Contents

1 Runtime Complexity

Main Idea. We give algorithms a "rating"(read: Big-O notation) based on how many instructions the algorithm takes as a function of the size of the input data. We always consider the *worst case scenario*.

> **Note.** Always give the Big-O notation in terms of the variables that you're given, i.e. if the size of the array is k, then your Big-O notation should be $O(k)$, *not* $O(n)$.

1.1 Single Input Algorithms

To get the Big-O of an algorithm, we only consider the most commonly performed operations, and ignore the rest. In other words, take the term with the highest degree and remove its coefficient.

When you know n is small, forget which Big-O is better and choose the algorithm that is *easiest* to program, as it won't make that much of a difference anyways.

1.2 Multi-input Algorithms

If an algorithm operates on two (or more) *independent* data sets, your Big-O must include *all* variables in the final notation, i.e. $O(c^2 + e)$.

1.3 STL Runtime Complexity

1.3.1 Vector

Purpose: A resizable array

- Inserting an item (top/middle): $O(n)$
- Inserting an item (bottom): $O(1)$
- Delete an item (top/middle): $O(n)$
- Delete an item (bottom): $O(1)$
- Access an item (anywhere): $O(1)$
- Finding an item: $O(n)$

1.3.2 List

Purpose: A linked list

- Inserting an item (top/middle*/bottom): $O(1)$
- Deleting an item (top/middle*/bottom): $O(1)$
- Access an item (top/bottom): $O(1)$
- Access an item (middle): $O(n)$
- Finding an item: $O(n)$

> **Note (*).** To get to the middle, you may have to first iterate through X items, at cost $O(X)$.

1.3.3 Set

Purpose: Maintains a set of unique items

- Inserting an item (anywhere): $O(\log_2 n)$
- Deleting an item (anywhere): $O(\log_2 n)$
- Finding an item: $O(\log_2 n)$

1.3.4 Map

Purpose: Maps one item to another

- Inserting an item (anywhere): $O(\log_2 n)$
- Deleting an item (anywhere): $O(\log_2 n)$
- Finding an item: $O(\log_2 n)$

1.3.5 Queue and Stack

Purpose: Classic stack/queue

- Inserting an item (anywhere): $O(1)$
- Popping an item (anywhere): $O(1)$
- Examining the top: $O(1)$

2 Sorting Algorithms I: Inefficient Sorts

> **Definition.** *Stable/Unstable Sorting*
> An *unstable* sorting algorithm re-orders the items without taking into account their initial ordering, whereas a *stable* sorting algorithm takes into account the initial ordering, maintaining the order of similarly-valued items.

2.1 Selection Sort

- Look at all N items, and find the "smallest" one
- Swap that item with the first item (the smallest item is in the front)
- Look at the remaining $N - 1$ items
- Swap that item with the second item
- Repeat...

Because you need N steps to find the "smallest item", and then $N - 1$ steps to find the second "smallest" item, etc., in total this algorithm runs in $O(N^2)$ time. If the items are already mostly sorted, selection sort still takes just as many steps. Selection sort is also unstable.

2.2 Insertion Sort

- Look at the first two items, and swap them if they are out of order
- Now look at the third item, and insert it into its proper spot, moving things around as necessary
- Repeat until the entire shelf has been sorted

Because you need 1 step to sort the first two books, then 2 steps for the first three, etc., this algorithm runs in $O(N^2)$ time. However, if the books are already sorted, then insertion sort doesn't need to move anything around, so it runs in $O(N)$ time. The worst case scenario is when the items are perfectly mis-ordered. Insertion sort is also stable.

2.3 Bubble Sort

- Look at the first two items, and swap if they're out of order
- Look at the next two items, and swap if they're out of order
- Repeat until you hit the end of the array (now the largest element is at the end)
  - If at least one swap was performed, start back at the top and repeat the process again

Because you need to pass through the list of N elements N times, this algorithm runs in $O(N^2)$ time. Just like insertion sort, this algorithm is really efficient on pre-sorted arrays and linked lists. Bubble sort is also stable.

