Sheffield Hallam University

Data Structures and Algorithms
Department of Computing
55-508226-AF-20223

Rory Boyes: c1054267

Contents

1 Project 1

1.1 Computational Complexity

Algorithm 1 Triple Nested For Loop with Variable Assignment and Increment

```
1: procedure ADL FOR ANALYSIS
2:     for i ← 1 to n by 1 do
3:         for j ← 1 to i by 1 do
4:             for k ← 1 to j by 1 do
5:                 x = x + 1
6:             end
7:         end
8:     end
```

- Let f represent the body of our function.
- Let n represent the input value of our function, an integer variable which tends to infinity.
- Let $f(n)$ denote the total number of instructions executed when f is applied to n.
- Let T represent time.
- Let $T(n)$ be the relative runtime of our function,(complexity/T).
  T(n) is a numerical value which represents the runtime of a given function relative to an arbitrary size of input, from which we can infer it's order of growth.
- Let θ be a tight bound on $T(n)$, which denotes both the asymptotic upper-bound, (O), as well as the asymptotic lower bound, (ω), of a given function, f.

It is clear from the outset that our space complexity is $O(1)$, as our integer assignments allocate a constant amount of memory, despite the incrementation of x. To contrast, if our inner statement involved the assignment of a new variable within a dynamic data structure which was constructed outside of the outer loop, then our space complexity would become dependant upon n.

We can derive $T(n)$ formally by expressing our function as the following triple summation;

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} c$$

Notice that the inner summation index, k, is not present within the operation of our innermost summation, and can therefore be represented as a constant, c.

We can therefore simplify our innermost k summation as following, where $T(n) =$

to the sum from $i = 1$ to n, of the sum from $j = 1$ to i, of the summand..(j) multiplied by the number of present terms (the top bound− the bottom bound)+1, multiplied by our operation constant, c.

$$\therefore T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} (j - 1 + 1)c$$

This can be further simplified as following;

$$\sum_{i=1}^{n} \sum_{j=1}^{i} jc$$

To begin resolving our j summation, it must be noted that the index, j is present within the inner operation. At this stage, we can either bound the summation, by deriving an upper and lower bound, or by applying a suitable formula. In our case, we are dealing with an arithmetic sum, (we are summing j), and thereby by factoring out c, we can directly apply the sum of natural numbers as following;

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\therefore T(n) \sum_{i=1}^{n} \sum_{j=1}^{i} c = \sum_{i=1}^{n} c \frac{i(i+1)}{2}$$

We may now arithmetically combine our constants while distributing i resulting in the following summation;

$$T(n) = \sum_{i=1}^{n} \frac{c}{2}(i^2 + i)$$

By factoring out $\frac{c}{2}$, and distributing our summation, we can make use of the following formula, the sum of squares of natural numbers;

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$T(n) = \frac{c}{2} \left[\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i \right]$$

$$\therefore T(n) = \frac{c}{2} \left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \right]$$

We now have a closed form expression, containing only basic operations therefore we can deduce its time complexity. We can see that our expression involves the addition of 2 products. Constants aside our first product is the result of the multiplication of 3 n terms, resulting in the product n^3, whilst our second

product is the result of the multiplication of 2 n terms, resulting in the product 2^n. As the addition operator is a linear operator, and we are only interested in the dominant term we can drop the constant factors and low-order terms, therefore we have determined that $T(n) \in \theta(n^3)$ and is thereby of exponential time complexity, more specifically, the time complexity is cubic, and therefore also of polynomial time complexity, more broadly speaking. By tightly bounding our function we have deduced both the upper, and lower bound simultaneously, as we may state that the function has a strict runtime, which may only deviate by a value asymptotically less than that of our complexites exponent, We can therfore state that both $O(n)$ and $\omega(n)$ of $f(n)$ are equivalent, and that taking the lower and upper bounds separately would be a provide no further insight.

1.2 Incorporation of Formative Feedback

2 Project 2

When devising an algorithm for sorting plates, one must consider not only suitable data structures, and their effect on time/space complexity, but also the real world implications of sorting these physical items. We must differentiate between the notion of space, in computing terms, and that of the physical realm.

2.1 Algorithms in ADL

Algorithm 2 Iterative Quick Sort

```
1: procedure SORTSTACK
2:    (stack:  Stack<Integer>,
3:    validPlateSizes:  HashSet<Integer>)
4:    returns Stack<Integer>
5:
6:      sortedStack := create a new Stack<Integer>
7:      while stack is not empty
8:        temp := stack.pop()
9:        if temp is in validPlateSizes
10:         while sortedStack is not empty
11:         and sortedStack.peek() < temp
12:           stack.push(sortedStack.pop())
13:         end while
14:         sortedStack.push(temp)
15:       end if
16:     end while
17:     return sortedStack
18: end procedure
```

The time complexity of the sortStack algorithm above is $O(n^2)$, where n is the number of elements in the input stack. This is due to the presence of nested loop which iterates over the elements of the stack. The inner loop, which is executed once for each element in the stack, has a time complexity of $O(n)$ as it iterates over all elements in the sorted stack. The outer loop also has a time complexity of $O(n)$ because it iterates over all elements in the input stack. Therefore, our overall time complexity is $O(n^2)$.

The space complexity of the algorithm is $O(n)$, as the algorithm uses a single stack to store the sorted elements. The size of the stack grows linearly with the number of elements in the input stack, so the space complexity is $O(n)$.

If space complexity was our primary concern, this approach would acceptable. Utilizing a hashed set for checking our valid plate sizes will prevent us from traversing during this operation, but despite this, our function is still $O(n^2)$, as the fastest growing term takes precedent.

Algorithm 3 Bucket Method with Hashing

```
1: procedure sortStackWithMap
2:    (map:  Hashmap<Integer, Integer>,
3:    stack:  Stack<Integer>)
4:
5:      popped:  Integer
6:      while stack is not empty
7:        popped <- stack.pop()
8:        if map contains key popped
9:          map.put(popped, map.get(popped)+1)
10:       end if
11:     end while
12:
13:     for(each key in map)
14:       while key > 0
15:         stack.push(key)
16:         map.put(key, map.get(key)-1)
17:       end while
18:     end for
19: end procedure
```

The time complexity of the algorithm above is $O(n)$, where n is the number of elements in the input stack. This is because the algorithm performs a single pass over the elements of the stack, counting the number of occurrences of each element inside of the Hash-Map. The time complexity of this operation is $O(n)$, as the hash map has an average-case time complexity of $O(1)$ for insertion and retrieval of elements due to usage of a hashing function. $\therefore$ the $T(n)$ grows linearly with the size of the input.

It is worth noting that if we were to use another data type such as a string to represent each plate type, a linked list value store implementation of a hash map could decay to $O(n^2)$ in the worst case, due to collisions. As we are using integers we may also state that $T(n) \in \theta(n)$

The space complexity of the function is also $O(n)$, as the hash-map used to store the counts of each element of the unsorted stack grows linearly with the number of elements in the input stack, i.e, there is no possibility that the number of elements within the hash-map could ever exceed that of the input stack. $\therefore$ the space complexity is $O(n)$.

Furthermore, had we not been working under the constant of only moving one element at any given time, the time complexity of the algorithm could be improved further to $O(nlogn)$ by sorting the elements in the hash-map before pushing them back onto the stack. However, this would be at the detriment of additional space complexity, as the usage of a sorting algorithm, such as quick or merge sort, would require additional space to store auxiliary structures such as partition indices.

2.2 Data Structures

Structure 1 HashMap

```
1: public class HashMap(<K, V>) {
2:
3:     private static final int DEFAULT_CAPACITY = 16;
4:     private static final double LOAD_FACTOR = 0.75;
5:     private int capacity;
6:     private int size;
7:     Entry<K, V>[] bucket;
```

A bucket is a term used to describe a space in the array that stores entries of the same hash code. Capacity determines how many buckets are available in the hash map, and therefore, how many Entry objects can be stored in the map before the map needs to be resized.

The default capacity constant field determines the initial size of the array used to store entries, (key value pairs). 16 has been chosen as a sane default, as it is a power of 2, and $\therefore$ allows for efficient index calculations using bit shifting, The default size is small enough to minimize memory overhead, while large enough to prevent resizing too often. The initial capacity can also be specified within the constructor, by passing the capacity as an argument. This could be useful in cases where it is known in advance that the hash-map will need to store a large data set, though a larger initial capacity will require more memory.

The load factor of a hash-map is the ratio of the number of entries in the hash map to the capacity of the hash map, and is calculated by dividing the number of entries in the map by the capacity of the map It is used to determine when the capacity of the hash map should be increased to maintain good performance. The default maximum load factor is defined to allow for the aforementioned resizing once it has been exceeded. A load factor too high could cause the map to perform poorly as collisions may occur causing our time complexity to decay. Contrarily, too low of a load factor is inefficient in regards to space complexity as the resizing will occur prematurely, resulting in unnecessarily memory allocation.

