❖ Greedy Algorithms

7.1 The Activity Selection Problem

In the last section, we used dynamic programming to efficiently prune the search space of optimization problems. Though this is a powerful technique, sometimes it can still be overkill, and simpler more efficient solutions can be designed. A **greedy algorithm** always makes the choice the looks best at that particular moment, without thinking about how the problem changing in the future might affect it.

We will start by looking at a variant of a familiar problem called the **activity selection problem**.

Problem (ACTIVITYSELECTION)**.**

INPUT: A set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to reserve a conference room. S is sorted by finish time.

OUTPUT: The maximum number of mutually compatible activities.

Question 104. Consider the following instance of activity selection. What is the correct solution?

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

Question 105. What is the difference between this problem and the weighted scheduling problem?

This problem can be solved using a dynamic programming approach. Let's briefly talk about why that is, by identifying optimal substructure. Let S_{ij} be the set of activities that start after activity a_i finishes, and that finish before activity a_j starts. Let A_{ij} be the maximum set of activities that can be scheduled in this range, and let's say it includes some activity a_k. We want to see if finding the maximal set of S_{ik} and S_{kj} relate to the solution to the full problem. In this case, if we let $A_{ik} = S_{ik} \cap A_{ij}$ and $A_{kj} = S_{ik} \cap A_{ij}$, we can say that

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1. \tag{24}$$

Now that we identified the optimal substructure, we can design a dynamic programming algorithm.

Question 106. Describe a dynamic programming algorithm to solve ACTIVITYSELECTION.

1. Subproblem domain:

2. Memo definition:

3. Goal:

4. Base cases:

5. Recurrence:

For some problems, there are simpler ways to solve the problem that allows us to bypass computing the many subproblems. This is possible when a problem can be solved just by considering **the greedy choice**.

Question 107. What is the greedy choice in the ACTIVITYSELECTION problem?

It turns out that repeatedly making the greedy choice in this problem yields the optimal solution! Why does this work? Let's state our result formally and try to prove it.

Theorem 7.1. Let S_k be the set of activities that start after activity a_k finishes, 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.

Let's design an iterative algorithm to solve this problem now that we have identified the greedy choice. We simply need to keep adding the earliest finishing task, as we show in the following python code snippet.

```
def greedy_activity_select(s, f, n):
    # s is an array of start times
    # f is an array of finish times
    # n is the number of activities
    A = [a[1]]
    k = 1  # k records the last activity added
    for i in range(2, n + 1):
        if s[m] >= f[k]:
            A.append(a[m])
            k = m
    return A
```

Question 108. What is the runtime of `greedy_activity_select`?

There can be many possible greedy choices to be made for a problem, and not all of them may yield the optimal solution. For each of the following, think about whether or not they yield the optimal solution, and if not, describe a counter example that shows that the greedy strategy does not work.

Question 109. Selecting the activity with the earliest starting time.

Question 110. Selecting the activity with the latest starting time.

Question 111. Selecting the shortest activity in the list.

7.2 Elements of the Greedy Strategy

There are two key properties that we need to identify to determine if the problem can be solved by a greedy algorithm.

7.2.1 Greedy-Choice Property

The first is the **greedy-choice property**, which states that you can assemble a globally optimal solution by making locally optimal (greedy) choices. We will often discover candidates based on intuition, kind of like we did in the activity selection problem.

Additionally, we need to prove that a greedy choice at each step yields a globally optimal solution. The typical way we will do this is to fix some globally optimal solution, and examine it in the context of some subproblem. We then want to show that making the greedy choice for this subproblem is not any worse than the globally optimal solution that we chose.

7.2.2 Optimal Substructure

The second property is a familiar one. To reiterate, we say a problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems. In the greedy setting, we can simplify this analysis by assuming that we generated our subproblem by a greedy choice, and simply show that the greedy choice combined with the solution to this subproblem yields the full solution.

