Data Structures and Algorithms

May 7, 2025

This document contains a collection of notes on Data Structures and Algorithms as taught in ESC190. I assume the reader knows how to code in C and Python and for the most part, topics covered in ESC180 are not explained unless I think I should. There are places where I supplement the ESC190 notes with my notes on competitive programming, so these notes do not replace Qilin's or attending lecture (I go to lecture... right?). Further, Qilin's notes contain more code examples because I was lazy.

These notes are not self-contained. Perhaps if I have time I will come back and flush them out, so someone can read it start to finish. Nevertheless, hopefully this helps someone!

This document is dedicated to the Zhang Wentao Fan Club ™

Good luck in second year!
Mike Guerzhoy

Figure 1: Michael Guerzhoy's Signature

The Prerequisites

Runtime Analysis

We want to compare different algorithms to see which ones perform "the best". We can compare algorithms on how much memory they take up and how fast they run. These types of analysis are called *space* and *time* complexity respectively. For the types of algorithms used in this course, space complexity isn't a big concern.

Time complexity is a measure of how fast an algorithm runs.

Since different compilers and processors take different amounts of time to run, it's helpful to look at the number of steps it takes for a program to run based on input size, and not compilation time. Specifically, we look at the greatest possible steps it takes for an algorithm to compile [1,2]. In other words, we are concerned with the worst possible case of an algorithm which we describe using Big Oh notation: $O(f(n))$, where n represents the number of inputs $f(n)$ represents the number of steps to compile. Figure shows $O(n)$ and $O(n^2)$ time complexities.

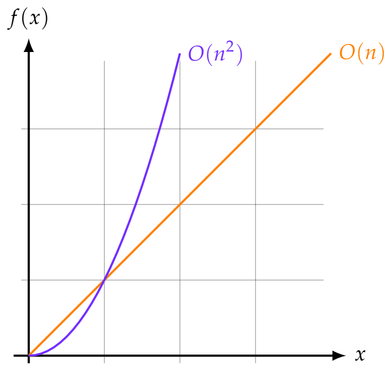


Figure 2: Example graphs of time complexities.

Importantly, Big Oh notation only considers the behaviour, or general shape of a function as the input size tends to infinity[3]. So, constant and lower order terms are ignored (think about taking the limit of $f(n)$ as n tends to infinity, only higher order terms are significant). The rules are summarized below. I was too lazy to derive them, but I hope they are at least somewhat intuitive from the idea

[1] Re: ESC180. Little explanation given here

[2] This is the asymptotic upper bound of compilation steps.

[3] For a more mathematically rigorous treatment of time complexity, check out Donald Knuth's Art of Computer Programming. But to summarize, we can find constants M and n_0 so that $O(f(n)) \leq M|f(n)|$ for all $n \geq n_0$ (this just states that $O(f(n))$ has an upper bound)

that big oh notation describes the shape of a function.

$$\begin{aligned}
f(n) &= O(f(n)) \text{ a function is bounded by itself} \\
c \cdot O(f(n)) &= O(f(n)) \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n))) &= O(f(n)) \text{ Bounding something that's already bounded does not change the bound} \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \text{ I don't have a good way of thinking of this yet}
\end{aligned}$$

You don't need to know the rules above, but they are cool.

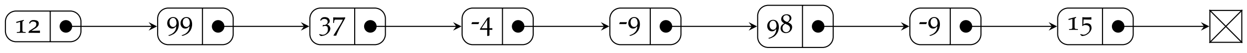
Data Structures

Data structures help organize and store "structural relations" data[4]. Without data structures, data would be stored as amorphous blobs which are hard to keep track of. ESC190 studies linked lists, stacks, queues, priority queues and arrays which are commonly used data structures.

[4] Relationships between different pieces of data

Linked Lists

Linked lists store a data point, and a pointer to the next data point. So suppose you have a list of integers: `{12,99,37,-4,-9,15}`, then you can visualize a linked list as follows:



One way to implement a linked list is to define a `node` data type that contains a data value, and a pointer to the next element in the linked list. You need to know how to append and delete elements at some index and search for nodes of specified values in a linked list. The terminating node always points to `null`[5]. The implementation for linked list is long and is included in the appendices.

[5] There's also something called *doubly-linked lists* which is linked list where each noted stores a pointer to the previous node in addition to a pointer to the next node.

