CHAPTER 1

ALGORITHMS

1.1 Analysing Algorithms

An algorithm is a well-defined procedure that is used to solve a problem in a finite period of time, e.g. sorting an array. It can be written in natural language, computer language (pseudocode or an actual language) or hardware design. There are many applications of algorithms, not just in computing!
Algorithm analysis aims to check the correctness and the running time of the algorithm.

Example of an algorithm: Insertion sort

The algorithm `insertionSort` is used to sort arrays. The following is a psuedocode for the algorithm:

```
int insertionSort(Array<int> array):
    // loop through the entire array
    for (int j = 1; j < n; j++):
        int key = array[j]
        int i = j-1
        // shift all the elements in the sorted sublist that are
        // bigger than key
        while (i >= 0 and array[i] > key):
            array[i+1] = array[i]
            i--
        array[i+1] = key
```

This algorithm works by increasing the sorted sublist from the left by comparisons. We explain this with an example. Assume that we want to sort the array `[5, 2, 4, 6, 1]` using insertion sort. At the start, our sorted sublist only consists of the first element 5. During the first iteration, we have the following setting for the array:

$$5 \;\Big|\; \underline{2} \quad 4 \quad 6 \quad 1$$

The number 2 (underlined black) is the `key`. We want to add it to the sorted sublist (to the left of the blue bar), so we must place it before 5.

$$\underline{5} \;\Big|\; 5 \quad 4 \quad 6 \quad 1$$

So, we make the element at the second position 5. The location where the `key` moves is highlighted in red (this is the value of `i+1`).

$$2 \quad 5 \;\Big|\; \underline{4} \quad 6 \quad 1$$

Now, the sorted sublist has 2 elements. We now aim to move the next element 4 into the sublist- it should be placed between 2 and 5.

$$2 \quad \underline{5} \,|\, 5 \quad 6 \quad 1$$

Because 4 is to be placed between 2 and 5, we only copy 5 to the third position; 2 doesn't get copied to the second position. We now add the element 4 at the second position.

$$2 \quad 4 \quad 5 \,|\, \underline{6} \quad 1$$

Now, we want to place 6 within the sorted sublist. It is at the correct position right now.

$$2 \quad 4 \quad 5 \,|\, \underline{6} \quad 1$$

Since 6 is already at the correct position, we have nothing to move- 6 stays where it is.

$$2 \quad 4 \quad 5 \quad 6 \,|\, \underline{1}$$

Now, we shall move 1 to the start of the array.

$$\underline{2} \quad 2 \quad 4 \quad 5 \,|\, 6$$

To move 1 to the start of the array, we shift all the elements within the sorted array one to the right. The element 1 gets placed at the first position.

$$1 \quad 2 \quad 4 \quad 5 \quad 6$$

After we place 1 at the start, there is nothing to the right of the sorted sublist. This means that we have sorted the entire array.
Insertion sort is a stable sorting algorithm, meaning that the relative order of the elements with the same value are maintained. Also, the sorting algorithm is in-place, meaning that we don't need to create another array in the process.

Analysing algorithms

We can look at the running time to analyse algorithms. We typically focus on the worst case.
This can be done using experimental studies- we run the algorithm with a single change (e.g. a larger size of array), and see how that affects the time it takes for the algorithm to run. Such studies are called empirical studies.
Experimental studies have many limitations. We need to implement the algorithm, and this might be difficult. It is not possible to conclude that the difference we see in the running time for two different inputs comes just from the difference (e.g. the operating system is more busy when running only one of the algorithms). Nevertheless, we consider algorithms run on the same hardware and software as being comparable.
Instead, we look into theoretical approaches. We don't need to implement the algorithm. Also, it is possible to consider all the input sizes. Moreover, we can evaluate the speed of the algorithm irrespective of the hardware and the software environment.

Counting Primitive operations

Primitive operations are basic computations performed in an algorithm. Their implementation is not important because we assume it takes a constant amount of time. Examples of primitive operations include assigning a value, indexing an array, calling a method, etc.
Consider the following algorithm to compute the element with maximum value in an array, which also highlights the primitive operations:

```
1 int max(Array<int> array):
2     int max = array[0]
3     for (int i = 1; i < array.length; i++):
4         if array[i] > max:
5             max = array[i]
6     return max
```

In line 2, we have 2 primitive operations- assigning a variable a value, and indexing an array. Similarly, returning a value in line 6 is a primitive operation. The for loop in line 3 isn't a primitive operation. This is because it is repeated `array.length` times. So, line 3 has `2n` primitive operations for assigning a value and checking whether `i < array.length`; and `2(n-1)` for assigning and incrementing `i`. Here, `n` is the length of the array. Similarly, any operation within the body of the for loop runs `n-1` primitive operations (since we're starting for `1`, not `0`). The if condition runs `n-1` times only if we're in the worst case; at the best case, the condition doesn't run once. In total, there are `8n-3` primitive operations in the worst case; in the best case, there are `6n-1` primitive operations.
To estimate the running time based on the primitive operations, we define the function T that is the running time of the worst-case of the algorithm. Then, $a(8n-3) \leq T(n) \leq b(8n-3)$. So, $T(n)$ is bounded by two linear functions.
When we change the hardware and the software environment, the function T is only affected by a constant factor- the function will still be linear. That is, the algorithm will always have a linear growth rate.
The growth rate of the running time of an algorithm isn't affected by constant factors and lower-degree terms.

Big-Oh Notation

If we have functions f and g, then we say that f is $O(g)$ if there are constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. That is, the big-oh notation gives us an upper bound up to a constant factor.
If $f = O(g)$, then g grows more than f or f and g have the same growth rate- it isn't possible for f to grow more than g. On the other hand, if $g = O(f)$, then f grows more than g or f and g have the same growth rate.
The Big-Oh notation allows us to perform asymptotic algorithm analysis. That is, we are comparing the algorithm with respect to the input size- it is evaluating the scalability of an algorithm. It is used to judge the asymptotic efficiency of an algorithm. In particular, the Big-Oh notion is used to express asymptotic upper bounds.
When we have a function f, the following are the rules for Big-Oh:

- If f is of degree d, then $f = O(n^d)$- we can drop lower order terms and any constant factors.

