Greedy Algorithms

- Greedy algorithms are used for optimization problems.
- An optimization problem can be solved using Greedy if the problem has the following property: *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.*
- If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming.
- But Greedy algorithms cannot always be applied. For example, Fractional Knapsack problem can be solved using Greedy, but 0-1 Knapsack cannot be solved using Greedy.
- The greedy algorithm method include assembly-line scheduling, activity- selection problem, fractional knapsack problem, Huffman codes, minimum-spanning-tree algorithms, Dijkstra's algorithm for shortest paths from a single source etc.

Elements of the greedy strategy

- A greedy algorithm makes a sequence of choices; each of the choices that seem best at the moment is chosen but not always produces an optimal solution.
- Two ingredients that are exhibited by most problems that lends themselves to a greedy strategy or characteristics:
  - Greedy-choice property: A global optimum can be arrived at by selecting a local optimum.
  - Optimal substructure: An optimal solution to the problem contains an optimal solution to subproblems.
- The second property may make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

Greedy Algorithms vs. Dynamic Programming

Greedy	Dynamic Programming
A greedy algorithm is one that at a given point in time, makes a local optimization.	Dynamic programming can be thought of as 'smart' recursion.,It often requires one to break down a problem into smaller components that can be cached.
Greedy algorithms have a local choice of the subproblem that will lead to an optimal answer	Dynamic programming would solve all dependent subproblems and then select one that would lead to an optimal solution.
A greedy algorithm is one which finds optimal solution at each and every stage with the hope of finding global optimum at the end.	A Dynamic algorithm is applicable to problems that exhibit Overlapping subproblems and Optimal substructure properties.
More efficient as compared,to dynamic programming	Less efficient as compared to greedy approach

Greedy algorithms and dynamic programming are similar; both generally work under the same circumstances although dynamic programming solves subproblems first. Often both may be used to solve a problem although this is not always the case.

Consider the 0-1 knapsack problem. A thief is robbing a store that has items 1..n. Each item is worth v_i dollars and weighs w_i pounds. The thief wants to take the most amount of loot but his knapsack can only hold weight W. What items should he take? This problem has optimal substructure.

Dynamic programming: We showed that we can solve this in O(nW) time, gives optimal value. Greedy algorithm: Take as much of the most valuable item first. Does not necessarily give optimal value!

Fractional Knapsack

A simpler version of the knapsack problem is solved optimally by this greedy algorithm: Consider the fractional knapsack problem. This time the thief can take any fraction of the objects. For example, the gold may be gold dust instead of gold bars. In this case, it will be hooves the thief to take as much of the most valuable item per weight (value/weight) he can carry, then as much of the next valuable item, until he can carry no more weight. Total value using this strategy and the above example is 8 of item 1 and 2 of item 2, for a total of $124.

In the ***fractional knapsack problem***, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0–1) choice for each item. You can think of an item in the 0–1 knapsack problem as being like a gold ingot, while an item in the fractional knapsack problem is more like gold dust.

Although the problems are similar, the fractional knapsack problem is solvable by a greedy strategy, whereas the 0–1 problem is not.

Moral: Greedy algorithm sometimes gives optimal solution, sometimes not, depending on the problem. Dynamic programming, when applicable, will typically give optimal solutions, but are usually tricker to come up with and sometimes trickier to implement.

Algorithm

Step 1: for i=1 to in steps of 1 do
X[i] = 0
End for loop

Step 2: weight = 0, i=1

Step 3: while weight < w do
If $W_i \leq W$ – weight then
i) X[i] = 1
ii) Weight = weight + W_i
Else
i) X[i] = (M – weight)/ W_i
ii) weight = M

return X

If the items are already sorted into decreasing order of V_i/ W_i then the while lop takes a time in O (n). Therefore the total time including the sort is O (nlogn).

Greedy solution for Fractional Knapsack

Given a set of item:

weight	w_1	w_2	. . .	w_n
cost	c_1	c_2	. . .	c_n

Let P be the problem of selecting items from I, with weight limit K, such that the resulting cost (value) is maximum.

