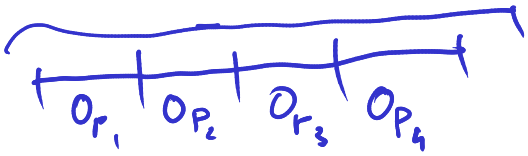
Lecture 12: Greedy Algorithms

MCS/CS 401 Computer Algorithms I — July 13, 2022
Instructor: Danko Adrovic

---

## Lecture Outline

- Greedy Algorithms
  - Components of the greedy strategy
  - Greedy method for data compression: Huffman codes and algorithm
  - Constructing a Huffman code
  - Greedy vs. dynamic programming
  - Limitations of the greedy strategy

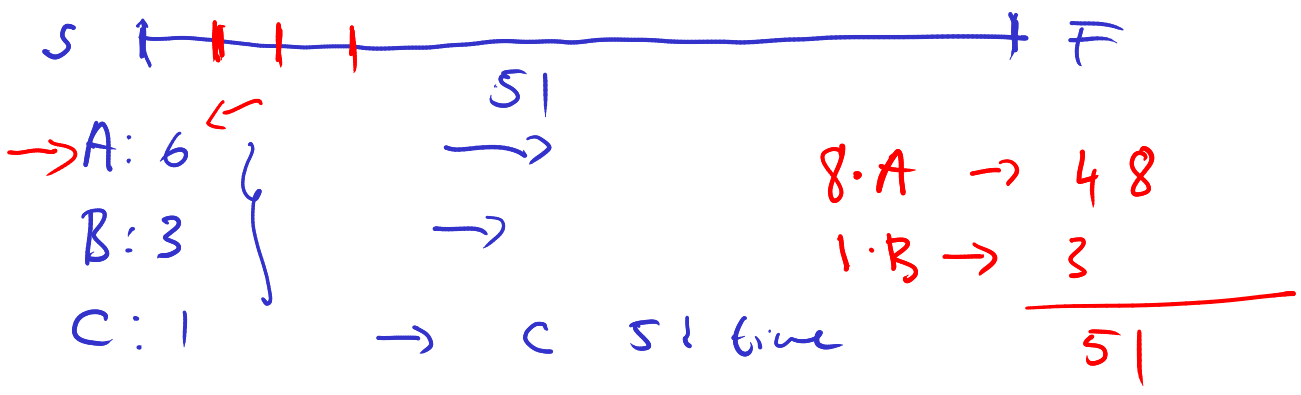



## 1 Greedy Algorithms

The algorithms that solve optimization problems usually go through a sequence of steps, making a set of choices at each step. We have already covered dynamic programming based algorithms but, often, they do more work than it is necessary to solve an optimization problem: recall that, in the case of *rod-cut problem*, the algorithm had to go through all cut configurations in order to find the optimal solution.

A different kind of algorithm, called **greedy algorithm**, approaches an optimization problem by selecting a solution, which is the best solution at that stage or moment. Then, at the next stage, it selects again the best solution available at that stage. Using this process of selecting a *locally* best solution, the *greedy algorithm* uses the idea that a sequence of *locally* best solution will lead to the best or optimal **global** solution.

A *greedy algorithm* does not always find the optimal solution for a problem. Certain problems have such a structure that, for example, *dynamic programming* based algorithms are better suited for. However, other problems are susceptible to a **greedy** approach and *greedy algorithm* will find an optimal solution to such problems. For example, scheduling, minimum-spanning-tree, or shortest path from a single source problems are all typically solved using the *greedy* method. Greedy method is also used in data compression algorithms, such as Huffman's algorithm, where the compression effectiveness of 20% to 90% occur on on a regular basis, depending on the type of data that is being compressed.



### 1.1 Components of the greedy strategy

In our approach to solve the activity selection problem we performed the following steps, which are not typical, as they were more involved than it is generally the case.

- Determine the optimal substructure of the problem
- Develop a recursive solution
- Show that if we make the greedy choice, then only one subproblem remains
- Prove that it is always safe to make the greedy choice
- Develop a recursive algorithm that implements the greedy strategy
- Convert the recursive algorithm to an iterative algorithm

Typically, the following set of steps are more frequently applied as they are more general in nature

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve
- Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe
- Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

There are two additional properties, which are essential for the greedy method, which worth emphasizing. First, **greedy-choice property**, which allows us to assemble a globally optimal solution by making locally optimal (greedy) choices. Second, **optimal substructure**, which a problem exhibits if an optimal solution to that problem contains within it optimal solutions to subproblem. The second property holds for both approaches, greedy and dynamic programming.