Stacks

Imagine a stack of plates. The last plate you add to the stack is the first to be taken out of the stack. If you take your plate to be data, you get a stack data structure![6]

[6] This type of data structure is called LIFO - last in first out

```python
class Stack:
    def __init__(self):
        self.data = []
    def add(self, val):
        self.data.append(val)
    def pop(self):
        return self.data.pop(val)
```

Python lists make the `pop` operation very easy.

Queues

Imagine a queue (lineup) of people. The first person to enter the line is the first to leave. Now imagine the people you add/remove from the queue is data. That's your queue data structure!

```
class Queue:
    def __init__(self):
        self.data = []
    def add(self, val):
        self.append(val)
    def remove(self)
        returnVal = self.data[0]
        del self.data[0]
        return returnVal
```

Please note I did not test the implementations for `Stack` and `Queue`

Graphs

Graphs store information about relations between pairs of objects in a collection of objects[7]. We can visualize graphs to contain objects called nodes and connections between nodes called edges. We can

[7] This is hard to understand, fix later

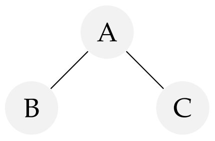


Figure 3: A simple graph.

have undirected graphs, where you can move freely between nodes, and directed graphs, where you move in specified directions. We can also have weights representing the strength of relationship? between nodes in weighted graphs.

Representing Graphs

Adjacency lists and adjacency matrices are the standard approaches to representing graphs. We will represent the following graph in both formats.

Adjacency List: a list for every node that contains all its neighbours.

Dynamic Programming

Dynamic programming (DP) is about reducing repeated computations. This is done by storing computations, and using them when needed. A classic example of where DP can be applied is to the recursive implementation of calculating the fibonacci sequence. Consider the definition of recursive function `fib(n)`:

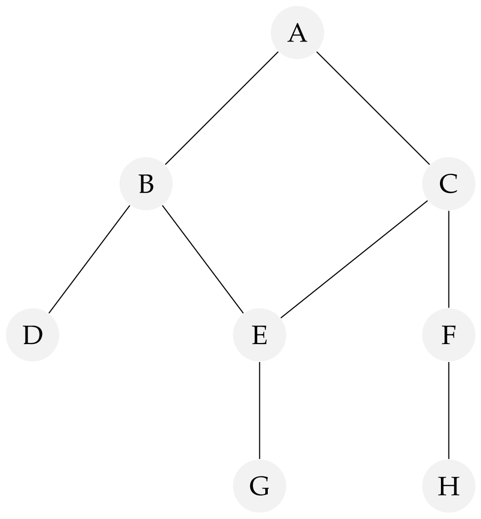


Figure 4: Example graph

```
def fib(n):
  if n == 2:
      return 1

  if n == 1:
      return 0

  else:
      return fib(n-1)+fib(n-2)
```

Look for repeated function calls in the call tree for `fib(5)`.

$$\begin{tikzcd}
 & & & & \text{fib}(5) \arrow[dll] \arrow[drr] & & & \\
 & & \text{fib}(4) \arrow[dl] \arrow[dr] & & & & \text{fib}(3) \arrow[dl] \arrow[dr] & \\
 & \text{fib}(3) \arrow[dl] \arrow[dr] & & \text{fib}(2) & & \text{fib}(2) & & \text{fib}(1) \\
\text{fib}(2) & & \text{fib}(1) & & & & &
\end{tikzcd}$$

`fib(3)` and `fib2` are repeated, for example. This is a waste of computation resources - unacceptable. Let's store all our computations, and before making a new one check if we've done it already. If we have, just use that value.

```
# just one way to do it!!
n = 5
memo = [-1]*(n+1) # use n+1 to avoid index out of bound error.
# we use -1 to check if memo[n] has been calculated yet.
memo[1], memo[2] = 0, 1

  def fib(n):
    if memo[n] != -1:
        return memo[n]
    if n == 2:
        return memo[2]

    if n == 1:
        return memo[1]

    else:
        memo[n] = fib(n-1)+fib(n-2)
        return memo[n]
```

Fun fact: storing recursive function calls is called *memoization* and storing iterative function calls is called *tabularization*. For ESC190, you should probably be able to recognize the DP solution to the coin combination problem.

Graph Theory

We learned about the graph data structure, and how it can be useful to represent paths between nodes. A common task is to find a path between two nodes. We can do this systematically with graph traversal algorithms. Depth First and Breadth First Search are most common. Play around with this simulation to visualize different graph traversal algorithms. You'll notice implementation wise, the only difference between Breadth First Search and Depth First Search is whether you use a queue or stack[8].

