Greedy Algorithms: Knapsack and Huffman Codes

Chapter 16.2, 16.3

Greedy Algorithms

For Activity Selection, we followed the procedure:

1. Determine optimal substructure
2. Develop a recursive solution
3. Prove that for any subproblem, one of the optimal choices is the greedy choice.
4. Show that all but one of the subproblems resulting from having made the greedy choice are empty.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

That's a lot of steps. In general, we can design greedy algorithms more simply.

Greedy design procedure

1. Express the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is always safe.
3. Show that the subproblem remaining after a greedy choice is such that if we combine an optimal solution to that subproblem with the greedy choice we made, we get an optimal solution to the original problem. This is optimal substructure applied to the greedy method.

Greedy-choice property

The Greedy-choice property is that a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

So in greedy algorithms we are making the choice that looks best in the current problem, without considering results from subproblems. This differs from dynamic programming.

Often proving the greedy-choice property involves proving you can modify an optimal solution to a subproblem to use the greedy choice, resulting in a solution that is just as good (or better!) than the optimal solution.

0-1 Knapsack problem

There is a collection of n items, with item i being worth v_i dollars and weighing w_i kilograms. You have a knapsack that will hold W kilograms of weight. Which items should you place in the knapsack to maximize the value of the items carried?

Here an item must either be taken (1) or not taken (0); you cannot take part of an item or take an item more than once.

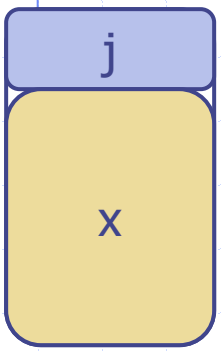
Fractional knapsack problem

There is a collection of n items, with item i being worth v_i dollars and weighing w_i kilograms. You have a knapsack that will hold W kilograms of weight. Which items or parts of items should you place in the knapsack to maximize the value of the items carried?

Here a fraction f of an item can be taken, $0 \leq f \leq 1$. The item could be "a kilogram of flour" and you might take "1/3 of a kilogram of flour".

Knapsack problems: optimal substructure

Both knapsack problems have the optimal substructure property. Consider the most valuable load that weighs at most W kilos.



- In the 0-1 problem, if we remove item j from this load, the remaining load must be the most valuable load weighing at most W - w_j made from all items except j.
- In the fractional problem, if we remove the fraction f_j of item j that was in the load, the remaining load must be the most valuable load weighing at most W – $f_j w_j$, taken from all items except j and (1-f_j) of item j.

If not, take the most valuable load of the remaining problem and add the removed amount of item j to it, getting an even more valuable load. This is a contradiction.

