

Solutions to Problem 1 of Homework 9 (10 + (6) points)

Name: GOWTHAM GOLI (N17656180)

Due: Tuesday, November 24

Consider the following greedy algorithm for the ACTIVITY-SELECTION problem. Select the activity a_i with the shortest duration $d_i = f_i - s_i$. Commit to scheduling a_i . Let S'_i consist of all activities a_j which do not overlap with a_i : namely, either $f_j \leq s_i$ or $f_i \leq s_j$. Recursively solve ACTIVITY-SELECTION on S'_i , scheduling the resulting activities together with a_i .

- (a) (2 pts) Give a simple example of the input where the proposed greedy algorithm fails to compute the correct optimal solution.

Solution:

Consider the following activities, a_1, a_2 and a_3 such that $(s_1, f_1) = (1, 5)$, $(s_2, f_2) = (4, 7)$ and $(s_3, f_3) = (6, 9)$. By using the above greedy algorithm, the activity selected will be a_2 i.e the activity with the shortest duration and a_1, a_3 will be excluded as they overlap with a_2 . Thus the output of the suggested greedy algorithm will be a_2 . But clearly, the optimal solution is to select the activities a_1 and a_3 . Therefore the proposed greedy algorithm fails to compute the optimal solution \square

- (b) (4 pts) Consider the following attempt to nevertheless justify the correctness of this algorithm using the “Greedy Always Ahead” method. Given a solution Z , let $F_i(Z)$ be the sum of the i shortest activities scheduled by Z , and ∞ if $|Z| < i$. Try to tell exactly where the claim “For any i and Z , $F_i(Z) \geq F_i(\text{greedy})$ ” fails. Is its base of induction? Or, if in the inductive step, for what smallest i does the transition from i to $i + 1$ fail? Justify your answer.

Solution:**Base Case**

For $i = 1$, The greedy strategy chooses the activity of the shortest duration. Hence $F_1(\text{greedy})$ will be lesser than F_1 of any other feasible solution $\implies F_1(\text{greedy}) \leq F_1(Z)$. Therefore the base case is true

Induction Hypothesis

Assume that $F_i(\text{greedy}) \leq F_i(Z) \quad \forall i \leq r$

Inductive Step

Let the activities that the greedy strategy selects are i_1, \dots, i_r, i_{r+1} and the activities that Z selects are j_1, \dots, j_r, j_{r+1} . We need to prove that $F_{r+1}(\text{greedy}) \leq F_{r+1}(Z)$

Let us assume that $F_{r+1}(\text{greedy}) > F_{r+1}(Z)$. From the induction hypothesis, we know that $F_r(\text{greedy}) \leq F_r(Z)$. Therefore for $F_{r+1}(\text{greedy}) > F_{r+1}(Z)$ to happen it must be the case that $d_{i_{r+1}} > d_{j_{r+1}}$. Also $d_{i_{r+1}} > d_{i_r}$. However we cannot make a comparison between d_{i_r} and $d_{j_{r+1}}$.

If $d_{i_r} < d_{j_{r+1}} \implies d_{i_r} < d_{j_{r+1}} < d_{i_{r+1}}$ then the greedy strategy would have selected j_{r+1} instead of i_{r+1} thus contradicting our assumption. Therefore, $F_{r+1}(\text{greedy}) \leq F_{r+1}(Z)$.

Else If $d_{j_{r+1}} < d_{i_r} \implies d_{j_{r+1}} < d_{i_r} < d_{i_{r+1}}$ then j_{r+1} would have already been selected by the greedy strategy i.e $j_{r+1} \in \{i_1, \dots, i_r\}$ thus our assumption might indeed be true. Therefore, $F_{r+1}(\text{greedy}) > F_{r+1}(Z)$

Also if greedy algorithm schedules k activities and Z schedules m activities where $m > k$ then if the given condition were true it would fail from k to $k+1$ as $F_{k+1}(\text{greedy}) = \infty$

□

- (c) (4 pts) Consider the following alternative attempt to justify the correctness of this algorithm using the “Local Swap” method. Given a solution Z , try to argue that it is always safe to substitute one activity in Z by the first activity of the greedy algorithm, and then argue that the resulting recursive subproblems are the same for the “perturbed” opt and the greedy. Where does your argument run into problem? Is it the swap, or the recursive subproblem part?

