	stable	in place	runtime	compares (n is descending)	swaps (n is descending)
insertion	✓	✓	$O(n^2)$	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$
selection		✓	$O(n^2)$	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\lfloor\frac{1}{2}n^2\rfloor$
bubble	✓	✓	$O(n^2)$	$n^2 - n$	$\frac{1}{2}n^2 - \frac{1}{2}n$
quick		✓	$O(n^2)$ $O(n \log n)$		
merge	✓	$O(n)$	$O(n \log n)$		
radix	✓		$O(n)$		
counting	✓		$O(n)$		
bucket	✓		$O(n^2)/O(n)$		

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

Hashing

Hash load factor (expected list length/chance for collision): $\frac{n}{m}$

- Open addressing: double table size when $\frac{n}{m} = \frac{1}{2}$ and halve the size when $\frac{n}{m} = \frac{1}{8}$
- Chaining: double table size when $\frac{n}{m} = 8$ and halve the size when $\frac{n}{m} = 2$

Data Compression

Impossible to guarantee listless compression for random files (pigeonhole principle)

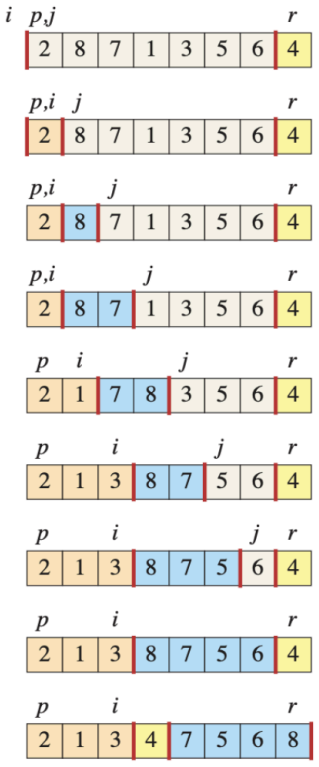
- 2^n binary strings of length n (pigeons)
- 2^{n-1} binary strings of length less than n (holes)

Huffman Coding:
Encoding O(n log n) where n unique characters
Decoding O(n log n) where n is longest bit length
Fixed: same length code for all characters
Variable: most frequent characters have shorter code
Best-case: uniform frequencies
Worst-case: characters in a chain
Optimal coding tree is not unique
Prefix code: no codeword is the prefix of another

- Queue characters by frequency (as trees)
- Pop 2 characters and join them with a parent node whose value is the sum of their frequencies
- Requeue the tree in the min-priority queue

Identities

log(n!) = Θ(n log n)
Median (unsorted): O(log n)
Compare sort: Ω(n log n)



- Repeat until one item left in queue

Heaps

Insert (swim up) and delete (sink down): O(log n)
Heap sort (heapify then sort down)
pop the min/max, swap the lowest element, and fix the heap
Bottom-up O(n):

- start at index: $\lfloor\frac{n}{2}\rfloor - 1$

Insertion/Top-down O(n log n):

- build heap using the insert and swap method

Solving Binary Tree Equations

Preorder (solve equation right to left): N, L, R
Inorder: L, N, R
Postorder (solve left to right): L, R, N

Big-O & Big-Ω

$f(n)=O(g(n))$ if $0 \leq f(n) \leq cg(n)$
$f(n)=\Omega(g(n))$ if $0 \leq cg(n) \leq f(n)$

Arrays

Double array size when full, half the size when quarter full
All operations take O(n) except for insert

- When the array is full insert necessitates a resize which takes O(n)
- Remove can also take O(n) if you downsize, but it is not always necessary

Stacks and queues take O(n) for all operations when implemented as linked lists. O(n) for all operations except push (and pop if you choose to downsize)

