Indian Institute of Technology Delhi

COL351 Analysis and Design of Algorithms: Assignment 1

Ishita Chaudhary: 2019CS10360

Mohammed Jawahar Nausheen: 2019CS10371

Contents

Minimum Spanning Tree

1.1 An Edge-Weighted Graph has a Unique Minimum Spanning Tree

Given G is an edge-weighted graph with distinct edge weights. To prove that G has a unique MST, it shall be shown that any MST of G is same as the MST obtained using the Kruskal's algorithm.

Claim 1: If T is a spanning tree and e' $\notin$ T, $T \cup \{e'\} \backslash \{e\}$ is also a spanning tree, where e $\in$ T is a part of the cycle created when e' is added to T.
Proof: $T \cup \{e'\} \backslash \{e\}$ is connected since any path from v to w that used the edge e = (x,y) can now be changed to path(v,x'). e'=(x',y'). path(y',w). Also the only cycle in $T \cup \{e'\}$ is the one containing e. On removing e the graph becomes acyclic. Thus $T \cup \{e'\} \backslash \{e\}$ is a spanning tree.

Let T be a MST of G and K be the MST obtained from Kruskal's. Let t_i and k_i denote the i^{th} smallest edge in T and K respectively.

Claim 2: $P(i) := t_j = k_j$ is true $\forall j \in [1, i], 1 \leq i \leq n-1$
Proof:Proof is by induction
Base Case: k_1 is the smallest edge in G. It can be shown that k_1 must be in T in the following way.

> *Case 1*: $k_1 \in T$, Nothing to be done
> *Case 2*: $k_1 \notin T$
> As T is a spanning tree (maximally acyclic), adding k_1 to T forms a cycle containing k_1. Replace any edge of the cycle with k_1 to get the graph T' = $T \backslash \{e\} \cup \{k_1\}$. From claim 1, T' is also a spanning tree. Also since k_1 is strictly lesser than any other edge in G, weight(T') < weight(T), which contradicts the fact that T is a MST.

Induction Hypothesis: For $m > 0$, $P(m)$ is true
Induction Step: Assume $t_{m+1} \neq k_{m+1}$, then $k_{m+1} < t_{m+1}$ from the choice of edges in Kruskal's. Add the edge k_{m+1} to T. A cycle C is created from the property of maximal acyclicity. $C \not\subseteq \{t_1, t_2, ...t_m, k_{m+1}\}$. If this was not the case, it follows from the Induction Hypothesis that $C \subseteq K$ which is false as K is a tree. Thus C contains at least one edge t_p where $p \geq m+1$. Since, $k_{m+1} < t_{m+1} \leq t_p$, replacing t_p with k_{m+1} gives a spanning tree of lesser weight than T, contradicting optimality of T.

$P(n-1) \Leftrightarrow K = T$

1.2 *O(n)* Algorithm to find Minimum Spanning Tree

We make use of the fact that G is a sparse graph of at most $n+8$ edges. Since, a spanning tree contains exactly $n-1$ edges, if we manage to select $|E|-(n-1)$ edges that are definitely not a part of any MST, what remains is a valid MST (Definite elimination is possible when there are unique edges, however we can use this to model the equal edge case as we shall see).

Thus we propose the following algorithm,

```
Function FindMST(edges E, number of vertices n):
    T := E
    while C := DetectCycle(T) ≠ φ do
        T = T − mostExpensiveEdge(C)
    end
    return T

Function DetectCycle(graph G):
    stack = φ
    For some v ∈ G
    return DFS(v, stack)

Function DFS(vertex v, stack stack):
    if stack ≠ φ then  predecessor = stack.top()
    else  predecessor = φ
    stack.push(v)
    visited(v) = true
    forall w ∈ neighbours(v) do
        if not visited(v) then
            S = DFS(w, stack)
            if S ≠ φ then  return S
        end
        else if visited(w) and w ≠ predecessor then
            C = φ
            C.insert(edge(w,v))
            while stack.top() ≠ w do
                x = stack.top()
                stack.pop()
                y = stack.top()
                C.insert(edge(x,y))
            end
            return C
        end
    end
    stack.pop()
    return φ
```

Claim 1: Given a cycle C $\in$ G with unique maximum weight edge e, e $\notin$ any MST of G.
Proof: Proof is by contradiction. Assume e = (x,y) $\in$ a MST M. Now if e is removed from M, M is divided into two components S containing x and S' containing y. It can now be claimed that $\exists e' \in C$ not a part of M that has one end in S and other in S', since the path C-e in G starts at x and ends at y and it must cross over from S to S' at some point. Now, M' = $M \backslash e \cup e'$

is connected since any vertex in S has a path to any vertex in S' using e'. Also M' has |G|-1 edges, thus it is a spanning tree of G. Since e' $\in$ C, weight(e') < weight(e), making M' cheaper than M, a contradiction.