0.75 is a fairly typical load factor for hash-map implementations as it provides a middle ground between these trade offs. The load factor can also be specified by passing a second argument to the constructor upon initialization.

Nested Class 1 Entry

```
1: private static class ENTRY(<K, V>) {
2:
3:    k key
4:    v value
5:    Entry<K, V> next
6:
7:    Entry(K key, V value, Entry<K, V> next) {
8:      this.key = key;
9:      this.value = value;
10:     this.next = next;
11:   {
12: {
```

The Entry class is a nested class within the HashMap class that represents a key-value pair in the map. Each Entry has a key and a value, as well as a reference to the next Entry in the bucket of entries, forming a linked list of Entries within each bucket. The constructor takes all three of these fields as arguments.

The key of each entry is passed through a hash function to determine which bucket it is to be stored in. If the bucket is empty, the new entry becomes the first entry in the bucket. If a collision occurs, the new entry is added to the end of the list within the corresponding bucket. Subsequently the hash function can again be used to determine which bucket to search to retrieve the desired key value pair based on the hash of the key.

Hashmap Constructor 1

```
1: public HashMap HASHMAP {
2:    this(DEFAULT_CAPACITY);
3: }
```

The default hashmap constructor is the most commonly used,and utliziles the aformented predetemined capacity to instatiate a new hashmap with no given arguments.

Hashmap Constructor 2

```
1: public HashMap HASHMAP(int capacity) {
2:    this.capacity = capacity;
3:    this.bucket = new Entry[capacity];
4: }
```

As mention the second constructor allows specification of initial capacity of the map upon instantiation. A new array of type Entry is also declared with the specified capacity assigned to the bucket field.

2.3 Software and its Presentation, including testing

https://github.com/Riverside96/-Project_2
https://youtu.be/s-Bb-ijeFlY

2.4 Descriptive Report, including artefacts

Hashmap Method 1

```
1: public void PUT(K key, V value) {
2:     int index = hash(key)
3:     Entry<K, V> entry = bucket[index];
4:     while (entry != null) {
5:       if (entry.key.equals(key)) {
6:         entry.value = value;
7:         return;
8:       {
9:       entry = entry.next
10:    {
11:    bucket[index] = new Entry<>(key, value, bucket[index]);
12:    size++;
13:    if (size > capacity * LOAD_FACTOR) {
14:      resize();
15: }
```

The put method is used to add a new key-value pair to the map, or update the value of an existing pair.

First, the hash code of the key argument is determined via the hash function, which will be later detailed. An int, index is assigned with the value of the argument hash corresponding to the index of the bucket where the entry should be stored. The first Entry at the specified index of the bucket array is then retrieved.

This is then used to iterate through the list of entries to check whether another entry with the same key as the argument key is present. If it is the case that another entry is present with the same key, the vale of that entry is updated, and return is called to leave the function.

Otherwise, this iteration continues until null is reached, signifying the end of the list. If an entry with the same key is not found, a new Entry is created with the given arguments. which is then added to the beginning of the linked list of entries for the bucket.

The size field of the hash map is then incremented, and a check is then performed to determine if the resize function is to be called. As mentioned previously, this is determined by the load factor, which must be exceeded before the resize function is called which will increase the capacity of the map and redistribute the entries between the new buckets.

Hashmap Method 2

```
1: public void GET(K key) {
2:     int index = hash(key);
3:     while (entry != null) {
4:       if (entry.key.equals(key)) {
5:         return entry.value;
6:       }
7:       entry = entry.next;
8:     }
9:     return null;
10: }
```

The get method is used to retrieves the value associated with a given key from the map.

It is used by many other methods of the HashMap class to access the values stored in the map.

The get method first calculates the hash code of the key argument, which is subsequently used to determine the index of the bucket where the entry with the matching key is stored.

A while loop is used to iterate through the linked list of Entry's for the given bucket to find the entry with the matching key. Again, a null check is used, similarly to the put method, to signifying the end of the list.

If an entry with the matching key is found, the function returns the value of the entry. Otherwise, if an entry with a matching key is not found, the function returns null.

Hashmap Method 3

```
1: boolean CONTAINSKEY(K key) {
2:     int index = hash(key);
3:     Entry<K, V> entry = bucket[index];
4:     while (entry !=null) {
5:       if (entry.key.equals(key)) {
6:         return true;
7:       }
8:       entry = entry.next;
9:     }
10:    return false;
11: }
```

Similarly to the get method, the containsKey method takes a key as an argument, and checks for its presence within the map, instead returning a boolean if it is found, else returning false if a null is reached, again signifying the end of the list.

Hashmap Method 4

```
1: public void REMOVE(K key) {
2:     int index = hash(key);
3:     Entry<K, V> entry = bucket[index];
4:     if (entry == null) {
5:       return;
6:     }
7:     if (entry.key.equals(key)) {
8:       bucket[index] = entry.next;
9:       size--;
10:      return;
11:    }
12:    While (entry.next != null) {
13:      if (entry.next.key.equals(key)) {
14:        entry.next = entry.next.next;
15:        size--;
16:        return;
17:      }
18:      entry = entry.next;
19:    }
20: }
```

The remove function is used to remove a key-value pair from the map, given the key. The method takes a key as an argument and uses it to determine the bucket in which the key-value pair is stored. It does this by calling the hash function to calculate the hash-code of the key.

A null check is first performed to determine whether the input argument is valid. If the input argument is null the function is left, and no action is performed on the object.

Next the index of the bucket is determined by taking the hash of the key argument. Once the index of the bucket is determined, the method traverses the linked list of Entry's stored in the bucket, searching for an Entry with a key that matches the given key.

If an entry with a matching key is found, it is removed from the linked list by updating the next field of the previous entry. If an Entry with a matching key is not found, return is called to leave the function. Once the Entry object is removed, the size field of the HashMap is decremented to reflect the fact that the map now has one fewer key-value pair.

Hashmap Method 5

```
1: public int SIZE {return size; }
```

The size function is self explanatory, it simply returns the current size field.

Hashmap Method 6

```
1: private void RESIZE {
2:    capacity *= 2;
3:    Entry<K, V>[] oldBucket = bucket;
4:    bucket = new Entry[capacity];
5:    size = 0;
6:    for (Entry<K, V> entry :  oldBucket) {
7:      while (entry != null) {
8:        put(entry.key, entry.value);
9:        entry = entry.next;
10:     }
11:   }
12: }
```

As previously mentioned, the resize function is used to increase the capacity of the map when the load factor (the ratio of the number of key-value pairs in the map to the capacity of the map) exceeds a certain threshold.

First the capacity field of the map is doubled. A new array of type Entry is then created with the updated capacity which is then assigned to the bucket field. The size field is then reset to zero.

The function then iterates through each key-value Entry pair of the old array, Each entry is then re-inserted into the map using the put method. This causes the key-value pair to be re-hashed and inserted into the appropriate bucket in the new array.

Resizing the map allows the map to function efficiently, even as the number of key-value pairs grows. It prevents the map from becoming too full, which would lead to collisions. This would cause the put and get methods to take longer to execute as aforementioned.

Hashmap Method 7

```
1: private int HASH(K key) {
2:     int hashCode = key.hashCode();
3:     return Math.abs(hashCode)
4:     }
5: }
```

The hash function of the is used to calculate the index of the bucket in which a key-value pair should be stored, based on the key. It is used by the put and get methods to determine where to store and retrieve key-value pairs.

The hash function takes a key as an argument and calculates the integer hashcode of the key using the hashCode method of the Object class.

The modulo operator is then used to map the hashcode to an index in the range [0, capacity), where capacity is the capacity of the map (i.e., the number of buckets in the map).

This is done by taking the absolute value of the hash-code and then calculating the remainder when it is divided by the capacity. This helps to distribute the key-value pairs evenly across the buckets, which helps to ensure that the map remains efficient as the number of key-value pairs grows.

The hash function is an important part of the HashMap implementation, as it determines how key-value pairs are stored and retrieved in the map. It's important to choose a good hash function that distributes the key-value pairs evenly across the buckets and minimizes the number of collisions (i.e., when two or more keys hash to the same index).

The use of hashing within the map allows us to achieve the aforementioned time complexity desired for out algorithm, saving precious time.