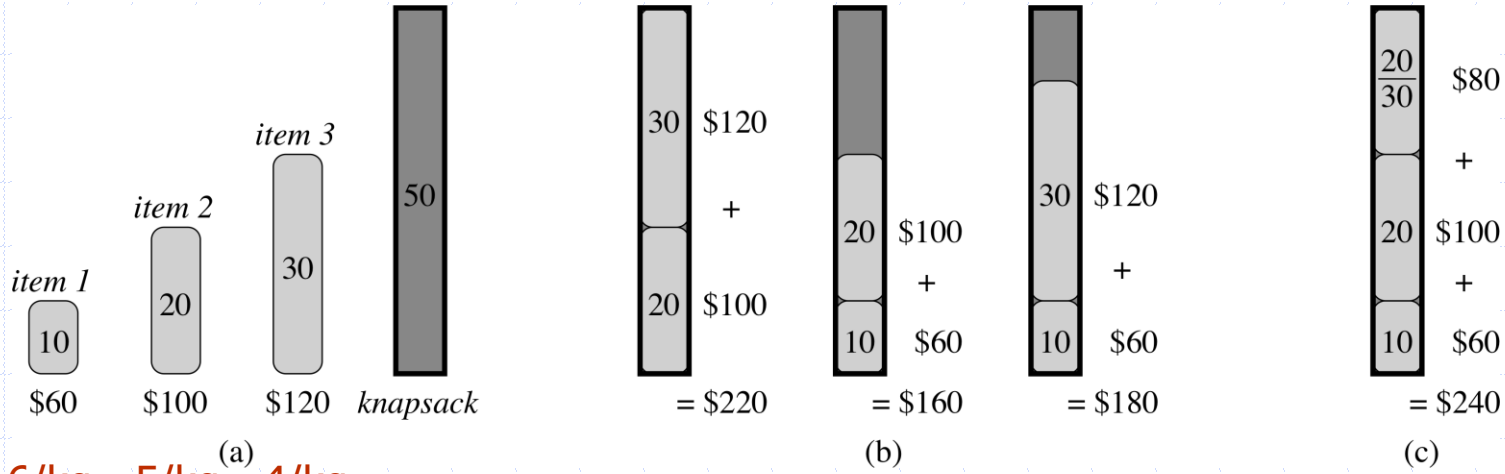
Knapsack problems: the greedy approach

The 0-1 knapsack problem is not amenable to the greedy approach, but the fractional knapsack problem is.

To solve the fractional problem, first compute the value per kilogram v_i/w_i for each item. Assume the items are sorted by this value: item 1 has the highest v_i/w_i, etc. Then take as much of the first item as the knapsack will hold. If the knapsack holds all of the first item, add as much of item 2 as will fit. Then as much of item 3 as will fit, etc.

0-1 Knapsack

The 0-1 knapsack problem is not amenable to the greedy approach.

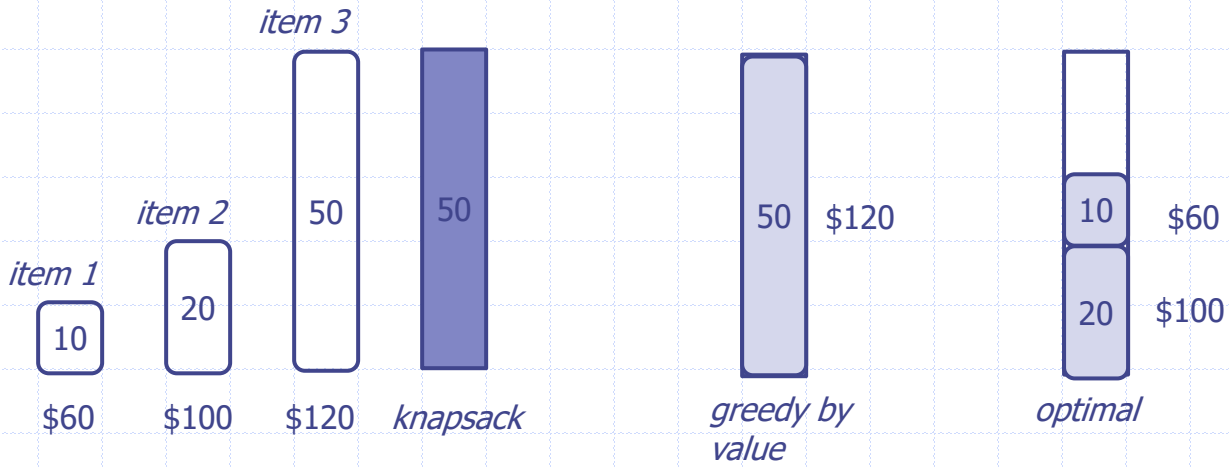


v_i/w_i 6/kg 5/kg 4/kg

(greedy by value/weight)

0-1 Knapsack

The 0-1 knapsack problem is not amenable to the greedy approach.



(greedy by value)

But 0-1 Knapsack also has overlapping subproblems, so DP still works on it.

Huffman codes

Huffman codes are an effective data compression technique, giving size savings from 20% to 90% depending on the type of data.

We consider the data to be a sequence of characters (or bytes).

Huffman's technique uses a table of the frequencies of occurences of each character to build an optimal way of representing each character as a binary string.