> **Note (Characteristics of Sorts).** After one round of selection sort, the smallest item is at the front. After one round of bubble sort, the largest item is at the end. After one round of insertion sort, the first two items are in the correct order.

3 Sorting Algorithms II: Electric Boogaloo

3.1 Quicksort

- If the array has only zero or one element, return
- Select an arbitrary element P from the array (typically the first)
- Move all elements that are less than or equal to P to the left of P, and all elements greater than P to the right of P
- Recursively repeat this process on the left and right sub-arrays

> **Note.** The "pivot" P will be in the correct position after doing quicksort once, as all the items to its left are smaller than P, and all the items to its right are greater than P.

Because partitioning the array costs N steps, and we do this $\log_2 N$ times (as we cut the array in half each time), this algorithm runs in $O(N \log_2 N)$ time. If the array is already (mostly) sorted, then the left sub-array is empty and the right sub-array contains $N - 1$ items, and we need to repeat N times, so the runtime complexity becomes $O(N^2)$. Quicksort is also unstable.

3.2 Merge Sort

- Sort the left and right sub-arrays
- Compare the next items in the left and right sub-arrays, and copy over the smaller one
- Repeat...
  - If there are no more items left in one sub-array, copy over all of the remaining items in the other sub-array

Because merging takes N steps to complete, and we are continuously dividing our pile in half each time, we need to run this process $\log_2 N$ times. Thus the runtime complexity of this algorithm is $O(N \log_2 N)$. Merge sort runs efficiently regardless of how the items are originally arranged, but takes up more space because it needs to store an auxiliary list. Merge sort is also stable.

4 Trees

Main Idea. Trees are a data structure that stores values in a hierarchical fashion. We often use linked lists to build trees (each vertex is a node with pointers), and trees can be used as an alternative to arrays and linked lists for storing data.

> **Definition.** *Binary Tree*
> A *binary tree* is a tree in which each node has *at most* two children nodes: a left child and a right child.

4.1 Traversals

- Pre-order Traversal: Process the current node, then traverse the left and right subtrees
- In-order Traversal: Traverse the left subtree, process the current node, then traverse the right subtree
- Post-order Traversal: Traverse the left and right subtrees, then process the current node
- Level-order Traversal: Traverse the nodes level by level, starting from the root node and working your way down

An easy way to remember the order of the traversals is to start at the root node and draw a loop counter-clockwise around all of the nodes. Pre-order will process a node everytime you pass the *left* side of it, In-order will process a node everytime you pass *below* it, Post-order will process a node everytime you pass the *right* side of it. Level-order traversal will go row by row.

4.2 Binary Search Trees

> **Definition.** *Binary Search Tree (BST)*
> A *binary search tree* is a binary tree such that for every node X in the tree,
>
> - All nodes in X's left subtree must be less than X
> - All nodes in X's right subtree must be greater than X

To find an item in a binary search tree, we only need to take $\log_2 N$ steps, because each step we take effectively removes half of the tree. In the worst case scenario, only left nodes or only right nodes are used, and it takes N steps to find an item. The runtime complexity for printing out all the items in a tree or freeing all the nodes in a tree is $O(N)$, because the algorithm needs to visit all N nodes in the tree.

To free all the items in a tree, we need to do a post-order traversal, to make sure that we delete all of the children before deleting the current node.

4.2.1 BST Node Insertion

- If the root is null, create a new node with the given value and make root point to it
- Otherwise, compare the given value with the current node's value
- If the given value is less than the current node's value, go to the left child (if it exists) and repeat. If it doesn't create a new node with the given value and make the current node's left child the new node.
- If the given value is greater than the current node's value, go to the right child (if it exists) and repeat. If it doesn't create a new node with the given value and make the current node's right child the new node.

4.2.2 BST Node Deletion

There are three cases to handle when deleting a node from a binary tree:

1. If the node is a leaf, set the parent's pointer to the leaf to be null and then delete the target node.
2. If the node has one child, relink the parent node to the target node's only child, then delete the target node.
3. If the node has two children, replace the target node with either the largest node in the left subtree, or the smallest node in the right subtree.