2.5 Incorporation of formative feedback

3 Project 3

3.1 Algorithms in ADL

Algorithm 1 Depth First Traversal

```
1: procedure depthFirstTraverse
2:    (start:Vertex, visitedList:List<Vertex>)
3:
4:    for e in startV.getEdges()
5:      thisNeighbour = e.getTo()
6:      if !visitedList.contains(thisNeighbour)
7:        visitedList.add(thisNeighbour)
8:        depthFirstTraversal(thisNeighbour, visitedList)
9:      end if
10:   end for
11: end procedure
```

This ADL implementation of a depth first traversal is given as arguments a Vertex object start, and a List of Vertex objects. The traversal is performed via iteration through the list of edges of the argument Vertex and visits its neighbors that have not yet been visited. If a neighbor has not been visited, it is added to the list and the function then calls itself recursively passing that neighbour as an argument.

The time complexity of this algorithm is $O(V + E)$ where V is the number of vertices and E is the number of edges, as each vertex and each edge at is visitied once at most.

The time compliexty increases with the number of vertices and edges in a parabolic fashion. The major component of the time complexity stems from the for-each loop *(for(Edge e : startV.getEdges()))*, that iterates through each edge, which will be done for each vertex.

The space complexity is $O(V + E)$, because two linked lists are utilized. One is used to keep track of the visited vertices, and the other to keep track of the path between the start and the end vertex, (the path). In the worst case, where every vertex is visited and all edges are traversed, both lists will have $V + E$ elements.

The other for loop that iterates through the path list and prints the path also adds to the time complexity, but it will only be run once at the end, and is therefore not considered a high order term, regardless if this is performed in a post/pre order or topological fashion.

The algorithm could alternatively be implemented recursively. In that case the space complexity will be $O(d)$ where d is the depth of the tree. This approach would be faster and use less space than the iterative version, but be limited by the maximum stack size and therfore prone to cause a stack overflow should we not allocate enough memory.

Algorithm 2 Breadth First Traversal

```
1: procedure BREADTHFIRSTTRAVERSE
2:     (start:Vertex, visitedList:List<Vertex>)
3:
4:     visitQueue = new Queue<Vertex>()
5:     visitQueue.enqueue(startV)
6:
7:     while visitQueue.notEmpty()
8:       current = visitQueue.dequeue()
9:       for e in current.getEdges()
10:        thisNeighbour = e.getTo()
11:        if !visitedList.contains(thisNeighbour)
12:          visitedList.push(thisNeighbour)
13:          visitQueue.enqueue(thisNeighbour)
14:        end if
15:      end for
16:    end while
17: end procedure
```

First a queue is instantiated, before adding the desired starting vertex. A while loop then encases the main body of the function, ceasing when the queue structure is empty. In each iteration of the loop, the procedure dequeues the next vertex from the queue, and iterates through its edges. For each neighbor of the current vertex, a check is performed to determine if it has already been visited. If it has not, the neighbour is pushed onto the visitedList and enqueued in the visitQueue. This continues until all vertices in the graph have been visited.

Much like the depth first search, time and space complexity of the breadth first search is determined by the number of vertices and edges in the graph. If the graph is dense (i.e., it has a large number of edges), the time and space complexity will be higher. If the graph is sparse (i.e., it has a small number of edges), the time and space complexity will be lower.

The time complexity of the breadth first search is algorithm is $O(V + E)$, where V is the number of vertices and E the number of edges in the graph. This is because for each vertex, each of its adjacent edges are visited before being added to the queue of vertices to visit. The maximum number of vertices that will be enqueued is V, and for each vertex that gets dequeued, all of its adjacent edges will be evaluated, which is a total of E operations.

The space complexity of this algorithm is $O(V)$, where V is the number of vertices. The maximum number of vertices that will be stored in the priority queue is V, which is the ampsymtotic upper bound of the space complexity.

This algorithm could easily be adapted to return a complete path to a given destination by first passing an additional list of vertices along with a destination vertex as arguments. Within the while loop, before entering the inner for, current could be added to the additional list to represent our path. An equality conditional could then be inserted before the inner for loop to determine whether current is the desired destination, which upon returning true would allow us to leave the function with our completed path.

Algorithm 3 Dijkstra's Algorithm

```
1: procedure DIJKSTRATRAVERSAL
2:     (graph: Graph, startV: Vertex): HashMap
3:
4:     distances <- new HashMap<String, Integer>
5:     previous <- new HashMap<String, Vertex>
6:     queue <- new PriorityQueue
7:     queue.add(start, 0)
8:
9:     for each v in graph
10:      if v not start
11:        in dist table put(v data, infinity)
12:      end if
13:      put in prevTable(v data, new null Vertex
14:    end for
15:    put in distTable(startV data, 0);
16:
17:    while queue.size is not 0
18:      current <- remove last vertex
19:      for each edge of current vertex
20:        alt <- get from distTable (current data + edge weight()
21:        neighbour <- edge destination data
22:        if alt greater than get distTable neighbour
23:          put in distTable(neighbour, alt)
24:          put in prevTable (neighbour, current)
25:          add to queue (edge destination, neighbour distance)
26:        end if
27:      end for
28:    end while
29: end procedure
```

The Dijkstra algorithm is an algorithm for finding the shortest path between two nodes (or vertices) in a graph.

By Utilizing two hash-maps to represent tables, we can keep track of the current shortest distance from the start vertex to each other vertex in the graph, as well as the previous vertex on the shortest path from the start vertex to each vertex in the graph.

First we add the start vertex to a priority queue, which is used to store the vertices that are yet to be processed. The distance from the start vertex to itself is set to 0, while the distances from the start vertex to all others are initialized at a value which represents infinity. The previous vertex for all vertices is initialized with null.

We then enter a loop, removing the top element from the priority queue (the vertex with the smallest distance), each time before processing it. For each vertex, we iterate over its edges, calculating the alternate distance to its neighbours (current distance + the weight edge weight. If this alternate distance is less than the current distance in the distance table for the neighbor, we update the distance and previous vertex for the neighbour in the respective tables, adding the neighbour to the priority queue with the updated distance.

We continue until the queue is empty, at which point we return the tables as an array. The final distance table will contain the shortest distances from the start vertex to all other vertices in the graph. The final previous vertex table will contain the previous vertices on the shortest paths from the start vertex to all other vertices in the graph.

Using this information we can then calculate the shortest distance between any two vertices with said tables. This prevents us from having re-traverse each time every time we wish to determine the shortest path to any given node and allows us to further make use of the resulting data, for visualisation purposes, etc.

The time complexity of this algorithm is $O((V + E)logV)$, here V is the number of vertices, and E is the number of edges.

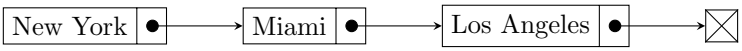
The while loop runs V times and the operations performed inside the loop is of $E + logV$. The while loop runs V times because the queue is polled V times, and each call to this method removes a vertex from the queue.

The inner for-loop runs E times in total, as it evaluates each edge for each vertex, and the $logV$ comes from the fact that when adding or removing elements from a PriorityQueue, the elements are automatically ordered, which takes $logV$ time.

The space complexity of this algorithm is $O(V)$ where V is the number of vertices. Two hashmaps, distTable and prevTable are utilized to store the distance and previous vertex for each vertex, which each occupy $O(V)$ space, and a priority queue which also occupies $O(V)$ space.

3.2 Data Structures

Seeing as we have already constructed our hashmap, we must next construct our basic data types to represent our graph. The first we will make use of is a linked list.



In the diagram above you can see that each node contains a data field with the location name presumably as a string. Additionally, each node also contains a reference or pointer to the next vertex in the list. The last item of the diagram represents a pointer to a null, representing the end of the list.

By utilizing the linked list we can begin to construct the graph data structure. By declaring class Vertex which, contains a data field, as well as a linked list Edges, we can add references to every other vertex which said vertex leads to. By also providing each edge instance with an integer field, weight, we can thereby construct a weighted graph.

If we were to create a bi-directional graph, we could simply add logic which is triggered upon edge addition to a given vertex, which also creates a pointers in the opposite direction, preventing user error and avoiding the accidental creation of a partially non directed graph.

This can be represented as following, where each node is a vertex, and each arrow leading from that vertex is a linked list of edges, each bearing a weight.

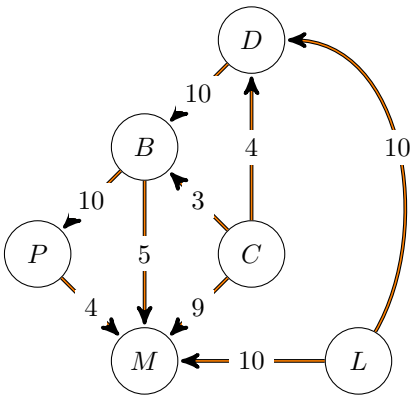


Figure 1: Graph

3.3 Software and its Presentation, including testing

https://github.com/Riverside96/-Project_3
https://youtu.be/J8OyIJxcmO0

3.4 Descriptive Report, including artefacts

To begin with constructing the artefacts necessary we will first start with the linked list.