### 1.2 Greedy method for data compression: Huffman codes and algorithm

Huffman algorithm is a data compression algorithm which, using a symbol frequency stored in a table, generates codes, called Huffman codes, to generate an optimal way of representing each symbol as a **binary** string. As an illustrative example, assume that we have a file consisting of 100,000 characters (= symbols) that we want to store compactly. We note that the characters in the file occur with frequency, given in the following table:

| | 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 |

Huffman code

$\lceil \lg(6) \rceil = \lceil 2.58 \rceil = 3$

We also note that we have just 6 characters and if we wanted to store the 6 characters using 1 or 0 bits, we could accomplish it using $\log_2(6) \approx 2.58 \approx 3$ bits. The 3 bit representation is given in the table on line called *Fixed-length codeword*. This means that we could compress the entire 100,000 character file using 300,000 bits. The question is, can we compress the file more efficiently than 300,000 bits? The answer is yes, using what is labeled in the table *Variable-length codeword*.

The variable-length codewords are the Huffman codes, that is, binary representation of the characters that use the least amount of bits to uniquely represent characters. In particular, Huffman code is a **prefix** code, that is, codes in which no codeword is also a prefix of some other code word. We will state it here without proof that a prefix code can always achieve the optimal data compression among any character code.

Encoding using Huffman codes is simple and it uses basic concatenation. In our table, the variable-length codes for a = 0, b = 101, and c = 100, so *encoding* a word "abc" results in $0 \cdot 101 \cdot 100 = 0101100$. If you wanted to *decoded* the message, you would immediately see the importance of a prefix code in such a process! Decoding any word that was encoded using a prefix code can be achieved without ambiguity, resulting always in a unique result.

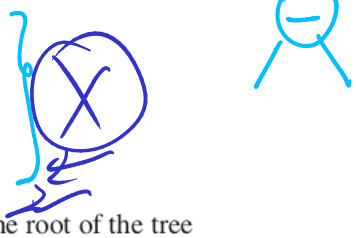
### 1.3 Constructing a Huffman code

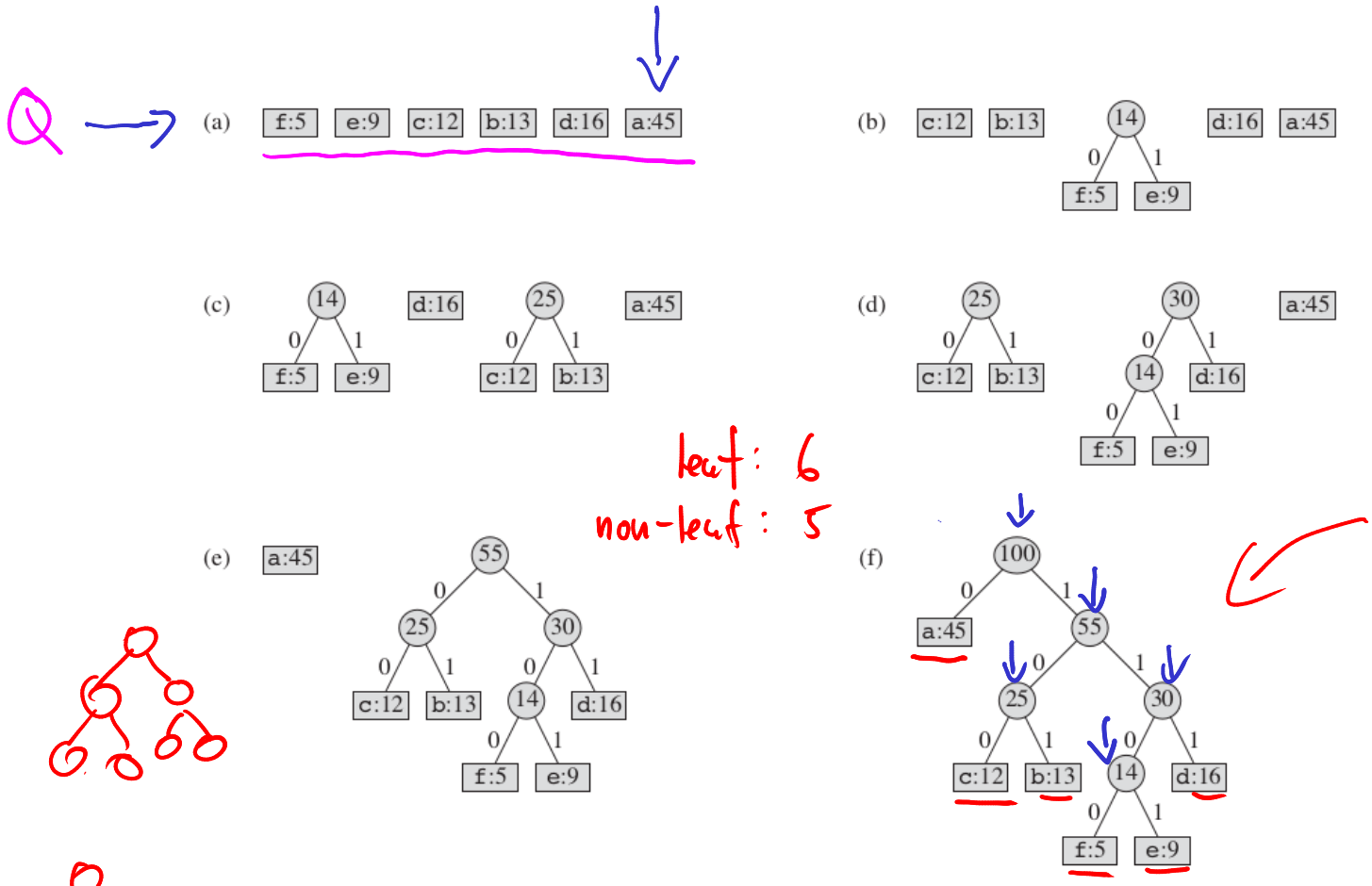
The following algorithm (Huffman) is a greedy algorithm that constructs an optimal prefix code called a Huffman code. As its input, it takes a C, a set of n characters and each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency.