- If $f = O(n)$ and $f = O(n^2)$, we just say $f = O(n)$ because we want the tightest bound to the function.
- If $f = O(3n + 5)$, we write $f = O(n)$- we drop lower order terms and constant factors.

Now, assume we want to show that $2n + 10$ is $O(n)$. First, define the functions $f(n) = 2n + 10, g(n) = n$. We now need to find constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. We can substitute and simplify as follows:

$$2n + 10 \leq cn \iff (c - 2)n \geq 10 \iff n \geq \frac{10}{c - 2}.$$

So, we can set $c = 3$ and $n_0 = 10$.
Now, let's say we want to show that $7n - 2$ is $O(n)$. So, we have functions $f(n) = 7n - 2, g(n) = n$. Also, define constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. Substituting, we get

$$7n - 2 \leq cn \iff (c - 7)n \geq -2 \iff n \geq \frac{-2}{c - 7}.$$

So, we can set $c = 8$ and $n_0 = 1$.
We can also try to show that n^2 is not $O(n)$. We do the calculations like above- set $f(n) = n^2$ and $g(n) = n$. Then, let c and n_0 be such that for $n \geq n_0 > 0$,

$$n^2 \leq cn \iff n \leq c.$$

But, we should be able choose $n \geq n_0$, which at some point will cross c, so n^2 is not $O(n)$.
We now try to show that $3n^3 + 20n^2 + 5$ is $O(n^3)$. So, let $f(n) = 3n^3 + 20n^2 + 5$ and $g(n) = n^3$, and define constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. Substituting, we get

$$3n^3 + 20n^2 + 5 \leq cn^3 \iff n^3(c - 3) \geq 20n^2 + 5 \iff n^3 \geq \frac{20n^2 + 5}{c - 3}.$$

So, we can set $c = 4$ and $n_0 = 3$.
Finally, we want to show that $3 \log n + 5$ is $O(\log n)$. So, let $f(n) = 3 \log n + 5$ and $g(n) = \log n$, and define constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. Substituting, we get

$$3 \log n + 5 \leq c \log n \iff (c - 3) \log n \geq 5 \iff \log n \geq \frac{5}{c - 3}.$$

So, we can set $c = 4$ and $n_0 = 2^5 = 32$.
We now look at another Big-Oh rule:

- If $T_1 = O(f)$ and $T_2 = O(g)$, then $T_1 + T_2 = O(\max(f, g))$.
- If $T_1 = O(f)$ and $T_2 = O(g)$, then $T_1T_2 = O(fg)$.
- If $T(n) = (\log n)^k$, then $T(n) = O(n)$ for all k.

Computing Running Time

There are many rules when we compute the running time. These are:

- If we have a loop, then the running time is the running time of the body of the loop multiplied by the number of times the loop is repeated.
- In particular, if we have nested loops, then the running time gets multiplied. For example, the running time of the function below is $O(n^3)$ since there are 3 loops of length n:

```
void algorithm(int n):
    int x = 0;
    for (int i = 0; i < n; i++):
        for (int j = 0; j < n; j++):
            for (int k = 0; k < n; k++):
                x++;
```

- If we have consecutive statements, we just take the maximum since it is additive.
- If we have many if-then-else clauses, we take the branch with the highest running branch.

We revisit the insertion sort now. Below is the psuedocode which highlights the number of operations per line.

```
int insertionSort(Array<int> array):
    // 2n + 2(n-1) operations
    for (int j = 1; j < n; j++):
        // 2(n-1) operations
        int key = array[j]
        // n-1 operations
        int i = j-1
        // (max) n(n+1) operations
        while (i >= 0 and array[i] > key):
            // (max) n(n+1)-1 operations
            array[i+1] = array[i]
            // (max) n(n+1)-1 operations
            i--
        // n-1 operations
        array[i+1] = key
```

In the best case, we never enter the while loop- the array is already sorted. The running time is $O(n)$. On the other hand, the `while` loop runs from `i=j-1` to `i=0` each iteration in the worst case, so in total it runs

$$1 + 2 + 3 + \cdots + n = \frac{1}{2}n(n+1)$$

times. So, the running time is $O(n^2)$.

Common growth rates

The following are common growth rates:

- $O(1)$ or constant;
- $O(\log n)$ or logarithmic;

- $O(\sqrt{n})$ or fractional power;
- $O(n)$ or linear;
- $O(n \log n)$ or quasilinear;
- $O(n^2)$ or quadratic;
- $O(n^3)$ or cubic.

They are listed in order of runtime with respect to growth rate- $O(1)$ is the fastest, while $O(n^3)$ is the slowest (in this array).
If we want to show that $\sqrt{n}$ lies between $\log n$ and n, we can show $\sqrt{n} = O(n)$ and $\log n = O(\sqrt{n})$. Taking $c = 1$ and $n_0 = 4$, we have $\sqrt{n} \leq cn$, while taking $c = 1$ and $n_0 = 8$, we have $\log n \leq c\sqrt{n}$.
Further complexities $O(2^n)$ (exponential) and $O(n!)$ (factorial) are possible, but are exceptionally slow in practice.
We now consider the running time of few algorithms. First, consider the following psuedocode

```
// computes the sum of the squares between 0 and n
int squaresSum(int n):
    int i = 0 // O(1)
    int sum = 0 // O(1)
    while (i < n): // O(n)
        i++ // O(n)
        sum += i*i // O(n)
    return sum // O(1)
```

The highest operation here is $O(n)$, so the function is $O(n)$. The following algorithm also performs the same computation, but using a formula

```
// computes the sum of the squares between 0 and n
int squaresSum(int n):
    int sum = n*(n+1)*(2n+1)/6 // O(1)
    return sum // O(1)
```

This function is $O(1)$. So, this function is intrinsically faster than the previous function. Now, consider the following function.

```
// computes the integer part of the square root of n
int integerPart(int n):
    int i = 0 // O(1)
    while ((i+1)*(i+1) <= n): // O(sqrt(n))
        i++ // O(sqrt(n))
    return i // O(1)
```