Nested Class 1 Node

```
public class Node {
       Node(T d){
          data = d;
          next = null;
      }
  T data;
  Node next;

  public void setNextNode(Node node) {
      this.next = node;
  }
  public Node getNextNode() {
      return this.next;
  }
 }
```

We start by first encapsulating a node class via class composition inside of our LinkedList. We pass data as a generic for function compostability s sake. Each node has a reference field to the next node, enabling the linking we seek, as well as a data field.

We can now begin our linked list functions. We will continue to pass generics as arguments to our functions as opposed to specializing to a Vertex data type for reasons aforementioned.

Function 1 Remove If

```
public void removeIf(Predicate<T> predicate) {
    Node current = head;
    while (current != null) {
        if (predicate.test(current.data)) {
            deleteValue(current.data);
        }
        current = current.next;
    }
}
```

This function allows us to remove a node from the linked list if its data field returns true for a given predicate which is passed in as an argument to the function.

By iterating through each node until we reach null, signifying the end of the list, we can then call an additional function, deleteValue, if our predicate is met, for each node.

This allows for finer control should we be tasked with deleting for example, all nodes which have a string data field value beginning with n, or a integer data value less than 10, for e.g.

Function 2 get

```
public T get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException(index+" OOB");
    }
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```

The get function is fairly straight forward and simply allows us to retrieve the data of a node at a specified index. An exception is thrown for the out of bounds case rather than printing an error to the user and/or returning null, as this allows us to separate the error handling code from the normal control flow aiding maintainability.

Function 3 contains

```
public boolean contains(T element) {
    Node current = head;
    while (current != null) {
        if (current.data.equals(element)) {
            return true;
        }
        current = current.next;
    }
    return false;
}
```

The contains function is again self explanatory, and simply iterates though each node until finding the null reference, returning true and terminating the function if the list contains the element passed into the function.

Function 4 peek

```
public T peek(){
    if (head == null){
    System.out.println("List is empty");
    return null;
  } else {
    return head.data;
  }
}
```

The peek function takes no arguments and simply returns us the head, otherwise informing us that the list is empty should this be true.

Function 5 peekEnd

```
public T peekEnd(){
    if (head == null){
      System.out.println("list is empty");
      return null;
  } else {
    Node current = head;
    while (current.next != null) {
      current = current.next;
    }
    return current.data;
  }
}
```

Similarly to the last function, this function instead traverses the list and returns us the tail of the list, again returning null if the list is empty.

Function 6 pop

```
public T pop(){
  if (head == null) {
    System.out.println("List is empty");
    return null;
  }
    T value = head.data;
    head = head.next;
    size--;
    return value;
}
```

The pop function allows us to delete the head of our list, decrementing the list size field in the process. It maintains the additional linkages by setting the class head value to be that of head.next. The function also returns deleted node to allow for further manipulation.

Function 7 popEnd

```
public MyLinkedList<T> popEnd(){
     if (head == null) {
     System.out.println("List is empty");
   }
    else{
      if (head.next == null){
        head = null;
        size--;

    } else {
      Node current = head;
      while (current.next.next != null){
        current = current.next;
      }
      current.next = null;
      size--;
    }
  }
    return this;
 }
```

Similarly to the last function, the popEnd function instead allowing us to delete the tail of the list by first traversing the list, instead returning the list instance itself.

Function 8 push

```
public MyLinkedList<T> push(T data){
    Node newNode = new Node(data);
    if (this.head == null) {
              this.head = newNode;
    } else {
      newNode.next = head;
      head = newNode;
  }
  size++;
  return this;
}
```

Push allows us to add a new head to the existing list with the passed in data argument, setting the previous head to head.next to maintaining our linkage.

Function 9 pushEnd

```
public MyLinkedList<T> pushEnd(T data)
 {
     // Instantiate node with argument
     Node newNode = new Node(data);

     // check for emptyness, set head if true
     if (this.head == null) {
         this.head = newNode;
     }
     else {
         // traverse the list & insert at the null position
         Node last = this.head;
         while (last.next != null) {
             last = last.next;
         }
         last.next = newNode;
     }
     size++;
     return this;
 }
```

Similarly, pushEnd allows us to a new node at the end of the list by first starting at the head and iterating through each node subsequently until null is found. The tail is then pointed to our new node, which then becomes our new tail.

Function 10 deleteAt

```
public  MyLinkedList<T> deleteAt(int index)
{
    // Store head node
    Node current = this.head, previous = null;
    // If index == 0, delete head & confirm deletion.
    if (index == 0 && current != null) {
        this.head = current.next; // Changed head
        size--;
        return this;
    }
    // if the index argument < than size of list
    // if counter = argument, connect previous & next
    // then confirm deletion
    // else traverse & increment tracker
    int tracker = 0;
    while (current != null) {
        if (tracker== index) {
            previous.next = current.next;
            size--;
            break;
        }
        else {
            previous = current;
            current = current.next;
            tracker++;
        }
    }
    if (current == null) {
        System.out.println(index + " position element not found");
    }
    return this;
}
```

The purpose of this function is to delete a node at a given position. First an emptiness check is performed. If the list is not empty we iterate through each node until we find our target node. We then reassign the previous nodes next field to the targets next. If a deletion takes place, the size is decremented.

It would be wise to update this function to utilize the size field of the structure to prevent us from iterating through the list should we already know that the argument index is our of bounds, leading to faster run time, though this is not strictly necessary at this early stage.

Function 11 deleteValue

```
public MyLinkedList<T> deleteValue(T value)
{
    Node current = this.head, previous = null;
    // check for emptyness. If the head contains the value, delete
    if (current != null && current.data == value) {
        this.head = current.next;
        System.out.println(value + "has been deleted");
        size--;
        return this;
    }

    // else iterate until val is found or end(null) is reached
    while (current != null && current.data != value) {
        previous = current;
        current = current.next;
    }
    // if iteration stopped before the end, connect previous & next
    if (current != null) {
        previous.next = current.next;
        System.out.println(value + " has been deleted");
        size--;
    }

    // if end is reached, value does not exist.
    if (current == null) {
        System.out.println("this list does not contain" + value);
    }
    return this;
}
```

This function allows us to delete a value with a specific value should it be present in the list. If a node is found with a data field corresponding to that of the arguments, we update the current.previous next field to point current.next.

This function is also utilized as a helper function by the removeIf function as aforementioned.

Function 12 getSize

```
public T removeHead() {
    Node removedHead = this.head;
    if (removedHead == null) {
      return null;
    }
    this.head = removedHead.getNextNode();
    return removedHead.data;
}
```

This function simply removes the head of the list, updates the new head to be the next in the list, and returns the data field of the deleted node.

Function 13 removeHead

```
public int getSize(){return size;}
```

This functions requires no explanation.

Function 14 displays

```
public void display()
{
    Node current = this.head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println("\n");
}
```

This function iterates through each node of the list and displays its data field. As we are a generic for our data field type, we ought to place guards here to prevent us from attempting something which has no means of doing so. Though, for our use case, this shall suffice for the time being.

Finally, as our class implements an Iterator, we must specify how we should implement these methods via overloading.

Nested Class 2 Iterator

```
  private class MyLinkedListIterator implements Iterator<T> {
    private Node current = head;

    @Override
    public boolean hasNext() {
      return current != null;
    }

    @Override
    public T next() {
      T data = current.data;
      current = current.next;
      return data;
      }
    }
  }
  @Override
    public Iterator<T> iterator() {
      return new MyLinkedListIterator();
    }
```

The purpose of this class is to provide an iterator object that can be used to iterate over the of a list. The use case of these functions will soon be discussed.

The hasNext function returns a boolean indicating whether there are elements remaining in the collection to be iterated over, by checking if null has been reached at which point false is returned.

The next function returns the next node in the list by returning the data stored in the current node and setting current to the next node in the list. An exception is then thrown should next be null.

Finally a constructor is defined to allow us to instantiate an instance of the nested class for usage.

This class allows us to extend the functionality of our node type. As we incorporated generic types into our linked list arguments, we can now create a LinkedList field of type Edge, (soon to discuss) for our Vertex.

Class 2 Vertex

```
public class Vertex {
  private String data;
  private MyLinkedList<Edge> edges;
  }
```

Vertex Function 1 Constructor

```
  public Vertex(String data){
    this.data = data;
    this.edges = new MyLinkedList<Edge>();
  }
```

Vertex Function 2 getData

```
  public String getData(){
  return this.data;
  }
```

Vertex Function 3 getedges

```
public mylinkedlist<edge> getedges(){return this.edges;}
```

Vertex Function 4 addEdge