The algorithm builds the tree T, representing 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. The algorithm uses a min-priority queue $Q$, keyed on the $freq$ attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

$C = \{a, b, c, d, e, f\} = |C| = 6$

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```






The function *EXTRACT-MIN* was covered in Chapter 6 when we covered **Heaps**, in particular when they are used in connection with queues, and in this case, a min-priority queue. A min-priority queue is implemented using min-heaps. Recall also that heaps are represented as arrays, which are often graphically represented as trees, much like what you see in the last image above.

The *greedy* strategy aspect of the Huffman algorithm involves keeping the least frequently occurring characters at or closer the bottom of the tree. Consequently, the more frequently occurring elements are forced to appear near or at the top of the tree, with the character "a", the most frequent of them, being located near the very top of the tree structure.

To analyze the running time of Huffman's algorithm, we assume that $Q$ is implemented as a binary min-heap. For a set $C$ of $n$ characters, we can initialize $Q$ (line 2) in $O(n)$ time using the BUILD-MIN-HEAP procedure (Chapter 6). The **for** loop (lines 3-8) executes exactly $n - 1$ times, and since each heap operation requires $O(lg(n))$ time, the loop contributes $O(n\ lg(n))$ to the running time. Therefore, the total running time of HUFFMAN algorithm on a set of $n$ characters is $O(n\ lg(n))$.

Using what are called *Emde Boas tree* instead of a *binary min-heap*, the running time can be reduced to $O(n\ lg(lg(n)))$.

The sub image (f) above, gives us the Huffman prefix code, the variable-length codeword in our frequency table. Using Huffman code, we can compress the original file using $1000 \cdot (1 \cdot 45 + 3 \cdot 13 + 3 \cdot 12 + 3 \cdot 16 + 4 \cdot 9 + 4 \cdot 5) = 224,000$ bits, a reduction of 25%.

**Proposition 1.** *If the characters in an alphabet are ordered so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.*

*Proof.* Proof by contradiction. Suppose that there are two codewords $c_1$ and $c_2$, where $c_1$ has a higher frequency but it has a longer codeword than $c_2$. This implies that $c_1$ has been merged more frequently than $c_2$, due to its longer code. Because we merge two elements with the lowest combined frequency, this leads to a contradiction as $c_1$ has a higher frequency than $c_2$. □

**Proposition 2.** *An optimal prefix code on a set $C = \{0, 1, ..., n-1\}$ of characters can be represented using only $2n - 1 + n\lceil lg(n) \rceil$ bits.*

*Proof.* A tree is full if every node that is not a leaf node has exactly two children. The associated full binary tree T of an optimal prefix has 2n-1 nodes (n leaf nodes, and n-1 internal nodes). By performing a pre-order traversal of the tree T, we can encode the structure of the tree in the following way: for every node visited in the pre-order traversal, write 0 if the node is an internal node, and write 1 if the node is a leaf node. Because the tree T is a full binary tree, the pre-order traversal is unique, and hence the encoding. Furthermore, recall there are n characters in C and that each character can be encoded using $\lceil lg(n) \rceil$ bits. Therefore, we have $n\lceil lg(n) \rceil$ bits (for the characters) plus $2n - 1$ bits (to specify the structure of the tree). □

### 1.4 Greedy vs. dynamic programming

Frequently, *dynamic programming* is the better option for a problem than the greedy strategy. Generally speaking, it can be tricky to determine which of these two options is better suited for a problem. This is mostly due to the fact that both strategies exploit *optimal substructure* property in which an optimal solution to the problem contains within it optimal solutions to subproblems. Therefore, a careful analysis is needed to avoid applying the wrong strategy.