Solution:

Let a be the activity of shortest duration in the given set of activities and S be the set of all the activities that overlap with a .

Let the activities scheduled by Z be $OPT = \{a_{j_1}, \dots, a_{j_k}, \dots, a_{j_m}\}$ and OPT^* be the set of activities obtained after substituting some activity in OPT by a . Now we have two cases

If OPT already contains a in it, then we could simply replace the existing a with a . In this case, $OPT^* = OPT$. But if we substitute some a_k by a , we have two a ’s in OPT^* . So exclude one of the duplicates then $|OPT^*| = |OPT| - 1$. Therefore, this is not an optimal solution anymore.

If OPT doesn’t contain a in it and it has one or more non-overlapping activities from the set S . Now replace some activity a_k in OPT with a . Now OPT^* has some overlapping activities in it. So eliminate all those activities that overlap with a from OPT^* . So $|OPT^*| < |OPT|$. Therefore, this is not an optimal solution anymore.

Therefore the argument runs into a problem when we swap some activity in Z with the first activity of the greedy algorithm

□

- (d) (**Extra credit**) (6 pts) Let t_{opt} be the size of the optimum solution and t_{bogus} be the size returned by the bogus greedy algorithm discussed. Argue that $t_{bogus} \geq t_{opt}/2$. To do this, argue that the first activity a scheduled by the bogus greedy algorithm overlaps *at least* one and *at most* two activities scheduled by the correct optimum algorithm.

In case it overlaps two activities a_1 and a_2 , argue that the recursive subproblem resulting from scheduling a has the optimum value at least as large as the one resulting from excluding a_1 and a_2 from opt. (Then use induction on the size of t_{opt} .)

In the the case of only one activity a_1 , argue that you can find at most one more activity a_2 in opt such that the recursive subproblem resulting from scheduling a has the optimum value at least as large as the one resulting from excluding a_1 and a_2 from opt. (Then use induction on the size of t_{opt} .) How to you find a_2 if you need it? (This is tricky.)

Solution:



Solutions to Problem 2 of Homework 9 (12 points)

Name: GOWTHAM GOLI (N17656180)

Due: Tuesday, November 24

You have n CDs in your library labeled $1, \dots, n$. You would like to arrange them on your CD rack in linear order $\pi(1), \dots, \pi(n)$, according to some permutation π on n numbers. After such an arrangement is finalized, the cost to access CD i is equal to $\pi(i)$, as you need to scan through the first $\pi(i)$ CDs until CD i is found. Given a sequence of k CD requests $r_1, \dots, r_k \in \{1 \dots n\}$, your job is to figure out the permutation π (i.e., an array A such that $A[i] = \pi(i)$) having the *smallest total access cost* $\sum_{j=1}^k \pi(r_j)$.

- (a) (4 pts) Describe in English a greedy algorithm for this problem.

Solution:

Greedy strategy is to pick that CD r_j that is most frequently used and place it on the top of the stack i.e assign corresponding $\pi(r_j)$ to be the least □

- (b) (4 pts) Prove the correctness of your algorithm using the local-swap argument.

Solution:

Let the greedy algorithm algorithm G outputs the permutation π^G and the optimal algorithm Z outputs the permutation π^Z . Let r_m be the CD with maximum number of requests then the greedy algorithm sets $\pi_{r_m}^G$ to 1. Now there is some CD in π^Z whose value is set to 1 by Z i.e $\pi_{r_p}^Z = \pi_{r_m}^G = 1$. Now substitute $\pi_{r_p}^Z$ with $\pi_{r_m}^G$ then the resulting permutation will have two 1's in it. So substitute $\pi_{r_p}^Z (= 1)$ with the old value of $\pi_{r_m}^Z$. Let the resulting permutation be some π^{Z^*} . Then π^{Z^*} still remains to be optimal as clearly $\sum_{j=1}^k \pi^{Z^*}(r_j) \leq \sum_{j=1}^k \pi^Z(r_j)$. Therefore, the **Lemma** is, In the optimal solution the value of π for the CD with maximum number of requests is set to 1

Theorem If $\pi^Z = \pi_{r_m}, \pi^{Z'}$, where $\pi_{r_m} = 1$ i.e r_m is the CD with maximum number of requests and $\pi^{Z'}$ is the optimal permutations of all the CDs starting from 2 i.e excluding r_m then we need to prove that π_z is also optimal.