```
public void addEdge(Vertex to, int weight){
  this.edges.push(new Edge(this, to, weight));
}
```

The addEdge function allows us to create a new edge for a Vertex, by passing in a reference to the calling Vertex, as well as a destination Vertex, which is used to denote the destination. Additionally we pass in the weight of the edge.

Vertex Function 5 deleteEdge

```
public void deleteEdge(Vertex rhsTo){
  this.edges.removeIf(edge -> edge.getTo().equals(rhsTo));
}
```

This calls the aforementioned removeIf function using a reference to the calling Vertex as well as an argument destination Vertex as a predicate, deleting the edge if the edge exists.

Vertex Function 6 print

```
  public void print(boolean showWeight){
    String toPrint = "";
    if(this.edges.getSize() == 0){
      System.out.println(this.data + "-->");
      return;
    }

    for(int i =0; i<this.edges.getSize(); i++){

      if (i == 0){
        toPrint += this.edges.get(i).getFrom().data + "-->";
      }
      toPrint += this.edges.get(i).getTo().data;

      if(showWeight){
        toPrint += " (" + this.edges.get(i).getWeight() + ")";
      }
      if (i != this.edges.getSize() -1){
        toPrint += ", ";
      }
    }
    System.out.println(toPrint);
  }
}
```

This function allows us to display all the edges of a given vertex, additionally including a boolean parameter which can be used to specify whether weighing for each should be shown. This allows for flexibility in the ways in which we display our data and prevents us from unnecessarily overloading the function.

Class 3 Edge

```
public class Edge {

  private Vertex from;
  public Vertex getFrom() {
    return from;
  }

  private Vertex to;
  public Vertex getTo() {
    return to;
  }

  private Integer weight;
  public Integer getWeight() {
    return weight;
  }

  public Edge(Vertex fromVertex, Vertex toVertex, Integer weight){
    this.from = fromVertex;
    this.to = toVertex;
    this.weight = weight;
  }
}
```

This class is fairly self explanatory and simply contains three fields. From contains a starting Vertex, to contains a destination Vertex, and finally weight.

Class 4 Graph

```
public class Graph {

  private MyLinkedList<Vertex> vertices;
  public MyLinkedList<Vertex> getVertices() {return vertices;}

  private boolean isGraphWeighted;
  public boolean isGraphWeighted() {return isGraphWeighted;}

  private boolean isGraphDirectional;
  public boolean isGraphDirectional() {return isGraphDirectional;}

  public Graph(boolean setIsGraphWeighted, boolean setIsGraphDirectional){
    this.vertices = new MyLinkedList<Vertex>();
    this.isGraphWeighted = setIsGraphWeighted;
    this.isGraphDirectional = setIsGraphDirectional;
  }
```

Finally we have the graph class itself. Storing a linked list of vertices aside, the class is fairly unremarkable besides the inclusion of two boolean fields, isGraphWeighted, and isGraphDirectional.

These fields allow us to specify the type of our graph once upon initialization, and the proceeding functions use those values accordingly to spare us either passing around null values, or writing separate implementations of the Graph class for our use case. This extensibility may well prove useful down the line.

```
public Vertex addVertex(String data){
    Vertex newOne = new Vertex(data);
    this.vertices.pushEnd(newOne); // push start or end?
    return newOne;
  }
  public void addEdge(Vertex from, Vertex to, Integer weight){
    if (!this.isGraphWeighted){
      weight = null;
    }
    from.addEdge(to, weight);
    if (!this.isGraphDirectional){
      to.addEdge(from, weight);
    }
  }
  public void deleteEdge(Vertex from, Vertex to){
    from.deleteEdge(to);
    if (!this.isGraphDirectional){
      to.deleteEdge(from);
    }
  }
  public void deleteVertex(Vertex vertex){
    this.vertices.deleteValue(vertex);
  }
  public Vertex getVertexWithVal(String val){
    for(Vertex v : this.vertices){
      if (v.getData() == val) {
        return v;
      }
    }
    return null;
  }
  public void print() {
    for(Vertex each : this.vertices){
      each.print(isGraphWeighted());
    }
  }
```

The usage of these boolean attributes can be seen here, the most noteworthy being addEdge. This function handles setting weight to null for each vertex to ensure consistency. Additionally, if we set isGraphDirectional to false, an edge pointer will be added to the start Vertex to the end Vertex, as well as from the end Vertex, to the start Vertex, denoted by from and to respectively. Further preventing inconsistency. Furthermore, should we need to create a graph in which only some edges are directed, we could overload the addEdge function to also take a boolean argument to determine whether that edge is directional.

3.4.1 Transitioning algorithms to implementation

We can now begin implementing our algorithms by first populating our graph.

```
Graph usa = new Graph(true, true);
Vertex newYork = usa.addVertex("New York");
Vertex chicago = usa.addVertex("Chicago");
Vertex denver = usa.addVertex("Denver");
Vertex dallas = usa.addVertex("Dallas");
Vertex miami = usa.addVertex("Miami");
Vertex sanFran = usa.addVertex("San Fran");
Vertex sanDiego = usa.addVertex("San Diego");
Vertex losA = usa.addVertex("LA");

usa.addEdge(newYork, chicago, 75);
usa.addEdge(newYork, denver, 100);
usa.addEdge(newYork, dallas, 125);
usa.addEdge(newYork, miami, 90);
usa.addEdge(chicago, sanFran, 25);
usa.addEdge(chicago, denver, 20);
usa.addEdge(miami, dallas, 50);
usa.addEdge(dallas, losA, 80);
usa.addEdge(dallas, sanDiego, 90);
usa.addEdge(denver, sanFran, 75);
usa.addEdge(denver, losA, 100);
usa.addEdge(sanDiego, losA, 45);
usa.addEdge(sanFran, losA, 45);
```

We will first start with depth first search, which is simple to implement but gives us no guarantees that any path we find will result in the lowest possible edge weight sum for the completed path, as there is no heuristics involved. For simplicity's sake we will follow our ADl logic and focus on the traversal steps, before adapting the algorithm to return the results that we desire.

Class 5 Depth First Traversal

```
public static void depthFirstTraversal(
Vertex startV, MyLinkedList<Vertex> visitedList){

    for(Edge e : startV.getEdges()){
      Vertex thisNeighbour = e.getTo();

      if (!visitedList.contains(thisNeighbour)){
        visitedList.push(thisNeighbour);
        Traverser.depthFirstTraversal(thisNeighbour, visitedList);
      }
    }
  }
```

To begin, a linked list to store visited vertices, visitedList is passed in rather than declared within the function, to allow us to make use of the call stack recursively.

We start by iterating through each edge of the starting Vertex, storing the vertex as thisNeighbour. If visitedList does not contain thisNeighbour, we add thisNeighbour to the visitedList.

We then recursively call the depthFirstTraversal function with thisNeighbour as the starting vertex and the updated visitedList.

We continue exploring the graph in this manner, pushing our function calls onto the call stack passing the next neighbouring vertex. When the function has finished processing the current vertex and all its neighbors, the function call is popped off the call stack and the program continues with the next function call on the stack until there are no more unvisited vertices. This will lead us to have visited all vertices in the graph which the starting vertex is connected to directly and/or indirectly.
Once the call stack is empty, the function terminates.

Class 6 Depth First Search

```
public static void depthFirstSearch
(Vertex startV, Vertex endV, MyLinkedList<Vertex> visitedList,
MyLinkedList<Vertex> path, int totalWeight){
  path.push(startV);

  if (startV.equals(endV)){

    for(Vertex v : path) {
      if(v != endV){
      System.out.print(v.getData() + " -> ");
      } else {System.out.println(v.getData());}
    }

    System.out.println(
      "This journey has a total weight of: "+totalWeight);

  } else {
      for(Edge e : startV.getEdges()){
          Vertex thisNeighbour = e.getTo();

          if (!visitedList.contains(thisNeighbour)){
            visitedList.push(thisNeighbour);
            Traverser.depthFirstSearch(
              thisNeighbour, endV, visitedList, path,
              totalWeight + e.getWeight());
          }
        }
      }
    //remove current vertex from path before returning
    path.deleteAt(path.getSize() -1);
  }
```

By also passing a desired end Vertex, a weight integer, as well as an additional linked list We can construct a path while accumulating our edge weight sums for said path.

We start by pushing the starting Vertex onto our path, before checking that we have not yet reached the target Vertex. If so we iterate through each Vertex within the path printing its data field using its getData method. Otherwise the function operated much like the traversal, besides the fact we are now passing more parameters to our call stack.

One advantage of using the call stack to implement this algorithm is that it allows the function to be implemented in a fairly simple manner, as the problem lends itself well to a recursive structure, however, if the graph is very large, or the recursion is very deep, the call stack could potentially consume an extreme amount of memory, when the problem does not strictly require memoization. This could lead to less performant code, and eventually a stack overflow.

Recursive algorithms can also be comparatively difficult to debug compared to iterative algorithms as it can be harder to fully understand the state of the program at each point of execution.

Recursive algorithms can also be slower than iterative algorithms because they involve the overhead of function calls and the maintenance of the call stack. Furthermore, while on some occasions a compiler can optimize a recursive algorithm, this is not always the case. Imperative algorithms are typically easier for the compiler to optimize due the imperative nature of assembly. Additionally tail call optimization is difficult for the JVM due to the necessity to always have a stack trace available.

Overall, whether recursion is an appropriate choice for implementing this algorithm will depend on the specific requirements and constraints of the problem at hand.

Class 7 Dijkstra Traversal