7.3 The Fractional Knapsack Problem

You go to a candy store that is running a deal. You will purchase a bag that has a weight capacity of W, and you are free to put any of the n items at the store as long as the bag can contain them. The catch is, that you assign a value v_i for each item as well depending on how much you like it. The goal is to maximize the total value that can fit inside the bag. This problem should sound familiar, as it is a variant of the truck loading problem, except that now we are trying to maximize over the value, not the weight. This problem is referred to as the **0-1 Knapsack Problem**, and can be solved using the same algorithm as the truck loading problem, with a minor modification.

There is a variant to this problem called the **fractional knapsack problem**. This time, instead of indivisible pieces of candy, you are allowed to take a fraction of the item instead. For example, we might be trying to maximize the value over different flavors of icecream that are available.

Question 112. Show that these problems exhibit optimal substructure.

To solve the fractional knapsack problem, we compute value per weight of each item v_i/w_i. Once this is done, we simply take the item with the maximum value per weight until either the item runs out, or we have no more room in our bag.

Question 113. Demonstrate the greedy strategy for the fractional knapsack problem for the following instance:

1. Icecream cup can hold $W = 500$ grams.
2. Three flavors, with happiness of $60, 100, 120$ and total weights of $100, 200, 300$ grams.

Question 114. Show that if the above instance was a 0-1 knapsack problem, the greedy solution does not work.

Question 115. Show that the fractional knapsack problem has the greedy-choice property. To find this, you need to characterize the following three points.

- A globally-optimal solution.
- Show greedy choice at first step reduces problem to the same but smaller problem.

  Greedy choice must be
  - Part of an optimal solution, and
  - Can be made first
- Use induction to show greedy choice is best at each step (i.e., optimal substructure).

7.4 Huffman Codes

Suppose you have a file with 100,000 characters whose options and frequencies are outlined in the table below. If we are using something like ASCII to encode the characters, we require 8 bits to identify each one, meaning we need 800,000 bits to store the file. Since we know which characters are in the file, it may be overkill to use something like ASCII, and we may want to design a way to compress the file so it is not as expensive to send.

We can represent n different characters using $\lceil \log_2 n \rceil$ bits, giving rise to a **fixed-length code**. In our instance, we can uniquely assign a bit string to each character using just $\lceil \log_2 6 \rceil = 3$ bits. This reduces our storage requirement down to 300,000 bits.

We can do even better by using a **variable length code**. This is a coding scheme where we represent frequent characters with less bits, and more common ones with more. The code in the table is one such example.

	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

Question 116. How many bits are we using to encode our document if we use the variable length code in the table above?

When we create a variable length code, we have to make sure that the code is **prefix-free**. This means that no codeword is the prefix of any other codeword. This also makes decoding efficient and unambiguous.

Question 117. What would the encoding for the word "face" be using the variable length encoding in the table?

We can describe our codes using binary trees. Let each internal node store the total number of occurrences of all its descendants. The codewords are then chosen by adding a 0 if we go left, and a 1 if we go right.

Is there an algorithmic way we can construct a tree that looks more like the one on the right? Turns out the answer is yes, and we can achieve this by using a greedy algorithm.

```
def huffman(C):
    # C is a set of n characters appearing in a file that also store their frequency.
    n = size(C)
    # Create a min-priority queue using the elements in C, keyed by their frequency.
    Q = priority_queue(C)
    for i in range(1, n):
        x = extract_min(Q)
        y = extract_min(Q)
        z = # new node
        z.left = x
        z.right = y
        z.freq = x.freq + y.freq
        Q.insert(z)
    return extract_min(Q)  # Return the root of the tree we created.
```

Question 118. Demonstrate a run of `huffman` where the input is

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

Question 119. Show that the Huffman coding problem has the greedy-choice property. To find this, you need to characterize the following three points.

- A globally-optimal solution.
- Show greedy choice at first step reduces problem to the same but smaller problem.

  Greedy choice must be

  - Part of an optimal solution, and
  - Can be made first
- Use induction to show greedy choice is best at each step (i.e., optimal substructure).

Question 120. Show that the Huffman coding problem has the optimal substructure property. That is, making a correct choice induces a subproblem whose solution builds into the full solution.