Tries

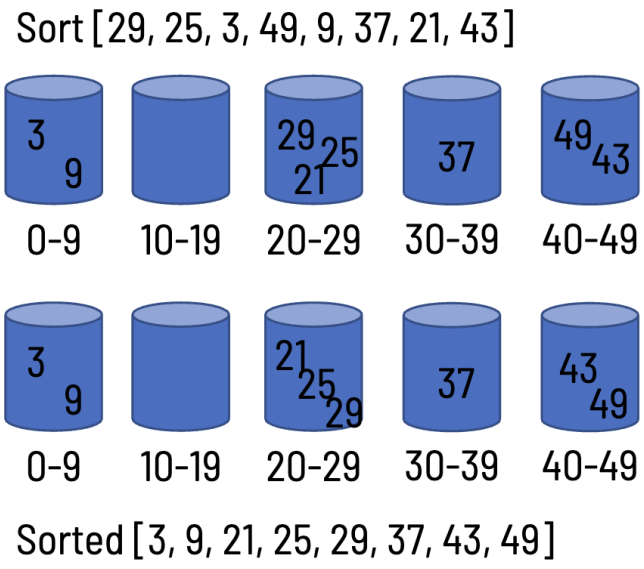
O(W) to add a word, W is the word length
O(L) to construct a trie, L is the average word length
Word matching: leaf value = start index of word in text
Suffix trie: used to find substrings & suffixes
PATRICIA: compressed trie, can be represented as indices of the strings (string index, start, end) inclusive

String Pattern Matching

Brute-force O(PT): compare first letter of pattern and text, move the pattern right by one if mismatch
KMP Algorithm O(P+T):

- Compute failure function O(P): length of the longest prefix that is also a suffix of the pattern string
- On mismatch find corresponding value for the letter of the pattern in failure function
- Go to that letter in the string (not index) and re-compare.
- If there is a mismatch with the first letter of the pattern, shift right by one

```
algorithm BucketSort(A, k)

   let B be an array of k empty buckets
   let M be the Max key in A

   for (i = 0, i < n, 1)
      j = floor(k * A[i] / M)
      insert A[i] to bucket B[j]
   end for

   for (i = 0, i < k - 1, 1)
      sort the bucket B[i]
   end for

   Concatenate the lists from each bucket
   into a single array and return it
```



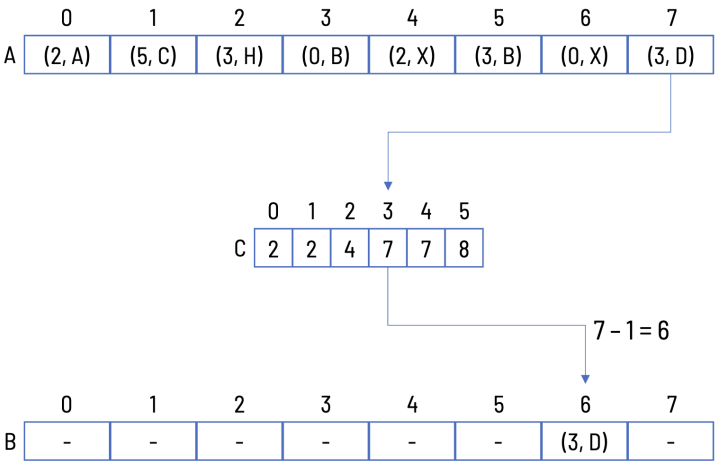
Step 1: Array C keeps the number of occurrence for each element in A.

Step 2: Count the occurrences of each item in A. Use A[i] as the indices of C.

Step 3: Accumulate the count values in C from left to right.

Step 4: Use values in C to determine the final index for each element in A.

```
algorithm CountingSort(A, k)

   let C be an array of length k + 1
   fill C with 0s

   let n be the length of array A

   for (i = 0, i < n, 1)
      C[A[i]] = C[A[i]] + 1
   end for

   for (i = 1, i <= k, 1)
      C[i] = C[i] + C[i - 1]
   end for

   let B be an array of size n

   for (i = n - 1, i >= 0, -1)
      B[C[A[i]] - 1] = A[i]
      C[A[i]] = C[A[i]] - 1
   end for

   return B
```



	Runtime	Height	Description
Quick-find	Find: O(1) Union: O(v)		Index is the vertex and the value is the group. Iterate through the vertices and update all members of the same group.
Quick-union	Find & union: O(v)	O(v)	Index is the vertex and the value is the parent index. The root of the first set points to the second vertex.
Weighted quick-union	Find & union: O(log v)	O(log v)	Smaller tree's root is added to the larger tree's root
Path Compression	Worst: O(log v) Average: O(α(v))		All vertices traversed in find() on both trees are adjusted to point directly to the root of the larger tree