```
public static HashMap[] dijkstra(Graph graph, Vertex startV) {

        HashMap<String, Integer> distTable = new HashMap<>();
        HashMap<String, Vertex> prevTable = new HashMap<>();
        PriorityQueue<PQObj> queue = new PriorityQueue<PQObj>();
        queue.add(new PQObj(startV, 0));

        for (Vertex v: graph.getVertices()) {
            if (v != startV) {
                distTable.put(v.getData(), Integer.MAX_VALUE);
            }
            prevTable.put(v.getData(), new Vertex("Null"));
        }

        distTable.put(startV.getData(), 0);

        while (queue.size() != 0) {
            Vertex current = queue.poll().vertex;
            for (Edge e: current.getEdges()) {
                Integer alt = distTable.get(
                  current.getData()) + e.getWeight();
                String neighbour = e.getTo().getData();
                if (alt < distTable.get(neighbour)) {
                    distTable.put(neighbour, alt);
                    prevTable.put(neighbour, current);
                    queue.add(new PQObj(
                      e.getTo(), distTable.get(neighbour)));
                }
            }
        }
        return new HashMap[]{distTable, prevTable};
    }
```

By utilizing Dijkstra's algorithm, we can ensure that the path we have found is indeed the shortest, by accumulating the weights for each possible route, and returning the one which has the smallest value.

We make use of a priority queue for this algorithm for which we have a custom priority queue object which we can pass to our queue structure, extending it's functionality, further enabling function composability.

It also utilize two HashMaps to represent tables used to store the distance from the start vertex to each vertex and the previous vertex in the shortest path from the start vertex to the current vertex.

- We begin by adding the start vertex and a distance of 0 to the priority queue. We then initialize the distances table as well as the previous vertex table for all vertices in the graph, setting the distance for all vertices except the start vertex to the maximum value to represent infinity, and the previous vertex for all vertices with null.
- We then enter a loop that continues. until the priority queue is empty. With each iteration, we remove the vertex with the smallest distance from the start vertex from the queue and examine its neighbours.
- For each neighbouring vertex, we calculate the total weight from the start vertex to the neighbor by adding the weight of the edge connecting the current vertex to the neighbor, to the distance of the current vertex from the start vertex.
- If this distance is less than the current distance recorded in the distance table for the neighbor, the distance and previous vertex for the neighbor are updated in the distance and previous vertex tables, and the neighbor is added to the priority queue.
- Once the loop is finished, the distance table and previous vertex table will contain the shortest distances from the start vertex to all other vertices in the graph and the previous vertices on the shortest paths, respectively. The algorithm returns these tables as an array of HashMap objects.

On the following page we will discuss the usage of a function which we will use to return the shortest path between any given pair of vertices.

Class 8 Dijkstra Traversal

```
public static void shortestPath(
Graph g, Vertex startV, Vertex endV) {
     HashMap[] tables = dijkstra(g, startV);
     HashMap distances = tables[0];
     HashMap previous = tables[1];

     Integer distance =
       (Integer)distances.get(endV.getData());
     System.out.println(
       "Shortest Distance between "
       + startV.getData()
       + " and " + endV.getData());
       System.out.println(distance);

     MyLinkedList<Vertex> path = new MyLinkedList<>();
     Vertex v = endV;

     while (v.getData() != "Null") {
         path.addAtIndex(0, v);
         v = (Vertex) previous.get(v.getData());
     }
     System.out.println("Shortest Path");
     for (Vertex pathVertex: path){
         System.out.println(pathVertex.getData());
     }
 }
```

We pass three arguments to this function, the starting vertex, the end vertex, as well as our graph. We then initialize an array of HashMaps and call our traversal function to populate said array. We then name each of our array instances with respectively for the sake of code clarity, before printing the distance from the starting vertex to the target vertex.

We then initialize a path to store the vertices in the shortest path. We then set a Vertex, v, to be equal to the target vertex. What is a field of a class before entering a while loop that continues until our end Vertex is equal to null. For each iteration of the loop we add the current value of v to the front of the path, and set v to be equal to the previous vertex in the shortest path from the starting vertex to v, as stored in the previous table.

Finally, we iterate through the path list, printing out the data field of each vertex in the path.

3.5 Incorporation of formative feedback

4 Project 4

4.1 Introduction

To begin solving our problem we must first contextualize the true meaning of our problem. The question at hand is truly how can we split a random set of numbers, so that the difference between the sum of the sets is as minimal as feasible, without exhaustively checking every possible configuration.

As we are working with a given unit, bricks, we can disregard the complexity of decimals and negative values, and simply focus on the natural numbers.

4.1.1 Weights and Load Problem

The most optimal solution will differ depending upon the characteristics of the numbers in the set, i.e.

- Are there any patterns within the set? (e.g. all primes)
- How large is there range of the set?.

We can not optimize for every circumstance, but we can generalize.

The problem is expected to become incredibly difficult with large sample sized sets, and we are therefore more likely to become more accepting of a less than optimal solution, that is, a solution which is not the global maximum. Furthermore, without using a brute force approach, (calculate the sum of every combination of numbers), which would prove prohibitively costly, if we do not cease iteration using via evaluation of a fitness value derived from a fitness function, there is a probability that we would continue indefinitely, never reaching a global maximum, expressed as a worst case time complexity of $(O(\infty))$.

Additionally, preprocessing could be prohibitive in larger sets, and if we intend to use a tree like structure, we have to consider the effect of pre-processing the input set on the balance of our structure.

One approach may be to first pre-process our set by adapting divide and conquer strategy, we can then begin implementing our strategy with an already reasonably distributed set. This strategy would involve sorting the numbers in ascending order before continually selecting the central element and placing it in the sub-set with the smallest current sum. This will ensure that the difference between the sums of the two sets is minimized and increases the chance of a fitness which is within an acceptable threshold without the need for additional processing.

It is important to note that this method does not always guarantee that the difference between the sums of the two sets will be minimized, but it is a good starting point and will often produce good results, while also being efficient with regards to time complexity, $(O(n))$.

We may begin by deriving a heuristic front the set. For this, we can first halving the sum of all numbers in the original set. Should this be a whole number, we know that if there exists any configuration of numbers which sums to this target then that would be an optimal splitting of the sets. Should the original set sum be an odd number, we can simply use integer division, wherein the 0.5 decimal value will be discarded, and also consider the target plus one to be valid.

We must decide whether to terminate our computation after a given length of time, after a certain threshold has been met, i.e within 10% of the target, or once we have established we have hit a local maximum. We shall initially aim try to achieve a combination of these three approaches, by running for a set time, and ceasing execution if at any point we believe we have reached a local maximum, at which point we will decide whether to attempt to further process depending on whether we are within a certain threshold of the target.

4.1.2 The Problem Proposed Solution and Data Structure

The initial proposal is as follows;

- Sort starting set by value. Data type must allow for multiples and be ordered.
- Create two subsets of an ordered data structure, again multiples must be allowed. Track their sums with a int field.
- Populate the structures by distributing the original set using a divide and conquer approach.
- Enter a loop which continues until either a global maximum has been found, or our fitness is below a set threshold.
- Begin small change strategy, placing the target floor value of the larger subset.
- If there is no value lower than the target present in the smaller set, randomly mutate by arbitrarily moving a number of values.
- Repeat until; A global maximum is found, a number of iterations has been surpassed, or a fitness value within a given threshold is found.

As our subsets require to both maintain order, as well as be reasonably efficient to search for the target floor upon each iteration of the random mutation, we shall use a Tree Map. This will generally provide us with $O(logn)$ time complexity, due to the nature of binary search. There is a possibility that our time complexity may decay to $O(n)$ if all of the nodes enter one branch of the tree, efficiently forming a linked list. This is known as the chain problem. By distributing our original set from the centre out we can prevent the likelihood of this occurring. While we do run the risk of our tree becoming unbalanced, we are unable to utilize hashing. By storing each node as a key value pair, we can increment the value of our key to allow for repeated integers.

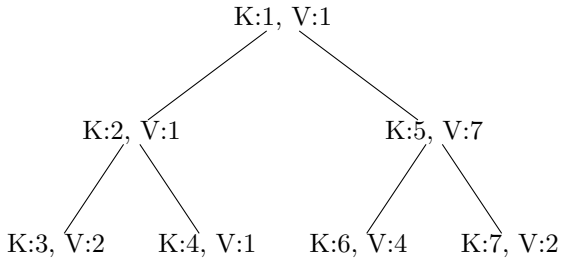


Figure 2: TreeMap

4.2 Heuristic Optimisation

To improve the results of our mutation, We can use the following additional heuristic. On each iteration, we will check if the larger set contains a value equal to the target, This shall be the predicate of our second heuristic. If the predicate is true so, make the swap and exit. This may reduce our effective run time, especially in cases where there is no global maximum, but it does improve our chances of finding it if possible.

4.2.1 Designed Fitness Function

Fitness Function 1 Fitness Function