Proof From the above Lemma we know that the optimal solution sets π_{r_m} to 1 i.e the CD with maximum number of requests is at the top of the stack so it means that π^Z might be an optimal solution. To show that it is indeed an optimal solution we need to prove that $\sum_{j=1}^k \pi^Z(r_j) \leq \sum_{j=1}^k \pi^{Z^*}(r_j)$.

$$\begin{aligned} \text{Let } \pi^{Z^*} = \pi_{r_m}, \pi^{Z^{**}} \text{ then as } Z' \text{ is optimal } \sum \pi^{Z'} &\leq \sum \pi^{Z^{**}} = \sum_{j=1}^k \pi^{Z^*} - \pi_{r_m} \\ \implies \sum_{j=1}^k \pi^Z(r_j) = \pi_{r_m} + \sum \pi^{Z'} &\leq \pi_{r_m} + \sum_{j=1}^k \pi^{Z^*} - \pi_{r_m} \leq \sum_{j=1}^k \pi^{Z^*} \end{aligned}$$

Therefore, π^Z is an optimal solution. Hence the proposed greedy algorithm is correct □

- (c) (4 pts) Write the pseudocode for implementation which runs in time $O(n + k)$. (Hint: At some point use counting sort.)

Solution:

```
1 Algorithm: ARRANGECDS( $n, r_1, \dots, r_k$ )
2  $freq \leftarrow \text{NEWARRAY}(n)$ 
3  $freq \leftarrow 0$ 
4 for  $i \leftarrow 1$  to  $k$  do
5    $freq[r_i] \leftarrow freq[r_i] + 1$ 
6 end
7  $CD \leftarrow \text{NEWARRAYTUPLES}(n)$ 
8 for  $i \leftarrow 1$  to  $n$  do
9    $CD(i) \leftarrow (i, freq[i])$ 
10 end
11  $\text{DESCENDINGCOUNTINGSORT\_TUPLES}(CD)$ 
12  $counter \leftarrow 1$ 
13 for  $i \leftarrow 1$  to  $n$  do
14    $\pi[CD[i](1)] \leftarrow counter$ 
15    $counter \leftarrow counter + 1$ 
16 end
17 Return  $\pi$ 
```

Algorithm 1: Greedy algorithm to minimize the total access cost of CDs in $O(n + k)$ time

In the above pseudocode $freq$ is an array such that $freq(i)$ maintains the frequency of the CD i 's requests. This step takes $O(n)$ time.

CD is an array of tuples such that $CD(i)$ stores the tuple $(i, freq(i))$. Therefore creating this list of tuples CD takes $O(n)$ time.

Now that we have a list of CDs we need to sort them using their frequencies. $\text{DESCENDINGCOUNTINGSORT_TUPLES}$ is a variant of Counting Sort that sorts a list of tuples in descending order based on the second element of the tuple i.e it sorts all the CDs based on their request frequencies. The frequency of the CD requests range between 0 to k and there are total n number of CDs. Therefore this step takes $O(n + k)$ time

Now we have a list of CDs sorted in a decreasing order based on their frequencies. The last *for* loop iterates through these sorted CDs and assigns them the value of π in an increasing order. This step takes $O(n)$ time. Therefore the total running time of the algorithm is $O(n + k)$

□

Solutions to Problem 3 of Homework 9 (8 points)

Name: GOWTHAM GOLI (N17656180)

Due: Tuesday, November 24

Design optimal Huffman codes for the following frequencies f_0, \dots, f_7 . In each case, draw the Huffman tree incrementally, until you arrive at your final solution. After you finish, which Huffman code is more “balanced”: “arithmetic” or “geometric”?

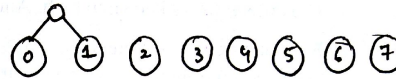
(a) (4 pts) Arithmetic: $f_i = 10 + i$, for $i = 0 \dots 7$.

Solution:

	0	1	2	3	4	5	6	7
f_i	10	11	12	13	14	15	16	17

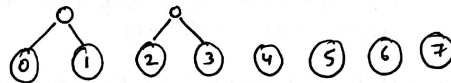
Merge 0 and 1.

