CS 251: Data Structures and Algorithms

Sum	*Closed Form*
$\sum_{k=0}^{n} ar^k \ (r \neq 0)$	$\frac{ar^{n+1}-a}{r-1}, r \neq 1$
$\sum_{k=1}^{n} k$	$\frac{n(n+1)}{2}$
$\sum_{k=1}^{n} k^2$	$\frac{n(n+1)(2n+1)}{6}$
$\sum_{k=1}^{n} k^3$	$\frac{n^2(n+1)^2}{4}$
$\sum_{k=0}^{\infty} x^k, \lvert x \rvert < 1$	$\frac{1}{1-x}$
$\sum_{k=1}^{\infty} kx^{k-1}, \lvert x \rvert < 1$	$\frac{1}{(1-x)^2}$

Figure 1: Formulas

Big-O

Big-$\mathcal{O}$: $f(n) \in O\big(g(n)\big) \iff$
$\exists\, c > 0,\ \exists\, n_0 > 0\ :\ \forall n \geq n_0,\ 0 \leq f(n) \leq c\, g(n)$.

Big-Ω: $f(n) \in \Omega(g(n))$,
$0 \leq c \cdot g(n) \leq f(n)$

Big-Θ: $f(n) \in \Theta(g(n))$,
$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

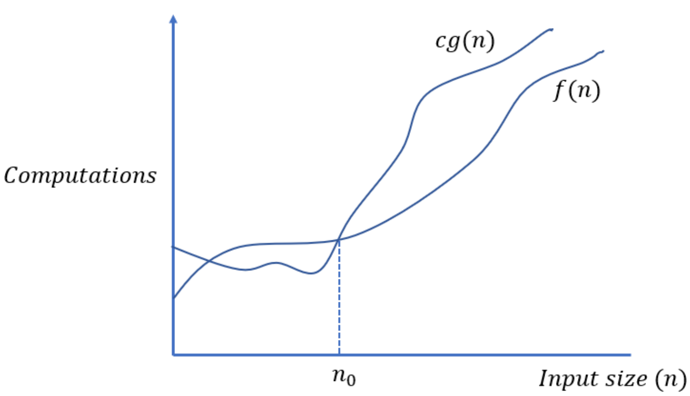


Figure 2: Big-O curve

Arrays

Access: $\Theta(1)$, Insert: $\Theta(1)$ (with tail) $\Theta(n)$ (no tail).

Resize and space of resize: $\Theta(n)$

Binary trees

Max nodes in BT (total in complete BT) $= 2^{h+1} - 1$. Leave nodes $= 2^h$

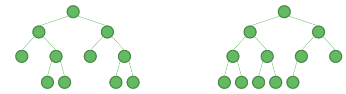


Figure 3: Full BT and Complete BT

→ Full BT: Each node has none or two children.

→ Complete BT: All levels full until last (left to right).

Balanced BT: At every node the height of the left and right subtree differs by at most 1:
$2^h \leq n \rightarrow h \leq \log_2 n \therefore h \in \mathcal{O}(\log(n))$, where n is the number of nodes.

Balanced BT Insertions, Searches & Deletions: $\mathcal{O}(\log(n))$

Binary Heap

Complete binary tree $\therefore h \in \Theta(\log(n))$.

Insertions & Deletions: $\mathcal{O}(\log(n))$ through single path.

→ Left: $2i + 1$, Right: $2i + 2$.

→ Parent: $\lfloor \frac{i-1}{2} \rfloor$.

Max heap (max on top). Min heap (min on top).

Insertions enter in complete BT order and then adjust (`sift up`).

Single insert is $\mathcal{O}\log(n) \therefore n$ inserts $\in \mathcal{O}(n\log(n))$

Heapify

`Sift down` on all nodes in an array from $[0, \lfloor \frac{n}{2} \rfloor - 1]$

Heap build: $\mathcal{O}(n)$

Heap sort

Call `Heapify` ($\mathcal{O}(n)$) and then swap the root with the last item and fix the heap ("lock" the last element after swap).

