ENGSCI 233 - Combinatorics
Lecture 2

Colin Simpson

Semester One, 2022

Department of Engineering Science
University of Auckland
(Beamer Theme: Metropolis)

Sorting Algorithms

Introduction to Sorting

There exists **many** different algorithms that can sort a sequence of values. We will specifically cover just two:

- Insertion sort
- Heap sort

There are a few things we need to consider when sorting:

- Ascending or descending order.
- Strings are sorted according to ascii codes.
- If also sorting a corresponding list, we may need to generate a rank table or index table.

Additional Algorithms

Some additional algorithms not covered here are:

- Bubble sort
- Shell sort
- Mergesort
- Quicksort
- Timsort

Why so many different algorithms - isn't there a best one? We will revisit this in the next topic on Performance.

Insertion Sort

If you've ever played cards and sorted a suit in your hand, you probably used an insertion sort. Consider the following example:

```
A = [5, 3, 7, 1, 2]
```

Initially our partial sorted list contains the first element. Starting with the second element, we find the appropriate location for each element and insert it. Quick aside to a visual demonstration.

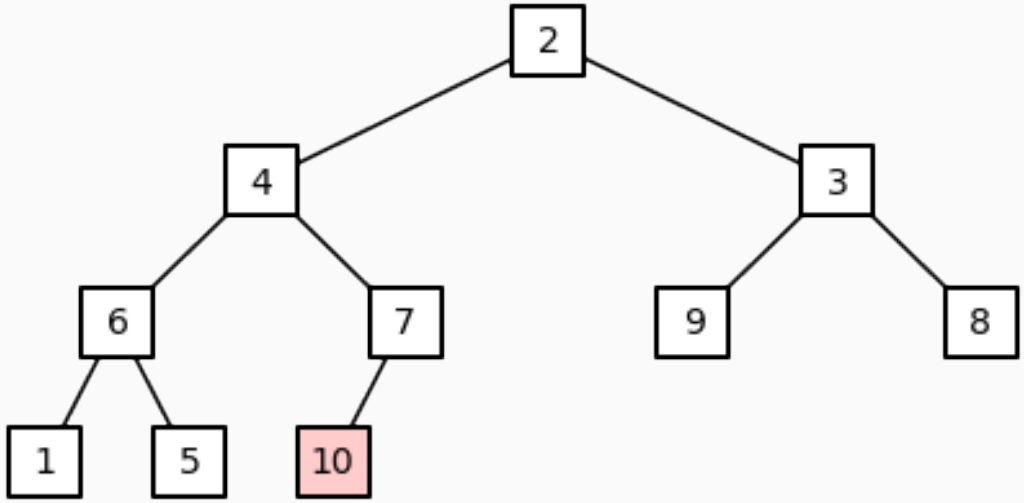
Insertion sort only performs well for small or mostly pre-sorted lists. A more advanced form is the shell sort.

Heap Sort

Heap sort first constructs a binary tree data structure (i.e. parent nodes have at most two children) from the unsorted list.

For example, the following unsorted list gives an initialised *heap*:

```
A = [2,4,3,6,7,9,8,1,5,10]
```



Sorting the Heap

Working backwards from bottom generation, we see if a node has any children. If yes, we perform a demotion operation:

- If parent has largest value, no change required.
- If child has largest value, swap with the parent. Then continue to demote that smaller value down the family line until it becomes the largest value.

For each generation, we ensure that largest value of each parent and its children (if any) is located in the parent position. We then iteratively work backwards through each generation.

Sorting a Corresponding List

Sorting Example

Consider an example of sorting a list of student names based on their Canvas quiz score. The example data are:

```
students = ['Anna', 'Bob', 'Charlie', 'Dave']
scores   = [8, 4, 10, 7]
```

Sorting the quiz scores into ascending order:

```
scores_asc = [4, 7, 8, 10]
```

How do we sort this corresponding list of names?

Index Table

```
students   = ['Anna', 'Bob', 'Charlie', 'Dave']
scores     = [8, 4, 10, 7]
scores_asc = [4, 7, 8, 10]
```

The index table contains the starting index of each sorted item:

```
indx = [1, 3, 0, 2]
```

Using this to sort the corresponding list:

```
students_indx = [None] * len(students)
for i in range(len(students)):
    students_indx[i] = students[indx[i]]
```

Rank Table

```
students   = ['Anna', 'Bob', 'Charlie', 'Dave']
scores     = [8, 4, 10, 7]
scores_asc = [4, 7, 8, 10]
```

The rank table contains the ending index of each unsorted item:

```
rank = [2, 0, 3, 1]
```

Using this to sort the corresponding list:

```
students_rank = [None] * len(students)
for i in range(len(students)):
    students_rank[rank[i]] = students[i]
```

Dijkstra's Shortest Path Algorithm

Shortest Path Algorithm

We will cover Dijkstra's algorithm for finding the shortest path from a source node to a destination node within a network.

If we assume that nodes are connected by single-directed arcs, we want the path with the lowest possible accumulated arc weight.

The algorithm can determine both the accumulated arc weight, as well as the path taken. It is a form of *greedy algorithm* i.e. one that selects the best option currently available.

Unvisited and Visited Sets

We will define two sets, which collectively contain all nodes in the network:

- Visited set
- Unvisited set.

A node can belong to only one set at a given time, but this can change throughout the steps of the algorithm.

Initially, all nodes are members of the unvisited set. Each iteration of the algorithm; one node will be transferred into the visited set.

Node Distance and Predecessor

The algorithm requires two pieces of information for each node:

- Smallest accumulated arc weight from source node to reach this node. Referred to as distance.
- Predecessor i.e. from which node did we leave to arrive at this node?

A single attribute, `node.value`, can store both pieces of information using a list. For example:

```
ndA.value = [0, None]
ndB.value = [2, ndA]
ndC.value = [4, ndA]
```

Algorithm Initialisation

Initial properties of each node in the algorithm:

- Source node has a distance of zero. All other nodes have a **very large** distance, `float("Inf")`
- All nodes have a predecessor of `None`.
- All nodes are members of the unvisited set.

These properties will be updated during the algorithm iterations.

Algorithm Iteration

Each iteration of the algorithm involves:

1. From the unvisited set, move the node with the smallest distance into the visited set.
2. For the Step 1 node, identify all nodes in the unvisited set that it connects outwardly to.
3. For each Step 2 node, calculate a proposed new distance to it i.e. add arc weight to Step 1 node distance. If this is smaller than the current distance, update this nodes' distance and set this nodes' predecessor to the Step 1 node.

Iterate until the destination node becomes a member of the visited set. This can be checked after Step 1 of the iterative procedure.

Algorithm Output

Once finished, we can follow the chain of predecessor nodes *in reverse* to re-create the shortest path through the network.

Note a couple of useful properties of this algorithm:

- If no stopping condition is specified, all nodes will be solved and we can find the shortest path to every possible destination node in the network from a given source node.
- With non-negative arc weights, this algorithm is guaranteed to produce the optimal solution.