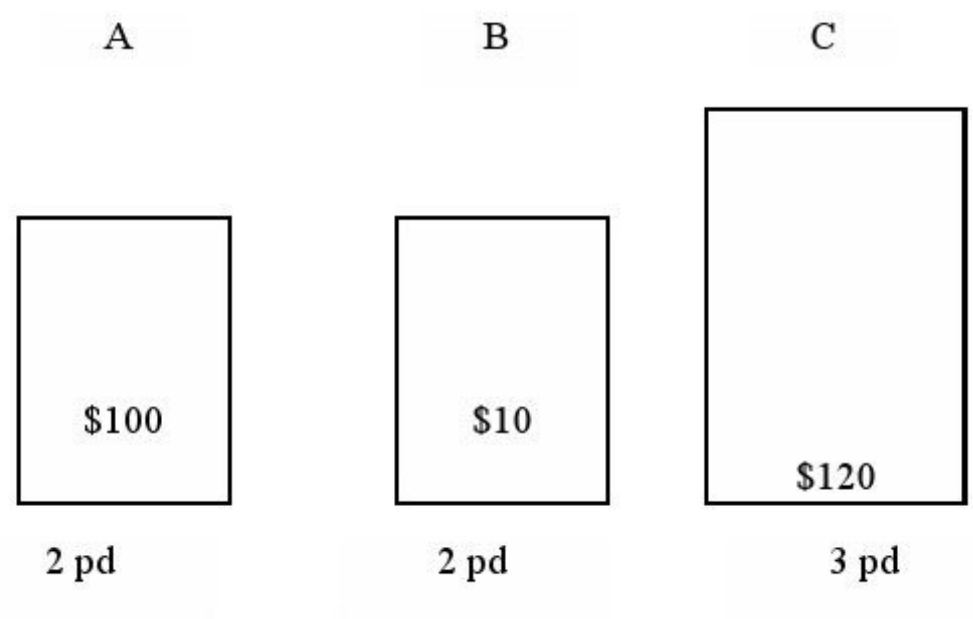
1. Calculate $v_i = \frac{c_i}{w_i}$ for $i = 1, 2, \ldots, n$.

2. Sort the items by decreasingv_i. Let the sorted item sequence be $1, 2, \ldots, n$, and the corresponding value and weight v_i and w_i respectively.

3. Let k *be* the current weight limit (Initially, k=K). In each iteration, we choose item i from the head of the unselected list. If $k \geq w_i$, we take item i, and $k = k - w_i$, then consider the next unselected item.

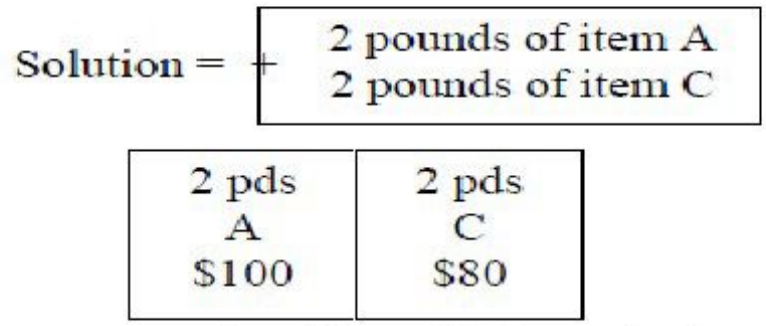
If $< w_i$, we take *a fraction f* of item i, i.e., we only take $f = \frac{k}{w_i}$ (< 1) of item i, which weights exactly k . Then the algorithm is finished.

Observe that the algorithm may take a fraction of an item, which can only be the last selected item. We claim that the total cost for this set of items is *an* optimal cost. Let us consider the same thief example. A thief enters a store and sees the following items:

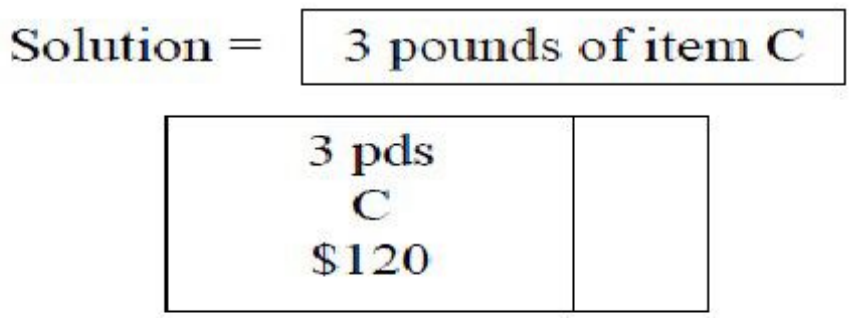


His Knapsack holds 4 pounds. What should he steal to maximize profit?

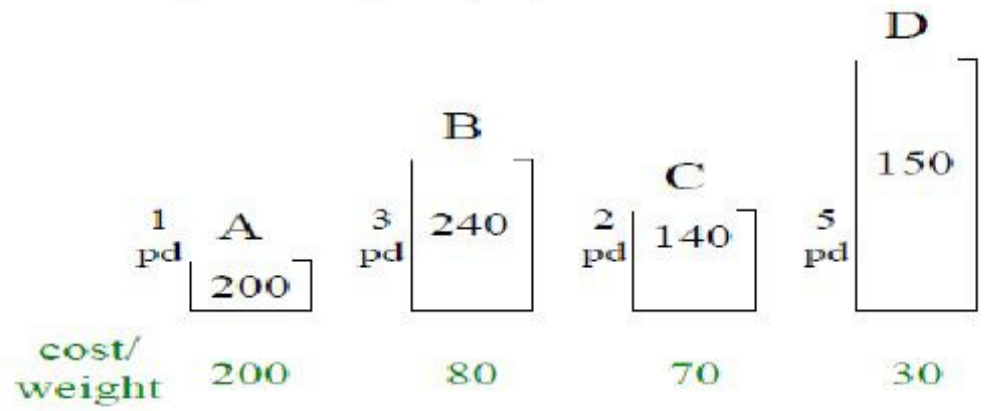
1. Fractional Knapsack Problem : Thief can take a fraction of an item.