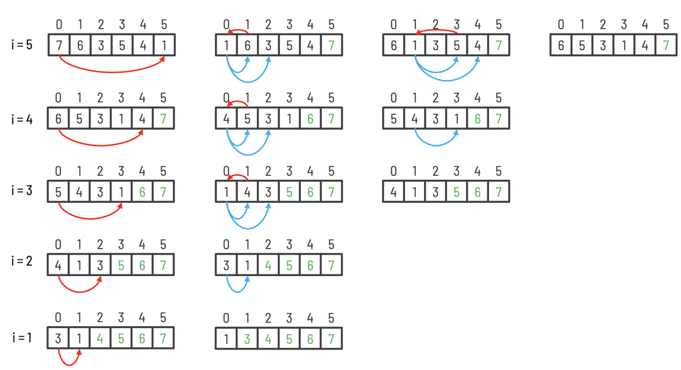


Figure 4: Heap sort example

Quick sort

Not stable and pivot selection is arbitrary.

Best case ($\mathcal{O}(n\log(n))$)→ even partitions.

Worst case ($\mathcal{O}(n^2)$)→ one partition has $n-1$ elements.

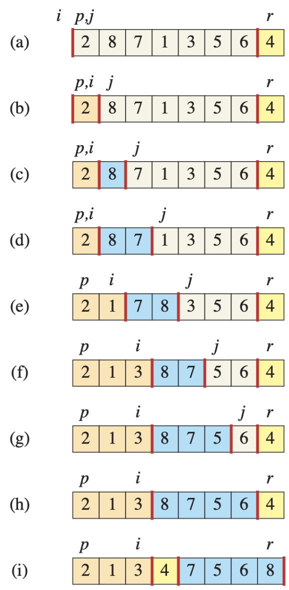


Figure 5: Quick sort; i (return) is moved when comparison is lower and swaps at j (comparison index)

Counting sort decision tree

Leaves are $n!$ permutations of the array (all possibilities).

Internal nodes are the conditional checks.

As full BT will be 2^h leaves $\therefore 2^h \geq n!$

$h \geq \log_2 n! \Rightarrow h \geq n\log_2 n \therefore h \in \mathcal{O}(n\log(n))$

Counting sort

Array → Freq. array → Cm. frq. array → Resulting array filled from cm. frq. array backwards.

Stable, bad with large max value of an array.

Runtime & Space: $\mathcal{O}(n+k)$, where n: array size; k: max value of array (freq. arr.).

Bucket sort

Worst case $\mathcal{O}(n^2)$ → unsorted un-uniform distribution.

Best case $\mathcal{O}(n)$ → one element per bucket or all sorted in one bucket.

Avg. case $\mathcal{O}(n+\frac{n^2}{k}+k)$

If there is one key per bucket (i.e. there exists a bucket for any key) then, $\mathcal{O}(n+k)$, where n: elements in array; k: buckets.

Extra work comes from sorting within each bucket.

Better for uniform distributions: fits things equally.

Radix sort

Sorting through LSD to MSD using a stable sorting algorithm (e.g. counting sort).

Runtime: $\mathcal{O}(d(n+k))$, where d: digits; $n+k$: counting sort.

Space: $\mathcal{O}(n+k)$

Using bucket sort with binary digits, create 2 buckets for 2 keys (1 and 0).

If there are constant lengths of digits then $\mathcal{O}(n)$ is possible.

E.g. $\mathcal{O}(d(n+2)) \in \mathcal{O}(n)$ if there are constant lenths of binary numbers.