We can find that the while loop is $O(\sqrt{n})$ by considering the number of iterations it takes for some integer n. For example, if $n = 13$, the loop runs 3 times (when the expression `(i+1)*(i+1)` equals 1, 4 and 9). Moreover, at the worst case, the loop runs $\sqrt{n}$. Since that is the highest operation here, the function is $O(\sqrt{n})$.
Now, consider the following algorithms:

```
void algorithm1(int n):
    int sum = 0 // O(1)
    for (i = 0; i < n; i++): // O(n)
        for (j = 0; j < i; j++): // O(n^2)
            sum++
```

Here, the inside loop runs for

$$1 + 2 + 3 + \cdots + n = \frac{1}{2}n(n+1) = O(n^2)$$

times. So, the running time of the algorithm is $O(n^2)$.
The next algorithm to consider is:

```
1 void algorithm2(int n):
2     int sum = 0 // O(1)
3     for (int i = 0; i < n; i++): // O(n)
4         for (j = 0; j < i*i; j++): // O(n^3)
5             for (k = 0; k < j; k++): // O(n^5)
6                 sum++
```

Here, the loop in line 3 runs n times; the loop in line 4 runs at most n^2 times for some `i`; while the loop in line 5 runs at most n^2 times for some `j`- multiplying, we find that the algorithm has runtime $O(n^5)$.
We now consider the final algorithm below:

```
void algorithm3(int n):
    int sum = 0 // O(1)
    for (int i = 1; i <= n; i++): // O(n)
        for (int j = 1; j <= i*i; j++): // O(n^3)
            if (j % i == 0): // O(n^3)
                for (int k = 1; k <= j; k++): // O(n^4)
                    sum++ // O(n^4)
```

The `if` condition is only true if `j` is `i`, `2i`, and so on up to `i*i`- this loop runs `i` times with respect to `j`. So, this algorithm has runtime $O(n^4)$.

More asymptotic notations

Now, we define more asymptotic notations:

- $f = \Omega(g)$ if there are constants c and n_0 such that $f(n) \geq cg(n)$ for $n \geq n_0$, i.e. f is bounded below by a factor of g.
- $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$, i.e. the functions grow at the same rate.

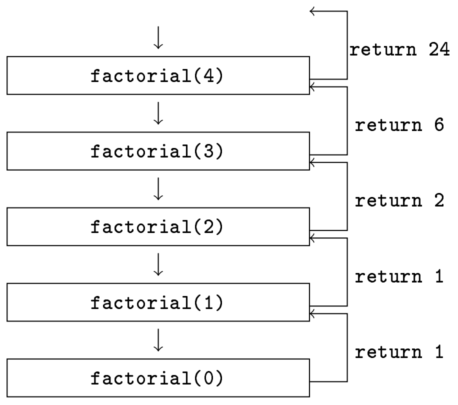
For example, let $f(n) = 5n^3 + 2$. Then, $f = O(n^3)$ (using $c = 6, n_0 = 2$), but also $f = O(n^4)$ or any bigger power. Similarly, $f = \Omega(n^3)$ (using $c = 4, n_0 = 0$) and $f = \Omega(n^2)$ or any smaller power. But, $f = \Theta(n^3)$ only, and not any other power- this is a tight bound.

1.2 Recursive Algorithms

A function is recursive if it is refers to itself from within the function. An example of this is the factorial function:

```
// returns the factorial of the number n
int factorial(int n):
    // base case
    if (n == 0):
        return 1
    // recursive case
    return n * factorial(n-1)
```

A recursive algorithm calls itself, but with different parameter(s) so that we get closer to the base case. Therefore, a recursive algorithm must have a base case for the algorithm to terminate.
One way to study a recursive algorithm is using a recursion trace. An example of the recursion trace for the `factorial` function is shown below with `n = 4`:



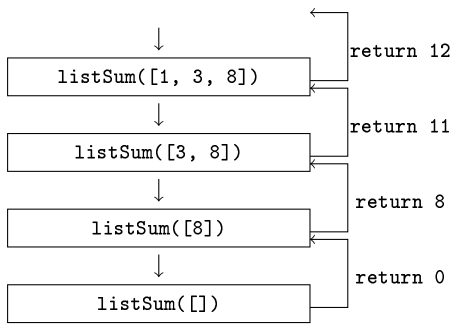
Therefore, the final output is 24.

Linear Recursion

A linear recursion makes at most one recursive call per invocation. The amount of space needed to track the nested calls grows linearly. Such functions are linear because it grows at the same rate as the input, e.g. the recursion trace grows one every time we increase the input.
Another example of a linear recursion is the sum of the values within an array.

```
// returns the sum of the array
int listSum(Array<int> array):
    // base case
    if (array.length == 0):
        return 0
    // recursive case
    return array[0] + listSum(array[1:])
```

This function is a terminating recursive function since there is a base case and the `array` always gets smaller. We now look at the recursive trace with `array = [1, 3, 8]`:



The output, in this case, is 12. Moreover, both the functions `factorial` and `listSum` are $O(n)$.

Tail Recursion

Recursion is very useful in designing short and intuitive algorithms. But, in practice, it has a cost of memory per recursive call- we keep track of all the calls in the recursive trace by a stack. We can always make a non-recursive algorithm for a recursive one if this is a big issue.
However, sometimes we can be more efficient by just using `tail recursion`. In tail recursion, the recursion is linear and the last operation in the algorithm. An example of tail recursion is the following algorithm:

```
// reverses an array
void reverse(Array<int> array, int i, int j):
    if (i < j):
        array[i], array[j] = array[j], array[i]
        reverse(array, i+1, j-1)
```

We start the algorithm with `reverse(array, 0, array.length-1)`. This is tail recursive since the final line is the only place where we find the recursive call, and the call is linear since there is only one call. The function `reverse` was also tail recursive for the same reason. However, the function `factorial` was not tail recursive since it ended with `return n * factorial(n-1)`- we perform an operation on the value. We can rewrite the function as shown below to make it tail recursive.

```
int factorial(int n, int acc):
    if (n == 0):
        return acc
    return factorial(n-1, n * acc)
```

We can also write this algorithm as an iterative algorithm:

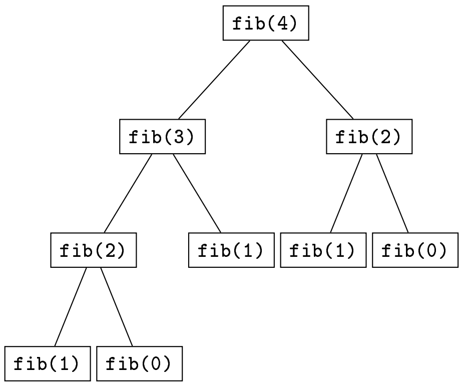
```
int factorial(int n, int acc):
    int i = n
    int factorial = 1
    while (i > 0):
        factorial *= i
        i--
    return factorial
```

Binary recursion

A binary operation makes two recursive calls. An example of this is computing the fibonacci number:

```
int fib(int n):
    // base cases (n = 1 or n = 0)
    if n <= 1:
        return n
    // binary recursion case
    return fib(n-1) + fib(n-2)
```

The algorithm is very inefficient. This has a recursion tree shown below for `n = 5`.



This implementation is very inefficient since there are a lot of duplicates. For example, the right branch of `fib(4)` is also the left branch of `fib(3)`. Since the recursion tree grows twice with respect to input size, the running time of this function is $O(2^n)$.

Analysing Recursive Algorithms

The running time $T(n)$ of a recursive algorithm can be described by a recurrence equation. For example, consider the factorial function below

```
int factorial(int n):
    // T(0) = O(1)
    if (n == 0):
        return 1
    // T(n) = T(n-1) + O(1)
    return n * factorial(n-1)
```

So, the base case has running time $T(0) = O(1)$- it is $T(0)$ since the base case applies when `n=0`. On the other hand, the recursive case has running time $T(n) = T(n-1) + O(1)$. This is annotated in the pseudocode above.
We can solve the recursive equation iteratively. We find that

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ &= (T(n-2) + O(1)) + O(1) \\ &= T(n-2) + 2O(1) \\ &= T(n-3) + 3O(1) \\ &= \cdots \\ &= T(0) + nO(1) \\ &= (n+1)O(1). \end{aligned}$$

Therefore, T is $O(n)$.

Another way of computing the running time is using recursion tree. We draw until the base case and then use its running time and add the running time for each level, and then multiplying it by the height of the tree.
Finally, we can apply the Master theorem to compute the running time. This is a generalisation of recursive tree. It states that if the recurrence is

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1, b \geq 1$, we have one of the three cases:

- If $f = \Theta(n^c)$, where $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$;
- If $f = \Theta(n^c)$, where $c = \log_b a$, then $T(n) = \Theta(n^c \log n)$;
- If $f = \Theta(n^c)$, where $c > \log_b a$, then $T(n) = \Theta(f)$.

1.3 Sorting Algorithms

There are many possible approaches in finding a sorting algorithm. For example, insertion sort takes an incremental approach- we assume that a sublist is sorted, and we expand that sublist one at a time. This is not the most optimal approach since it is of order $O(n^2)$.

Selection Sort

Selection sort is another incremental approach that aims to expand the sublist by adding the smallest element to the sorted sublist from the unsorted subsection by performing swaps. It has the following pseudocode:

```
void selectionSort(Array<int> array):
    for (int i = 0; i < n-1; i++):
        // compute the index of the minimum element in the
        // unsorted sublist
        int min = i
        for (int j = i; j < n; j++):
            if (array[j] < array[min]):
                min = j
        // swap the element in this index with the minimum element
        array[i], array[min] = array[min], array[i]
```

We now visualise how the algorithm works using the array `[2, 5, 3, 1, 4, 7]`. The initial state of the array, after entering the loop, is:

$$\vert\; 2 \quad 5 \quad 3 \quad \underline{1} \quad 4 \quad 7$$

The blue wall represents the sorted sublist- with every iteration, it increases by 1. At the start, there is no element in this sublist. We now add the smallest element `1` into the first index by swapping it with the element just after the blue wall.

$$1 \;\vert\; 5 \quad 3 \quad \underline{2} \quad 4 \quad 7$$

Now, we swap the smallest element `2` with the element after the wall `5`.

$$1 \quad 2 \;\vert\; \underline{3} \quad 5 \quad 4 \quad 7$$

Here, the smallest element is the element just after the blue wall- we perform no swaps.

$$1 \quad 2 \quad 3 \;\vert\; 5 \quad \underline{4} \quad 7$$

Then, we swap the elements `5` and `4`.

$$1 \quad 2 \quad 3 \quad 4 \;\vert\; \underline{5} \quad 7$$

Here also, we make no swaps since `5` is the smallest element in the unsorted sublist. Then, we end up at the last element in the array- this element must have been bigger than all the elements before it. Therefore, we have sorted the array. The sorted array is

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 7$$

In the pseudocode, the maximum number of iterations with respect to `i` is `n-1`, while the maximum number of iterations with respect to `j` is `n-1`. So, the running time of selection sort is $O(n^2)$.

Merge sort

We now look at divide-and-conquer approaches of sorting. This is typically a recursive algorithm and has a better running time than an incremental approach. First, we divide a problem into smaller subproblems; solve the subproblems recursive, until we get to the base case; and combine the solutions to form the solution to the bigger problem. Such algorithms are more efficient than incremental algorithms. Examples of this approach are merge and quick sort.
Merge sort is a divide-and-conquer algorithm, so we first divide the input into subarrays of half the length; sort the subarrays recursively using merge sort; and finally combine the merged subarrays to sort the entire array.
The combination process, called `merge` is given below:

```
// array[p:q] is the left subarray, while array[q:r] is the right
// subarray. Both subarrays are sorted.
void merge(Array<int> array, int p, int q, int r):
    Array<int> left = array[p:q]
    Array<int> right = array[q+1:r]
    left.add(Integer.MAX_VALUE)
    right.add(Integer.MAX_VALUE)
    int i = 0
    int j = 0
    for (int k = p; k <= r; k++):
        // if the left elt is smaller, add that
        if (left[i] <= right[j]):
            array[k] = left[i]
            i++
        // otherwise, add the elt from the right
        else:
            array[k] = right[j]
            j++
```