2. 0-1 Knapsack Problem : Thief can only take or leave item. He can't take a fraction



3. Sort items by decreasing cost per pound



If knapsack *holds k = 5 pds*, solution is:

1 pds A
3 pds B
1 pds C

Huffman codes

Huffman codes are a widely used and very effective technique for compressing data; savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given as shown in below. That is, only six different characters appear, and the character a occurs 45,000 times.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

There are many ways to represent such a file of information. We consider the problem of designing a ***binary character code*** (or ***code*** for short) wherein each character is represented by a unique binary string. If we use a ***fixed-length code***, we need 3 bits to represent six characters: a = 000, b = 001, ... , f = 101. This method requires 300,000 bits to code the entire file. Can we do better?

A ***variable-length code*** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f.

This code requires $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix codes.***

Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with the variable-length prefix code. We code the 3-character file abc as $0 \cdot 101 \cdot 100 = 0101100$, where we use "$\cdot$" to denote concatenation.

Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string 001011101 parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$, which decodes to aabe.

The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Figure 9 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

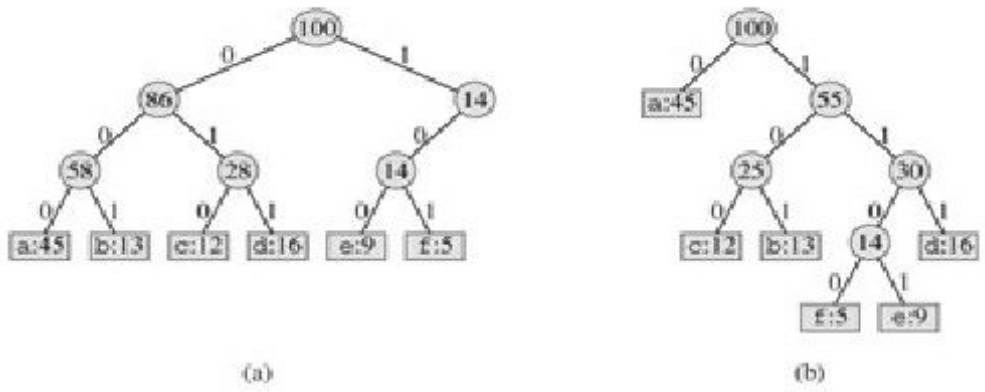


Figure 9: Example of Huffman code

Figure 9: Trees corresponding to the coding schemes as shown in above table. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. *(a)* The tree corresponding to the fixed-length code a = 000, ..., f = 101. *(b)* The tree corresponding to the optimal prefix code a = 0, b = 101, ..., f = 1100.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children. The fixed-length code in our example is not optimal since its tree, is not a full binary tree.

Given a tree T corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file. For each character c in the alphabet C, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus,

$$B(T) = \sum_{c \in C} f(c) d_T(c) ,$$

which we define as the ***cost*** of the tree T.