[8] To help remember which data structure to use for BFS or DFS: BFS explores all the neighbours of a node first. So the first node you see is the first node you explore –> Queue. DFS explores one path at a time, so all neighbours of a node are explored later –> Stack.

Breadth First Search

Search along the "breadth" of the graph. Again, the algorithm visualizer linked above is useful to visualize this. Essentially, you explore all the neighbours of a node at a time.

Depth First Search

Search along the "depth" of a graph. This time, you pick a node, and explore one path until it terminates[9].

[9] Good enough reason for 190

Dijkstra's Search

Dijkstra's search is used when we deal with graphs of with weighted edges. The weighted edges could represent something like the cost to make a certain decision (if you think of flowcharts), or maybe the time of travel to get to a node (if you think nodes as destinations, and the cost represents how long to travel between destinations).

Our goal is to get from one node to another by taking the lowest *cost* path, where we define *cost* as the sum of edge weights between two nodes[10].

[10] Note to self: terminology can be more precise. But hopefully the idea is clear.

Dijkstra's search uses a greedy approach[11] to find this minimum

[11] The greedy approach is when you pick the minimum cost choice at every step of your algorithm.

cost path between two nodes. To perform Dijkstra's search:

1. Pick a start and target node.
2. Set every node to infinity other than the start node.
3. Explore all the neighbours of the starting node. Store `min(current cost, new cost)` to go to each neigbhour node. Note the cost is cumulative.
4. Pick the neighbour node with the smallest cost.
5. Repeat the same type of exploration at the node you picked.
6. Repeat until you reach the target node.

Qilin Xue posted the pseudocode for Dijkstra's which I'll include. Some notation: $G = (V, E)$ represents a connected graph G that has interconnected vertices V and edges E. The notation $V§$ refers to the set of vertices that are in V but not in S. $|(u, v)|$ is the weight of the edge connecting u and v.

```
Dijkstra(G = (V,E), startNode)
    S = startNode # S is the set of explored nodes
    d(startNode) = 0 # d(v) is the shortest path from startNode to v

    while S != V
        choose v in (V \ S) such that d(u) +|(u,v)| is minimized where u is
     in S
        Add v to S
        Set d(v) = d(u) + |(u,v)|
# Taken from Qilin's notes
```

Random thoughts:

- BFS is just Dijkstra's if all edge weights are equal
- Use a priority queue to efficiently retrieve the next node to explore
- Time complexity as written: $\mathcal{O}(|V|^2)$[12]. This is because in the worst case
- Time complexity using a pirority queue

[12] $|V|$ is the size of the graph, or number of vertices

if all the node weights are the same, is dijkstras the same as bfs?
- implementation - time complexity - pq implementation - time complexity - closing thoughts: - BFS is just Dijkstra's algorithm if all the edge weights are equal

Bonus: Bellman Ford Algorithm

The Bellman Ford search algorithm is similar to Dijkstra's but handles graphs with negative weights. This is done by checking for the minimum cost to each node's neighbours $N - 1$ times, where N is the number of nodes in the graph. The idea is that by $N - 1$ iterations, you are guranteed to have found the minimum distances to every node. It's not exactly repeating Dijkstra's algorithm multiple times, and each iteration your graph input is the output of the previous iteration[13]. The difference is that Dijkstra's explores the lowest cost[14] unvisited neighbours to a node; Bellman Ford explores all neighbours.

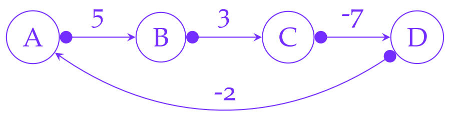
[13] For the first iteration, set all node distances to infinity

[14] You can think of this as the closest neighbouring nodes

```
BellmanFord(G = (V,E), startNode)
    # initialize distance to every node to infinity except for the start
     node
    d() = inf #d(v) is the shortest path from startNode to v
    d(startNode) = 0 # d(v) is the shortest path from startNode to v

    for x in range |G|-1 # |G| is the number of nodes in graph G
        for each edge (u,v) that has weight w
            if distance(u) + w < distance(v)
                Set d(v) = d(u) + w
```

The pseudocode does not include an additional step to check for *negative-weight cycles*. This happens when the cost to take a path decreases every iteration. For instance, consider the following graph cycle:



The sum of costs in this cycle is $5 + 3 - 7 - 2 = -1$. When the Bellman Ford algorithm checks the costs of this cycle again (remember it does it $|V| - 1$ times). In this case, the least cost path is undefined. If the cycle sum is positive, the sum jumps to infinity which is not a problem, since our goal is to find the least cost path from the starting node to any other node on the graph.