Using `Integer.MAX_VALUE`, we can avoid extra conditions to check when the two arrays have been scanned. We show how this function works by considering how it merges the array `[1, 3, 4, 2, 5, 6]` with `p=0`, `q=2` and `r=5`:

1 3 4 | 2 5 6

By the definition of `p`, `q` and `r`, we can divide the array into 2 left and right sublists, both of which are sorted. We make a copy of these arrays, and enter the while loop:

`array` $\underline{1}$ 3 4 2 5 6 `left` $\underline{1}$ 3 4 `right` $\underline{2}$ 5 6

The variable `k` keeps track of the indices within the array; the variable `i` keeps track of the indices within the left sublist; and the variable `j` keeps track of the indices within the right sublist. Since the highlighted element in the left sublist `1` is smaller than `2`, we place `1` in the first position of the array. We then increment `k` and `i`.

`array` 1 $\underline{3}$ 4 2 5 6 `left` 1 $\underline{3}$ 4 `right` $\underline{2}$ 5 6

Now, since `2` is smaller than `3`, we add the element from the second array into the array. This time, we increment `k` and `j`.

`array` 1 2 $\underline{4}$ 2 5 6 `left` 1 $\underline{3}$ 4 `right` 2 $\underline{5}$ 6

Since `3` is smaller than `5`, we now add `3` into the array.

`array` 1 2 3 $\underline{2}$ 5 6 `left` 1 3 $\underline{4}$ `right` 2 $\underline{5}$ 6

Still, the element `4` in the left sublist is smaller than the one in the right sublist, we add `4` into the array.

`array` 1 2 3 2 $\underline{5}$ 6 `left` 1 3 4 $\underline{\ }$ `right` 2 $\underline{5}$ 6

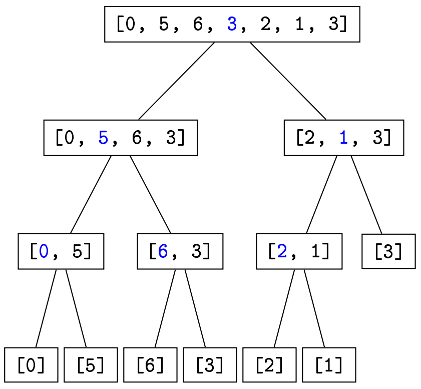
We have gotten to the end of the left sublist, so we add the elements from the right sublist in the order:

`1 2 3 2 5 6`

The two sublists is now merged.
Since we iterate over all the elements, the merge algorithm is $O(n)$. Moreover, the procedure is stable, but not in place. It requires linear memory. Now, we consider the overall merge sort, using the merge procedure:

```
void mergeSort(Array<int> array, int p, int r):
    if (p < r):
        int q = (p + r)/2
        mergeSort(array, p, q)
        mergeSort(array, q+1, r)
        merge(array, p, q, r)
```

The merge sort divides the array, and aims to merge all the divided arrays. We can see how this function works by considering `mergeSort([0, 5, 6, 3, 2, 1, 3])`:



The element highlighted in blue is the value of `q`. After breaking all the elements, we merge the arrays to get `[0, 1, 2, 3, 3, 5, 6]`.
Like the merge algorithm, merge sort is stable and not in-place. Its running time is $O(n \log n)$ in both the best and the worst case. We can prove this using

iterative substitution. The base case (i.e. $p \geq r$) has running time $T(1) = O(1)$, while the recursive case has running time $T(n) = T(n/2) + T(n/2) + O(n) = 2T(n/2) + O(n)$- the sorting of the smaller sublists is $T(n/2)$, while the merge is $O(n)$. We now substitute this iteratively:

$$\begin{aligned}
T(n) &= 2T(n/2) + O(n) \\
&= 2(2T(n/4) + O(n/2)) + O(n) \\
&= 4T(n/4) + 2O(n) \\
&= 8T(n/8) + 3O(n) \\
&= \cdots \\
&= 2^k T(n/2^k) + kO(n).
\end{aligned}$$

The base case happens when $n = 1$, i.e. $n = 2^k$. So, $k = \log n$, meaning that $T(n) = nO(1) + \log nO(n)$. This implies that merge sort is $n \log n$.
We could also have computed this using recursion tree. We would find that, at each level, we have running time of $O(n)$. Since the tree has height $\log n$, the running time of the entire method is therefore $O(n \log n)$.
Similarly, we can also use the Master theorem. In this case, we have $c = 1$ and $\log_2 2 = 1 = c$. So, $T(n) = \Theta(n \log n)$, as we found.
We can improve merge sort in many ways:

- We can make the algorithm in place by amending the `merge` algorithm.
- We can make the `mergeSort` algorithm iterative.
- We can make use of `insertionSort` when the array size is small. Cutoffs are typically set between 5 and 20 elements.

Quick sort

Quick sort is another divide-and-conquer sorting algorithm. We choose a pivot element and the division aims to break an array into two sublists- one containing all the elements smaller than the pivot, while the other one contains all the elements bigger than the pivot. Doing this recursively ensures that we end up with a sorted array- there is no combination step. The following is the way we partition the array, choosing the pivot to be the last element.

```
int partition(Array<int> array, int p, int r):
    int x = array[r]
    int i = p-1
    for (int j = p; j < r; j++):
        if (array[j] <= x):
            i++
            array[i], array[j] = array[j], array[i]
    array[i], array[i+1] = array[i+1], array[i]
    return i+1
```

We illustrate how this algorithm works using the array `[0, 7, 5, 2, 4, 1, 3]` and `p=0`, `r=6`. We start with the following state:

$$|\,\underline{0}\ \ 7\ \ 5\ \ 2\ \ 4\ \ 1\ \ \boxed{3}$$

The pivot `x` is the element in the box. The blue wall keeps track of the sublist that is smaller than the pivot and represents the variable `i`. Within the for loop, `j` is represented by an underline. When we first enter the while loop, we check whether `1` is smaller than the pivot- since `1` is smaller, we swap it with the first element after the blue wall, i.e. with itself. We then move the blue wall.