The above claim can be used even in the case when there isn't a unique maximum edge in C. We can argue that one of the maximum weight edges must be removed, by adding small perturbations to the weights without changing their relative order.

Claim 2: DetectCycle correctly identifies a cycle in G.
Proof: Since G is a undirected graph, non-tree edges can only be from a vertex to its ancestor in DFS tree but not from one sub-tree to other (If there were such an edge then it would have been a tree edge of the first visited sub-tree). If there is a non-tree edge e between v and its ancestor w, e along with the chain from v to w in the DFS tree forms a cycle. This is exactly what DetectCycle returns.
Using the above claims we prove the correctness of given algorithm.

Proof of termination: From claim 1 and 2, in each iteration, the algorithm eliminates one edge correctly from T, that should not be a part of MST, reducing the problem from **opt**(T) to **opt**(T-e). Since the edge removed is part of a cycle, T always remains connected. When T contains exactly |V|- 1 edges, T cannot contain a cycle since it is connected. Thus the loop runs exactly |E|- |V|+ 1 times, which is $\leq$ 9 for G.

Complexity Analysis: As stated above the loop iterates a maximum of 9 times. In each iteration DFS is performed which takes $O(m+n)$ time. Since m $\leq$ n+8, it can be reduced to O(n). Thus the complexity of the algorithm is $O(9*n) \equiv O(n)$

Huffman Encoding

2.1 Optimal Binary Huffman Encoding with Fibonacci Numbers as Frequencies

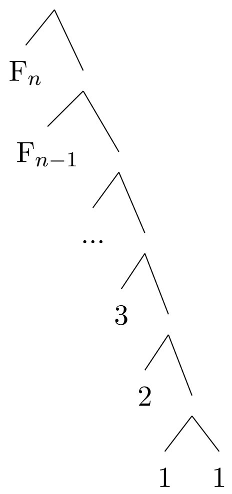
Given n letters with the frequency vector F as the first n Fibonacci numbers, with F_1=1 and F_2=1. We have to find the optimal binary Huffman encoding for this setting.

As we know, the property of Fibonacci sequence's i^{th} term is, $F_i = F_{i-1} + F_{i-2}$.

Claim: $P(n) := F_{n+1} = \sum_{i=1}^{n-1} F_i + 1$ is true $\forall n \geq 2$
Proof is by induction on n.
Base Case: Consider F={1,1,2}, n=2. $F_{n+1} = \sum_{i=1}^{1}\text{F}_i + 1$. Implying, $F_3 = 1 + 1 = 2$. This holds true.
Induction Hypothesis: Assume the claim is holds for n=k, i.e., $P(k)$ is true.
Induction Step: We will prove the correctness of P(k+1).
Since F_{k+2}=F_{k+1} + F_k
F_{k+2}= $\sum_{i=1}^{k-1}\text{F}_i + 1 + F_k$
F_{k+2}= $\sum_{i=1}^{k}\text{F}_i + 1$
P(k+1) holds true.
As our claim is true for n=2 and true for k $\Rightarrow$ true for k+1.
Our claim holds true $\forall n \geq 2$.

Since, $F_{n+1} = \sum_{i=1}^{n-1}\text{F}_i + 1, \implies F_{n+1} > \sum_{i=1}^{n-1}\text{F}_i$. .... eq(1)

The optimal binary Huffman tree is trivial for n=1 (only a single node) and n=2 (both nodes at the same level) Fibonacci numbers.
Definition: T_n is defined as the sideways growing tree depicted in the figure, that has F_1, F_2 at depth n-1 and F_n at depth 1.
Claim: For the first n Fibonacci numbers, after the k^{th} iteration of Huffman's algorithm, we are left with $F_{k+2}, ...F_{n-1}, F_n$ and T_{k+1}. The above property holds at every iteration k while building the Huffman tree.
Proof is by induction on number of iterations.