Proofs

Induction:

1. Let P(n) = proposition
2. Base Step: show P(n) holds for initial n value
3. Inductive step: Assume P(k) is true and show P(k+1) is true
4. Inductive hypothesis: P(k) = P(n)
5. Algebra
6. Conclusion

Direct: $q \Rightarrow p$

Contraposition: $\neg q \Rightarrow \neg p$

Contradiction: $p \wedge \neg q$

Counter example: case or proof

Graphs

- Simple: no loops/self connections & only one edge between two vertices
- Bipartite: no edge that connects two vertices in the same set (graph can be divided into 2 disjoint sets)
- Eulerian graph: all vertices have an even degree
- Eulerian cycle: a path that uses every edge exactly once
- Articulation Point: vertex whose removal disconnects the graph
- Bridge: edge whose removal disconnects the graph (cannot be contained in a cycle)
- Undirected Graph: 2 * edges = sum of vertices degrees

Binary Search Trees (BSTs)

Properties:

- Children: 2k + 1 & 2k + 2
- Parent: $\lfloor \frac{k-1}{2} \rfloor$
- Max nodes: $2^{h+1} - 1$
- Max leaves: 2^h
- Max height for complete BST: $\lfloor \log_2(n) \rfloor$

Insert, delete, and search:

- Average: O(log n) if reasonably balanced
- Worst-case: O(n) if the node DNE or unbalanced

R-B Trees

- No nodes have 2 red links
- All paths have the same number of black branches
- A red nod represents a red link to the parent
- Root node is always black
- Insert, delete, and search: Worst-case: O(log n)

2-3 Trees

- All nodes are on the same level
- Worst-case height is max height for complete BST (never taller than min-height for binary tree)
- Insert, delete, and search: Worst-case: O(log n)
- Max nodes = $\frac{1}{2}(3^{h+1} - 1)$
- Max keys = $3^{h+1} - 1$

MST

- All vertices are connected without cycles & the sum of weights is minimized
- Cut: if there are multiple edges connecting two subgraphs, only the minimum is in the MST
- Cycle: if there is a cycle, the max-edge is not in the MST

Algorithms

BFS/DFS: list O(v + e) time or matrix O(v^2) time

Topological Sort: O(v + e)

- Maintain an array of the indegree to all vertices. Remove the vertex with an indegree of 0, add it to the topological ordering, and decrease the indegree to all of its neighbors.

Prim's: O(e log v) time & O(e) space

- Add all vertices of the current node to a min-priority queue based on the weight of their minimum discovered in-edge. Pop the lowest weight vertex and update the queue with newly discovered vertices and new minimum in-edges.

Kruskal's: O(e log e) time & O(e) space

- Iterate through a list of edges from min-weight to max-weight and call find on the vertices of the edge. If find(a) ≠ find(b) then union(a, b).

Strong connectivity: O(v + e)

- DFS on a starting node. If there is an unvisited node then return false. Reverse the graph. Run DFS from the same starting vertex. If there is an unvisited vertex return false, else return true.

Kasaraju's SCC: O(v + e) time

- Run DFS on nodes in lexicographical order if they are not visited already. Add nodes to a stack as they are completed. Reverse the graph. Repeatedly call DFSMark on the top of the stack if it is not already marked.

Dijkstra's: O(e log v) time or O(e + (v log v)) with Fibonacci heap

- Checking if a vertex is in the queue O(1)
- Resetting values in queue: O(e log v)
    - O(log v) to update a value, repeat for all adjacent edges (e) to a vertex

Bellman-Ford: Θ(ve) time & Θ(v) space

- Check the vertices in lexicographical order at most v times. If the distance to the current node is unknown then skip it, otherwise, update the minimum distance to adjacent nodes. If there are no updates in a pass, end the function early.

DAG Distances: O(v + e) time

- Topologically sort the graph (assumes graph has no cycles). Iterate through the edges in topological order and update the minimum distance (current distance + edge weight).

Floyd Warshall All Pairs Shortest Path: O(v^3) time & O(v^2) space

- Use an adjacency matrix to represent the minimum path weight between two vertices. Start with intermediate paths including no nodes and increment to node v - 1 (include nodes 0, 0-1, 0-2, etc.)