Fixed-length and variable-length codes

Suppose we have file of 100,000 bytes, but only 6 different bytes occur in it. (This happens quite a lot, say, in computational biology.) We want to design a binary code for it: each character will be represented by a unique binary string.

- A fixed-length code has every character code the same length. In our example, we'd need 3 bits/character.
- A variable-length code allows the character codes to be different lengths. We could have some characters be represented by 2 bits, some by 3, etc.

A Huffman code is a variable-length code. It does considerably better than a fixed-length code.

Prefix codes

Huffman codes have the property that no codeword is a prefix of another codeword. Such codes are called prefix codes. Optimal data compression can be achieved with a prefix code.

Suppose we have the simple prefix code a:0, b:101, c:100. Then we would encode abc as $0 \cdot 101 \cdot 100 = 0101100$, where we use "$\cdot$" to denote "followed by" (concatenation).

This encoding step is the same even if the code is not a prefix code.

Prefix codes

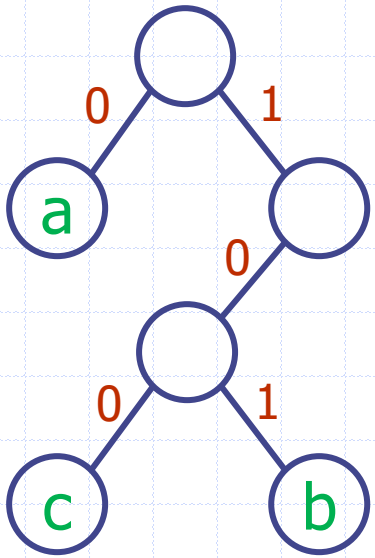
Prefix codes simplify decoding. Since no codeword is a prefix of any other, the codeword first detected at the start of a file is unambiguous. We can identify this codeword, translate it back to the original character, and repeat the decoding process on the part of the file that remains.

In our example (a:0, b:101, c:100), the string 10101010100 breaks down uniquely as 101 · 0 · 101 · 0 · 100, decoding to babac.

Prefix codes as trees

It is convenient to represent a prefix code as a binary tree, where the leaves are the given characters. The codeword for a leaf is the path from the root to that leaf, where 0 means "left child" and 1 means "right child".

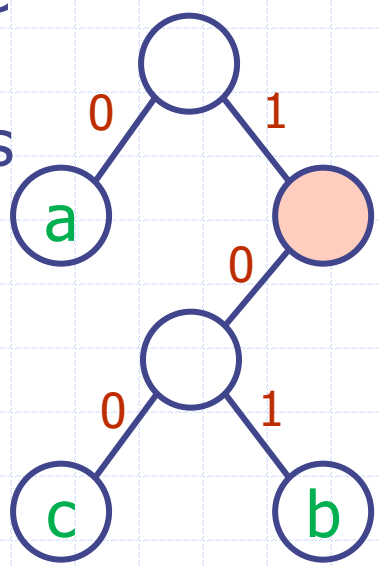
Our example code (a:0, b:101, c:100) has the tree shown.



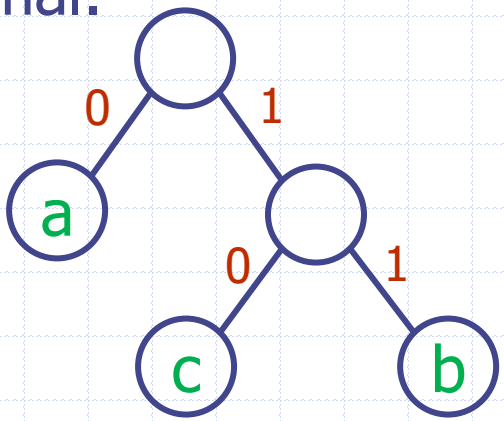
Optimal prefix codes

Optimal prefix codes are always represented by a full binary tree, where by full we mean each nonleaf node has two children.