Base Case: We start building up the tree from the lowest frequency, i.e., go from left to right in F. Combine 1 and 1 to get a tree, with weight 2 as a root node, which is essentially T_2
We are now left with $F_3, ..F_{n-1}, F_n$ and T_2.

T_2 is a sideways growing tree as depicted in figure above. Our claim holds true.
Induction Hypothesis: Assume that the claim is true i= k^{th} iteration.
Induction Step: We will prove the correctness of claim for i= $k+1^{th}$ iteration.
After k iterations, we will have combined all the nodes till $k+1$ in T_{k+1}, the root node will be of weight $\sum_{j=1}^{k+1}\mathrm{F}_j$. Using eq(1), we can state that weight(T_{k+1}) $< F_{k+3}$.
Thus the two minimum weights at this stage are that of F_{k+2} and T_{k+1}. The algorithm combines them to form T_{k+2}. We are now left with $F_{k+3}, ..F_{n-1}, F_n$ and T_{k+2}. The claim holds for k + 1.
Our claim is true for k=1 and true for k iterations $\implies$ true for k+1 iterations, hence, proved by induction. Our claim holds ∀1≤k ≤ n-2.
After $(n-2)^{th}$ iteration, we are left with only F_n and T_{n-1}. These are combined to get T_n , the optimal Huffman tree for F

From above proofs, we can conclude that F_n will be at depth 1, F_{n-1} will be at depth 2 and so on, in the optimal Huffman encoding. Both F_1 and F_2 will be at the same depth.
The optimal Huffman encoding for first n Fibonacci numbers will be (taking left branch as "0" and right branch as "1") :

F_n : 0

F_{n-1} : 10

F_{n-2} : 110

...

3 : 11..(*depth of tree-3* times)0

2 : 11..(*depth of tree-2* times)0

1 : 11..(*depth of tree-1* times)0

1 : 11..(*depth of tree* times)

Since, at each level (other than the last level) there is only one number of the Fibonacci sequence and at the last level there are two numbers. The depth of the tree will be one less than the total number of Fibonacci numbers, i.e., n-1.
The encoding of first two numbers is 111...(n-1 times) and 111...(n-2 times)0. For $F_k, k \geq 1$ encoding is 111..(n-k times)0

2.2 Compression of a File using Huffman Encoding vs Fixed-length Encoding

Given a file with 16-bit characters, which represents 2^{16} different characters. We aim to compress such a file, given the constraint that the maximum character frequency is strictly less than twice the minimum character frequency.

Fixed-length Encoding
Since there are 2^{16} different characters, each character will be encoded by 16-bits, i.e., the size of the file would be 16 X $\sum f_i$ bits.

Huffman Encoding
Claim: For 2^n different characters such that (max character frequency) $<$ 2(min character frequency), the Huffman tree will be a perfect binary tree with height n $\forall$n $\geq$ 1.

We prove correctness of claim by a loop invariant proof using the following invariant:
Invariant: After each loop iteration i involving combining two frequencies to result in one tree (with frequency = sum(frequencies of right child and left child)), the resulting list containing 2^{n-i} frequencies maintaining the property (max character frequency) $<$ 2(min character frequency) if there are at least two frequencies in the list.
Initialization: Initially, when zero loop iterations were made. The invariant states that the list contains 2^n frequencies which hold the property (max character frequency) $<$ 2(min character frequency). This is given and hence, is true.
Maintenance: Let the loop invariant holds after loop iteration j. The list has $m = 2^{n-j}$ frequencies with (max character frequency) $<$ 2(min character frequency).
The list is sorted in the ascending order, $\{f_1, f_2, ..., f_{m-1}, f_m\}$ given $f_m < 2f_1$.
For $j + 1^{th}$ loop iteration, we start by combining the two smallest frequencies and placing the resultant frequency F_{sum} in its sorted position in the list.
$\implies F_1 = f_1 + f_2$

> ***Claim:*** $F_1 > f_m$.
> ***Proof:*** Given $f_m < 2f_1$. Since, f_1 is the minimum frequency, $f_1 \leq f_2$.
> $\implies f_1 + f_2 \geq 2f_1$
> $\implies f_1 + f_2 > f_m$
> $\implies F_1 > f_m$
> Hence, proved.

The frequency list now looks like $\{f_3, f_4, ..., f_{m-1}, f_m, F_1\}$.

