Week 4, Lecture 1

Algorithm Analysis and Design

Pratyay Suvarnapathaki, 2020111016

Activity Selection Problem

Given a set of activities $S = \{a_1, a_2, \cdots, a_n\}$, such that a_i needs some resource during the period $[s_i, f_i)$ i.e. [start_time, finish_time), Select the largest set of non-overlapping, i.e. **mutually compatible** activities.

The Greedy Solution

Applying the basic principles of greedy design:

- **The greedy choice property** Can we know for sure what a good first-step towards the solution is? i.e. Can we ascertain an activity that will *always* be present in an optimal solution?
- **The optimal substructure property** Can we now re-state the remainder of the problem post the first step? i.e. After ascertaining the requisite always-present activity, can we find the rest of the activities compatible with it?

Step 1

We can say for sure that the activity with the *least* ***finish-time*** will **always** be a part of an optimal solution. Proof: Let A be the largest set of mutually compatible activities from S. Let a_0 be the activity with the earliest `finish_time`. Sort A in ascending order of `finish_times`. Now, if a_0 has an earlier `finish_time` than the first element of A, say a_k, then insert a_0 in place of a_k. The resulting set has the same number of activities as earlier (i.e. it is still an optimal solution), but now it also contains a_0. Hence, proved.

Step 2

Pseudocode for finding and adding a_0, and then adding the rest of the activities compatible with a_0

```
# s,f are arrays containing the start_time and finish_time of activities, such that f is sor
# The activity a_i is assumed to be denoted by the index i.
def activity_selector(s,f):
A=[0] # we are certain the 0th activity is present in an optimal solution
i=0
for m in range (1,n):
if s[m]>=f[i]:
A.append(a[m])
i=m
return A
```

An example:

i	0	1	2	3	4	5	6	7	8
s	1	2	4	1	5	8	9	11	13
f	3	5	7	8	9	10	11	14	16

Here, the largest mutually compatible sets are: $\{a_0, a_2, a_5, a_7\}$ and $\{a_1, a_4, a_6, a_8\}$

Huffman Coding

The central idea here is to come up with a scheme for lossless compression of a long string, based on a pre-defined alphabet. Naturally, the optimal solution here would be a scheme that is the most economical. A first-cut idea would be to map letters in the string to definite binary sequences. Eg. if the alphabet is $\{A, B, C, D\}$ then we can have an encoding that maps $A \to 00, B \to 01, \dots$

However, we can definitely do better than that, using **variable-length encoding.** Herein, the letter of the alphabet that appears most frequently in the string is assigned less bytes, and vice-versa. This is accomplished using a prefix-free encoding (no code-word can be a prefix of another code-word, as that may lead to ambiguity => inaccurate decoding) that has the form of a full binary tree.

Eg. if the order of frequency is A>D>C>B, we can map $A \to 0, B \to 100, C \to 101$ and $D \to 11$

With this idea we can pose the problem in a more exact way: 'Given the frequencies $f_1, \cdots, f_n$ of n symbols (the 'alphabet'), a tree whose leaves correspond to a symbol and which minimizes the length of the encoding, is to be constructed. Cost of tree $= \Sigma_{i=1}^{n} f_i *$depth of i^{th} symbol in tree (as the number of bits required for a letter is equal to its depth)

The Greedy Solution

Applying the basic principles of greedy design:

- **The greedy choice property** Can we know for sure what a good first-step towards the solution is? Well yes, naturally, in the optimal tree, the two symbols with the smallest frequencies must be at the bottommost depth.
- **The optimal substructure property** Can we now re-state the remainder of the problem post the first step?

For this, we can now define the cost of every *internal* node (although we have been provided with just the frequencies of leaves), as the sum of frequencies of its descendant leaves (as one bit of 'cost' is generated on one traversal of an internal node) Thus the total cost now is equal to the **sum of all non-root nodes** To summarise the greedy approach: Take symbols with least frequencies, say f_1 and f_2. Make them the child of a node. This new node has frequency $f_1 + f_2$. Now, the cost of the tree is $f_1 + f_2$ plus the cost of the subtree with $n - 1$ leaves with frequencies $f_1 + f_2, f_3, \cdots, f_n$.

Pseudocode:

```
# H is a priority queue with priority decided by frequencies 'fi'
# let the number of frequencies = number of leaves in the resultant tree be n
for i in range (n):
 insert(H,i)
for k from (n + 1) to (2n  1):
i = deletemin(H), j = deletemin(H)
create a node numbered k with children i, j
f[k] = f[i] + f[j]
insert(H, k)
```

The above algorithm accepts the array of n frequencies, and outputs the resultant Huffman encoding tree. It takes $O(nlogn)$ time if the priority queue `H` is implemented using a binary heap.

Entropy in Huffman Coding

The randomness of a given probability distribution is correlated with its compressibility in the following manner (intuitively): more compressible $\equiv$ less random $\equiv$ more predictable Considering a distribution of m outcomes picked from a sample $p_1, \cdots, p_n$, then the number of bits required to encode the sequence is $\Sigma_{i=1}^{n} m p_i log(1/p_i)$, and the average number of bits to encode one draw from the distribution is $\Sigma_{i=1}^{n} p_i log(1/p_i)$