The steps listed in the Section *Components of the greedy strategy* can be, generally, used to determine the applicability of the greedy strategy. However, the minimalist nature and its simplicity make the greedy strategy very appealing over the more complicated dynamic programming strategy.

### 1.5 Limitations of the greedy strategy

The *greedy strategy* may not always be applicable to solve a problem. As an illustration, consider the following problem, called the **0-1 knapsack problem**.

*A thief robbing a store finds n items. The ith item is worth $v_i$ dollars and weighs $w_i$ pounds, where $v_i$ and $w_i$ are integers. The thief wants to take as valuable a load as possible, but he can carry at most $W$ pounds in his knapsack, for some integer $W$.*

Which items should the thief take? We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once. We will use this problem to illustrate that the

greed strategy does not always work.

Consider three items: item 1 (worth \$60, weighs 10 pounds), item 2 (worth \$100, weighs 20 pounds), item 3 (worth \$120, weighs 30 pounds), and one knapsack capable of holding 50 pounds.

```
Item 1  $60,  10 lb,  60/10 = 6 $/lb value
Item 2  $100, 20 lb, 100/20 = 5 $/lb value
Item 3  $120, 30 lb, 120/30 = 4 $/lb value
```

Greedy strategy dictates that the thief take most valuable item, namely Item 1, as it has the largest dollar per pound value. However, as the following image (b) shows, any selection of Item 1 will lead to the sub-optimal solution! This shows that for the $0-1$ knapsack problem, the greedy method does not work.

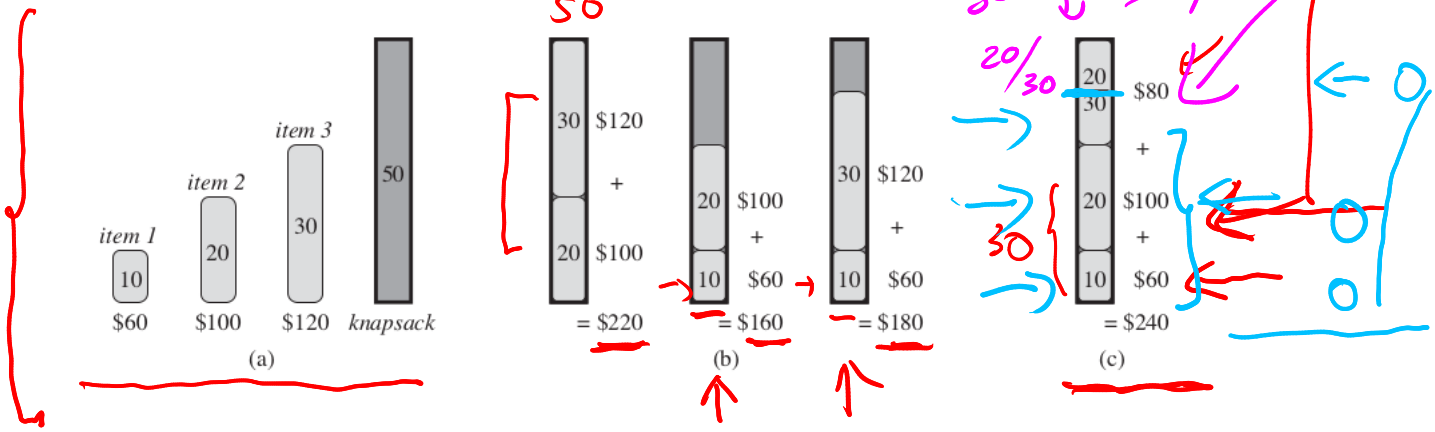



**Figure 16.2** An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

The reason for failing to reach optimal solution with greedy strategy is that the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load. In the 0-1 problem, when we consider whether to include an item in the knapsack, we must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before we can make the choice. The problem formulated in this way gives rise to many overlapping subproblems. Any time we have an overlapping of subproblems, dynamic programming is the method to use.

Consider now an related problem, called **fractional knapsack problem**. In the fractional version 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. We can think of an item in the 0-1 knapsack problem as being like a gold chunk and an item in the fractional knapsack problem as more like gold dust.

In the *fractional knapsack problem*, the greedy strategy works! Using the greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as

possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W. This is illustrated in the picture (c) above.

The greedy-choice property states that we can assemble a globally optimal solution by making locally optimal (greedy) choices. Prove the following statement.

**Proposition 3.** *Fractional knapsack problem has the greedy-choice property.*

*Proof.* Let the optimal solution of the fractional knapsack problem have the highest total density value (like \$/*lb* in problem above). Add as much of the element with the highest total density value to the solution as is available. When that is exhausted, start adding as much of the element with the next highest total density value to the solution as is available, and so on. This approach always makes locally optimal solution, and builds the global solution using only optimal solutions. Therefore, the global solution consists of only optimal choices and is, therefore, optimal. Hence, the fractional knapsack problem has the greedy-choice property. □