> ***Claim:*** $F_1 \leq 2f_3$.
> ***Proof:*** Given $f_m < 2f_1$ and $F_1 = f_1 + f_2$. Since, f_1 was the minimum frequency initially, $f_1 \leq f_2 \leq f_3 \leq f_4$.
> $\implies f_1 \leq f_2$ and $f_2 \leq f_3$
> $\implies f_1 + f_2 \leq 2f_3$
> $\implies F_1 \leq 2f_3$
> Hence, proved.

$\implies F_2 = f_3 + f_4 \geq 2f_3$
$\implies F_2 \geq F_1$
The frequency list now looks like $\{f_5, f_6, ..., f_m, F_1, F_2\}$.
The above condition will again hold true, and can be proved similarly. Hence, in $j + 1^{th}$ loop iteration, the first two frequencies will continue to combine together till we combine f_{m-1} and f_m to get $F_{m/2}$. The final list after the $j + 1^{th}$ loop iteration, which is equivalent to $m/2$ Huffman iterations, looks like $\{F_1, F_2, ..., F_{m/2}\}$.

Claim: $F_{m/2} < 2F_1$.
Proof: Given $f_m < 2f_1$, F_1= f_1 + f_2 and $F_{m/2}$= f_{m-1} + f_m. Since, f_1 was the minimum frequency initially, $f_1 \leq f_2$ and f_m was the maximum frequency initially $f_{m-1} \leq f_m$.
Since, $f_m < 2f_1$ and $f_1 \leq f_2 \implies f_m < 2f_2$
Since, $f_{m-1} \leq f_m$ and $f_m < 2f_2 \implies f_{m-1} < 2f_2$
$\implies f_{m-1} + f_m < 2f_1 + 2f_2$
$\implies F_{m/2} < 2F_1$
Hence, proved.

Hence, after the $j + 1^{th}$ loop iteration, the total frequencies in the list are half of initial number, i.e., m/2 = 2^{n-j-1} and the property (max character frequency) < 2(min character frequency) still holds.
The invariant is maintained.
Termination: The loop terminates when there is no further scope of combining two frequencies, i.e., there is only 1 element left in the list. Implying, $2^{n-i} = 1$.
That is, after n loop iterations, the loop will terminate.

From the above proof we can conclude that if we start with 2^n different characters such that (max character frequency) < 2(min character frequency), after one loop iteration we will have 2^{n-1} perfect binary trees of depth 1 holding the property (max root frequency) < 2(min root frequency).
After i such iterations, we will get 2^{n-i} perfect binary trees of depth i holding the same property. Hence, after termination, i.e., n iterations, we will get 1 perfect binary tree with depth n.

In case of n=16, we have 2^{16} different characters. We will get one perfect binary tree of depth 16, with the characters as leaves. Since, it's a perfect binary tree, all leaves will be at the last level.
The Huffman coding for each leaf will be a 16-bit number. The size of the compressed file would be 16 X $\sum$f$_i$ bits.

The size of the compressed file obtained are same in Fixed-length coding and Huffman coding. Hence, proved.

Graduation Party of Alice

3.1 Largest subset with degree constraints

According to the problem, a pair of people either know each other or do not each other. So we define a undirected graph G with each person as a vertex wherein vertices x,y are adjacent iff x and y know each other. It follows that the number of people x knows is given by its degree in G, $degree(x)$. Note that the number of people x doesn't know is the degree of x in the complementary/inverse graph of G, i.e, $degree^{-1}(x) = |G| - degree(x) - 1$.

Now the required subset can be considered as an induced sub-graph S of G of maximum size where ∀ x ∈ S ,$degree(x) \geq 5$ and $degree^{-1}(x) \geq 5$

Claim 1 : Any vertex x, $degree(x) < 5 \vee degree^{-1}(x) < 5$ cannot be a part of our solution.
Proof :
Case 1: $degree(x) < 5$
It is clear that x does not satisfy the condition of $degree(x) \geq 5$ in G. It follows that x cannot satisfy the condition in any induced sub-graph of G either, since the degree can only be lesser in a sub-graph. Thus x cannot be a part of our solution.
Case 2: $degree^{-1}(x) < 5$
The same can be claimed if the $degree^{-1}(x) < 5$, i.e, |G|- $degree(x) - 1 < 5$.
Consider an induced sub-graph, G' of G on V(G)-y, y ≠ x.