$$0 \mid \underline{7} \;\; 5 \;\; 2 \;\; 4 \;\; 1 \;\; \boxed{3}$$

Now, we check whether `7` is smaller than the pivot- it isn't, so we just move onto the next element.

$$0 \mid 7 \;\; \underline{5} \;\; 2 \;\; 4 \;\; 1 \;\; \boxed{3}$$

Since `5` is also bigger than `3`, we do nothing in this iteration.

$$0 \mid 7 \;\; 5 \;\; \underline{2} \;\; 4 \;\; 1 \;\; \boxed{3}$$

Now, since `2` is smaller than the pivot, we swap it with the first element after the blue wall. We then increment the blue wall.

$$0 \;\; 2 \mid 5 \;\; 7 \;\; \underline{4} \;\; 1 \;\; \boxed{3}$$

As `4` is bigger than the pivot, nothing changes in this iteration.

$$0 \;\; 2 \mid 5 \;\; 7 \;\; 4 \;\; \underline{1} \;\; \boxed{3}$$

In the final iteration, we swap `1` and `5`.

$$0 \;\; 2 \;\; 1 \mid 7 \;\; 4 \;\; 5 \;\; \boxed{3}$$

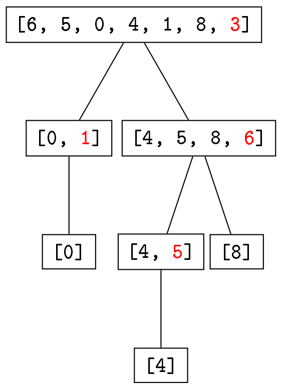
After exiting the while loop, we swap `3` with the first element after the blue wall to give the partitioned array.

$$0 \;\; 2 \;\; 1 \mid 3 \mid 4 \;\; 5 \;\; 7$$

We now recursively partition the two sublists until we end up with a sorted array. The number returned gives us access to the start and the end of these sublists.
The running time of the partition function is $O(n)$. The function is in place, but not stable. Using the algorithm, we can define the quicksort algorithm:

```
void quickSort(Array<int> array, int p, int r):
    if (p < r):
        int q = partition(array, p, r)
        quickSort(array, p, q-1)
        quickSort(array, q+1, r)
```

The recursion tree for the array `[6, 5, 0, 4, 1, 8, 3]` is:



The element highlighted in red is the pivot.
The running time of median depends on the choice of the pivot. In the best case, the pivot is always the median. Then, the recurrence equation is $T(n) = 2T(n/2) + O(n)$. This is the same equation as merge sort, so it is $O(n \log n)$.
On the other hand, if we are at the worst case, the pivot is very unbalanced- we end up with one sublist of size `n-1` and another of size `0`. The recurrence equation in that case is

$$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n)$$

since $T(0) = O(1)$. We find that

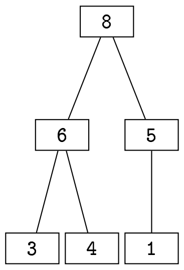
$$\begin{aligned} T(n) &= T(n-1) + O(n) \\ &= T(n-2) + 2O(n) \\ &= \cdots \\ &= T(0) + nO(n). \end{aligned}$$

So, $T(n) = O(n^2)$. This only happens when the array is already sorted. On average, the algorithm is $O(n \log n)$.
We could make the algorithm better by choosing a different element, e.g. the middle. This could still degrade to $O(n^2)$. We can also consider the median of 3 numbers- the start, the middle and the end. It is a bit slower, and we could still degrade to $O(n^2)$. We could also choose the pivot randomly, although that is not always feasible/fast.
We can use a cutoff like in the case of merge sort. Here, we call insertion sort on the entire array- insertion sort performs better with nearly sorted arrays. We can tail-optimise and then we only have one recursion call. We can also make the quicksort iterative.
Finally, we can use a 3-way quicksort, where we have a third sublist containing all the elements equal to the pivot. It is a good algorithm if there are many duplicates.

Heap sort

Heap sort is an iterative algorithm that makes use of the heap data structure in order to sort the array. It iteratively extracts the maximum element from the unsorted sublist. Unlike selection sort, it is better because the heap data structure makes the search a linear operation.

A heap is a nearly complete binary tree that satisfies the heap property- if `p` is a parent node of `c`, then the value of `p` is greater than or equal to the value of `c`. We can implement a heap as an array. For example, the following represents a heap:



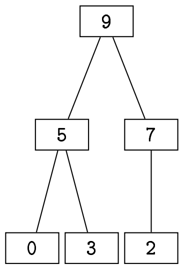
Here, we always find that the parent node of an element is bigger than the element, and vice versa. The heap is called nearly complete since all the level, except for the final one, are complete. We store it as an array going top to bottom and then from left to right. In this case, the array will be:

```
8 6 5 3 4 1
```

For an element in the array, its left child is in index $2n+1$ while its right child is at index $2n+2$.
Now, we consider the algorithm for heapsort. The pseudocode is:

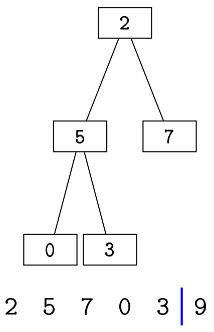
```
void heapSort(Array<int> array):
    createHeap(array)
    for (int i = array.length-1; i >= 0; i--):
        array[0], array[i] = array[i], array[0]
        heapify(array, 0, i)
```

First, we aim to transform an array into a heap structure. As a heap, we know that the maximum element is at the start. So, we swap it with the last element in the unsorted (left) sublist. Then, we re-establish the heap structure in the array and continue finding the maximum element.
We now consider how the algorithm works with the array `[3, 5, 2, 0, 9, 7]`.

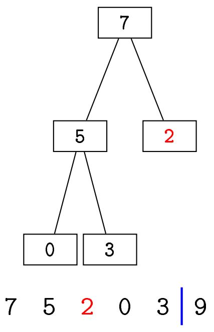


```
9 5 7 0 3 2
```

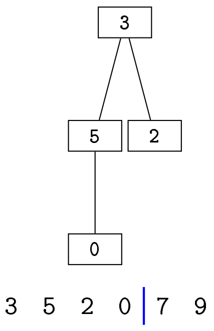
First, we transform the array into a heap.