Polynomial rolling hash

Where S: string of length m; a: multiplier for polynomial.

$$H(S,a) = s_0a^{m-1} + s_1a^{m-2} + ... + s_{m-1}a^0 = \sum_{i=0}^{m-1} s_ia^{m-i-1}$$

If using a fixed sized window m then do rolling updates on S: $H_1 = (H_0 - s_{old} \cdot a^{m-old}) \times a + s_{new}$

Hashing functions

Division method: $h(k) = k \; mod \; m$ (avoid 2^P and $10^P = m$), where m: size of hash table.

Multiplication method: $h(k) = \lfloor m(kA \; mod \; m) \rfloor$ ($A \approx \frac{\sqrt{5}-1}{2}$; m is not critical)

Aim to have keys be distributed uniformly (no clusters).

Collision strategies

Under the Simple Uniform Hashing Assumption

→ Average number of keys per bucket (Load Factor): $\alpha = \frac{n}{m}$
→ $n \in \mathcal{O}(m) \therefore \alpha \in \mathcal{O}(1)$, n: keys, m: buckets.

Chaining

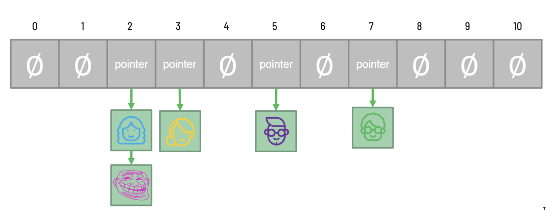


Figure 6: Chaining collision management

Worst case $\mathcal{O}(n)$: traversing all items in the table.

Runtime $\Theta(1 + LengthOfChain) = \Theta(1+\frac{n}{m})$.

Open addressing

Probing sequence for $h(k,i) \Rightarrow h(k,0), h(k,1), ..., h(k,m-1)$

$\rightarrow$ Linear: $h(k,i) = (h(k)+i)\ mod\ m$.

$\rightarrow$ Quadratic: $h(k,i) = (h(k)+c_1 i + c_2 i^2)\ mod\ m$, where c: coefficients chosen.

$\rightarrow$ Double hashing: $h(k,i) = (h(k)+ih_1(k))\ mod\ m$.

Element deletion will mess up the hash (offset from probing impossible to reach).

When load factor is too high there is a cluster overhead.

Insertions probe: $\frac{1}{1-\alpha}$

Successful search probes: $\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$

Unsuccessful search probes: $\frac{1}{1-\alpha}$

Rehashing

Chaining:

$\rightarrow$ Goal: α constant.

$\rightarrow$ Double m when $\alpha \geq 8$.

$\rightarrow$ Halve m when $\alpha \leq 2$.

Open addressing:

$\rightarrow$ Goal: $\alpha \leq \frac{1}{2}$.

$\rightarrow$ Double m when $\alpha \geq \frac{1}{2}$.

$\rightarrow$ Halve m when $\alpha \leq \frac{1}{8}$.

B-trees

The order (num. of children) is m, determined from $m-1$ keys in the parent.

Height: $h \in \mathcal{O}(\log_m n)$.

B-trees are always "complete"—all levels full.

Insertions do not increase heigh unless m order limit is reached in the root.

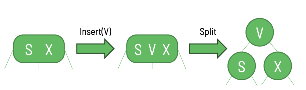


Figure 7: 2-3 B-tree insertion

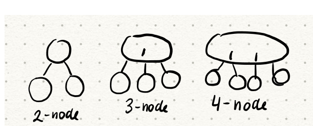


Figure 8: B-tree types

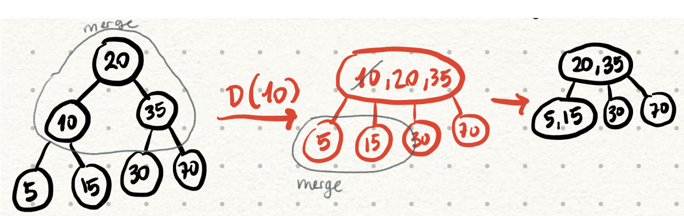


Figure 9: B-tree deletion with high redux

Red-Black trees

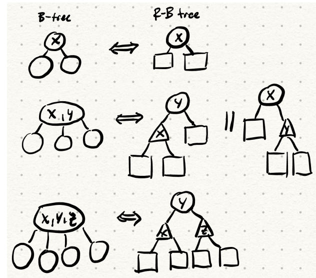


Figure 10: B-tree to R-B tree conversion

Root and null links are black.

