Homework-4

Submitted by
→ Indukuri Vijay Anand Varma (VXI210000)
→ Ananda kumar, karthik Ragunath (KXA200005)

1) → We should choose jewels such that size will be less than robber bag size (b).

Since we can take any fraction of any jewel, we can write a greedy algorithm using $v[i]/s[i]$
$s[1...n]$ is size array
$v[1...n]$ is values array.
→ Idea is to calculate the ratio of $v[i]/s[i]$ for each item and sort them in descending order.

Algorithm:
→ For each item, calculate Value/Size ratio.
→ Arrange all items in non-increasing order of their ratio.
→ Start putting items into knapsack beginning from the item with highest ratio.
→ Insertion happens untill sum of sizes reaches b.
→ If sum becomes more than b, then instead of adding whole jewel, we add fraction of last jewel.
value will increase by $\text{value}_i \times \frac{(\text{remaining size})}{\text{size of this jewel}}$

Complexity:
After sorting, algorithm will get completed in linear time.
But for sorting, ~~we~~ it takes $O(n \log n)$ time.
So, complexity will be $O(n \log n)$

```
procedure jewelselect (V[], S[], b):
    for i=1 to n:
        P[i]= V[i]/s[i]
    sort_array_by_P_value (P)
    size=0, total = 0
    for i= 1 to n:
        if size + s[i] ≤ b then
            total = v[i]+ total
            size = s[i] + size
        else
            total += [(b-size)/s[i]] x v[i]
            size = b
            break
    return total
```

1b) let's say our algorithm gives a total value of v, which is optimal. we will prove that any other arbitrary solution will have value less than v. let v' be the value of any arbitrary solution. We need to prove that $v' \leq v$, which proves that our greedy solution is optimal.

→ $\sum x'_i s_i \leq b$

we know that $\sum x_i s_i = b$ (since v is optimal)

$\Rightarrow \sum (x_i - x'_i) s_i \geq 0$

→ let's say k be the least index with $x_k < 1$
if $i < k$ then $x_i = 1$ and if $i > k$ then $x_i = 0$

→ Since we first sort objects in non-increasing ~~nondecreasing~~ order of ratio v_i/s_i, we can say that
for $i > k$, $v_i/s_i < v_k/s_k$ —— ①

$\Rightarrow$ for $i > k$, $x_i = 0$ and thus $x_i - x'_i \leq 0$. —— ②

from ① & ② we have

$$\left(\frac{v_i}{s_i} - \frac{v_k}{s_k}\right)(x_i - x'_i) \geq 0$$

$$\Rightarrow (x_i - x'_i)\left(\frac{v_i}{s_i}\right) \geq (x_i - x'_i)\left(\frac{v_k}{s_k}\right) \text{ —— ③}$$

Thus difference in the profits is

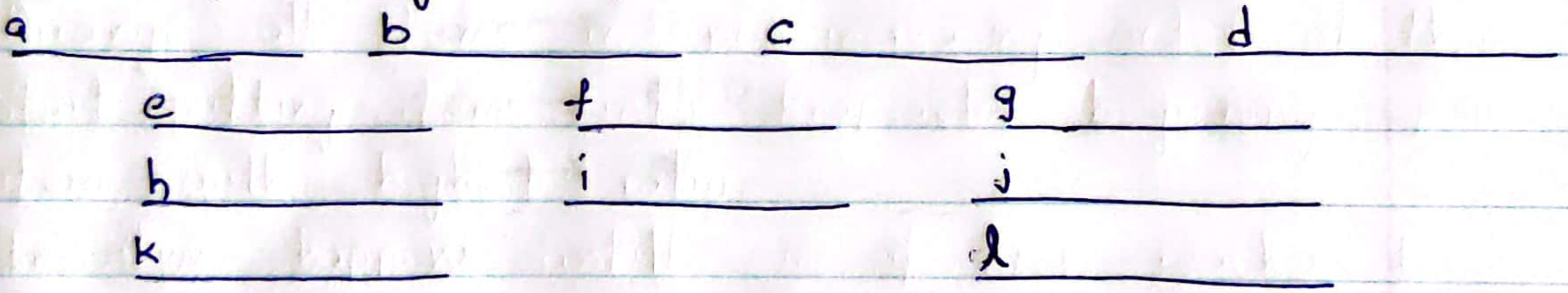
$$v - v' = \sum_i (x_i - x'_i) v_i \geq 0 \quad \text{(from equation ③)}$$

$$\Rightarrow v \geq v' \text{ —— ④}$$

from 4, we can say that v is maximum possible optimal profit.
Thus, our greedy algorith is correct and gives optimal solution.