So, Dijkstra's takes a greedy approach and Bellman Ford takes a brute force approach. We can prove we find the optimal solution after $N - 1$ iterations.

Greedy Best First Search

Greedy Best First Search is like Dijkstra's except we choose the next node to explore based on an heuristic function. An heuristic function is something we define for a given problem that tells us approximately how far away we are from our goal. We pick the node to explore that has the minimum value in our heuristic function.

For instance, consider a maze-finding algorithm. We are at a fork in the maze. Our heurestic function is the euclidean distance from where we are to the end of the maze. We choose the path that has the minimium distance to the end of the maze first, before exploring the longer path. **Greedy Best First Search does not gurantee you find**

the least path cost between two nodes but is *usually* quicker than Dijkstra's since your choice of node to visit takes you closer to your destination.

A Search*

A* is like Greedy Best First Search but we incorporate the edge weight too. See this link for more information.

Binary and Binary Search Trees

Bonus: An Introduction to Statistical Learning

NOTE: I'm speedrunning this section and should probably be studying for exams right now. As such, I'm skipping a lot of definitions, explanations, and equation typesetting. A book I like that explains this in depth is *An Introduction to Statistical Learning* by Gareth M. James, Daniela Witten, Trevor Hastie and Robert Tibshirani.

We collect a lot of data and want to draw conclusions from it. Statistical learning is a branch of mathematics focussed on finding conclusions from data.

The simplest problem involves taking two variables we think are correlated, and trying to find how to get from one variable value to the other. For example, consider a set of observations $X = \{X_1, X_2, X_3, ..., X_n\}$ which we may be related to another variable $Y = \{Y_1, Y_2, Y_3, ..., Y_n\}$. We aim to find a function f that maps $X_i \in X$ to $Y_i \in Y$. So, $f(X) = Y$.

Finding f is helpful for *inference* (estimating model parameters)[15] and *prediction* (predicting what Y would be for some X_k that isn't in X).

[15] this can be explained better

Clustering

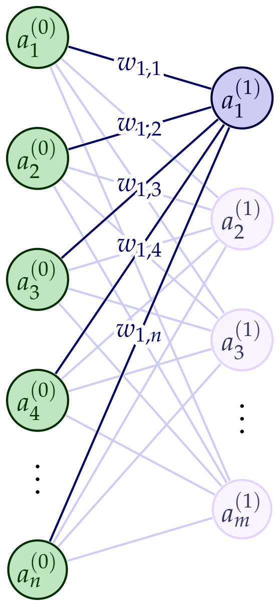
Sometimes we have a lot of data that seems to organize itself into distinct groups.

UPDATE FIGURE: change colouring, double check the matrix math stuff

Linked Lists Implementation

Python implementation

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None # points to null by default
```



$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma \left[\begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix} \right]$$

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{a}^{(0)} + \mathbf{b}^{(0)}\right)$$

```
class LL():
    def __init__(self):
        # store head node
        self.head = None

    def get_i(self, i):
        x = 0
        cur = self.head
        while x < i:
            cur = cur.next
            x += 1
        return cur.data

    def append(self, value):
        # traverse to end of linkedlist and add a node
        newNode = Node(value) # create new node
        # case 1: head node is None
        if self.head == None:
            self.head = newNode

        else:
            # find the last node
            cur = self.head
            while cur.next != None:
                cur = cur.next
            cur.next = newNode
    def insert(self, value, i):
        newNode = Node(value)
        if i == 0:
            tmp = self.head
            self.head = newNode
            newNode.next = tmp # this can be done in two lines by the way
        else:
            cur = self.head
            for i in range (i-1):
                cur = cur.next
            newNode.next = cur.next
            cur.next = newNode
    def printLL(self):
        cur = self.head
        if cur == None:
            print("empty")
        while (cur != None):
            print(cur.data, " ", end="")
            cur = cur.next

# driver code
hi = LL()
hi.append(4)
hi.append(5)
print("hello")
hi.printLL()
print("\n")
hi.insert(5,2)
hi.printLL()
print(hi.get_i(2))
```

C implementation