Greedy Algorithms

Greedy algorithms is a strategy to solve ***optimization problems***. In an optimization problem, you are given certain constraints and an objective function, and your goal is to find a "solution" to minimize or maximize the objective function, such that all the constraints are satisfied.

Example. You have n homework assignments. Homework i is worth p_i points, require t_i days to finish, and the deadline is at day d_i. You cannot work on two assignments at the same time. If you do not finish the assignment before the deadline, you lose all the points for that assignment. Now the goal is to maximize the total number of points you can get.

In this example, the constrains are 1) you need to spend t_i days before day d_i to earn p_i points, and 2) you can only work on one assignment at a time. Your goal is to maximize the total number of points.

You can use many strategies to make plans for the assignments. "Greedy" strategy means to make decisions one by one, and at any decision, we find the current "best" solution based on some criteria. Clearly, there can be multiple "criteria" that you can use to decide which assignment to work on first. You could start with the assignment with the earliest deadline, or the one with the highest score, or the one that takes the shortest time, etc.

Note that, although the greedy strategies usually looks simple, ***they are not necessarily optimal***. For the problems where certain greedy strategies can give optimal solutions, we need to *prove* their optimality.

In class, we will use four algorithms as examples for greedy algorithms: buying candies, kayaking, activity selection, and Huffman code. We will provide the proofs for two problems below as examples.

Buying Candies

You plan to buy some candies with s dollars of budgets. There are n candies in the store, and we denote the price as p_i for candy i. For each candy, you plan to buy at most one of them. Your goal is to buy as many candies as possible.

The constraints of this problem are 1) your budget - the total price is no more than s, and 2) that you can buy each candy at most ones (no duplicates in your solution). Your objective is to maximize the number of candies.

In this optimization problem, one simple greedy strategy is to buy candies from the cheapest to the most expensive. But will this give you the *best (optimal) solution*? And, if so, how can we *prove this*?

Note that, there can be *multiple solutions that are all optimal*, i.e., they all give the largest number of candies. As long as your greedy strategy can give one of such solutions, it is "optimal".

A General Framework to Prove the Optimality of Greedy Algorithms

There are many ways to prove that a greedy strategy is optimal. In the textbook, a simple "framework" is given, which could (hopefully) make the prove easier. In particular, you only need to show the following two properties are true for your algorithm.

- **Greedy Choice.** This means that your first choice based on your greedy strategy is valid - i.e., there exists an optimal solution that contains your greedy strategy. Therefore, making this greedy choice can give you an optimal solution - as long as you make correct decisions later.
- **Optimal Substructure.** Optimal substructure means that the optimal solution of a problem contains optimal solutions of subproblems. In other words, after making your first greedy choice, the rest of decisions are equivalent to finding the optimal solution with a smaller input (e.g., the original input without your first choice). This property suggests that after your greedy strategy, you could just solve a subproblem recursively, which means to repeatedly make greedy choices.

Buying Candies - Proof for Optimality

Greedy Choice Property

Our greedy choice is the candy that has the minimum cost among all candies that have not been bought. To show the Greedy Choice Property, we must show that ***there exists an optimal solution that contains the cheapest candy***.

Let S be the input set of all candies. Let $A = \{c_1, c_2, \ldots, c_m\}$ be an optimal solution, which is a subset of S and contains m candies. Let c^* be the cheapest candy. We will show that either A contains the cheapest candy, or we can construct another optimal solution contains the first candy. There are two cases.

1. $c^* \in A$, i.e., A contains the cheapest candy. In this case, it is true that there exists an optimal solution that contains our greedy choice (which is A itself!). So we are done.
2. $c^* \notin A$, i.e., A does not contain the cheapest candy c^*. Let B be a set of candies created by replacing c_1 in A with the cheapest candy c^*. In other words, $B = \{c^*, c_2, c_3, \ldots, c_m\}$. We will show that B is also an optimal solution.

   First we must show that the set of candies B is a valid solution to the problem. First of all, since $c^* \neq A$, we have $c_j \neq c^*$ for all $1 \leq j \leq m$. Therefore, by replacing c_1 with c^*, we are still buying each candy at most once. We will then show that the total cost of B is within our budget s. Since c^* is the cheapest one, it must be true that $p_{c^*} \leq p_{c_1}$. Note that we only replace a candy in A to a cheaper (or equal price) candy, so the total cost of A should be greater than the total cost of B. Considering that A is a valid solution, so the cost of A is within s. Therefore, the cost of B is no more than the cost of A, which is no more than s. Therefore, we have shown that B is a valid solution.