4.3 Huffman Encoding

Count the frequencies of every character in your text file, and save it into a list. Then to create a Huffman tree, we perform the following process:

- Take the two items with the lowest frequency off of the bottom of the list, and make them the children of a parent node (whose frequency is the sum of its children)
- Add this back into the list
- Repeat...

When you only have one item left in your list of frequencies, that becomes the root node of your Huffman tree. To get the bit representation of a character in the Huffman tree, just trace a path from the root node to that character in the tree. Start with an empty string, and everytime you go down the left branch, add a 0 to your string, and everytime you go down the right branch, add a 1 to your string. The end result is how you would encode that character using Huffman encoding.

4.4 Balanced Search Trees

The only thing you need to know is that balanced BSTs are always $O(\log_2 N)$ for insertion and deletion. When encountering BSTs in job/internship interviews, ask if the BST is balanced.

5 Hash Tables

Main Idea. Hash tables are often *the most efficient* way to search for data.
The goal here is to have a more efficient way to find data than a BST (which is $\log_2 N$). One way we could do this is to just have a massive boolean array and store a `true` in the index corresponding to the value we want, but this is very space inefficient. An alternative way to do this is to map our items to a smaller range of numbers using the modulus operator.

5.1 Collision Avoidance

Definition. *Collision*
A *collision* is a condition where two or more values both map to the same bucket in the array.

There are two ways to get around collisions: linear probing insertion and the open hash table.

5.2 Linear Probing

To insert an item into the closed hash table, first check if the target bucket is empty. If it is, store the value into the bucket. If it isn't, just keep moving down the array until you find an empty slot. If you go past the end of the array, loop back up to the top of the array.

To search our hash table, we use a similar approach. We first compute a target bucket number using our mapping function and then look in that bucket for our value. If we find it, great! Otherwise, probe linearly down the array until we either find our value or hit an empty bucket. If you run into an empty bucket, it means your value isn't in the array.

Deleting an item in a closed hash table is garbage, so we want to use an "open hash table".

5.3 Open Hash Table

Idea. Instead of storing our values directly in the array, each array bucket points to a linked list of values.

As before, compute a bucket number using your mapping function. Search the linked list at `array[bucket]` for your item. If you reach the end of the linked list, the item is not in the table. To delete an item from the hash table, just remove the item from the linked list.

5.4 Hash Table Efficiency

Question. How large must our hash table be so that it runs quickly?

Definition. *Load Factor*
The *load* of a hash table is the maximum number of values you intend to add divided by the number of buckets in the array.

$$L = \frac{\text{Max number of values to insert}}{\text{Total number of buckets}}$$

The load tells you what portion of your buckets are actually going to be utilized.

Note. Open hash tables are almost *always* more efficient than closed hash tables.

The average number of tries it'll take for you to insert/find an item in a hash table is

$$\underbrace{\frac{1}{2}\left(1 + \frac{1}{1-L}\right) \text{ for } L < 1.0}_{\text{Closed Table}} \qquad \underbrace{1 + \frac{L}{2}}_{\text{Open Table}}$$

Note. When choosing the exact size of your hash table, always try to make it a prime number for more distribution and even fewer collisions.

5.5 A Hash Function for Strings

We could use our own hash function for mapping strings to integers, but we're lazy. Let's use the one provided by C++ by using `#include <functional>`.

We can create a string hashing object with `std::hash<std::string> str_hash`, and then get a value out of it with `unsigned int hashValue = str_hash(hashMe)` (between 0 and 4 billion). We can then apply our own modulo function and return a bucket number that fits into our hash table's array.