Red parents have black children.

All paths, excluding null links, have the same number of black nodes $h \in \mathcal{O}(\log(n))$ (black height).

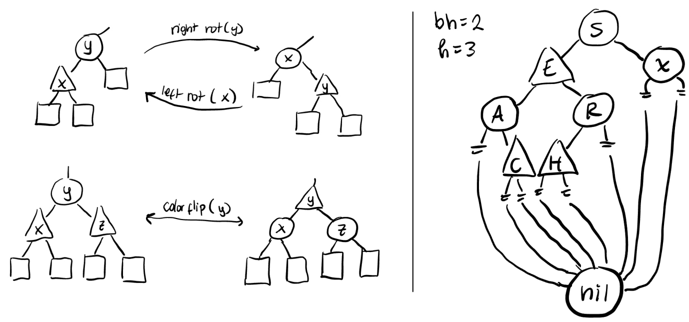


Figure 11: R-B Tree operations

Black height increase in an RB-tree, correlates to a height increase in a 4-node b-tree (2-3-4 tree).

Undirected graphs

Handshake theorem: $\sum_{v \in V} deg(v) = 2 \cdot |E|$

Edge count: $|E| \leq \frac{|V|(|V|-1)}{2}$, considering $|V|-1$ edges on V edges.

Complete graph has vertexes with degree $|V|-1 \therefore |E| = \frac{|V|(|V|-1))}{2}$

Paths and cycles

Simple path/cycle have no repeated edges/vertices.

Euler path visits all edges once.

Euler cycle is a Euler path that starts/ends on the same v.

Hamiltonian path visits all vertices once.

Hamiltonian cycle is a H. path that starts/ends on teh same v.

Length is the number of edges.

Edge representations

Edge list

List: $(u,v),(u,w),(v,x),...$

Adding edge: $\mathcal{O}(1)$

Space, Adj. vertex check, & Adj. iteration: $\mathcal{O}(|E|)$

Adjacency matrix

Cells are 1 or 0, and A^k gives the num. of k-length paths between vertices.

Edges go $i \to j$

	U	V	W	X	Y	Z
U	0	1	0	0	0	0
V	0	0	1	1	0	0
W	1	0	0	0	1	0
X	0	0	1	0	0	1
Y	0	0	0	1	0	0
Z	0	0	0	1	0	1

Figure 12: Adjacency matrix

Space: $\mathcal{O}(|V|^2)$

Adj. iteration: $\mathcal{O}(|V|)$

Adding edge & adj. vertex check: $\mathcal{O}(1)$

Adjacency list

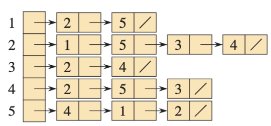


Figure 13: Adjacency list

Space: $\mathcal{O}(|V| + |E|)$

Adding edge: $\mathcal{O}(1)$

Adj. vertex check & Adj. iteration: $\mathcal{O}(deg(V))$

Types of graphs

Connected graphs have paths for all pairs of vertices.

Spanning subgraph has all V in G

Trees are undirected, connected, and acyclic graphs.

Forests are collections of trees (acyclic).

Spanning tree are spanning subgraphs of G that are trees.

Graph characteristics

Sparse graphs $\to |E| << |V|^2$ (use lists)

Dense graphs $\to |E| \approx |V|^2$ (use matrices)

Strong connectivity is that every v is connected to every other $v \in V$ (with reflexive, symmetric, and transitive properties).

