Lecture 11: Greedy Algorithms

MCS/CS 401 Computer Algorithms I — July 11, 2022
Instructor: Danko Adrovic

Lecture Outline

- Greedy Algorithms
  - Activity-selection problem
  - The greedy approach to solve the problem
  - Recursive greedy algorithm
  - Running time of RECURSIVE-ACTIVITY-SELECTOR
  - Iterative greedy algorithm
  - Components of the greedy strategy
  - Greedy method for data compression: Huffman codes and algorithm
  - Constructing a Huffman code

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 Activity-selection problem

The following problem consist of of scheduling several competing activities that require *exclusive* use of a common resource. The optimal solution to this problem consists of selecting a maximum-size set of mutually compatible activities, that is, activities whose time duration do not overlap.

For our problem setup, assume we have a set $S = \{a_1, a_2, \ldots, a_n\}$ consisting of n proposed activities that wish to use a single resource, such as a lecture hall. The resource, the lecture hall, can serve only one activity at a time. Each activity a_i has a fixed start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. Any selected activity a_i takes place during the *half-open* time interval $[s_i, f_i)$. Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. In other words, activities a_i and a_j are **compatible** if $s_i \geq f_j$ or $s_j \geq f_i$. In addition, we will assume that the activities are sorted in monotonically increasing order of the *finish time*: $f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$. This assumption on sorting will provide as with a great advantage, as we will see later.

The goal of the **activity-selection problem** is to select a **maximum-size subset** of mutually compatible activities. As our running example, we will use the following set S of activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Figure 1: Activities of set S, sorted according to the finish time f_i.

In this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities, however, it is not a *maximum* subset as the subset $\{a_2, a_4, a_9, a_{11}\}$ is larger. Later, using the algorithm we are developing, we will discover that the optimal solution to this problem is the subset $\{a_1, a_4, a_8, a_{11}\}$. Note that subsets $\{a_2, a_4, a_9, a_{11}\}$ and $\{a_1, a_4, a_8, a_{11}\}$ are both equally optimal, it's just that, due to the sorting, that $\{a_1, a_4, a_8, a_{11}\}$ will selected. This brings us to another aspect of greedy algorithms: they find an optimal solution but the solution may not be unique.

1.2 The greedy approach to solve the problem

If we were to apply the *dynamic programming* approach, we would analyze the subproblem structure and design a recursive algorithm with memoization. We could also directly apply the bottom-up approach in an iterative fashion, filling up the array entries as we move along, to solve the original problem. This approach, however, would overlook a key property of greedy algorithms - **the greedy choice:** *the first activity to finish is always part of an optimal solution*! Now you can see why the assumption that activities are sorted is so important.

What is the greedy choice for the activity-selection problem? Intuitively, we think that we should choose an activity that leaves the resource available for as many of the other activities as possible. Of the activities we end up choosing, one of them must be the first

one to finish. Because of this, our intuition tells us to select then the one which finishes first right away, that is, choose the activity in S with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible. (If there is a tie between choices that finish first, choose any such activity among them). In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1 by default.

One very appealing aspect of making the *greedy choice* is that we have only one remaining subproblem to solve, namely that of *finding activities that start after* a_1 *finishes*. Note that we do not have to consider activities that finish before a_1 starts, since we have that $s_1 < f_1$, and f_1 is the earliest finish time of any activity. Therefore, no activity can have a finish time less than or equal to s_1. Consequently, all activities that are compatible with activity a_1 must start after a_1 finishes.

Intuition is nice and good but how do we know that our intuitive reasoning has lead us to the correct, or in this case, optimal solution? Is the first activity to finish always part of some optimal solution? Yes, it is.

Theorem 1. *Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k .*

Proof. Let A_k denote a maximum-size subset of mutually compatible activities in S_k, and let a_j denote the activity in A_k with the earliest finish time. If $a_j = a_m$, then we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A_k^{'} = A_k \setminus \{a_j\} \cup \{a_m\}$, (that is $A_k^{'}$ is the set A_k in which we have replaced a_j with a_m). The activities in $A_k^{'}$ are disjoint since the activities in A_k are disjoint (that is, because A_k is a subset of mutually compatible activities). In addition, we have a_j is the first activity in A_k to finish, and $f_m \leq f_j$. The replacement of a_j with a_m did not change the the size of the set $A_k^{'}$, that is, $|A_k^{'}| = |A_k|$. With $|A_k^{'}| = |A_k|$, we conclude that $A_k^{'}$ is maximum-size subset of mutually compatible activities of S_k and $A_k^{'}$ includes a_m as an element. □

With the proof that the first activity to finish is always part of an optimal solution, we do not need to come up with the dynamic programming algorithm to solve the activity-selection problem. The greedy method is simpler to construct. In fact, it does not need to perform a bottom-up run, like a dynamic programming-based algorithm would do. It simply performs top-down run, choosing an activity to put into the optimal solution before solving the subproblem in the same way, only making sure that next chosen activity (in the subproblem) is compatible with those that have already been chosen.