Scanned with CamScanner

1c) To prove a greedy strategy won't work, we can think of a counter example for which strategy doesn't work.
Consider following example:

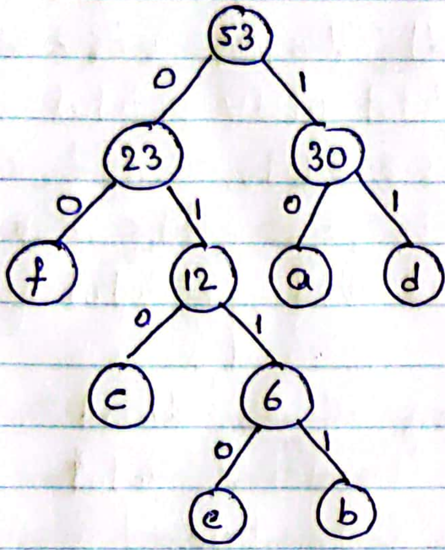


Optimal solution in this example will be {a, b, c, d}
By using given greedy ~~so~~ algorithm, we first choose f. Then we take b, c, i out of picture since they are overlapping with f. Next, lets choose "a" and remove e, h, k since they are overlapping with a. Then our greedy algorithm will choose "d" and exits. So, our greedy algorithm picks {f, a, d} which gives length of 3. But optimal solution is 4.
Thus, this greedy strategy is not optimal.

1d) Huffman coding:

characters: a b c d e f

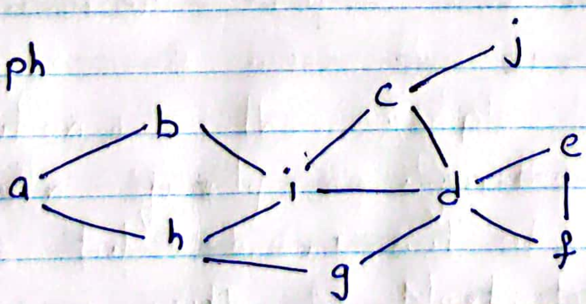
frequencies: 13 4 6 17 2 11



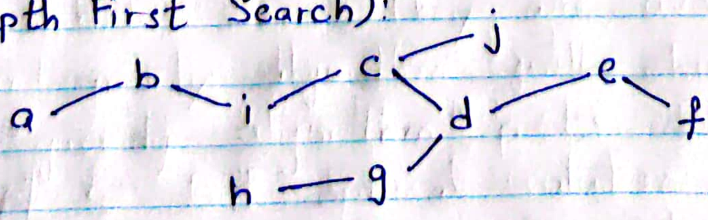
Binary Encoding:

a → 10
b → 0111
c → 010
d → 11
e → 0110
f → 00

2 a) Given graph

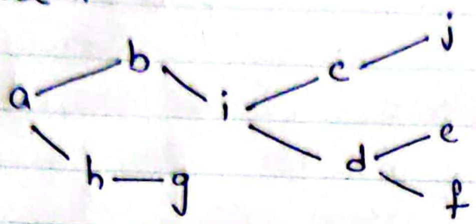


DFS (Depth First Search):



Order: a, b, i, c, d, e, f, g, h, j

BFS Tree :



Order : a, b, h, i, g, c, d, j, e, f

2b) 1 → g, a, b, h, c, d, f, e
2 → h, a, b, g, f, c, d, e

3a) Given a directed acyclic graph. Path is uniquely identified by the subset of vertices.
Given a graph consisting of n vertices (including s and t)
Path from s to t might have 0, 1, 2 ... n-2 vertices.
And each vertex has 2 options : present/not present in path.
Thus, total number of paths will be $2^{(n-2)}$.

3b) Given a directed acyclic graph with n vertices and m edges Algorithm is to ~~vis~~ start from s and visit all ~~unvisited edges and increase~~ adjacent nodes and increase count when an unvisited node is found.

We can solve this code recursively.
let's say function name be pathcount (s, t)
Base case will be when ~~a~~ path reaches target, i.e. when s = t, we add one to our counter.
Otherwise, we visit all adjacent edges going out and call pathcount on next vertex.
We store this count in an array, so that we don't have to calculate pathcount for nodes which are already visited.