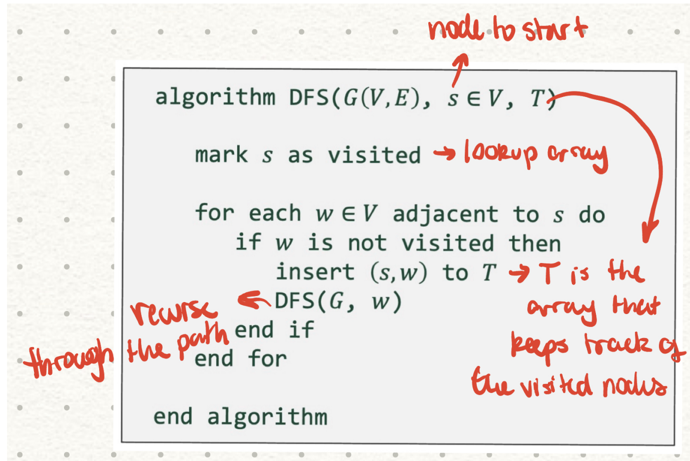
Test strong conn. by running DFS on G and G' (inverted pointing), if all nodes are visited then there is strong conn.

Depth-first search (DFS)

Traverses $|E|$ twice $(2 \cdot |E|)$ by backtracking.

If non-visited node remains, then G is not connected.

Runtime: $\mathcal{O}(|V| + |E|)$ (adj. list)

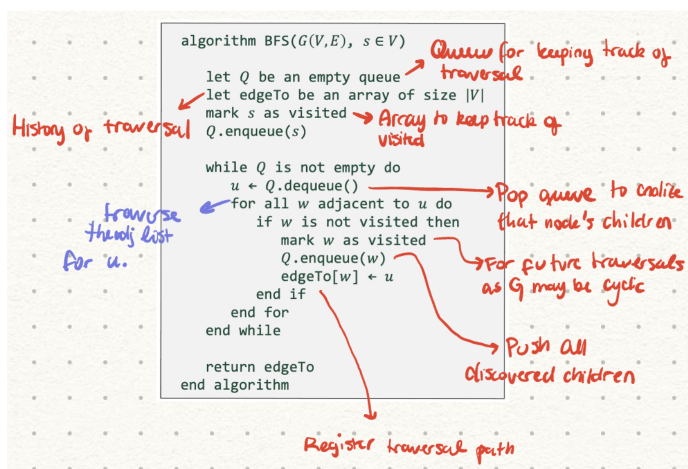


Breath-first search (BFS)

Traverses all V and E.

Runtime: $\mathcal{O}(|V| + |E|)$

Terminates when the queue is empty or all vertices are discovered.



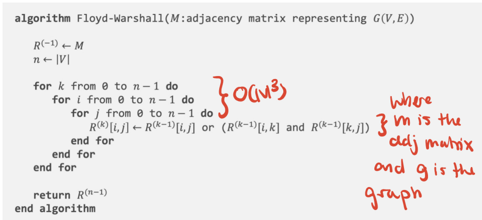
Transitive closure

G^* shows the paths where $v \neq u$ has a directed path (u, v)

Warshall uses 1s in diagonal if self loops or cycles.

Floyd-Warshall algorithm builds G^* matrix.

R^k to denote progression from $k = -1$.



`m[i][j] = g[i][j] || (g[i][k] && g[k][k]);`

`final = init || (indirect path i->k->j);`

Space: $\mathcal{O}(|V|^2)$ (matrix)

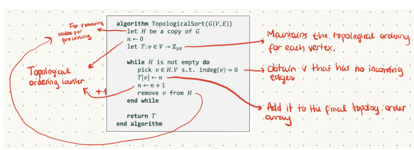
Runtime: $\mathcal{O}(|V|^3)$ (i, j, k)

Topological ordering

Directed acyclic graph (DAG): digraph with no cycles.