This top-down approach of a greedy algorithm is typical: make a choice, then solve the subproblem. This is in contrast to the bottom-up approach applied to by dynamic programming based algorithm, where solving of subproblems before making a choice takes place.

1.3 Recursive greedy algorithm

Next we want to put all this information together and build a recursive greedy algorithm to solve the *activity-selection problem* in an optimal way. We will call this algorithm

RECURSIVE-ACTIVITY-SELECTOR.

Let s denote the start time and let f denote the finish time of the activities. In practice, s and f will be represented by *arrays*. Let the index k define the subproblem S_k to be solved and let n denote the size of the original problem (ie. the number of activities). In addition, we will assume that the activities are already sorted according to their finish time. If they are not, we will sort them, using $O(n\ lg(n))$ time.

In order for the algorithm *RECURSIVE-ACTIVITY-SELECTOR* to start running, we will introduce a fictitious activity a_0 and its finish time $f_0 = 0$, so that the subproblem S_0 is the entire set of activities S. This initial call is performed by *RECURSIVE-ACTIVITY-SELECTOR(s,f,0,n)*, which will solve the entire problem.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

1.4 Running time of RECURSIVE-ACTIVITY-SELECTOR

If we assume that the activities have already been solved, the running time of the *RECURSIVE-ACTIVITY-SELECTOR* is $\Theta(n)$: for every recursive call the algorithm makes, each activity is checked exactly once inside the while loop. Specifically, an activity a_i is checked in the last call made in which $k < i$. The index variable i is bounded by $1 \leq i \leq n$.

1.5 Iterative greedy algorithm

It is relatively easy to convert the recursive algorithm *RECURSIVE-ACTIVITY-SELECTOR* to an iterative version, based on what we have observed so far. For example, note that, after sorting, the resulting first activity a_1 is always part of the solution by default. This is a direct consequence of the Theorem we proved earlier. In addition, the inclusion into the optimal solution set A is permitted only for the activities which do not overlap with those already in the set A (line 5, where s[m] is the start time of the activity being considered and f[k] is finish time of the last activity added to A).