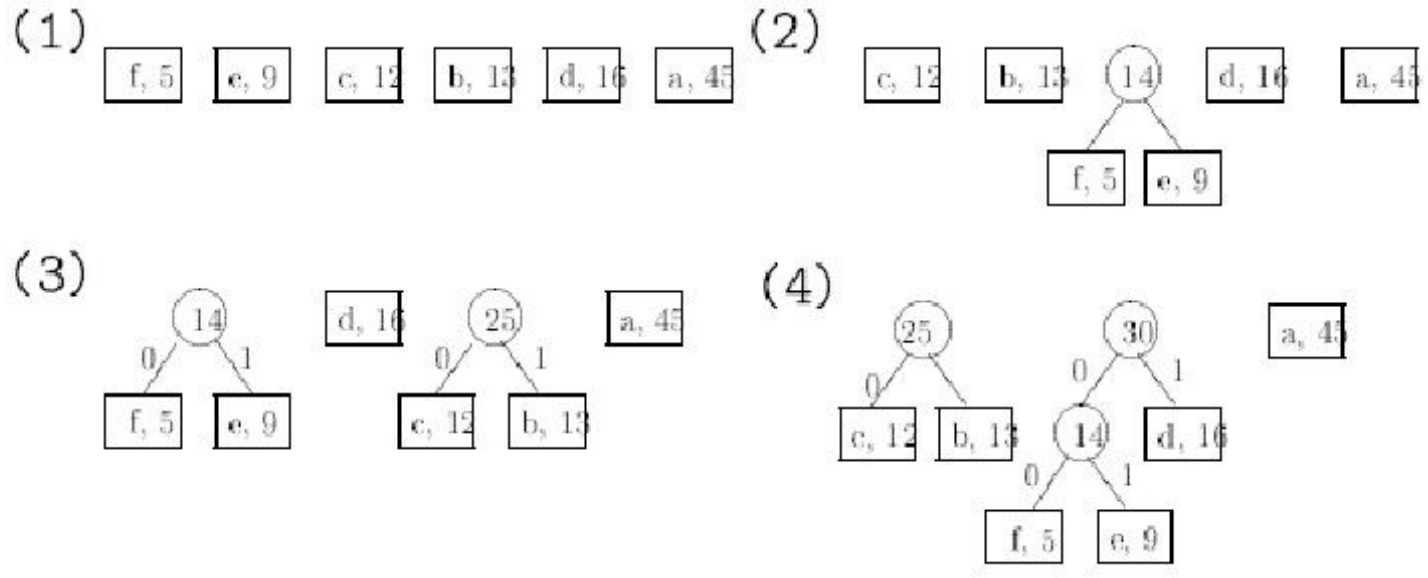
Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a ***Huffman code***. In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree. A min-priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN(C)
1 n ← |C|
2 Q ← C
3 for i 1 to n - 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN (Q)
```

6 $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN (Q)
7 $f[z] \leftarrow f[x] + f[y]$
8 INSERT(Q, z)
9 **return** EXTRACT-MIN(Q) (Here, return the root of the tree.)

The total running time of HUFFMAN on a set of n characters is O (n lg n).Continue the same example can be represented by using Huffman's algorithm as shown in figure 10. Since there are 6 letters in the alphabet, the initial queue size is $n = 6$, and 5 merge steps are required to build the tree. The final tree represents the optimal prefix code. The codeword for a letter is the sequence of edge labels on the path from the root to the letter.



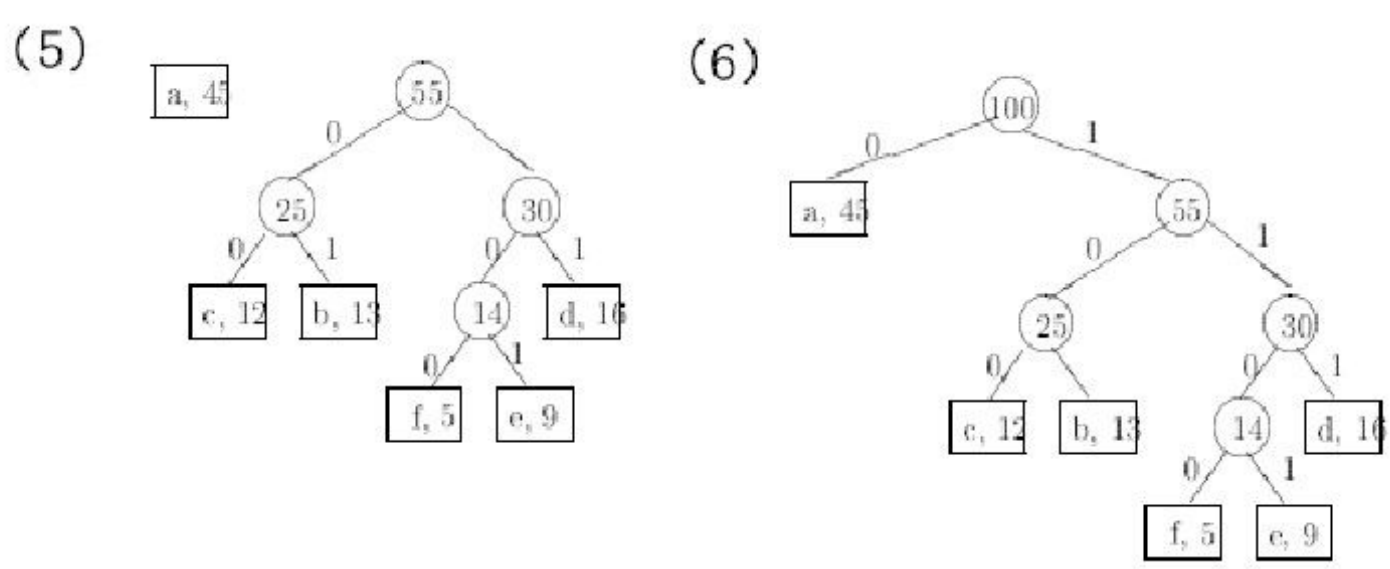


Figure 10: Huffman algorithm using variable length code.