Our example tree is not optimal. The red node has only 1 child.



Here's a better tree. It may or may not be optimal.



The cost of a code or tree

Suppose we are given a file F on alphabet C. For each character c $\in$ C, let f(c) denote the frequency of that character in the file.

Let T be a tree corresponding to a prefix code for C. For each character c $\in$ C, let $d_T(c)$ denote the depth of c's leaf in the tree. $d_T(c)$ is also the length of the codeword for c.

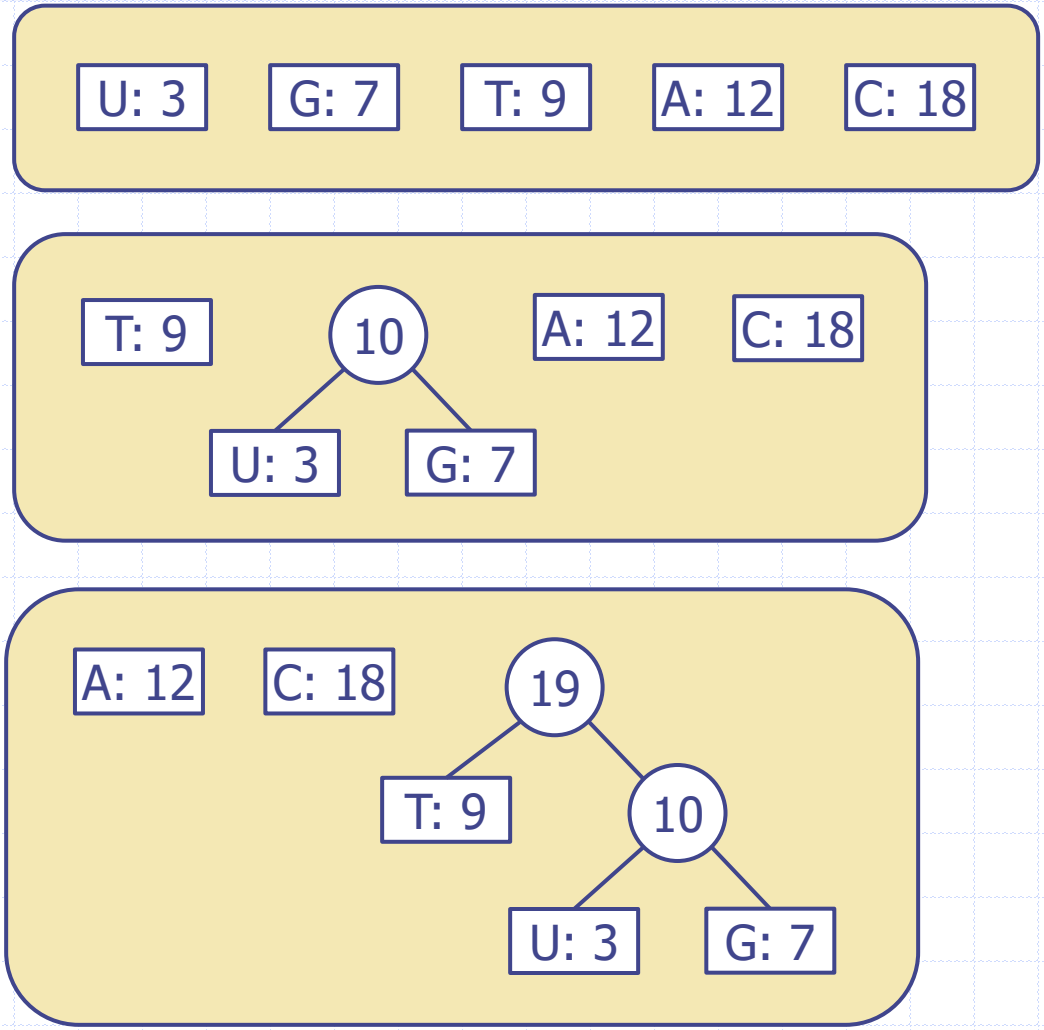
We define the cost of the tree for the file as:

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

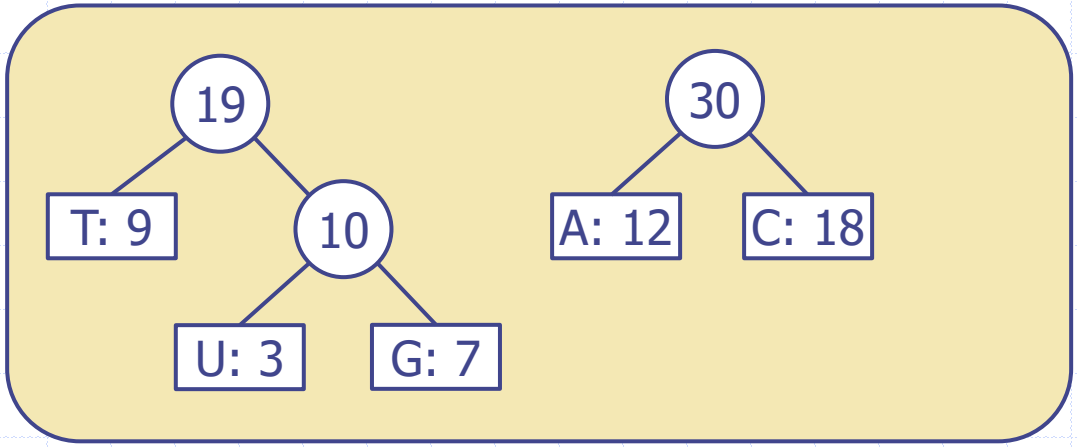
Huffman codes

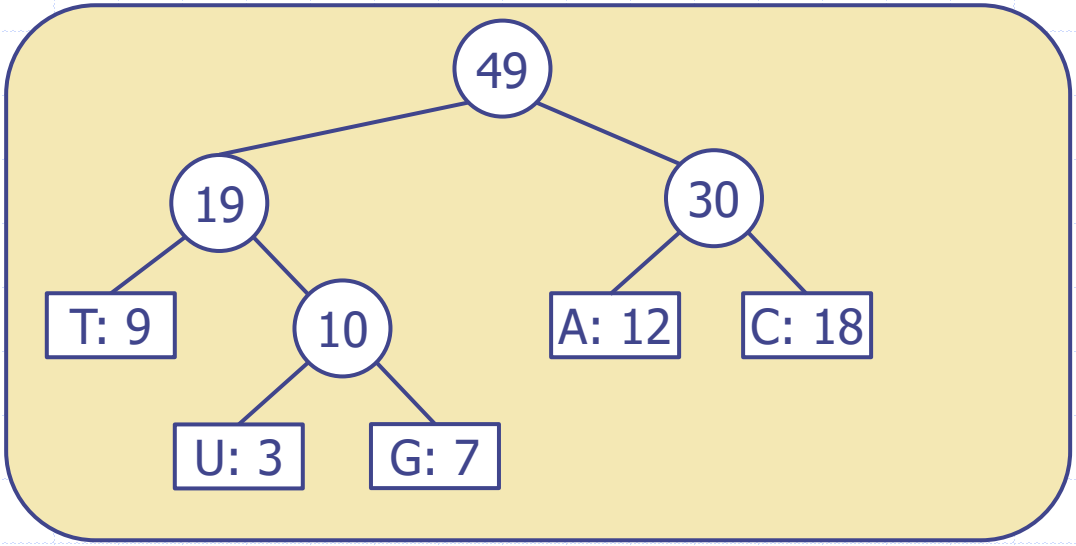
```
HUFFMAN(C, f)                          // C is alphabet, f frequencies
create new min-priority-queue Q
for each element c of C
        Q.INSERT(c, f[c])              // insert c with key f[c]

while(Q.size() > 1)
        allocate a new tree node z
        left[z] = Q.EXTRACT-MIN()
        right[z] = Q.EXTRACT-MIN()
        f[z] = f[left[z]] + f[right[z]]
        Q.INSERT(z, f[z])              // insert z with key f[z]

return Q.EXTRACT-MIN()
```