Numbering $v_1, ..., v_n$ such that every (v_i, v_j) is $i < j$.

By inspection, remove nodes with `indeg(0)` in order.



Runtime: $\mathcal{O}(V + E)$ using DFS, $\mathcal{O}(1)$ for insertion lists.

Dijkstra's algorithm

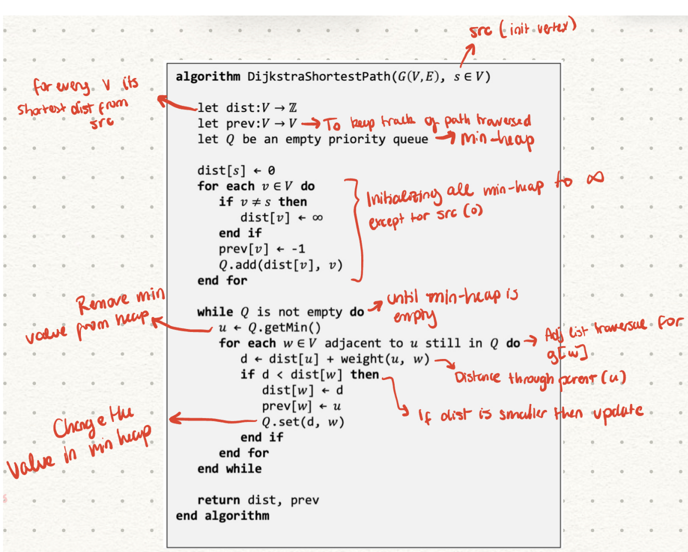
Greedy shortest path, that uses min-heap for inspected nodes and their weights.

→ Removes when discovered, updated when finding path with lesser cost.

Min-heap removal: $\mathcal{O}(|E| \log(|V|))$

Update: $\mathcal{O}(log(|V|))$

Runtime: $\mathcal{O}((|V| + |E|) \log(|V|))$

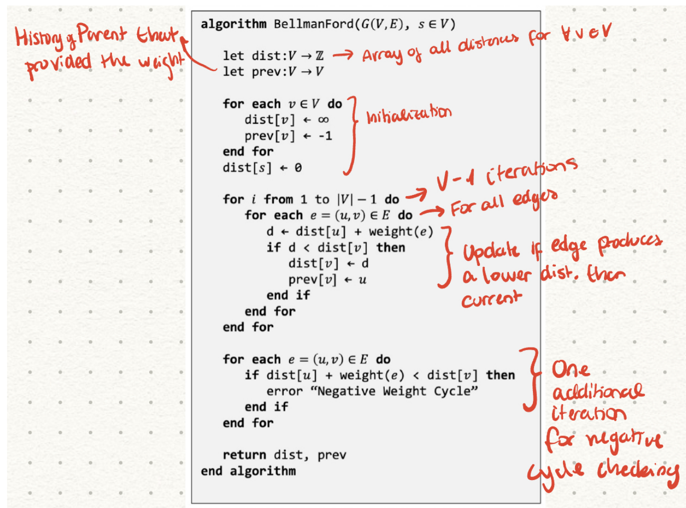


Bellman-Ford algorithm

Shortest path with negative weights, that iterates $V - 1$ times $+1$ for negative cycle checking.

Checks every node, and updates neighbors weights until no nodes are updated.

Runtime: $\mathcal{O}(|V| \cdot |E|)$



MST & Kruskal's MST algorithm

Minimum spanning trees (MSTs) are spanning trees with minimal weights.