5.6 Creating a Custom Hash Function

If we are creating a hash function for our own data type, we want it to satisfy a few criteria:

1. The hash function must always give us the same output for a given input value
2. The hash function should disperse items throughout the hash array as randomly as possible
3. When coming up with a new hash function, always measure hw well it disperses items

5.7 Hash Tables versus Binary Search Trees

	Hash Tables	Binary Search Trees
Speed	$O(1)$ regardless of the number of items	$O(\log_2 N)$
Simplicity	Easy to implement	More complex to implement
Max Size	Closed: Limited by array size Open: Not limited, but high load impacts performance	Unlimited size
Space efficiency	Wastes a lot of space if you have a large hash table holding few items	Only uses as much memory as needed (one node per item inserted)
Ordering	No ordering (random)	Alphabetical ordering

6 Tables

Main Idea. Tables are the building block of databases—they are very useful for organizinig large amounts of data and making it quickly searchable.

> **Definition.** *Table Definitions*
> A group of related data in a table is called a *record*. Each record has a number of *fields* that can be filled with values. A number of these records is what we call a *table*.

6.1 Implementing Tables

We can just use a struct or class to hold the values for each record, and then an array or vector to hold our records.

To make this efficient, we add a new data structure that lets us associate each person's field types with their slot number in the vector, i.e. name with position, ID number with position, phone number with position, etc. Each of these secondary data structures is called an *index*, and helps us efficiently find a record based on a particular field.

> **Note.** Remember that any time you delete or update a record's searchable fields, you also you to update your indexes.

While hash tables are more efficient than BSTs, BSTs can be more useful if you need to process elements in order, i.e. if you need to print out the names in order.

7 Priority Queues

Main Idea. Useful for prioritizing different types of data for processing based on their importance.

> **Definition.** *Priority Queue*
> A *priority queue* is a special type of queue that allows us to keep a prioritized list of items. In such a queue, each item you insert has a "priority rating" that indicates how important it is. In contrast to a regular queue, when you dequeue an item from a priority queue, it dequeues the item with the *highest priority*, instead of the first item inserted.

A priority queue has three operatons:

- Inserting a new item into the queue
- Getting the value of the highest priority item
- Removing the highest priority item from the queue

For a limited set of priorities, we can use n linked lists, one for each priority level (i.e. high, medium, low). If we have a large number of priorities, we can use a *heap* data structure, which is a special kind of binary tree.

> **Note.** While a heap uses a binary tree to store its data, it is *not* a binary search tree.

> **Definition.** *Complete Binary Tree*
> A binary search tree is *complete* if the top $n-1$ levels of the tree are completely fileld with nodes, and all nodes on the bottom-most level must be as far left as possible (with no empty slots between nodes).

7.1 Types of Heaps

There are two kinds of heaps, *minheaps* and *maxheaps*:

- Maxheap:
  - Quickly insert an item into the heap
  - Quickly retrieve the *largest* item from the heap
- Minheap:
  - Quickly insert an item into the heap
  - Quickly retrieve the *smallest* item from the heap

7.2 The Maxheap

A *maxheap* is a binary search tree satisfying:

1. The value contained by a node is *always greater than or equal to* the values of the node's children
2. The tree is a *complete* binary tree

Notice that by definiton, the largest (highest priority) item is always at the root of the tree.

7.2.1 Extracting the Largest Item

1. If the tree is empty, return error
2. Otherwise, the top item in the tree has the largest value (save it for later)
3. If the heap has only one node, delete it and return the saved value
4. Copy the value from the right-most node in the bottom-most row to the root node
5. Delete the right-most node in the bottom-most row
6. Repeatedly swap the just-moved value with the larger of its two children until the value is greater than or equal to both of its children (this is called *sifting down*)
7. Return the saved value to the user

7.2.2 Adding a Node

1. If the tree is empty, create a new root node and return
2. Otherwise, insert the new node in the bottom-most, left-most position of the tree (so it's still a complete tree)
3. Compare the new value with its parent's value
4. If the new value is greater than its parent's value, then swap them
5. Repeat steps 3-4 until the new value rises to its proper place

> **Note.** This process is also known as *reheapification*.

7.3 How to Implement a Heap

We use an array to store our tree, row by row. The root node goes in the first slot of the array, the next two nodes in the next two slots, etc.