Huffman code example



Huffman code example





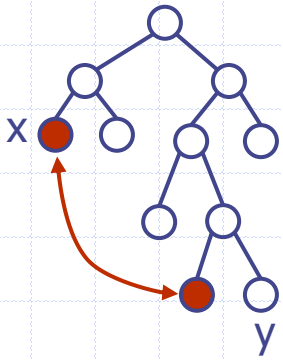
Code:
T: 00
U: 010
G: 011
A: 10
C: 11

Correctness: Greedy-choice Property

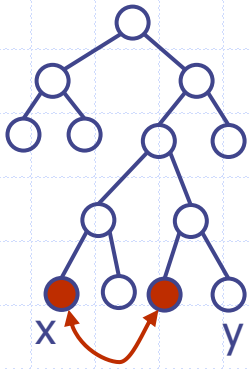
Lemma: Let C be an alphabet with each character c having frequency f[c]. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ by only one bit.

Proof (sketch):

Let T be an optimal tree.



If x is not at bottom level, switch it with a non-y vertex that is. Similarly for y.

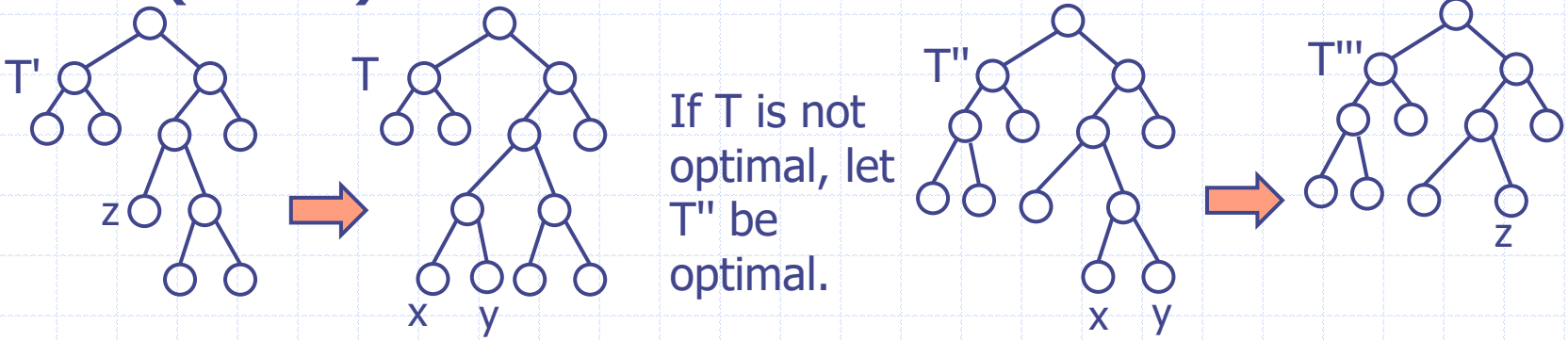


Both x and y are at bottom level. Switch x with y's sibling.

Correctness: Optimal Substructure

Lemma: Let C be an alphabet with each character c having frequency f[c]. Let x and y be two characters in C having the lowest frequencies. Let C' = C – {x,y} + {z} with f[z] = f[x] + f[y]. If T' is an optimal tree for C', then replacing z in T' with an internal node having the children x and y yields a tree T that is optimal for C.

Proof (sketch):



B(T) = B(T') + f[x] + f[y]
B(T') = B(T) – f[x] – f[y]

B(T''') = B(T'') – f[x] – f[y]
< B(T) – f[x] – f[y]
= B(T') contradiction

Correctness

Theorem: Procedure HUFFMAN produces an optimal prefix code.

Textbook note: You are not responsible for sections 16.4 and 16.5 (Matroids and Task scheduling).