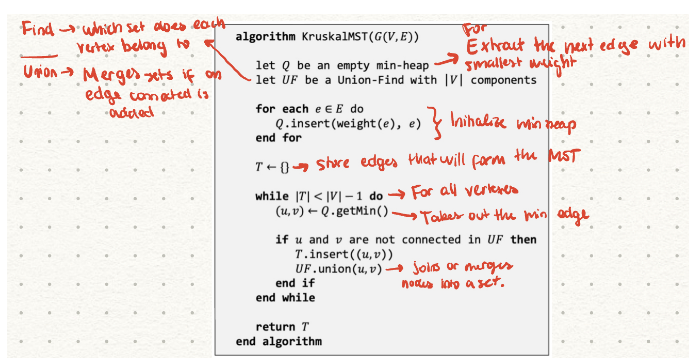
→ Splitting vertices into two groups, the min weight edge is part of the MST of G.

→ On cycles the biggest weight is never in the MST of G.

Kruskal's algorithm finds the MST by using Union-Find on nodes at increasing weights.

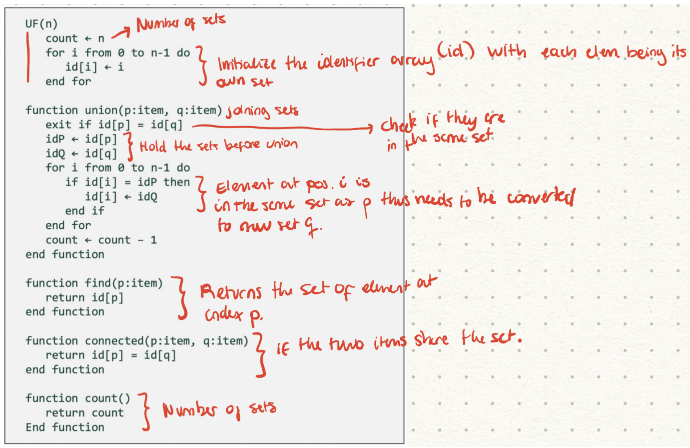
Uses min-heap to obtain the weights in increasing order.

Runtime: $\mathcal{O}(|E| \log(|V|))$



Union-find

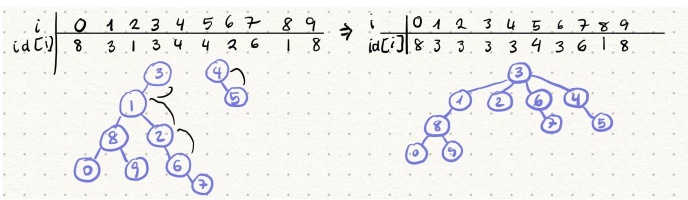
Quick-Find merges by grouping nodes under a "parent" node label, and finds by an array lookup.



```
UF(n)
   count ← n
   for i from 0 to n-1 do
      id[i] ← i
   end for

function union(p:item, q:item)
   exit if id[p] = id[q]
   idP ← id[p]
   idQ ← id[q]
   for i from 0 to n-1 do
      if id[i] = idP then
         id[i] ← idQ
      end if
   end for
   count ← count - 1
end function

function find(p:item)
   return id[p]
end function

function connected(p:item, q:item)
   return id[p] = id[q]
end function

function count()
   return count
End function
```


Quick-Union, using path compression, joins nodes into their corresponding tree.

`Union(6, 5)` will look up to the root of 6 in `Find(6)`: $6 \to 2 \to 1 \to 3$ and will attach all intermediaries directly to the root (3). Does the same with `Find(5)` $5 \to 4$ and joins the smaller tree as a child of the bigger tree (4 child of 3).




```
UF(n):
   count ← n
   for i from 0 to n-1 do
      id[i] ← i
   end for

function union(p:item, q:item)
   idP ← find(p)
   idQ ← find(q)
   exit if idP = idQ
   id[idP] ← idQ
   count ← count - 1
end function

function find(p:item)
   while p ≠ id[p] do
      p ← id[p]
   end while
   return p
end function

function connected(p:item, q:item)
   return find(p) = find(q)
end function

function count()
   return count
end function
```

```
function find(p:item)
   if p = id[p] then
      return p
   end if
   id[p] ← find(id[p])
   return id[p]
end function
```

Quick-Find runtime: $\mathcal{O}(tree\_depth)$ or worst degen $\mathcal{O}(n)$

Quick-Union runtime: $\mathcal{O}(1)$ and path comp: $\mathcal{O}(\alpha(n))$

Brute-force string matching

Takes a substring and compares it contiguously.