Some properties of our array-based tree are:

1. We can always find the root node in `heap[0]`
2. We can always find the bottom-most, right-most node in `heap[count - 1]`
3. We can always find the bottom-most, left-most node in `heap[count]`
4. We can add or remove a node by simply setting `heap[count] = value;` and/or updating our count

To find the indices for the left and right children of a node, we have

$$\text{leftChild(parent)} = 2 \cdot \text{parent} + 1 \quad \text{and} \quad \text{rightChild(parent)} = 2 \cdot \text{parent} + 2.$$

To find the parent of a child node, we have

$$\frac{\text{child} - 1}{2} = \text{parent}.$$

> **Note.** The above only works if you use integer division.

Because our tree is a complete binary tree, it will have a height of $\log_2 N$. Inserting and extracting requires "bubbling" the items up and down the tree, so inserting/extracting all N items takes $2N \log_2 N$ steps, so the runtime complexity is $O(N \log_2 N)$.

8 Heapsort

A naïve way to do this is to first insert all of the items into a heap and then extract them all, but this runs in $O(N \log_2 N)$ time.

8.1 Efficient Heapsort

Given an array of N numbers that we want to sort:

1. Convert our input array into a maxheap
2. While there are numbers left in the heap:
   (a) Remove the biggest value from the heap
   (b) Place it in the last open slot of the array

To make this more efficient, start checking at the parent of the right-most node in the bottom-most row to avoid checking for all of the single-node subtrees.

The runtime complexity of heapsort is $O(N \log_2 N)$ because converting the array to a heap takes N steps, and extracting the items takes $N \log_2 N$ steps.

9 Graphs

> **Definition.** *Graph*
> A *graph* is an ADT that stores a set of entities and also keeps track of the relationships between all of them.

There are two main kinds of graphs, *directed* and *undirected* graphs. Edges in directed graphs can only go on one direction, while edges in undirected graphs can go in either direction.

The easiest way to represent a graph is with a double-dimensional array. The size of both dimensions of the array is equal to the number of vertices in the graph, and each element in the array indicates whether or not there exists an edge between vertex i and vertex j. This is called an *adjacency matrix*. Furthermore, if you raise the adjacency matrix to the nth power, you get which vertices are n edges apart.

9.1 Another Way to Represent a Graph

We can also use an array of n linked lists (an *adjacency list*) to represent a graph. If we add a number j to the ith list, this means there is an edge from vertex i to vertex j.

9.2 Which Representation to Use?

Use an adjacency matrix if you have lots of edges between vertices but few vertices ($< 10,000$ vertices). Use an adjacency list if you have few edges between vertices and lots of vertices ($> 10,000$ vertices). A graph that has many edges is called a *dense graph*, and one with few edges is called a *sparse* graph.

9.3 Graph Traversals

There are two main types of graph traversals (both of which we've seen before): *depth-first traversal* and *breadth-first traversal*. The former employs a stack, and the latter a queue.

9.4 Weighted Graphs

Definition. *Weighted Graph Definitions*
A *weighted graph* is a graph where each edge connected vertices u and v has a *weight* or *cost* associated with it. The *weight* of a path from u to v is the sum of the weights of the edges between the two vertices. The *shortest path* between two vertices is the path with the lowerst total cost of edges between the two vertices.

9.5 Finding a Shortest Path

To find a shortest path, we use Dijkstra's Algorithm. It relies on the basic idea of *settled vertices* (we know the minimal distance to the start) and *unsettled vertices* (we don't know the minimal distance to the start). The algorithm proceeds as follows:

- Set all vertices to have a distance of $-\infty$
- Let the start vertex A have a value of 0
- Let all of A's neighbors have a tentative value of the weight of the edge connecting A and the vertex
- Moving to the neighbor of A with the smallest value, repeat this process (this vertex is now settled)
  - If you find a path to a vertex that is less than the vertex's original value, update the table with the lower value

To actually implement this in code, we perform the same algorithm with two arrays—an integer array holding the lengths of the minimal paths from the start vertex to that vertex, and a boolean array that holds whether each vertex is "settled" or not.