Sorting Algorithms

- Take a number of elements in any order and output them in a logical order.
- This is usually numerical or lexicographic (phonebook style ordering).
- Most output elements in ascending order, but can typically be slightly altered or their output reversed in order to produce an output in descending order.

Bubble Sort

- Makes comparisons and swaps between pairs of elements.
- The largest element in the unsorted part of the input is said to “bubble” to the top of the data with each iteration of the algorithm.
    - Starts at the first element in an array and compares it to the second.
    - If they are in the wrong order, the algorithm swaps the pair.
    - The process is then repeated for every adjacent pair of elements in the array, until the end of the array is reached.
- This is one pass of the algorithm.
- For an array with n elements, the algorithm will perform n passes through the data.
- After n passes, the input is sorted and can be returned.

Implementation of Bubble Sort

```
A = Array of data
    for i = 0 to A.length - 1:
        for j = 0 to A.length - 2:
            if A[j] > A[j+1]:
                swap A[j] and A[j+1]
    return A
```

- Can be modified to improve efficiency.
- A flag recording whether a swap has occurred is introduced.
- If a full pass is made without any swaps, then the algorithm terminates.
- With each pass, one fewer element needs comparing as the n largest elements are in position after the Nth pass.
- Bubble sort is a fairly slow sorting algorithm, with a time complexity of O(n^2).

Insertion Sort

- Places the elements into a sorted sequence.
- In the i-th iteration of the algorithm the first i elements of the array are sorted.
    - Although the i elements are sorted, they are not the i smallest elements in the input.
- Starts at the second element in the input, and compares it to the element to its left.
- When compared, elements are inserted into the correct position in the sorted portion of the input to their left.
- This continues until the last element is inserted into the correct position, resulting in a fully sorted array.
- Has the same time complexity as a bubble sort, O(n^2).

Implementation of Insertion Sort

```
A = Array of data
    for i = 1 to A.length - 1:
        elem = A[i]
        j = i - 1
        while j > 0 and A[j] > elem:
            A[j+1] = A[j]
            j = j - 1
        A[j+1] = elem
```

Merge Sort

- Example of a “divide and conquer” algorithm.

- Formed from two functions. MergeSort and Merge.
  - MergeSort divides its input into two parts and recursively calls MergeSort on each of those two parts until they are of length 1.
  - Merge is then called
  - Merge puts groups of elements back together in a special way, ensuring that the final group produced is sorted.
- The exact implementation of merge isn't required, but knowledge of how it works is.
- A more efficient algorithm than bubble sort and insertion sort, with a worst case time complexity of 0(n logn).

Implementation of Merge Sort

```
A = Array of data
    MergeSort(A)
        if A.length ≤ 1:
            return A
        else:
            mid = A.length / 2
            left = A[0...mid]
            right = A[mid+1...A.length-1]
            leftSort = MergeSort(left)
            rightSort = MergeSort(right)
            return Merge(leftSort, rightSort)
```

Quick Sort

- Works by selecting an element, often the central element (called a pivot), and dividing the input around it.
- Elements smaller than the pivot are placed in a list to the left of the pivot and others are placed in a list to the right.
- This is then repeated recursively on each new list until all elements in the input are old pivots themselves or form a list of length 1.
- Quick sort isn't particularly fast, with time complexity O(n^2).

Searching Algorithms

- Used to find a specified element within a data structure.
- Numerous different algorithms exist, each of which is suited to a particular data structure or format of data.
- Different algorithms are sued depending on each individual scenario.

Binary Search

- Can only be applied on sorted data.
- Works by finding the middle element in a list of data before deciding which side of the data the desired element is to be found in.
- the unwanted half of the data is discarded and the process repeated until either:
    - The desired element is found.
    - Or it is known that the desired element doesn't exist in the data.
- With each iteration, half of the input data is discarded.
- The algorithm is very efficient.
- The time complexity of a binary search is O(log n)

Implementing a Binary Search

A = Array of data

x = Desired element

```
low = 0
high = A.length -1
while low ≤ high:
    mid = (low + high) / 2
    if A[mid] == x:
        return mid
    else if A[mid] > x:
        high = mid - 1
    else:
        low = mid + 1
    endif
endwhile
return "Not found in data"
```

Linear Search

- The most basic searching algorithm.
- Looks at elements one at a time until the desired element is found.
- Doesn't require the data to be sorted.
- A great deal of pot luck, but easy to implement.
    - Sometimes gets lucky and finds the desired element almost immediately.
    - In other situations, the algorithm is incredibly inefficient.
- Time complexity of O(n).

Implementing a Binary Search

A = Array of data

x = Desired element

```
i = 0
while i < A.length:
    if A[i] ==x:
        return i
    else:
        i = i + 1
    endif
endwhile
return “Not found in data”
```

Pathfinding Algorithms

Dijkstra’s Shortest Path Algorithm

- Dijkstra’s algorithm finds the shortest path between two nodes in a weighted graph.
- Graphs are sued as an abstraction for real life scenarios.
- Nodes and edges can represent different entities.
- Implemented using a priority queue, with the smallest distances being stored at the front of the list.

Example

Step 1

- Starting from the root node (A), add the distances to all of the immediately neighbouring nodes (B, C, D) to the priority queue.
- The table below should be used to keep track of visited nodes and distances.
- Begin by filling in the ‘node’ column with the nodes connected to the root node.
- The ‘From’ column should contain the node that you are travelling from.
- The ‘distance’ column contains the distance between these two nodes.
- The ‘total distance’ is the sum of the distances from the root node to that particular node.
- The node which is the shortest distance away from the root node is highlighted and selected for the next stage.

Step 2

- Remove the node the shortest distance away, from the front of the queue.
- Now traverse all of the nodes connected to the removed node C.
- If the total distance passing through the removed node to the neighbouring node is smaller than the distance currently stored with this node, update this value to the smaller distance.
- C has two neighbours: B and E. The shortest total cost of travelling to E so far is 13, as it is less than the infinite value previously allocated to E.
- Travelling from A to C to B adds up to a total cost of 5, while travelling from A to B has a cost of 7. B’s cost value is therefore updated to 5.
- We can ignore nodes we have visited, such as B.
- We can remove routes that are not the shortest way to get to a particular node.

Step 3

- Continue repeating Step 2 until the goal node has been reached.

- Once again, we repeat this same process for node D.

- As we have now visited all of the nodes on the graph, we can confirm by tracing back through the table above that the shortest path is ADE.

A* Algorithm

- TA general path-finding algorithm which is an improvement of Dijkstra’s algorithm and has two cost functions:
  - The actual cost between two nodes - the same cost as is measured in Dijkstra’s algorithm.
  - An approximate cost from node x to the final node, called a heuristic, which aims to make the shortest path finding process more efficient.
- The approximate cost is added onto the actual cost to determine which node is visited next.

Example

Step 1

- The method used here is very similar to the method used in Dijkstra’s algorithm.
- The difference is that the heuristic cost is added onto the actual cost to calculate the total cost.
- Again, the route with the lowest total cost is selected to traverse further.

Step 2

- The node D is then selected.
- Note that the heuristic cost of the previous node is not added on to the new distance.
- As 11 is the shortest total distance, the algorithm terminates.
- The shortest route is found to be ADE, at a cost of 11.

- The heuristics used here allow the shortest path to be found much faster than when using Dijkstra’s algorithm.
- How effective the A* algorithm is depends on the accuracy of the heuristics used.