   Now we will show that B is optimal to the problem. Notice that $|A| = |B|$, since A is an optimal solution and B has the same number of candies as A. Therefore, B is an optimal solution.

Optimal Substructure

To show the optimal substructure, we must prove the statement

> If we have an optimal solution A that starts with the greedy choice c^* for a set of candies S, then $A' = A - \{c^*\}$ (a set of candies created by removing the greedy choice in A), must be optimal to $S' = S - \{c^*\}$ and a new budget $s - p_{c^*}$.

To show the statement above, we will use proof by contradiction. So we will assume that the following statement is true.

> Assume that we have an optimal solution A that contains the greedy choice for a set of candies S and budget s, and let $A' = A - \{c^*\}$ (a set of candies created by removing the first choice in A) is not optimal to $S' = S - \{c^*\}$ and budget $s' = s - p_{c^*}$.

Since we are assuming that A' is not optimal to S' and s', this implies the existence of some different set $B' \neq A'$, that is optimal to S' and b'. From this we can see that $|A'| < |B'|$. Now consider when we add in the candy c^* into B'. Let $B = B' + \{c^*\}$. Since B' is a solution with budget $s' = s - p_{c^*}$, the total cost of B' is no more than s'. Therefore the total cost of B is no more than s. So B is also a solution to the original problem. However, we will show that B is a better solution than A because

$$
\begin{aligned}
&|A'| < |B'| \\
\Rightarrow &|A' + \{c^*\}| < |B' + \{c^*\}| \\
\Rightarrow &|A| < |B|
\end{aligned}
$$

We see that B has more candies in it than A. This contradicts our assumption that A is an optimal solution to the original problem. Therefore, our assumption is not true, and A' is optimal to S'.

Kayaking

n students went to kayaking. The maximum weight for each kayak is w, and there can be at most two people on one kayak. Given the weight of each student a_i, find the minimum number of kayaks we need.

Algorithm. There are multiple valid greedy strategies. For example, we can start with the lightest student X, and pair X with the heavies student that fits in the same kayak with X. If there does not exist such a student (all the other students are too heavy), then X will take a single-person kayak.

In class, we show an outline to prove this strategy is optimal. You could use this as a practice to write a formal proof for greedy algorithms using the ideas of *greedy choice* and *optimal substructure*.

Activity Selection

Given a set of activities represented as time intervals $\{s_i, e_i\}$ (start time s_i and end time e_i), we want to find a subset A of non-overlapping activities, and maximize the number of activities we select.

Algorithm. Our greedy strategy is ***earliest finish***. In particular, we will first select the activity with the earliest end time and add it to the output set A. Then we remove all activities that overlaps with this activity. For the rest of activities (they are all "compatible" with the current result A), we repeat this process and select the earliest-finish activity, so on so forth.

Activity Selection Proof

Greedy Choice Property

Our greedy choice is the activity that has the earliest end time. To show the Greedy Choice Property, we must show that ***there exists an optimal solution that starts with our greedy choice***.
Suppose we have our set of activities $S = \{\{s_1, e_1\}, \{s_2, e_2\}, \{s_3, e_3\}, ...\}$ sorted in increasing order of end time. Let A be some optimal solution. There are two cases.

1) The first activity in A is $\{s_1, e_1\}$. In this case, it is true that there exists an optimal solution that starts with our greedy choice. So we are done.

2) The first activity in A is not $\{s_1, e_1\}$ and instead some activity $\{s_i, e_i\}$, where $e_1 \neq e_i$. Let B be a set of activities created by replacing $\{s_i, e_i\}$ in A with $\{s_1, e_1\}$. In other words, B is a set of activities identical to A, with the exception of the first element being replaced by the greedy choice.

   First we must show that the set of activities B is a valid solution to the problem. Notice that since the activities in S, A, and B are sorted according to increasing end time, it must be true that $e_1 < e_i$. Since $\{s_i, e_i\}$ does not overlap with the second activity in B, it must be true that an activity that ends even earlier $\{s_1, e_1\}$, would also not overlap with the second activity in B. Since none of the activities overlap in B, it is a valid solution.

   Now we will show that B is an optimal solution. Notice that $|A| = |B|$, since B is an identical set to A, with the difference being that the first activity in B is the greedy choice. Since A is an optimal solution and B has the same number of activities as A, B is an optimal solution.

Optimal Substructure

To show the optimal substructure, we must prove the statement

> If we have an optimal solution A that starts with the greedy choice for a set of activities S, then $A' = A - \{s_1, e_1\}$ (a set of activities created by removing the first choice in A), must be optimal to $S' = S - \{\text{All activities that overlap with } \{s_1, e_1\} \text{ in } S\}$.