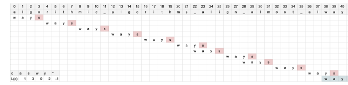
Runtime: $\mathcal{O}(m) \cdot \mathcal{O}(n) \in \mathcal{O}(nm)$

Boyer-Moore algorithm

T (text) and backwards P (pattern) comparison.

Creates a numeric rep. for the occurrence of each char, and then compares and moves at offset: $m - min(j, 1 + L[c])$.

Number of comparisons are the total times the last char is compared + offsets (example has 15).

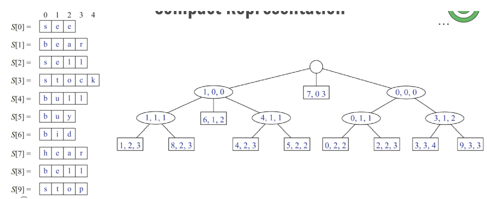


Tries

Prefix/suffix tries contain all possible combinations.

For a single string: $\mathcal{O}(|w|)$

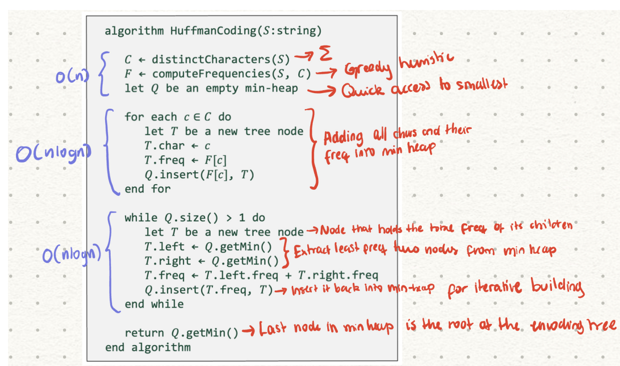
For PATRICIA tries, a node is redundant and can be grouped if it's not the root and has one child (include \$).



Huffman

Greedy, min-heap, that encodes high freq. with short codewords.

Runtime: $\mathcal{O}(n\log(n))$.



```
algorithm HuffmanCoding(S:string)

   C ← distinctCharacters(S)
   F ← computeFrequencies(S, C)
   let Q be an empty min-heap

   for each c ∈ C do
      let T be a new tree node
      T.char ← c
      T.freq ← F[c]
      Q.insert(F[c], T)
   end for

   while Q.size() > 1 do
      let T be a new tree node
      T.left ← Q.getMin()
      T.right ← Q.getMin()
      T.freq ← T.left.freq + T.right.freq
      Q.insert(T.freq, T)
   end while

   return Q.getMin()
end algorithm
```


K-d trees & quadtrees

K-dimensions where non-leaf nodes split into half-spaces. Each level alternates dimensions (e.g. 0:x, 1:y, 2:x).

Quadtrees split into 4 spaces (NW, NE, SW, SE).

Insertions use level discriminant to locate parent child nil (e.g. `Insert(70, 50)` uses 70 for x levels, and 50 for y levels).

Deletions swap node with minimum value of discriminant at the level in the right subtree (if the min node has children then the same process is done for that node). Leaf is deleted.

The expected depth of the quadtree at each level: $4^D \approx n \therefore D \approx \frac{1}{2}\log_2(n)$, where n: num. points.

Range query result is determined by square query s intersection with square region: $\frac{UpperBound}{2^D} = \frac{UpperBound}{n^{1/2}}$.

$\frac{s}{1/n^{1/2}} = s \cdot n^{1/2} \therefore s^2 \cdot n$ are the points intersecting (leaf nodes).