	2	3	4	5	6	7	01
f_i	12	13	14	15	16	17	21



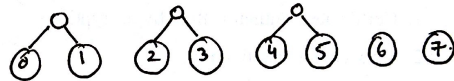
Merge 2 and 3

	4	5	6	7	01	23
f_i	14	15	16	17	21	25



Merge 4 and 5

	6	7	01	23	45
f_i	16	17	21	25	29



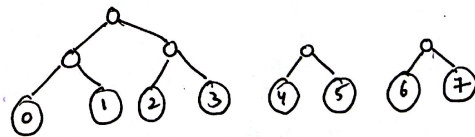
Merge 6 and 7

	01	23	45	67
f_i	21	25	29	33



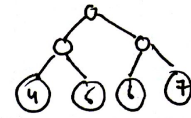
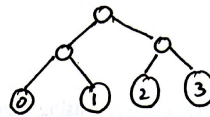
Merge 01 and 23

	45	67	0123
f_i	29	33	46



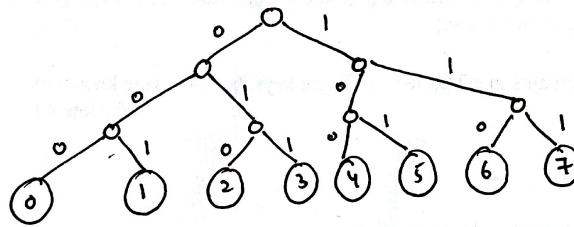
Merge 45 and 67.

f_i 0123 4567
 46 62



Merge 0123 and 4567.

\therefore The final tree is



0 \leftarrow 000 , 1 \leftarrow 001 , 2 \leftarrow 010
 3 \leftarrow 011 , 4 \leftarrow 100 , 5 \leftarrow 101
 6 \leftarrow 110 , 7 \leftarrow 111

□

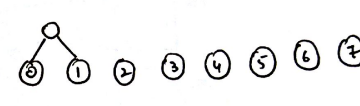
(b) (4 pts) Geometric: $f_i = 10 \cdot 2^i$, for $i = 0 \dots 7$.

Solution:

	0	1	2	3	4	5	6	7
f_i	10	20	40	80	160	320	640	1280

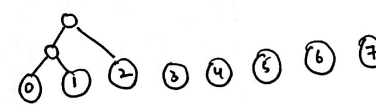
Merge 0 and 1.

	0	1	2	3	4	5	6	7
f_i	30	40	80	160	320	640	1280	



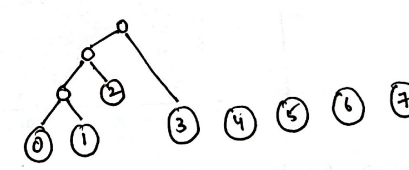
Merge 01 and 2

	0	1	2	3	4	5	6	7
f_i	70	80	160	320	640	1280		



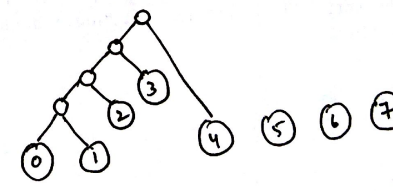
Merge 012 and 3

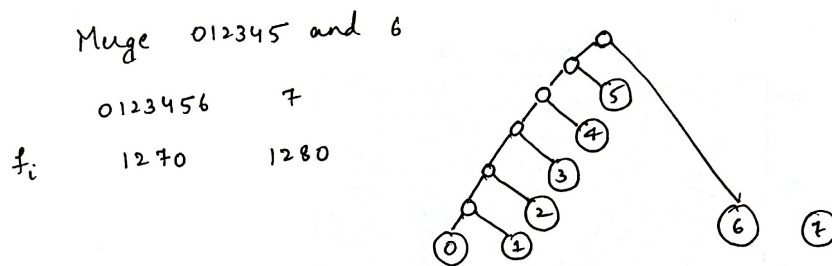
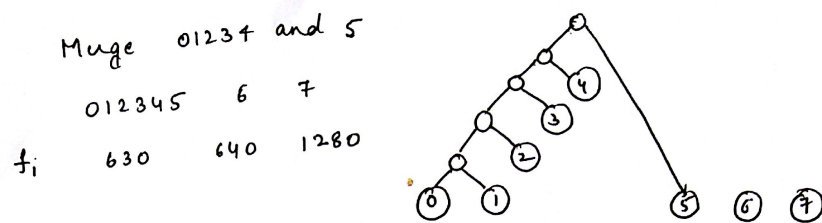
	0	1	2	3	4	5	6	7
f_i	150	160	320	640	1280			



