)
  MergeSort(A, q + 1, r)
  Merge(A, p, q, r)
```

And `Merge` defined as

```
n1 = q - p + 1
n2 = r - q

L = new Array(1 .. n1 + 1)
R = new Array(1 .. n2 + 1)

L[1 .. n1] = A[p .. q]
L[n1 + 1] = infinity
R[1 .. n2] = A[q + 1 .. r]
R[n2 + 1] = infinity

i = j = 1

for k = p to r
  if L[i] <= R[j]
    A[k] = L[i]
    i = i + 1
  else
    A[k] = R[j]
    j = j + 1
```

---

- If the length of the array is not a power of 2, pad $\infty$ to the end so that it is.
- After sorting, remove the added $\infty$ at the end of the sorted array.

The input array is modified, `Merge` has no return value.

**Recurrence Relations**

The input size is lengh of the region to be sorted $n = r - p + 1$

Let $T(n)$ be the cost of solving $\text{MergeSort}(A, p, r)$

- If $p = r$, $T(1) = 1$
- If $p < r$

<table>
<tr><th>Action</th><th>Cost</th></tr>
<tr><td>Calculate $q$</td><td>$\Theta(1)$</td></tr>
<tr><td>Calls itself on 2 subproblems</td><td>$T(n/2) \times 2$</td></tr>
<tr><td>Calls $\text{Merge}(A, p, q, r)$<br><table><tr><th>Action</th><th>Cost</th></tr><tr><td>Creates 2 arrays of length $n + 2$</td><td>$\Theta(n)$</td></tr><tr><td>Loop $n$ iterations: assign into array and increment $i$ or $j$</td><td>$\Theta(n)$</td></tr></table></td><td>$\Theta(n)$</td></tr>
</table>

$$\begin{aligned} T(1) &= 1 \\ T(n) &= \Theta(1) \text{ work} + 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \text{ work} \\ &= k_1 + 2 \cdot T\left(\frac{n}{2}\right) + k_2 \cdot n \end{aligned}$$

**Definition**

A **closed form solution** is not defined in terms of itself through direct or indirect recursion.

$$\begin{aligned} T(n) &= k_1 + k_2 \cdot n + 2 \cdot T\left(\frac{n}{2}\right) \\ &= k_1 + k_2 \cdot n + 2 \cdot \left(k_1 + k_2 \cdot \frac{n}{2} + 2 \cdot T\left(\frac{n}{4}\right)\right) \\ &= k_1 + k_2 \cdot n + 2 \cdot \left(k_1 + k_2 \cdot \frac{n}{2} + 2 \cdot \left(k_1 + k_2 \cdot \frac{n}{4} + 2 \cdot T\left(\frac{n}{4}\right)\right)\right) \\ &\vdots \\ &= k_1 \cdot \underbrace{(1 + 2 + 4 + ...)}_{\log n \text{ terms}} + k_2 \cdot n \cdot \underbrace{(1 + 1 + 1 + ...)}_{\log n \text{ times}} + 2^{\log n} \cdot T(1) \\ &= k_1 \cdot (n - 1) + k_2 \cdot n \log n + n \\ &\in \Theta(n \log n) \end{aligned}$$

**Note**

We preserved the equal signs instead of saying “this term dominates” so we know $T(n) \in \Theta(f(n))$ instead of just $O(f(n))$

If the array length is not a power of 2

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + k_1 + k_2 \cdot n$$

Which gives the same solution.

**The Master Theorem**

Let $a \geq 1$ and $b > 1$ be constants.

- $T(1) = 1$
- $T(n) = a \cdot T(n/b) + f(n)$

**Note**
$n/b$ can be interpreted as ceil or floor, it doesn't matter.

$$f(n) \in O\left(n^{-\varepsilon + \log_b a}\right) \text{ for some } \varepsilon > 0 \Longrightarrow T(n) \in \Theta(n \log_b a)$$

$$f(n) \in \Theta\left(n^{\log_b a}\right) \Longrightarrow T(n) \in \Theta(n \log_b a \cdot \lg a)$$

$$f(n) \in \Omega\left(n^{\varepsilon + \log_b a}\right) \text{ for some } \varepsilon > 0 \text{ and } f(n/b) \leq c f(n)$$
$$\text{for some } c > 1 \text{ for all sufficiently large } n \Longrightarrow T(n) \in \Theta(f(n))$$

**Note**
There is an extended master theorem for conditions between case 2 and 3.

---