1 Introduction to Greedy Algorithms

Greedy alogirthms are a class of algorithms where we build a solution one step at a time making locally optimal choices at each stage in the hope of finding a global optimum solution. So no back tracking is done.

```
def GreedyAlgorithm(input):
    initalise result                                              # initalisation
    determine order in which to consider input
    for each element i of the input (in abover order) do          # make greedy choices
            if element i improves result then
            update result with element i
    return result
```

1.1 The Fractional Knapsack Problem

Given a set S of n items, with each item i having b_i (a positive benefit) and w_i (a positive weight), and a knapsack of capacity W, find a subset of S that maximises the total benefit of the items in the subset. Let x_i denote the amount we take of item i.

Required to maximise: $\sum_{i \in S} b_i(x_i/w_i)$

```
def FractionalKnapsack(b, w, W):
    x ← array of size |b| of zeros                                # initalisation
    current ← 0
    for i in descending b[i]/w[i] order do                        # make greedy choices
        x[i] ← min(w[i], W - current)                             # Don't take more than we can
        current ← current + x[i]
    return x
```

Require O(n log n) time to sort the items and then O(n) time to process them in the for-loop.

1.2 Task scheduling

Given a set of n lectures, each with a start time s_i and finish time f_i, find the minimum number of classrooms required to schedule all lectures so that no two occur at the same time in the same room.

The depth of a set of open intervals is the maximum number that contain any given time. Therefore, the number of classrooms needed $\geq$ depth of the set of intervals.

```
def TaskScheduling(S):                                          # O(n log n)
    sort S by increasing startng time order                     # initalisation
    rooms ← 0
    for each lecture i in S do                                  # make greedy choices
        if lecture i is compatible with some classroom k then
            schedule lecture i in classroom i ≤ k ≤ d
        else
            allocate a new classroom rooms + 1
            schedule lecture i in classroom rooms + 1
              rooms ← rooms + 1
    return rooms
```

RTP: Greedy algorithm is optimal

- rooms = number of classrooms that the greedy algorithm allocates.
- Classroom rooms is opened because we needed to schedule a job, say i, that is incompatible with all rooms-1 other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_i.
- Thus, we have rooms lectures overlapping at time s_i.
- All schedules use $\geq$ rooms classrooms.

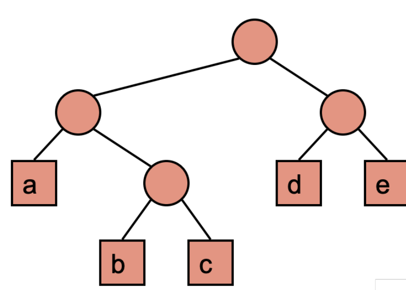
2 Text Compression

Given a string X, efficiently encode X into a smaller string Y that saves memory and/or bandwith.

2.1 Encoding Trees

A binary code is a mapping of each character of an alphabet to a binary code-word. A prefix code is a code such that no code-word is the prefix of another code-word. An encoding tree represents a prefix code, whereby each external node stores a character and the code word of a character is the path from the root to the external node.

00	010	011	10	11
a	b	c	d	e



2.2 Huffman encoding - Encoding Tree Optimization

- Let C be the set of characters in X.
- Compute frequency f(c) for each character c in C.
- Encode high-frequency characters with short codes and low-frequency characters with long codes.
- No code word is a prefix of another code word.
- Use an optimal encoding tree to determine the code words.

Given a string X, Huffman's algorithm constructs a prefix code that minimizes the size of the encoding of X. It runs in time O(n + d log d), where n is the size of X and d is the number of distinct characters of X. The algorithm builds the encoding tree from the bottom up, merging trees as it goes along, using a priority queue to guide the process.

$$\textbf{Minimises: } \sum_{c \in C} f(c) \times \text{depth of c in tree}$$

```
def Huffman(C, f):                                        # O(|C|log|C|)
    Q ← priority queue of C                               # initalisation
    for c in C do
        T ← single-node binary tree storing c
        insert T into Q with key f(c)
    while Q.size() > 1 do                                 # make greedy choices
        f1, T1 ← Q.removeMin()
        f2, T2 ← Q.removeMin()
        T ← new binary tree with T1 and T2 as left and right subtrees.
         f ← f1 + f2
         insert T into Q with key f
     return Q.removeMin()                                 # returns encoding tree
```