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

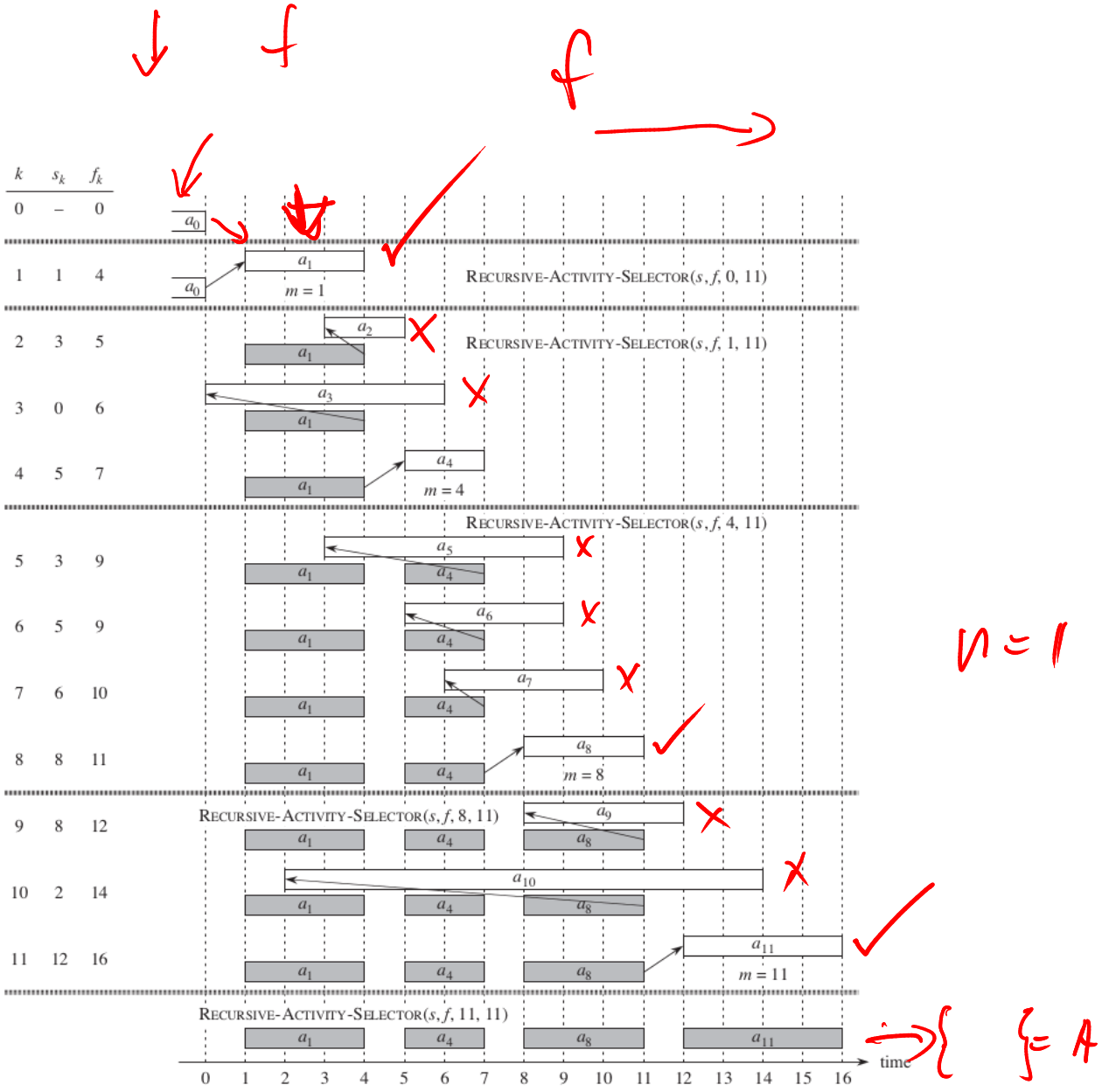


Figure 2: Run of the RECURSIVE-ACTIVITY-SELECTOR on the set S.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Figure 3: Activities of set S, sorted according to the finish time f_i.

Given the description of the iterative version of the algorithm for activity selection, determining the running time is equally straight-forward. With a single for loop, where all other operations take constant time, the running time is again $\Theta(n)$. It's Θ, rather than just O, because we have to consider all activities to determine whether they belong to the optimal solution set A or not, of which there are n. This can be seen in lines 4 and 5 in the *GREEDY-ACTIVITY-SELECTOR* algorithm above. Again, this bound holds only under the assumption that the activities have already been sorted. Otherwise, it is $O(n\ lg(n))$.

1.6 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.

1.7 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

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

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.

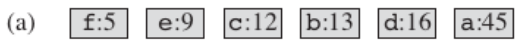
Encoding using Huffman codes is simple and it uses basic concatenation. In our table, the variable-length codes for a = 0, b = 101, and c = 100, so *encoding* a word "abc" results in $0 \cdot 101 \cdot 100 = 0101100$. If you wanted to *decoded* the message, you would immediately see the importance of 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.

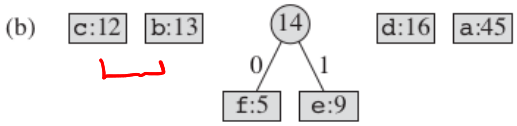
1.8 Constructing a Huffman code

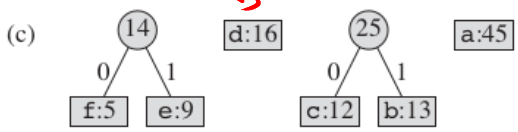
The following algorithm (Huffman) is a greedy algorithm that constructs an optimal prefix code called a Huffman code. As its input, it takes a C, a set of n characters and each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency.

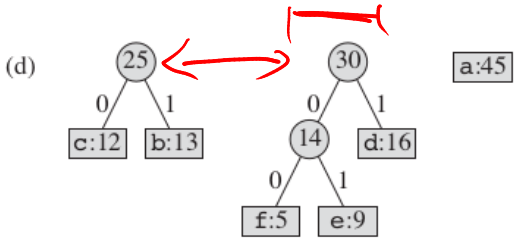
The algorithm builds the tree T, representing the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two 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.

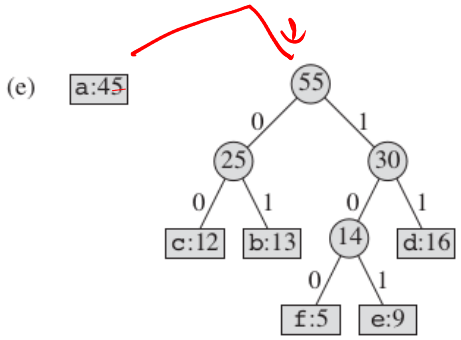
```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

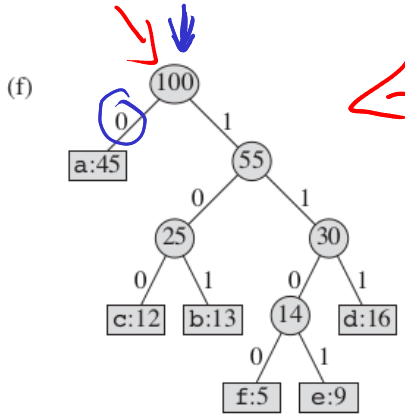












The function *EXTRACT-MIN* was covered in Chapter 6 when we covered **Heaps**, in particular when they are used in connection with queues, and in this case, a min-priority queue. A min-priority queue is implemented using min-heaps. Recall also that heaps are represented as arrays, which are often graphically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

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 $O(n\ lg(lg(n)))$.

In the last image, the sub image (f), gives us the Huffman prefix code, also called 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, which represents a reduction of 25%.