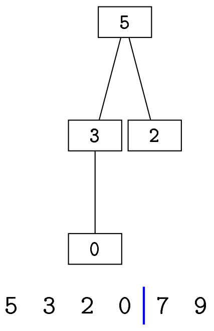
Now, we enter the for loop. The first thing we do is swap the first element 9 with the element at the end of the unsorted sublist (blue wall) 3. The heap now has one fewer element.



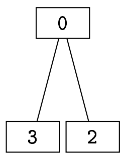
We then restore the heap structure to the array.



Next, we swap 7 and 3 and expand the sorted sublist.

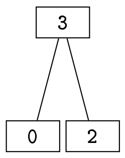


Again, we restore the heap structure.



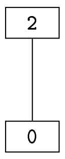
0 3 2 | 5 7 9

We now swap the maximum element 5 with 0.



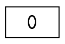
3 0 2 | 5 7 9

Again, we restore the heap property.



2 0 | 3 5 7 9

Now, we swap 3 and 2. This configuration still obeys the heap structure.



0 | 2 3 5 7 9

Finally, we have arrived at a heap with height 1. This implies that the array is sorted. So, the sort gives us

0 | 2 3 5 7 9

The algorithm is analogous to a variant of selection sort. Now, we consider the algorithms **`heapify`** that restores the heap structure.

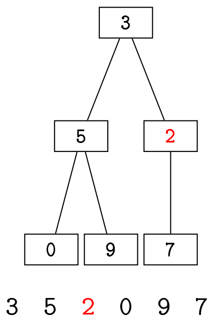
```
void heapify(Array<int> array, int i, int length):
    int left = 2*i + 1
    int right = 2*i + 2
    int largest = null
    // left bigger than parent -> the largest is left child
    if (left < length && array[left] > array[i]):
        largest = left
    else:
        largest = i
    // right bigger than left/parent -> the largest is right child
    if (right < length && array[right] > array[largest]):
        largest = right
    // swap largest with i and recurse if different
    if (largest != i):
        array[i], array[largest] = array[largest], array[i]
        heapify(array, largest, length)
```

It checks whether the left or the right child is larger than the element- if it is, we swap the child with the parent, and recurse on that child to ensure the heap structure is still satisfied.

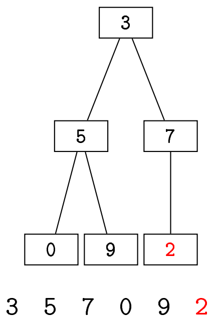
Also, the function `createHeap` makes use of `heapify` to convert the array into a heap.

```
void createHeap(Array<int> array):
    for (int i = (array.length/2) - 1; i >= 0; i--):
        array[0], array[i] = array[i], array[0]
        heapify(array, 0, i)
```

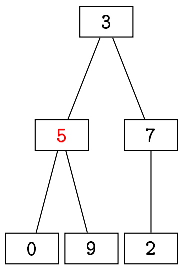
We now consider how we create the heap structure for the array `[3, 5, 2, 0, 9, 7]`.



Originally, the array doesn't satisfy the heap property. We enter the for loop and ensure the indices 2, 1 and 0 satisfy the heap property. We start with index 2. Since the (left) child of 2 is bigger, we swap the two elements.

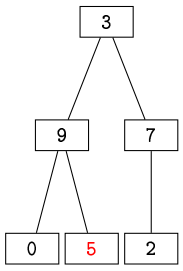


Now, we ensure that 2 satisfies the heap property. It has no children, so there is nothing to check here.



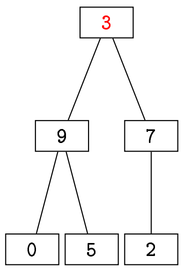
3 5 7 0 9 2

Now, we move onto index 1. The right child of 5 is bigger, so we swap the two elements.



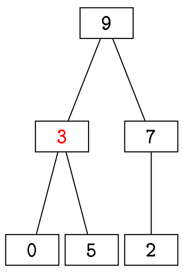
3 9 7 0 5 2

The element 5 has no children, so there is nothing to check here.



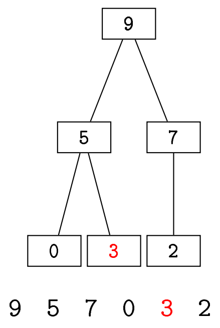
3 9 7 0 5 2

Finally, we move onto applying the `heapify` function on index 0. 3 is smaller than both 7 and 9, but 9 is bigger than the other child. So, we swap 3 and 9.



9 3 7 0 5 2

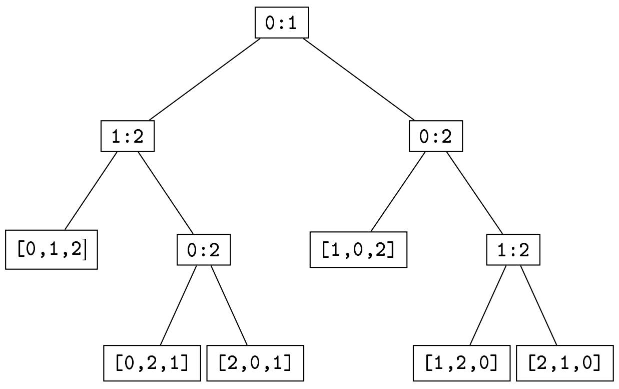
`3` is smaller than `5`, so we swap them.



Since `3` has no children, there is nothing more to check. We have acheived the heap structure.
The function `createHeap` is $O(n)$. The function `heapify` is $O(\log n)$- this is an improvement from the $O(n)$ version in selection sort. Since there are `n-1` iterations, the complexity of the algorithm is $O(n \log n)$. This is in both the best and the worst case. The algorithm is in place, but not stable.