Scanned with CamScanner

```
procedure pathcount(s,t):
    if s=t:
        return 1
    if count[s] is undefined:
        count[s] = 0
        for each edge s→v:
            count[s] += pathcount(s,t)
    return count[s]
```

count[ ] is kind of a dynamic array which stores pathcount of each vertex.

Running time:
In worst case all vertices are visited once and all edges leading from s→t are processed.
Since we visit each vertex and each edge, running time will be $O(n+m)$

4) Given a directed graph G, we need to find all vertices that can be reached through patriotic walks from a given vertex v.
The idea here is to remove all non-patriotic paths from G. We create a new graph G' from G, which contains only patriotic paths.
Hence we can find all vertices ~~from~~ by doing DFS on G' ~~from~~ v.

Constructing G' with only patriotic walks:

```
procedure   construct(v, G, count):
      if v is visited:
            return
      update v to visited
      for u in adjacent(v):
            if color of (v→u) is color[count]:
                  add (v→u) to G'
                  ~~pro construct(~~
                  construct(u, G, (count+1) % 3)      (# Recursive call)
```

Here color[] is an array of colors.
color = [red, white, blue]
We set count to zero initially because path starts with red and color[0] = red.
Function call to construct G':
construct (v, G, 0)

Now, G' will only contain those walks which has red → white → blue ~~edges~~ edges.

So, After getting G' we can directly do DFS from v and add all vertices that come after a blue edge to our result.

Time complexity:
Since constructing G' from G is a simple recursive call that goes through all edges and vertices, this will be linear.
And then getting required nodes from G' is simple DFS which can be done in linear time.
Thus, this algorithm runs in linear time. i.e. $O(V+E)$

5) a) It is mentioned to verify if it is possible to legally drive from any intersection to any other intersection.

This is same as checking if whole city is a strongly connected component of a graph in which directed edges are one-way streets.

After converting all directed edges to one-way streets, if we get more than one strongly connected components, then mayor's claim is wrong. If we get one strongly connected component, then mayor's claim is not wrong.

We can compute the number of strongly connected components in linear time using the algorithm discussed in class.

We can imply that the last vertex in post order of Rev(G) is in a source component of Rev(G), hence a sink component of G.

```
CountAndLabel(G):
  count = 0
  for all v in reverse post
    order of Rev(G) do
    if v unmarked then
      count += 1
      Label(v, count)

Label(v, count)
  mark v
  v.component = count
  for all v→w do
    if w unmarked then
      Label(w, count)
```

```
Main(G):
  if CountAndLabel(G) = 1 :
    # Mayor's claim is true
  else:
    # Mayor's claim is false
```

Time complexity:

We can compute strong components in $O(n+m)$ time by computing post order of Rev(G) and then repeatedly extract strong components by looping through vertices in reverse post order.

5b) This can be done by constructing a ~~strongly~~ strong component graph SC(G). This is done by collapsing each strong component to a vertex.

→ For each strong component, make a vertex, and add edge from S_1 to S_2 iff there is $u \in S_1$ and $v \in S_2$ s.t. $u \to v \in G$

Observe that an intersection x is good if and only if the vertex v_x corresponds to a sink vertex in the strong component graph SC(G).
Thus, our algorithm should compute the total number of vertices present in all sink vertices of strong component graph SC(G).
If more than 95% of verteces are present in sink components, then mayor's claim will be valid.

```
procedure goodIntersections(G):
    count = 0        # Initializing counter.
    Construct strong component graph of G.
  # let strong component graph be denoted by SC(G)
    Mark all sink vertices of SC(G).
    create array connected[] from SC(G).
  # connected[v] gives component containing vertex v.
    for each vertev v ∈ V:   (# Iterate through all vertices)
        if connected[v] is marked:
            count += 1
    return count.
```

→ We are basically marking all sink component vertices first and then iterating through all the vertices and incrementing counter by one whenever a marked vertex is encountered.

```
procedure Main(G):
  if (goodIntersections(G) > 0.95(#V)):
      # mayor's claim is true
  else:
      # mayor's claim is false.
```

Time complexity:

~~Con~~ Building a strong component graph will be linear. Even marking of vertices and for-loop takes linear time. So, we can verify mayor's claim in linear time.