

# 1 Sort

## 1.1 Insertion Sort

Insertion sort works by scanning from right to left and when an item is out of place, “bubbling” that item to the left until it is in its rightful place.

Basically what happens is an item (the leftmost of the unsorted portion) is selected, and then scanned through the already sorted portion of the list until its rightful place is found.

- stable? Yes.
- time complexity: Worst case is  $\Theta n^2$ .
- space complexity:  $\Theta 1$  as everything is done in place.
- best-case input: A nearly-sorted array, as almost all needed work is done.
- worst-case input: An array in reverse order.  $\sum_{k=1}^n k = 1 + 2 + 3 + \dots + n$ , because the item we are looking at *belongs* at the new beginning of the list, but we have to look at *everything* on the way over.

## 1.2 Selection Sort

Selection sort scans through the entire *unsorted* part of the list looking for the next smallest item. Therefore the running-time of this algorithm is given by the series  $\sum_{k=1}^n k$  where  $k$  is the number of compares.

- Not stable
- Upper-bounded by  $O(n^2)$ , actually, always runs in  $\Theta(n^2)$
- The only benefit of selection sort over other sorts: it minimizes the number of array swaps

## 1.3 HeapSort

Heapifies data (classically bottom-up), then builds the sorted list from the last item to the first.

Runs in absolute worst case  $O(n \cdot \lg(n))$ , so good choice if time constraints are tight and Quicksort with shuffle can't be used.

- Not stable
- $\Theta(n \cdot \lg(n))$

## 1.4 MergeSort

The fkn shit.

- Stable
- $\Theta(n \cdot \lg(n))$

- $\Theta(n)$  extra space for arrays
- $\Theta(\lg(n))$  extra space for your mergeable datastructure of choice

## 1.5 QuickSort

Picks an item as a pivot (this was classically the leftmostmost item, but this causes worst case performance on already sorted input). Partitions  $x$  around pivot  $n$  like dis: 

$x < n$	$x = n$	$n < x$
---------	---------	---------

Naive implementations run in  $n^2$  on already sorted arrays (which is apparently common in real world situations), so caution must be exercised with this sort.

There are many situations where we might use Quicksort: When we want faster overall runtime, better memory usage, when we don't care about stability, or when we have many duplicates.

- Not stable<sup>1</sup>
- Upper-bounded by  $O(n^2)$  on already sorted (and not shuffled arrays, but is pretty dependably  $\Theta(n \cdot \log(n))$  with carefullness

## 1.6 MSD & LSD

Items are sorted by *Least Significant Digit* or *Most Significant Digit*. Arrays are partitioned into subarrays based on digit place being looked at (i.e. tens, hundreds, thousands) and placed in order based on that place. Stability ensures that they are in place.

---

<sup>1</sup>sometimes it is, but sketchy things must be done for this to be the case.

## 2 Search

Adjacency list: list of nodes that can be *immediately* reached from the current node. A.K.A. connections or outgoing edges.

Adjacency matrix: table or matrix of *nodes* · *nodes* where each intersection is either *true* or *false*

	a	b	c
a	0	1	1
b	1	0	0
c	1	0	0

### 2.1 Dijkstra's

time complexity:  $O(|E|\log|V|)$  according to Hug, and  $O(|E| + |V|\log|V|)$  according to Wikipedia. Hug's justification is as follow::

- we do  $V$  insertions, which each cost  $O(\log|V|)$  time.
- we do  $V$  min-deletions, costing  $O(\log|V|)$  time each.
- we do  $E$  “priority decreases”, costing  $O(\log|V|)$  time each.
- so our total runtime is  $O(V \cdot \log|V| + V \cdot \log|V| + E \cdot \log|V|) \Rightarrow O(2V + E \cdot \log|V|)$
- which becomes  $O(E \cdot \log|V|)$

Algorithm for finding shortest path in a graph. *Greedy* algorithm in that it enqueues the lowest cost path th

### 2.2 A\*

Dijkstra's but with heuristics added to edge-weights.

### 2.3 Kruskal's

time complexity:  $V$  is at most  $E^2$ , so it follows that  $O(\log V^2) \Rightarrow O(2\log V) \Rightarrow O(\log V)$ .

Algorithm for finding the MST of a graph.

Pseudocode:

```
insert all edges into MinPQ
while (not all nodes are connected):
    dequeue edge
    if (edge doesn't create a cycle):
        add edge
```

One issue with this algorithm is that it runs the risk of adding edges that are disconnected to the larger whole of the graph if not careful about enqueueing edges. So for Kruskal's to be the ideal algorithm we must be sure that the graph is connected.

## 2.4 Primm's

time complexity:

- with binary heap and adjacency list:  $O(|E| \log |V|)$
- with adjacency matrix  $O(|V|^2)$

Another algorithm for finding the MST of a graph. Algorithm of choice if not sure that graph is connected. As Prim's is run from an arbitrary vertex, it will only find vertices that are connected to starting vertex.

Pseudocode:

```
start at a vertex
while (MST isn't found):
    visit closest unvisited vertex to any visited vertex
    add edge to MST
```

So what happens is essentially that we pick a place to start, then visit the node closest to the start if we haven't already visited it. We will then continue to do this until we've found the MST for the graph.

## 3 Addenda

### 3.1 Asymptotics

$$\text{Taking : } \frac{r(\alpha)}{r(\beta)} = \left(\frac{\alpha}{\beta}\right)^b \text{ as } \gamma = \delta \quad (1)$$

$$\text{we can simplify to : } \log_{\gamma} \delta = b \text{ and solve for } b \quad (2)$$