Merge 0123 and 4

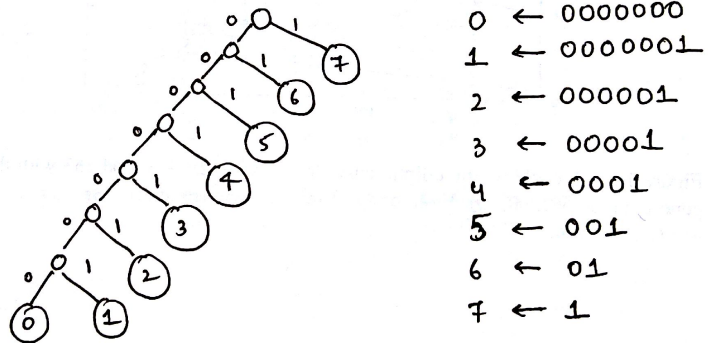
	0	1	2	3	4	5	6	7
f_i	310	320	640	1280				





Merge 0123456 and 7.

\therefore The final tree is



The Huffman code for arithmetic is more balanced than that of geometric

□

Solutions to Problem 4 of Homework 9 (20 Points)

Name: GOWTHAM GOLI (N17656180)

Due: Tuesday, November 24

Consider the problem of merging k sorted lists $L_1 \dots L_k$ of sizes n_1, \dots, n_k , where $n_1 + \dots + n_k = n$. We know that using a priority queue of size k , we can implement this merge in time $O(n \log k)$ (by repeatedly extracting smallest element from priority queue and replacing it by the next element of the list it came from).

Here we will design an alternative algorithm which repeatedly finds two sorted lists and merges them, so that after $k - 1$ merges of two lists we are left with a single sorted list. Assume merging two lists of size ℓ_1 and ℓ_2 takes time $\ell_1 + \ell_2$ (irrespective of the actual elements inside the lists). We would like to find the order of the $k - 1$ merges which minimizes the total cost.

E.g., when $k = 3$, we have three choices depending on which two lists we merge first. If we start with L_1 and L_2 (costing $n_1 + n_2 = n - n_3$), and then merge the result with L_3 (cost $(n_1 + n_2) + n_3 = n$), we pay $(n - n_3) + n = 2n - n_3$. Similarly, if we start with L_2 and L_3 (costing $n_2 + n_3 = n - n_1$), and then merge the result with L_1 (cost $(n_2 + n_3) + n_1 = n$), we pay $(n - n_1) + n = 2n - n_1$. Finally, if we start with L_1 and L_3 (costing $n_1 + n_3 = n - n_2$), and then merge the result with L_2 (cost $(n_1 + n_3) + n_2 = n$), we pay $(n - n_2) + n = 2n - n_2$. Thus, the best cost achievable is $\min(2n - n_1, 2n - n_2, 2n - n_3) = 2n - \max(n_1, n_2, n_3)$, which means we should exclude the largest list from the first merge (i.e., merge the two smallest lists first).

- (a) (4 pts) The first (naive) hope is that the order of the merges does not matter “too much”. For any $k < n$, given an example of k inputs $n_1 \dots n_k$ summing to n , and a really poor choice of the merge order, so that the total cost of the merges is $\Theta(nk)$. I.e., for $k > \log n$ this is much worse than simply sorting the n total numbers from scratch!

Solution:

Let $k = 8$ and $n_1 = 1, n_2 = 2, n_3 = 3, n_4 = 4, n_5 = 5, n_6 = 6, n_7 = 7, n_8 = 8 \implies n = 36, k > \log n$. Merge lists in the descending order i.e start merging n_8 with n_7 then with n_6 and so on. Therefore the total cost of merging these k lists will be $(15 + 21 + 26 + 30 + 33 + 35 + 36) = 196 = \Theta(nk)$ \square

- (b) (4 pts) Represent any valid order of $(k - 1)$ merges as a binary tree, where the k leaves are labeled by the k initial lists with sizes $n_1 \dots n_k$, and every merge of two lists corresponds to creating a parent node of size equal to the sum of the two lists (children) just merged. Given a particular tree (i.e., order of merges), write the total cost of all the merges as a function of $n_1 \dots n_k$ and the depths $d_1 \dots d_k$ of the initial k lists (leaves) in this tree.