Case (i): y is adjacent to x in G.
$degree^{-1}_{G'}(x)$ = |G'|- $degree_{G'}(x)$ -1 = (|G|- 1) - ($degree_G(x)$ - 1) -1 = $degree^{-1}_G(x)$
Thus $degree^{-1}(x)$ remains same when an adjacent vertex is removed.
Case (ii): y is not adjacent to x.
$degree^{-1}_{G'}(x)$ = |G'|- $degree_{G'}(x)$ -1 = (|G|- 1) -$degree_G(x)$ -1 = $degree^{-1}_G(x)$ - 1
Thus $degree^{-1}(x)$ reduces by one when a non-adjacent vertex is removed from G.

In any case, the $degree^{-1}(x)$ does not increase for any induced sub-graph containing x, i.e, $degree^{-1}(x) < 5$ in G implies $degree^{-1}(x) < 5$ in any subset of V(G) which implies x cannot be a part of the solution.

Thus we propose the following algorithm. The algorithm uses a min-heap that stores a tuple of vertex, $degree(x)$ and $degree^{-1}(x)$ w.r.t S, i.e (x, degree(v), |S|- degree(v)-1). The priority on two tuples (x, d_1, d_2) and (y, e_1, e_2) is defined as min(d_1,$d2$) < min(e_1, e_2). Since the operations used are extractMin and updatePriority, using a Fibonacci Heap can be useful.

Definition: $minDegree(x)$ is defined as the minimum of $degree(x)$ and $degree^{-1}(x)$.
Claim:Assuming the optimal solution for G is given by **opt**(G), the subset returned by the proposed algorithm S = **opt**(G).
Case 1: $\forall x \in G, degree(x) \geq 5 \wedge degree^{-1}(x) \geq 5$
Since G itself satisfies the constraints of the problem, **opt**(G) = G.
Also, $degree(x) \geq 5 \wedge degree^{-1}(x) \geq 5 \Rightarrow minDegree(x) \geq 5 \forall x$. Thus the condition of while loop fails and G is returned by the algorithm.
Case 2: Let x ∈ G be vertex with lowest minDegree, minDegree(x) ¡ 5.
$minDegree(x) < 5 \Rightarrow degree(x) < 5 \vee degree^{-1}(x) < 5$. It follows directly from Claim 1, x ∉ **opt**(G), thus **opt**(G) = **opt**(G - x).
This is exactly what is done in the algorithm.

Proof of termination: The while loop in the algorithm reduces the size of S by exactly 1 in each iteration. Thus the number of iterations is bounded by the number of vertices V.

Complexity analysis: Building the heap takes $O(n)$ time.
In each iteration, the minimum is extracted and the priority of each of the |S|-1 vertices is updated, and each operation takes log(|S|) time. Thus, |S|log(|S|) operations are performed in each iteration. From above, the loop runs a maximum of |V|times, hence the complexity of the algorithm is $O(n^2 logn)$

```
Function FindMaximalSubset(pairs P, vertices V):
    S = V
    forall v ∈ V do
        adjacencyList[v] = φ
    end
    forall (x, y) ∈ P do
        adjacencyList[x].add(y)
        adjacencyList[y].add(x)
    end
    forall v ∈ V do
        minHeap.insert(v, degree(v), |V|- degree(v) - 1)
    end
    while minHeap ≠ φ∧ minDegree(minHeap.top()) < 5 do
        v := minHeap.top()
        S.remove(v)
        forall w ∈ S do
            if adjacencyList[v].contains(w) then minHeap.updatePriorityBy(w, 1, 0)
            else minHeap.updatePriorityBy(w, 0, 1)
        end
    end
    return S
```

3.2 *O(n)* Greedy Algorithm to Divide a Group into Subgroups Restricted to Constraints

Given n_o people with ages in range [10,99], should be divided into minimum number of groups, such that:

1. Each group has a size of at most 10.
2. The age difference among the members in a group can be at most 10.

The list S contains the age of people, not necessarily in a sorted order. The size of S is n_o.

```
Function FindMinNumOfGroups(list S):
    freq:= [0]*100
    numOfGroups, count:=0
    forall v ∈ S do freq[v]+=1
    i:=10, minAgeOnTable:= -1
    while i < 100 do
        if freq[i]==0 then i+=1
        continue

        if minAgeOnTable ==-1 then minAgeOnTable =i
        if i ≤ minAgeOnTable + 10 then
            freq[i]-=1
            count+=1
            if count == 10 then
                numOfGroups+=1
                if freq[i] ≠ 0 then minAgeOnTable = i
                else minAgeOnTable = -1
                count = 0
            end
        end
        else
            numOfGroups+=1
            count=0
            if freq[i] > 0 then minAgeOnTable = i
            else minAgeOnTable = -1
        end
    end
    if count > 0 then numOfGroups+=1
    return numOfGroups
```

Let T be a seat allocation and $minT_i$, $maxT_i$ denote the minimum, maximum age on table T_i. Sort the tables in increasing order of $minT_i$.Now T_i denotes the table in i^{th} position in this order.
Claim: There exists an optimal solution T in which ages are allotted to tables in increasing order of age, i.e,
$\exists$ T, for $i < j$, $a \in T_i$ and $b \in T_j$, $a \leq b$.
Proof: Consider an optimal solution S where for $i < j$, $a \in T_i$ and $b \in T_j$, $a > b$
Case 1: $a > minT_i$, $b > minT_j$.
Since T is a valid solution, a - $minT_i \leq 10$, b - $minT_j \leq 10$.
b can be placed in T_i instead of a.

$b > minT_j \geq minT_i \Rightarrow b - minT_i \geq 0$
$minT_i \geq a - 10 > b - 10 \Rightarrow b - minT_i < 10$
a can be placed in T_j instead of b. $b > minT_j$, $a > b \Rightarrow a - minT_j > 0$
$a - minT_i \leq 10, minT_i \leq minT_j \Rightarrow a - minT_j \leq 10$
Case 2: a $> minT_i$, b $= minT_j$.
b can be placed in T_i instead of a.
$b = minT_j \geq minT_i \Rightarrow b - minT_i \geq 0$
$minT_i \geq a - 10 > b - 10 \Rightarrow b - minT_i < 10$
a can be placed in T_j instead of b. Since b was the minimum on T_j, let the new minimum on T_j be $minT_j^*$
$minT_j^* \leq a \Rightarrow a - minT_j* \geq 0$
$a - minT_i \leq 10, minT_i \leq minT_j \leq minT_j^* \Rightarrow a - minT_j^* \leq 10$
Case 3: a $= minT_i$, b $\geq minT_j$
$a = minT_i > b \geq minT_j \Rightarrow minT_i > minT_j$, a contradiction.
Thus if a and b are exchanged, S still remains a valid allocation and |S|does not change. Hence S remains optimal.

Let the ages in increasing order be $a_1, a_2, ...a_n$.
Define $range(T_j) := max(k) \forall a_k \in T_j$
Claim 2: Let S be a seat allocation achieved by greedily allocating people in order of age at maximum capacity whenever possible (as done by the algorithm). Let T be an optimal solution that satisfies condition in claim 1. Then $range(S_i) \geq range(T_i)$
Proof: Case i = 1 is trivial since S is greedy. Assuming claim to be true for i-1.
Let k = $range(T_{i-1})$, k' = $range(T_i)$ and s = $range(S_{i-1})$. Then k'-(k+1) $\leq$ 10, $a_{k'} - a_{k+1} \leq 10$.
Since s $\geq$ k, k' -(s+1) $\leq$ 10 and $a_{k'} - a_{s+1} \leq 10$. Thus $range(S_i) \geq range(T_i)$.

Claim 3: In the above claim, $|S| = |T|$
Proof: Assume, $|T| = p < |S|$. From claim 2, $range(S_p) \geq range(T_p) = n$. Thus n people can be covered in p tables of S. Since S is greedy it should have selected all the n people. Therefore, $|T| < |S|$ is not possible.

Complexity Analysis: The for loop takes $O(n_0)$ time. For the while loop
Let S(j) := $\sum freq[i]$ + 100 - i in iteration j. Then S(j+1) = S(j) - 1 since if freq[i] $\neq$ 0, then freq[i] is decremented else i is incremented. Note that when i = k, freq[j]=0 $\forall$ j < k. Thus at termination, freq[p] = 0 $\forall$ p and i = 100 implies S(j) = 0. The number of iterations are hence bounded by $\sum freq[i]$ + 100 - 10 = n_0 + 90. Each iteration takes constant time making the complexity $O(n_0)$.