Non-comparison Sorts

All the sorting algorithms we've looked at have been comparison sorts. That is, we compare elements to determine the permutation of the array that gives us a sorted array. We can consider all comparison sorts abstractly as a decision tree model. We only look at the comparisons in that case.
A decision tree is a complete binary tree that represents comparisons between the elements with respect to the input size. We expect the elements within the array to be distinct. Using the binary tree, we can find a lower bound to comparison sorts. For example, for an array of 3 elements, the decision tree is:



We only indicate the indices above. The leaf nodes represent the indices of the sorted array, while all the other nodes are decision nodes. For a condition `a:b`, if we go to the left, we have `a <= b`. On the other hand, if we go to the right, `a > b`. Depending on the comparisons, we can end up with all the 6 possible permutations of the three elements.
For a comparison sort, it must be possible to go down every path. Therefore, the worst running time is the height of the decision tree. We know that $n^{n/2} \leq$

$n! \le n^n$, and that a binary tree of height h has at most 2^h leaves. Therefore,

$$h \ge \log(n!) \ge \log(n^{n/2}) = \frac{n}{2}\log n = \Omega(n \log n).$$

That is, a comparison sort has at lowest $O(n \log n)$. We say that merge sort and heapsort are asymptotically optimal- it is possible to improve these algorithms just by a comparison factor.

Counting sort

In search for a better running time, we now look at non-comparison sorts. Counting sort is an efficient sorting algorithm for integers between 0 and k. Its running time is $O(n + k)$- it is linear since it has no comparisons. Moreover, it is a stable algorithm. It requires additional $O(n+k)$ memory. The pseudocode for counting sort is given below:

```
Array<int> countingSort(Array<int> array, int k):
    Array<int> count = []
    // we initialise the array count with zeros
    for (int i = 0; i < k; i++):
        count.add(0)
    Array<int> sorted = []
    // we also initialise the array sorted with zeros
    for (int i = 0; i < array.length; i++):
        sorted.add(0)
    // count the values between 0 and k-1 in the array
    for (int i = 0; j < array.length; j++):
        count[array[i]]++
    // update the count to make it accumulative
    for (int i = 1; i < k; i++):
        count[i] += count[i-1]
    // add to the sorted while updating the count
    for (int i = n-1; i >= 0; i--):
        sorted[count[array[i]]-1] = array[i]
        count[array[i]]--
    return sorted
```

There are 3 parts to the algorithm: we first count the occurrences of each number between 0 and k; we then make the array cumulative; and finally we copy the values into a new array. Clearly, there are no comparisons here.
We illustrate how the algorithm works by sorting the array `[1, 3, 7, 1, 4, 2]` with `k=7`. The first step is counting the number of elements between 0 and 7 in the array:

`array`	1	3	7	1	4	2		
`count`	0	2	1	1	1	0	0	1

We traverse through the `array` and increment the array `count` at the relevant index if we encounter an element with that value. The array `count` states that there are no 0; two 1; one 2; one 3; one 4; no 5; no 6; and one 7. Now, we make the count cumulative:

`count`	0	2	3	4	5	5	5	6

Using these values, we can now correctly index and place the elements within the array `sorted`.

array	1	3	7	1	4	$\underline{2}$		
count	0	2	$\underline{3}$	4	5	5	5	6
sorted	_	_	_	_	_	_		

We traverse the `array` from the end. Since the element is `2`, we find the value of `count[2]=3`- it is `3`. So, we place the number `2` at position `3-1=2` in `sorted` and decremented `count[2]`.

array	1	3	7	1	$\underline{4}$	2		
count	0	2	2	4	$\underline{5}$	5	5	6
sorted	_	_	2	_	_	_		

Now, we place the element `4` in the array `sorted`. Going by the `count`, we find that it should be placed at position `4`. We then decrement the value in the array `count`.

array	1	3	7	$\underline{1}$	4	2		
count	0	$\underline{2}$	2	4	4	5	5	6
sorted	_	_	2	_	4	_		

Then, we place the element `1` in `sorted`. We need to place `1` in index `count[1]-1 =1`.

array	1	3	$\underline{7}$	1	4	2		
count	0	1	2	4	4	5	5	$\underline{6}$
sorted	_	1	2	_	4	_		

Now, we add `7` to `sorted`. We place it in index `counted[7]-1=5`.

array	1	$\underline{3}$	7	1	4	2		
count	0	1	2	$\underline{4}$	4	5	5	5
sorted	_	1	2	_	4	7		

Now, we add to the array `sorted` `3` at index `3`.

array	$\underline{1}$	3	7	1	4	2		
count	0	$\underline{1}$	2	3	4	5	5	5
sorted	_	1	2	3	4	7		

Finally, we add `1` at the first position and the array is sorted. We return the array `sorted`.

```
1  1  2  3  4  7
```

The running time of the loops are either $O(n)$ or $O(k)$, depending on the loop. So, the running time of the entire algorithm is $O(n+k)$. In practice, $k = O(n)$, so $T(n) = O(n)$.

Radix sort

Radix sort sorts elements with respect to their digit. We first set `d` as the highest number of digits present in the array of integers. We then apply a stable sort on the array with respect to digit `i`. The following is the algorithm for it.

```
void radixSort(Array<int> array, int d):
    for (int i = 0; i < d; i++):
        stableSort(array, digit = i+1)
```

If the stable sort is counting sort, then the running time is $O(d(n+k))$. If we're in base 10, $k = 9$, so we can treat it as a constant. We can also take d to also be a constant- then its running time is $O(n)$.
We illustrate how this algorithm works by sorting the array `[1, 23, 424, 536, 120, 7]`. The maximum number of digits here is `d=3`, and because the numbers are in base 10, `k=9`.

```
001   023   424   536   120   007
```

We enter the loop and first sort with respect to the final digit.

```
120   001   023   434   536   007
```

Now, we sort by the second digit.

```
001   007   120   023   434   536
```

Since the sort was stable, the numbers `1` and `7` are in order. Finally, we sort by the first digit and get a sorted array.

```
001   007   023   120   434   536
```

Summary of sorting algorithms

The following table summarises the properties of the sorting algorithms:

Sort	Best Case	Worst Case	Average Case	Stable	In-place
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Merge	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	Yes	No
Quick	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	No	Yes
Heap	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	No	Yes
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	Yes	No
Radix	$O(d(n+k))$	$O(d(n+k))$	$O(d(n+k))$	Yes	No