```
public static double fitnessFunction
(int aSum, int bSum, int target, int brickSum) {
  try {
      if (!isEven(brickSum)) {
          if (aSum == target || aSum == target + 1
          || bSum == target || bSum == target + 1) return 0;
      } else {
          if (aSum == target || bSum == target) return 0;
      }
      return diffAsPosiPerc(smallestDiff(aSum, bSum, target), target);
  } catch (Exception e) {
      System.out.println(
      "An error occurred while calculating the fitness function: "
      + e.getMessage());
      return 0;
  }
}
```

This fitness function utilizes function composition to return the fitness of the function defined as the difference between the target/s and the sub set sum with the smallest difference. The functions it is comprised of are written with referential transparency in mind to aid in testing.

Fitness Function 2 Fitness Evaluation

```
public static boolean fitnessEvaluator(double fitness){
      return fitness < 0.3 ? true : false;
  }
```

4.2.2 Introduction to Hill Climbing

Hill climbing algorithms are a class of optimization algorithms that are used to find the maximum or minimum value of a function. These algorithms are called "hill climbing" because they operate by iteratively making small changes to the input of the function in an effort to "climb" to the highest (or lowest) point on the function's curve.

Hill climbing algorithms are generally simple to implement and can be effective at finding good, local solutions to optimization problems. However, they can sometimes get stuck in local optima, meaning that they may not find the global maximum or minimum of the function. There are many variations of hill climbing algorithms, including simple hill climbing, steepest ascent hill climbing, and simulated annealing. These algorithms can be applied to a wide range of optimization problems, including finding the shortest path between two points, fitting a model to data, and many others.

Overall, hill climbing algorithms are a useful tool for finding good solutions to optimization problems, but they may not always find the absolute best solution.

4.2.3 Small Change Strategy

Small Change 1 Transfer Smallest Value of Largest Subset

```
procedure smallChange(
    lhs: TreeMap[int, int], rhs: TreeMap[int, int],
    lhsSum: int, rhsSum: int, target: int) -> bool:

    lhs increment(rhs first value)
    lhsSum += (rhs first value)
    rhsSum -= (rhs first value)
    rhs decrement(rhs first value)

    if (targetFound)
      return True
    else
      return False

end procedure
```

By continually shifting the smallest value of the larger a subset, a high level of fidelity can be achieved. This is an alternative to continually traversing the tree on every iteration to find a value closest to the target. Used in conjunction with the random mutation, we can very quickly arrive at a close approximation of the global maximum at very little computational cost.

4.3 Incorporation of Formative Feedback

5 Project 5

5.1 Experiment Strategy

To better understand the real run time implications of our implementation we must define metrics in which to test a number of variables against. and determine our control methods.

We aim to analyse the effectiveness of our algorithm as a whole, as well as the bearing of changes that we make. For this reason we will contrast and compare the addition of the random mutation on our algorithm.

We will run each variant of the program 100 times, taking the mean of the final fitness of each set of 10 iterations, before recording our tabulating our results, and subsequently plotting them on line chart for ease of comparison.

It is imperative that we isolate the effect of the independent variable on the dependant variable by holding all other variables constant. Therefore we will ensure that both variants of the program receive the same inputs as arguments

- **Independent Variable** We will contrast the use and non use of random mutation to highlight the concept of local optima.
- **Dependant Variable:** We will take the mean fitness of each 10 iterations measured in % difference from the target.
- **Control:** We will ensure both programs will take the same arguments as input, and both will be ran 100 times and averaged.

5.2 Experiment Results

Figure 3: Sample size of 20

5.3 Discussion

It can be seem from the figure above that the inclusion of random mutation within the algorithm consistently achieved much better results.

Both variants of the algorithm perform fairly adequately considering the worst of 3% within the target for the non-mutated variant.

Depending on requirements, as well as the size of the dataset, it could in some cases be preferable to simply aim to reach a local maxima and cease execution there. As our dataset is relatively small, the somewhat costly tree traversal required to randomly select values is arguably justified.

Of course the brute force exhaustive search not be ideal, even with a data set of this size. There (2^n) different ways to place n numbers into two sets. For our sample size of 20 it would take significantly more than 1048576 operations to calculate every configuration, since each operation would involve at least one comparison and one assignment, and these operations would have to be repeated for each number in the set.

5.4 Largest Dataset Experiment

5.4.1 Dataset

For this experiment, random numbers within the range of the previous and maximum size of the previous dataset. Rather than the 20 that were used previously, we will instead generate 2000.

5.4.2 Results

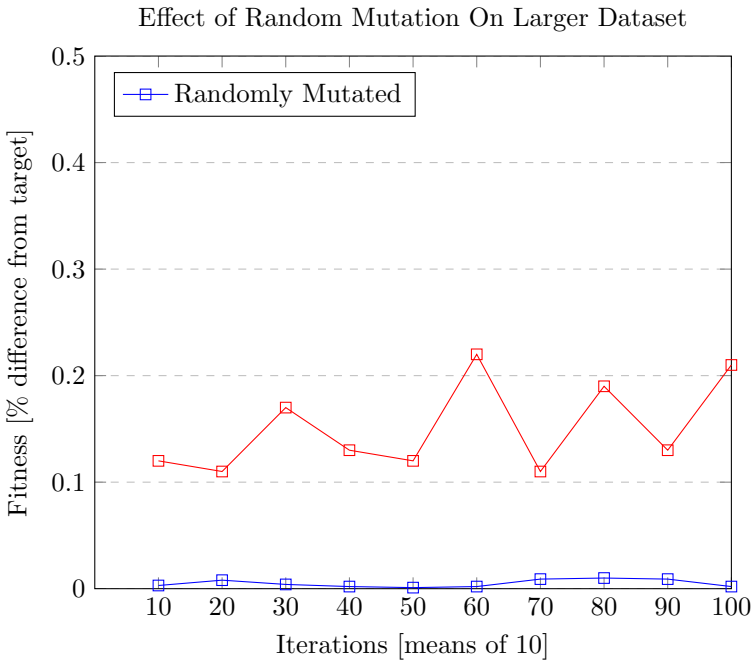


Figure 4: Sample size of 2000

5.4.3 Discussion

It is evident from the results that both the mutated and non mutated variants of the algorithms perform favourably when presented with a larger data set.

This is most likely due to the fact that the algorithm performed a check for a global maxima. The larger dataset results in a larger configurations of numbers, and thus the likelihood of finding at any stage a value which corresponds to the target is significantly increased.

It can be seen in the non mutated variant, that while the inability to deviate from a local maxima does negatively impact the resulting fitness, the previous statement is still true, in that the increased fidelity allows us to incidentally find a value closer to the target.

5.5 Software and its Presentation, including testing

`https://github.com/Riverside96/-Project_4`
`https://youtu.be/CVqa1Hsr4aE`

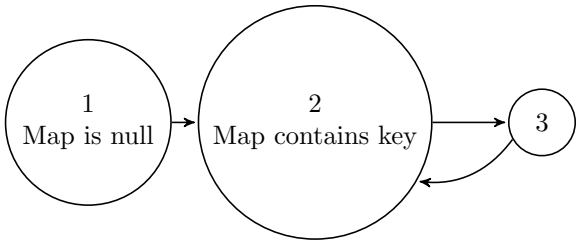
The program consists of 6 distinct steps, namely;

1. **Initialization:** This path includes the creation of the bricks list and the assignment of random values to it.
2. **Distribution:** This path includes the distribution of the bricks into the a and b TreeMaps.
3. **Small Change:** This path includes the transfer of TreeMap key floors.
4. **Mutation:** This path includes the random mutation of elements between the a and b TreeMaps.
5. **Evaluation:** This path includes the evaluation of the fitness of the current configuration of the a and b TreeMaps.
6. **Termination:** This path is taken when the program terminates, either because the target sum has been reached or the maximum number of iterations has been reached.

The cyclomatic complexity of a function takes into account only the independent paths within the function itself, not the paths taken through any functions that it calls.

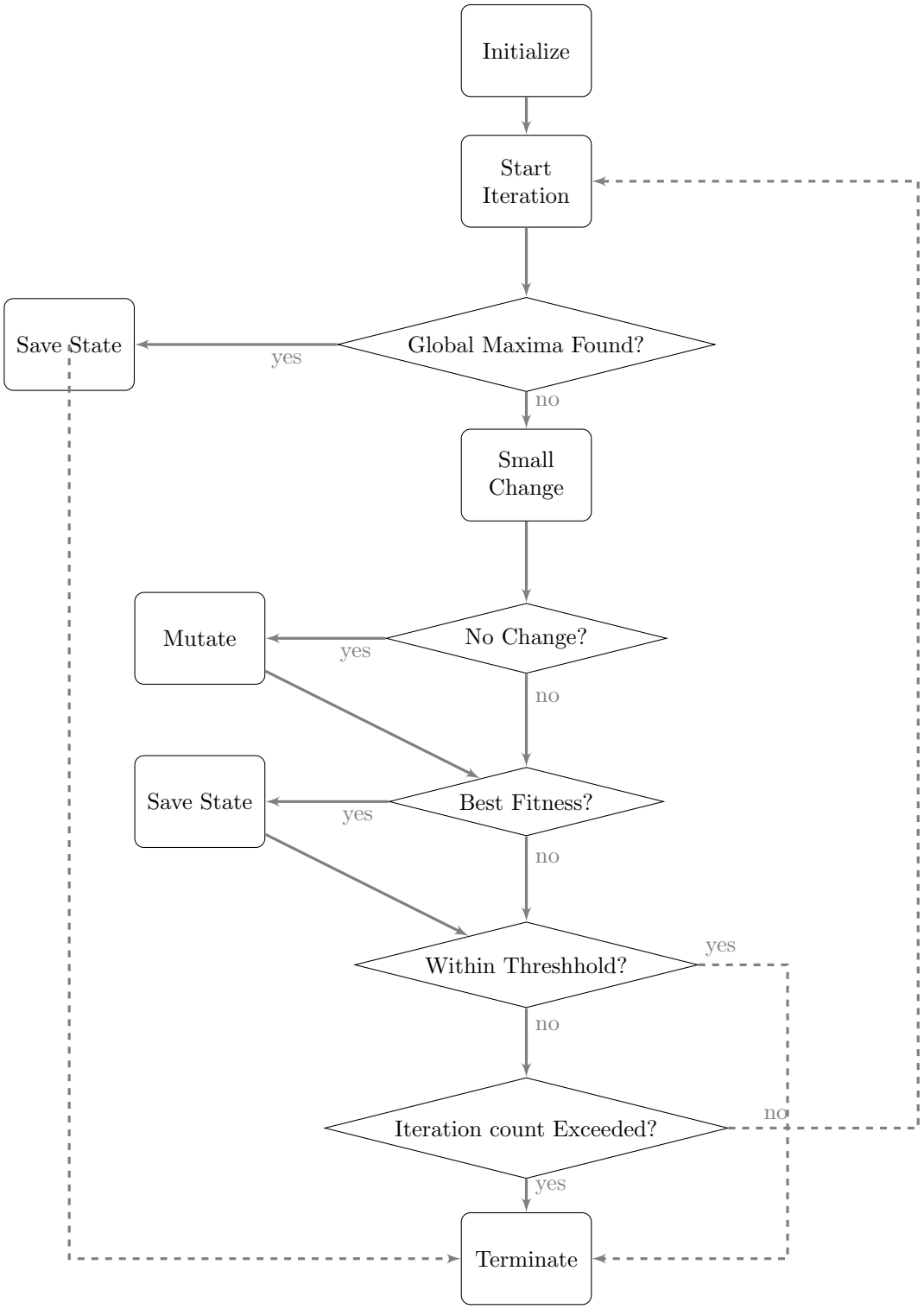
For example, the cyclomatic complexity of the incrementMapVal function is 2. and can be expressed formally as $M = E - N + 2P = 2 - 1 + 2 * 1 = 2$.

However, the isEmpty and contains methods, which are called by incrementMapVal function, have their own cyclomatic complexities based on the independent paths within those functions.



To calculate the overall cyclomatic complexity of the program, we would need to sum the cyclomatic complexities of all the functions in the program, including any functions that they call.

Figure 5: Control Flow Chart for Hill Climb Algorithm



Calculating the cyclomatic complexity for each individual function would be laborious. We can get a general sense of the complexity using the figure of the previous page. The control flow graph contains 17 edges, 12 nodes, and one exit point. We can therfore deduce that the cyclomatic complexity $= M = E - N + 2 * P = 17 - 12 + 2$.

The algorithm begins with initialization;

- Generate set
- Calculate sum
- Initialize Tree Maps
- Populate Tree Maps
- Instantiate array to store states
- Determine target
- Calculate current difference
- Further initialization.