To show the statement above, we will use proof by contradiction. So we will assume that the following statement is true.

> Assume that we have an optimal solution A that starts with the greedy choice for a set of activities S and let $A' = A - \{s_1, e_1\}$ (a set of activities created by removing the first choice in A) is not optimal to $S' = S - \{\text{All activities that overlap with } \{s_1, e_1\} \text{ in } S\}$.

Since we are assuming that A' is not optimal to S', this implies the existence of some different set B', that is optimal to S'. From this we can see that $|A'| < |B'|$. Now consider when we add in the activity $\{s_1, e_1\}$ into A' and B'. Let $B = \{s_1, e_1\} + B'$ and recall that $A = A' + \{s_1, e_1\}$

$$|A'| < |B'|$$
$$|A' + \{s_1, e_1\}| < |B' + \{s_1, e_1\}|$$
$$|A| < |B|$$

We see that B has more activities in it than A. However, this is a contradiction, since A is an optimal solution, so there should not exist another set that has more activities than it! Thus by proof by contradiction, we have proven the original statement.

Huffman Code

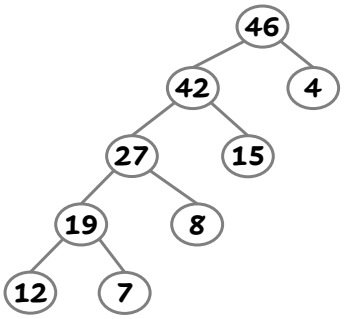
Our next example is about Huffman Code and Huffman tree. We start with a simple "pebble merging" problem:

Pebble Merging. Given some piles of pebbles, each with size a_i, your goal is to merge them together to finally form one pile. You can merge two piles at a time. Merging two piles of sizes x and y will take you $x + y$ units of energy, and form a new pile with size $x + y$. You want to spend minimum energy to finish the task.

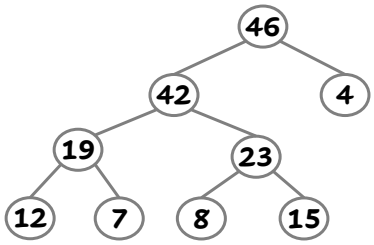
First of all, note that, although finally you always get a pile with size $\sum a_i$ pebbles, the merging order does affect the energy cost. For example, consider five piles with sizes $12, 7, 8, 15, 4$ and the following two strategies:

- Solution 1 (merge them one by one in the given order):
  1. Merge 12 with 7 (cost = 19). Now we have four piles $19, 8, 15, 4$. Total energy cost: 19.
  2. Merge 19 with 8 (cost = 27). Now we have three piles $27, 15, 4$. Total energy cost: $19 + 27$.
  3. Merge 17 with 15 (cost = 42). Now we have two piles $42, 4$. Total energy cost: $19 + 27 + 42$.
  4. Merge 42 with 4 (cost = 46). Now we have the final piles. Total energy cost: 19+27+42+46 = 134.
- Solution 2 (merge every two of them):
  1. Merge 12 with 7 (cost = 19). Now we have four piles $19, 8, 15, 4$. Total energy cost: 19.
  2. Merge 8 with 15 (cost = 23). Now we have three piles $19, 23, 4$. Total energy cost: $19 + 23$.
  3. Merge 19 with 23 (cost = 42). Now we have two piles $42, 4$. Total energy cost: $19 + 23 + 42$.
  4. Merge 42 with 4 (cost = 46). Now we have the final piles. Total energy cost: 19+23+42+46 = 130.

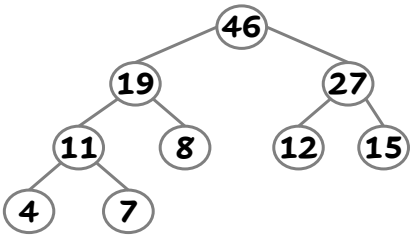
So by changing the order, we can save some energy!
This process can be illustrated in a tree-like structure:



Solution 1: Energy cost = 134



Solution 2: Energy cost = 130



Optimal Solution: Always combine two smallest piles
Energy cost = 103

Each pile is represented as a tree node. When we merge two piles, it's illustrated as connecting the two nodes by a parent, which is the new piling combining them.

