Knapsack

Shelflife [ ],

double franctionalKnapsackBrutforce [weights[], value[], capacity

// Input :- ⇒ Array of weight of each item,
Array of values of each item,
Array of shelflife
Total ~~Comptox~~ capacity

n = number of items.
max_value = 0

// Generate all subsets of items.
for each subset of item:
subset_weight = 0
subset_value = 0

// Calculate total weight and value for this subset
& for each item in subset

if subset_weight + weight[item] <= capacity
subset_weight += weight[item]
subset_value += value[item]

else
remaing_capacity = capacity - subset_weight
subset_value += values[item] * [remaing_capacity / weights[item]]

subset_weight = capacity.
break.

if subset_value > max_value

max_value = subset.value.

Time Complexity

1) As we first generate all possible subsets of given items.

∴ Generation of subsets = $O(2^n)$

2) Subset Evaluation:

For each subset the algorithm runs for the subsets size times.

3) Total Time complexity :-

$O(n \cdot 2^n)$

Knapsack - Greedy Approach

double ~~funct~~ fractionalKnapsack (weight [], value [], shelflife [], capacity)

// Input :- 1) Array of weight of each item
2) Array of value of each item.
3) Array of shelflife of each item.
4) Total capacity

// Output :- Maximum value.

// Sort according to ~~shelfli~~ shelflife/value ratio

vector < pair < double, pair < int, int >>> v

for (i <- 0 ; i < weights.size() ; i++) :

double ratio = ~~shelflife [i] / value [i]~~ value[i] / shelflife * weight[i]

v <= { ratio, { weights [i], values [i] }

// Sort vector v

Sort (v.r begin, v.r end)

current - weight = 0

current - value = 0

for (i <- 0 : i < v.size (); i++) :

w = v [i] . second . first

v = v [i] . second . second.

if (w + current_weight <= capacity)

current value += v

else

remaing_capacity = capacity - current_weight

currentvalue += v * (remaing_capacity / w)

current_weight = capacity.

break.

return current value.

* Time Complexity :

1) As we first generate all possible subset of given items.

∴ Generation of

Time Complexity.

1) Creating a vector pair of ratio of shelflife / value and weight and value of each item.
   = O(n)
2) Sorting this created vector
   = O(n log n)
3) Calculating answer using greedy approch
   = O(n)

Total time complexity :-

= O(2n + n log n)

= O(n log n)

. Test - cases.

Capacity	weight	value	shelflife	Max value
50	[10, 20, 30]	[60, 100, 120]	[2, 3, 5]	230
100	[10, 20, 30]	[10, 20, 115]	[2, 3, 5]	145
100	[10, 20, 30, 40, 50 60, 70]	[10, 20, 5, 8, 55, 9, 12]	[2, 3, 5, 1, 9, 3, 2]	25.2857
50	[10, -20, 30]	[60, -100, 120]	[2, 3, 5]	error.
-12	[10, 20, 30]	[60, 100, 120]	[2, 3, 5)	error.

Huffman Node Tree.

HuffmanNode* buildHuffmanTree (const string & text)
unordered_map <char

// objective :- To find out the frequency of char and and formation of tree

// Input :- A string name text.

// Output :- return the top element of the tree

// calculate frequency of each element.

for (char ch : text) do
if (isalpha(ch)) do
freqMap[ch]++;

// Priority queue to build the tree based on frequency.

priority_queue <HuffmanNode*, vector <HuffmanNode*>, Compare > minHeap.

// Insert each char and its frequency as a node into the minHeap.

for (const auto & pair : freqMap) do
minHeap.push (new HuffmanNode (pair.first, pair.second));

// Build Huffman tree

while (minHeap.size() > 1) do {

HuffmanNode* left = minHeap.top()

minHeap.pop()

HuffmanNode* right = minHeap.top()

minHeap.pop()

HuffmanNode* merged = new HuffmanNode ('10', left->freq + right freq)

merged->left = left

merged->right = right

minHeap.push(merged)

return minHeap.top();

// function to generate Huffman Code.

void generateCode (HuffmanNode* root, const string& code, map<char, string)& huffman code)

Objective:- To generate Huffman Code.

Input :- 1) root node

2) a string name code

3) a map. hashmap.

Output :- ~~to traverse~~ void.

if (! root) return

```
if (root -> character != '\0') do
        huffmanCodes [root -> character] = code
gene
generateCodes(root -> left, code + "0", huffmanCodes)
generateCodes(root -> right, code + "1", huffmanCode)

// function to compress text using Huffman coding.

String compress(const string & text, map < char, string > &
                                                huffman code)
// Objective :- To compress text
// Input :- a string, a hashMap
// output :- a string.

    for (~~char~~ ch : text) do
        if (huffmanCode.find(ch) != huffmanCode.end)
            compressedText += huffmanCode[ch];

returns compressedText.
```

Time Complexity.

1) Count frequency of every letter using loop = O(n)
2) ~~#~~
2) To build priority queue for each unique character K we will have = O(K log K)

3) for merging to the loop run (K-1) times, = O(K)

~~Over~~ Overall :-

Time complexity = O(n) + O(K log K) + O(K)
= O(K log K)

Test Case :-

	File type	Original text size	Compressed size	Compressed ratio
1)	Text	1016	441	0.43
2)	Docx	50 216	21390	0.43
3)	HTML	50 490	21433	0.42
4)	PDF	2680	526	0.43
5)	~~nd~~ PDF(empty)	no - content extracted.		