Week9 — Greedy method

Knapsack Problem

- Given n objects and a "knapsack"
- Item i weights $w_i > 0$ kilograms and has value $b_i > 0$
- Each items will be have two option to choice: *take* or *not take*
- Knapsack has capacity of W kilograms
- **Goal**: fill knapsack so as to maximize total value

"Fractional" Knapsack Problem

- We can take partial amount from item i , denote as x_i, $0 \leq x_i \leq w_i$
- Choose items with maximum total benefit of weight at most W

> [benefit/weight]: Select items with highest benefit to weight ratio.

```
def fractional_knapsack(b, items, W):
  x <- empty list
  curr <- 0
  i <- 0
  preprocess items and sorted in discending order of [benefit/weight]

  while curr < W do
    weight_can_take <- min(weight of items[i], W - curr))
    x.append( (items[i], weight_can_take))
    curr <- curr + weight_can_take
    i <- i+1
  return x
```

Time Complexity: $O(nlogn)$ time to sort items and $O(n)$ time to process them in the loop

---

How to solve actual Knapsack problem?

> Dynamic Programming (DP) — comming soon
>
> - Using Buttom-up recurrence

Task Scheduling

- Given set S of n lectures
- Lecture i starts at s_i and finishes at f_i
- **Goal:** find minimum number of classrooms to schedules so that no two occur at the same time in same room

> **Definition**: The depth of a set of open intervals is the maximum number that contain any given time.
>
> **Observation**: Number of classrooms needed $>=$ depth

Greedy Attempt: Sorting each lectures in increasing order or start time

- Question: Is that optimal? Is there any counter-example?

Implementation: $O(nlogn)$

More Problem: Weighted Interval Scheduling

- Greedy algorithm works if all weights are 1.
- Job j starts at s_j, finishes at f_j, and has weight v_j.
- Two jobs compatible if they don't overlap.
- $Goal$: find maximum weight subset of mutually compatible jobs.

> **Observation.** Greedy algorithm can fail if arbitrary weights are allowed.

How to solve the Problem?? 🤔🤔 — Comming in following weeks

Text Compression

- Given a string X
- Goal: efficiently encode X into a smaller string Y
  - (saves memory and/or bandwidth)

Huffman encoding:

- 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 code words
- No code word is a prefix for another code
- Use an optimal encoding tree to determine the code words

Endcoding Tree:

- A $code$ is a mapping of each character of an alphabet to a binary code-word

- A $prefix\ code$ is a binary code such that no code-word is the prefix of another code-word
- An $encoding\ tree$ represents a prefix code
  - Each external node stores a character
  - The "code word" of a character is given by the path from the root to the external node storing the character

Prolem: Given a text string X, we want to find a prefix code for the characters of X that yields a small encoding for X.

- **Frequent characters should have short code-words**
- **Rare characters should have long code-words**

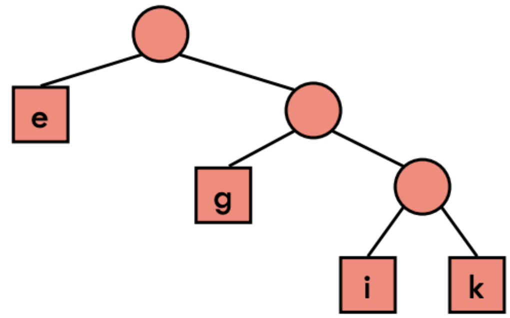
```
def huffman(C, f):
  # C, distinct characters
  # f, frequency of each distinct character
  #initialize the priority queue
  Q <- empty priority queue
  for c in C do
    T <- single-node binary tree storing c
    Q.insert(f[c], T);

  # merge trees while at least two trees
  while Q.size() > 1 do
    f1, T1 ← Q.remove_min_item()
    f2, T2 ← Q.remove_min_item()
    T ← new binary tree with T1/T2 as left/right subtrees
    f ← f1 + f2
    Q.insert(f, T)

  f, T ← Q.remove_min_item()
  return T
```

$TimeComplexity:$

- Dominated by the priority queue ops, which using heap takes $O(|C|log|C|)$
- Total upper bound running time: $O(n + dlogd)$, where n is the size of X and d is the number of distonct charcters of X.

Obs: In an optimal tree encoding T for any a and b in C, if $depth_T(a) < depth_T(b)$ then $f(a) \geq f(b)$.

$$\sum_{c \text{ in } C} f(c) * depthT(c)$$



For example, if f(e) < f(g) then swapping them leads to shorter encoding