Algorithm. Our greedy algorithm is very simple: when we need to choose two piles to merge, we always choose the two smallest ones. In the above example, this gives solution as:

- Optimal Solution (always merge two smallest piles):
  1. Merge the smallest two piles with sizes 4 and 7 (cost = 11). Now we have four piles $11, 8, 12, 15$. Total energy cost: 11.
  2. Merge the smallest two piles among $11, 8, 12, 15$, which are 8 and 11 (cost = 19). Now we have three piles $19, 12, 15$. Total energy cost: $11 + 19$.
  3. Merge the smallest two piles among $19, 12, 15$, which are 12 and 15 (cost = 27). Now we have two piles $19, 27$. Total energy cost: $11 + 19 + 27$.
  4. Merge the last two piles with size 19 and 27 (cost = 46). Total energy cost: $11+19+27+46 = 103$.

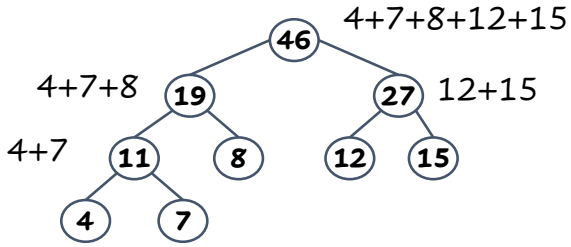
This is better than all the other strategies that we tried!

The tree built based on this strategy is ***Huffman tree*** (see the illustration above).

Intuition of Optimality. Why this greedy strategy gives you the best solution? Intuitively, it is because the total cost can also be viewed as (see explanation and illustration in slides):

$$\sum_i a_i \cdot d_i$$

where d_i is the depth of the leaf corresponding to pile i in the tree.



Cost: 103 = (4+7+8+12+15) + (4+7+8) + (4+7) + (12+15)
= 4*3 + 7*3 + 8*2 + 12*2 + 15*2

Therefore, the intuition is to put smaller piles as deep as possible. Based on this, we can prove that the two smallest piles must be the two deepest leaves - this means that our greedy choice is good.

Huffman Code. Huffman code refers to the encoding scheme for strings. All characters stored in your computer are represented in integers (more precisely, in binary form of 0s and 1s). There are many encoding schemes used in practice. The most famous one is the ascii code, which is fixed length, i.e., all codewords have the same length l. The advantage of fixed-length codes is that they are trivially unambiguous - for decoding, you just read every l bits, and translate it to a character. However, it is probably wasting space, since for all characters, we need l bits.

Some variable length codes avoid this by allowing different lengths for codewords. For example, for those more frequent letters (e.g., "`e`"), we can make them shorter, so that overall we can make the code of a string shorter. One example is the Morse Code (represented as dots and dashes). However, for variable length codes, it may incur ambiguity in decoding since we do not know how many bits represents each character. As a result, we may need to insert separators in the codes.

The ***prefix codes*** can avoid ambiguity in variable length codes. In prefix codes, no codeword can be the prefix of another codeword. Therefore, once we recognize a codeword in the encoded message, we can decode it.

The prefix codes can also be represented by a tree structure. Each letter can be represented by a leaf. The left/right branches means `0` and `1`, respectively. Then the codeword of a letter is the path from the root to its corresponding leaf. When encoding some text, we hope the encoded message can be as short as possible.

Then what is the total length of the encoded message? Let f_i be the frequency of each letter. Note that the length of its codeword is depth in the tree, then the total message length is

$$\sum_i f_i \cdot d_i$$

This is exactly the same as the pebble merging problem! If we view the frequency of a letter as the size of a pebble pile, then to minimize the encoded message size, it is equivalent to find the minimum energy cost in the pebble merging problem.

Therefore, to encode some text with the shortest prefix code, we can just do the same thing - we find the two letters with the smallest frequency f_1 and f_2, remove them and combine them to a "virtual letter" with frequency $f_1 + f_2$, and repeat until there is only one "letter" left. To find the two letters with the smallest frequency, we can use a priority queue. By using an efficient priority queue (e.g., a binary heap), inserting an element takes $O(\log n)$ time. Finding and removing the smallest element also takes $O(\log n)$ time. In total, the algorithm to construct a Huffman tree is $O(n \log n)$ when n letters and their frequencies are given. The pseudocode is given below.

```
1  Huffman(C[1..n]) {
2    PriorityQueue Q(C)  // construct a priority queue of all letter's frequency
3    for i = 1 to n-1 {
4      allocate a new node z
5      z.left = x = ExtractMin(Q) // find the smallest frequency in x and remove it
6      z.right = y = ExtractMin(Q) // find the (next) smallest frequency in y and remove it
7      z.freq = x.freq + y.freq
8      Insert(Q, z);
9    }
10   return ExtractMin(Q) // Root of the tree
11 }
```