Algorithms Notes

paraphrased by Tyler Wright

1 Bounding

1.1 Racetrack Principle

For $f, g : \mathbb{N} \to \mathbb{N}$ functions, n, k in $\mathbb{N}$ we have that:

$$\left.\begin{array}{l} f(k) \geq g(k) \\ f'(n) \geq g'(n) \quad (\forall n \geq k) \end{array}\right\} \Rightarrow f(n) \geq g(n) \quad (\forall n \geq k)$$

If a function f is greater than another function g at a value k and has a greater gradient for all values after and including k, f is greater than g for all values after and including k.

1.2 Big O Notation

1.2.1 Definition of the big O notation

For $g : \mathbb{N} \to \mathbb{N}$ a function, $0(g)$ is a set of functions $f : \mathbb{N} \to \mathbb{N}$ such that each for f in $O(g)$:

$$\exists\, c \in \mathbb{R}, n_0 \in \mathbb{N} \text{ such that } \forall n \in N,$$
$$(n \geq n_0) \Rightarrow (0 \leq f(n) \leq cg(n)).$$

1.2.2 The big O notation under multiplication

For $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{N}$ functions where:

- $f_1 \in O(g_1)$
- $f_2 \in O(g_2)$,

we have that:

- $f_1 + f_2$ is in $O(g_1 + g_2)$
- $f_1 \cdot f_2$ is in $O(g_1 \cdot g_2)$.

1.2.3 Closure of the big O notation

For $g : \mathbb{N} \to \mathbb{N}$ a function, $O(g)$ is closed under addition (this follows from the above).

1.2.4 Polynomials and the big O notation

For $p : \mathbb{N} \to \mathbb{N}$ a polynomial of degree k, p is in $O(n^k)$.

1.3 Θ Notation

1.3.1 Definition of the Θ notation

For $g : \mathbb{N} \to \mathbb{N}$ a function, $\Theta(g)$ is a set of functions $f : \mathbb{N} \to \mathbb{N}$ such that each for f in $\Theta(g)$:

$$\exists\, c_0, c_1 \in \mathbb{R}, n_0 \in \mathbb{N} \text{ such that } \forall n \in N,$$
$$(n \geq n_0) \Rightarrow (0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)).$$

f is sandwiched by multiples of g.

1.3.2 Equivalency of the Θ notation

For $f, g : \mathbb{N} \to \mathbb{N}$ functions:

$$f \in \Theta(g) \iff g \in \Theta(f).$$

1.3.3 Θ and O notation

For $f, g : \mathbb{N} \to \mathbb{N}$ functions:

$$f \in \Theta(g) \iff f \in O(g).$$

Which also means $g \in O(f)$ by the above equivalency.

1.3.4 Definition of the Ω notation

For $g : \mathbb{N} \to \mathbb{N}$ a function, $\Omega(g)$ is a set of functions $f : \mathbb{N} \to \mathbb{N}$ such that each for f in $\Omega(g)$:

$$\exists\, c \in \mathbb{R}, n_0 \in \mathbb{N} \text{ such that } \forall n \in N,$$
$$(n \geq n_0) \Rightarrow (0 \leq c g(n) \leq f(n)).$$

1.3.5 Equivalency of the Ω notation

For $f, g : \mathbb{N} \to \mathbb{N}$ functions:

$$f \in \Omega(g) \iff g \in O(f).$$

2 Runtime

2.1 Best-case Runtime

Considering all the inputs for a given algorithm, the best-case runtime is the runtime of the input that the algorithm takes the least amount of time to process.

2.2 Worst-case Runtime

Considering all the inputs for a given algorithm, the worst-case runtime is the runtime of the input that the algorithm takes the most amount of time to process.

2.3 Average Runtime

Considering all the inputs for a given algorithm, the average runtime is the average of the runtimes of all the inputs.

3 Data Structures

3.1 Trees

3.1.1 Definition of a tree

A tree T of size n is defined as $T = (V, E)$ where:

$$V = \{v_1, \dots, v_n\} \text{ is a set of nodes}$$
$$E = \{e_1, \dots, e_{n-1}\} \text{ is a set of edges,}$$

with the properties that for i in $\{1, \dots, n-1\}$, j, k in $\{1, \dots, n\}$ with $j \neq k$ we have $e_i = \{v_j, v_k\}$, and for all i in $\{1, \dots, n\}$, there exists j in $\{1, \dots, n-1\}$ such that v_i is in e_j.

Basically, we have n nodes and $n - 1$ edges where each node has least one edge and the edges can't branch between identical nodes.

3.1.2 Rooted trees

A rooted tree is defined as $T = (v, V, E)$ where $T = (V, E)$ is a tree and v in V is the root of T.

3.1.3 Leaves and internal nodes

A leaf in a tree is a node with exactly one incident edge. If a node isn't a leaf, it's an internal node.

4 Searching

4.1 Linear Search

4.1.1 Information on Linear Search

Input	An array of integers and an integer x both in $[0, n)$ for some n in $\mathbb{N}$
Output	1 if x is in the array, 0 otherwise
Best-case Runtime	$O(1)$
Average Runtime	$O(n)$
Worst-case Runtime	$O(n)$

4.1.2 Process of Linear Search

Iterate through the array comparing the input value with the current array value. If it's equal, return 1. If we reach the end of the array, return 0.

4.2 Binary Search

4.2.1 Information on Binary Search

Input	A sorted array of integers and an integer x both in $[0, n)$ for some n in $\mathbb{N}$
Output	1 if x is in the array, 0 otherwise
Best-case Runtime	$O(1)$
Average Runtime	$O(\log_2(n))$
Worst-case Runtime	$O(\log_2(n))$

4.2.2 Process of Binary Search

Look at the middle value of the array, if equal to the input value then return 1. If the value is greater than our input value, repeat the process with the lesser half of the array. Otherwise, repeat with the greater half of the array.

This works because the array is sorted.

5 Sorting

5.1 Properties of Sorting Algorithms

5.1.1 In place

A sorting algorithm is in place if at any moment at most $O(1)$ array elements are stored outside the array.

5.1.2 Stable

A sorting algorithm is stable if any pair of equal values appear in the same order in the sorted array (this may be important if this value is tied to some overarching data structure).

5.2 Insertion Sort

5.2.1 Information on Insertion Sort

Input	An array of integers in $[0, n)$ for some n in $\mathbb{N}$
Output	An ascending, sorted array
Best-case Runtime	$O(n)$
Average Runtime	$\Theta(n^2)$
Worst-case Runtime	$O(n^2)$
In place	✓
Stable	✓

5.2.2 Process of Insertion Sort

Iterate through the array A, when at position i, place $A[i]$ into the array at some index in $\{0, \ldots, i\}$ such that $A[0, i]$ is sorted.

5.3 Merge Sort

5.3.1 Information on Merge Sort

Input	An array of integers in $[0, n)$ for some n in $\mathbb{N}$
Output	An ascending, sorted array
Best-case Runtime	$O(n \log_2(n))$
Average Runtime	$O(n \log_2(n))$
Worst-case Runtime	$O(n \log_2(n))$
In place	$\times$
Stable	✓

5.3.2 Process of Merge Sort

If the array size is less than 3, reorder the elements and return. Otherwise, split the array into two, perform merge sort on the two halves and combine them.