Solution:

Without loss of generality assume that the merge order is $n_1, n_2, n_3, \dots, n_k$. In the first step when n_1, n_2 are merged, the cost is $n_1 + n_2$. Now we have a new list of size $n_1 + n_2$ that will be merged with n_3 . So the total cost until now is $(n_1 + n_2) + (n_1 + n_2 + n_3)$. After $k - 1$

merges the the cost will be $(n_1 + n_2) + (n_1 + n_2 + n_3) + \dots + (n_1 + n_2 + \dots + n_k)$. Notice that n_i would have occurred exactly d_i times in this summation. Therefore the total cost of merging these k lists will be $\sum_{i=1}^k n_i d_i$ \square

- (c) (4 pts) Consider the Huffman code problem with k characters $c_1 \dots c_k$, where frequency of c_i is $f_i = n_i/n$. Using part (b), argue that the optimal tree (order or merges) for the list merging problem is *identical* to the optimal tree (i.e., prefix-free code) for the Huffman code problem.

Solution:

In the optimal tree for the Huffman code, first we merge the two characters with the least frequencies from all the given set of characters i.e if c_k, c_p are the first two characters to be merged then $f_k \leq f_p < \dots = n_k \leq n_p < \dots$ and then form a new alphabet c_{kp} whose frequency is $f_k + f_p$ and add this new alphabet to the existing set of alphabets.

In the optimal tree for the merging problem, first we merge the two lists with the least number of elements from all the given set of lists i.e if L_k, L_p are the first two lists to be merged then $n_k \leq n_p < \dots$ and then form a new list L_{kp} whose size is $n_k + n_p$ and add this new list to the existing set of lists.

So there is a one to one correspondence between these two problems, therefore the optimal tree for the list merging problem is identical to the optimal tree for the Huffman code problem \square

- (d) (4 pts) Based on part (c), develop an optimal greedy algorithm for the list merging problem. Express the running time of the list merging solution (not just determining the order or merges, but also the merges themselves!) as the function of n, k and V , where V is the optimal solution *value* for the Huffman code problem introduced in part (b). Do not forget to count the time use to actually solve the Huffman code problem!

Solution:

Maintain a pointer for every leaf node to it's corresponding array and when any two leaf nodes are merged into a single node, this merged node contains a pointer to the merged array.

Proceeding this way up until the root, we finally get the resulting list formed by merging all the given k lists. This is identical to Huffman code problem and additionally at every step we are also merging two lists. Therefore the running time will be running time of Huffman code problem (with k leaf nodes $= k \log k$) and time taken to merge all the k lists

From part(b), we know that the cost of all the merges is $\sum_{i=1}^k n_i d_i = n \sum_{i=1}^k f_i d_i = nV$ where $V = \sum_{i=1}^k f_i d_i$. Hence the total running time is $O(k \log k + nV)$ \square

- (e) (4 pts) Prove that $V \leq \log k$ (think of one solution which is always an option),¹ and substitute $V = \log k$ into the formula you got in part (d). How does it compare with the original $O(n \log k)$ solution?

Solution:

If the tree is a perfectly balanced binary tree then $d_i = \log k \ \forall i = 1 \text{ to } k$
 $\implies V = \frac{1}{n} \sum_{i=1}^k n_i d_i = \frac{\log k}{n} \sum_{i=1}^k n_i = \log k$

If the tree is not perfectly balanced i.e like the tree in Problem-3, part(b), i.e $d_i = i$, then
 $V = \frac{1}{n} \sum_{i=1}^k n_i d_i = \frac{1}{n} \sum_{i=1}^k n_i i \leq \log k$.

Therefore $V \leq \log k$. Substitute this value of V in part (d), we get the running time
 $T(n) = O(k \log k + n \log k) = O(n \log k)$ i.e the greedy algorithm is not asymptotically significant than the original solution □

¹It turns out that one can prove a much tighter bound on V : $V \in [H, H+1]$, where $H = \sum_{i=1}^k \frac{n_i}{n} \log_2(\frac{n}{n_i})$ is called the *entropy* of the probability distribution $(\frac{n_1}{n}, \dots, \frac{n_k}{n})$. Moreover, for many “skewed” distribution, $H, V \ll \log k$.