```
public static double[] hillClimbing(){
    Stack<Integer> bricks = getBricks();
    final int brickSum = bricks.stream().reduce(0, Integer::sum);
    TreeMapObj a = new TreeMapObj(new TreeMap<Integer, Integer>());
    TreeMapObj b = new TreeMapObj(new TreeMap<Integer, Integer>());
    populateTrees(a, b, bricks);
    int resultSums[] = new int[2];

    int target = (a.sum + b.sum)/2;
    int diff = (smallestDiff(a.sum, b.sum, target));
    double fitness=0, bestFitness=(Double.MAX_VALUE);

    int iterations = 0, change = 0;
```

We then enter a while loop, and subsequently;

- Traverse tree in search of value which matches target, if so, save state and terminate.
- Begin small change by looking for the greatest key in the larger subset which is lower than the target
- If there is no such value, mutate.
- Evaluate Fitness
- If Fitness is lower than best fitness, save state.
- If fitness is within threshold save state and terminate.
- Repeat while iteration value has not been met.

```
do{
   // indicates perfect solution
   if (globalMaximaFound(a, b, diff, target, brickSum)){
     saveState(a, b, resultSums);
     bestFitness = fitnessFunction(a.sum, b.sum, target, brickSum);
     diff = updatedDiff(a.sum, b.sum, diff);
     break;
   }
   // small change is default
   if(a.sum > b.sum)  change = smallChange(a, b, diff);
   else change = smallChange(b, a, diff);
   //no change indicates local maxima
   if (change == 0) mutate(a, b, bricks);
   // determine fitness, evaluate, save if new best fitness
   fitness = fitnessFunction(a.sum, b.sum, target, brickSum);
   if (betterThanLastFitness(fitness, bestFitness)){
     saveState(a, b, resultSums);
     bestFitness = fitness;
   }
   // break if fitness within defined threshold
   if (fitnessEvaluator(fitness)){
   break;
   }
 diff = updatedDiff(a.sum, b.sum, target);
 iterations++;
 } while(iterations < 100);
 double[] toReturn = new double[5];
 toReturn = saveResults(
   toReturn, resultSums[0], resultSums[1], target, iterations, bestFitness);
 presentResults(resultSums[0], resultSums[1], target, iterations, bestFitness);
 return toReturn;
 }
```

The primary concern for determining the time and space complexity of this function is the number of iterations it takes to reach the solution, and the amount of memory used to store the state of the algorithm. The expected runtime is expected to be highly dependant upon the input data.

The time complexity is likely to be of the order $O(n)$ or $O(n^2)$ depending on the number of iterations, as each iteration of the while loop performs a number of simple operations. One important factor to consider is that the loop does not exit unless a condition is met. the iterations have a limit of n and thus time complexity would be $O(n)$.

In terms of space complexity, the function primarily uses data structures such as a stack, two tree maps, and several integers and doubles to store the state of the algorithm, as well as some other variables used for iteration and storing results. The space complexity would be $O(n)$ for the stack which holds the bricks.

If we assume the input is somewhat random it's likely that the treemap would be close to balanced the complexity would therfore be $O(logn)$ on average for all the operations performed on it.

In terms of time complexity, the time taken to populate the TreeMaps with the bricks would be $O(nlogn)$ due to the combination of the $O(n)$ time complexity of iterating over the stack and the $O(logn)$ time complexity of inserting each element into the TreeMap.

In terms of space complexity, the space used by these TreeMap objects is likely to be of order $O(n)$, where n is the number of unique bricks in the input stack, as each key-value pair in the TreeMap takes up a given amount of memory.

5.6 Incorporation of Formative Feedback