Algorithms Notes

paraphrased by Tyler Wright

*An important note, these notes are absolutely **NOT** guaranteed to be correct, representative of the course, or rigorous. Any result of this is not the author's fault.*

1 Bounding

1.1 Racetrack Principle

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

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

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:

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

1.3.3 Θ and O notation

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

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

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:

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

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:

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

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

3.1.4 Other definitions

- The **parent** of a node is the closest node on the path from the node to the root (the root has no parent)
- The **children** of a node are all its neighbours barring its parent
- The **height** of a tree is the length of the longest path connecting the root with a leaf
- The **degree** of a node is the number edges incident on the node
- The level of a node is the length of the unique path from the root to it plus one.

3.1.5 k-ary trees

A k-ary tree is a rooted tree where each node has at most k children. A k-ary tree is:

- **Full** if all internal nodes have exactly k children
- **Complete** if all levels except the last are full.
- **Perfect** if all levels are full.

Complete and perfect k-ary trees have heights of $O(\log_k(n))$.

3.2 Priority Queues

3.2.1 Definition of a priority queue

Priority queues are data structures that allow for the creation of a data structure from an array of values and the extraction of said data structure's maximum.

3.2.2 A tree-oriented priority queue

An array could be interpreted by a priority queue as a complete binary tree with the indices from top to bottom, left to right. So, where applicable, for a node index i, the parent of the node has index $\lfloor i/2 \rfloor$ and the children have indices $2i$ and $2i+1$.

3.2.3 Heaps

If we have a tree-oriented priority queue and we add the condition (heap property) that the values of nodes must be greater than their children we get a tree with the maximum at the root. We call this tree a heap.

3.2.4 Producing a heap

Given a binary tree, we can transform it into a heap using a `heapify` function. This function takes a node and its children and ensures the maximum value of these nodes lies in the parent node.

Input	A binary tree and an index
Output	A binary tree such that the value at the given index is greater than or equal to the values of its children
Runtime	$O(\log_2(n))$

Using this, we traverse through the internal nodes, performing `heapify`. If the function makes a change, we perform `heapify` on the child node that was swapped with. This produces a heap.

So, we can see that the heap building function has the following properties:

Input	A binary tree
Output	A heap
Runtime	$O(n \log_2(n))$

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 Lower Bound for Comparison-Based Sorting

When sorting comparatively, we can only sort an array length n at best in $O(n\log(n))$ time.

5.3 Insertion Sort

5.3.1 Information on Insertion Sort

Input	An array of integers length 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	✓

5.3.2 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.

5.4 Merge Sort

5.4.1 Information on Merge Sort

Input	An array of integers length 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	✓

5.4.2 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.

5.5 Heap Sort

5.5.1 Information on Heap Sort

Input	An array of integers length n in $\mathbb{N}$
Output	An ascending, sorted array
Runtime	$O(n\log_2(n))$
In place	✓
Stable	$\times$

5.5.2 Process of Heap Sort

Produce a heap from the array and extract the maximum from it until it is empty (ensuring that it remains a heap between extractions).

5.6 Quick Sort

5.6.1 Information on Quick Sort

Input	An array of integers length 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	$O(n^2)$
In place	✓
Stable	$\times$

(Based on a random pivot selection procedure)

5.6.2 Process of Quick Sort

Choose a pivot, move all values greater than the pivot into indices greater than the pivot's and move all values less than the pivot into indices less than the pivot.

This can be done by storing the smallest index i such that it's value has not been swapped then iterating through the array (excluding the pivot) and checking if the value is less or equal to the pivot. If so, move it to position i and increment i. Once at the end of the array, the pivot can be moved into position i.

Then, we perform quick sort on all values less than the pivot and on all values greater than the pivot.

5.7 Counting Sort

5.7.1 Information on Counting Sort

Input	An array of non-negative integers length n in $\mathbb{N}$
Output	An ascending, sorted array
Runtime	$O(n))$
In place	$\times$
Stable	✓

5.7.2 Process of Counting Sort

Create an array A with its length equal to the maximum of the input array. At each index, set the value equal to the amount of values less than the index in the input array. Then, we can fill our input array with the correct amount of each integer in increasing order.

5.8 Radix Sort

5.8.1 Information on Radix Sort

Input	An array of non-negative integers length n in $\mathbb{N}$
Output	An ascending, sorted array
Runtime	$O(n))$
In place	✓
Stable	✓

5.8.2 Process of Radix Sort

Using a stable sorting algorithm, sort the least to the most significant digits.

6 Recurrences

6.1 Motivation

For an increasing function $T : \mathbb{N} \to \mathbb{N}$ that produces the worst-case runtime for an algorithm for an input n. If we have that T is defined in terms of itself:

$$\begin{aligned} T(1) &= c_1 \\ T(n) &= c_2 T(f(n)) + c_3 g(n), \end{aligned}$$

where c_1, c_2, c_3 are in $\mathbb{N}$ and $f, g : \mathbb{N} \to \mathbb{N}$. We would like to bound this function.

6.2 Substitution

We can guess what bounds the function. Consider the following:

$$\begin{aligned} T(1) &= c_1 \\ T(n) &= 2T(\lfloor n/2 \rfloor) + c_2 n. \end{aligned}$$

Suppose we guess that $T(n) \leq C[n \log_2(n)]$, we will use induction to prove our claim (as we are looking for an asymptotic comparison we can take any n in $\mathbb{N}$ as our base case):

Base case ($n = 2$):

$$\begin{aligned} T(2) = 2T(1) + 2c_2 &= 2(c_1 + c_2) \\ C[2 \log_2(2)] &= 2C \end{aligned}$$

So, for $C \geq (c_1 + c_2)$, $T(2) \leq C[2 \log_2(2)]$.

Inductive step, suppose the claim holds for all $n < k$:

$$\begin{aligned} T(k) &= 2T(\lfloor k/2 \rfloor) + c_2(k) \\ &\leq 2(C[k/2] \log_2(k/2)) + c_2(k) \\ &= C[k \log_2(k/2)] + c_2(k) \\ &= Ck[\log_2(k) - 1] + c_2(k) \\ &= Ck \log_2(k) + k(c_2 - C) \end{aligned}$$

So, for $C \geq c_2$, we have:

$$T(k) \leq C[k \log_2(k)].$$

So, for $C \geq (c_1 + c_2)$, the claim holds for all $n \geq 2$ by induction.