Chapter 16
The Greedy Method

We should know well about the divide n’ conquer technique, and will now discuss another general technique, the greedy method, on designing good algorithms.

We will go over the basic scenarios, in which it is appropriate to apply this technique, and a few concrete applications.

When solving optimization problems, we are given a set of constraints, and an optimization function.

Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called an optimal solution.
The greedy method

It is one way to construct a feasible solution for such optimization problems, and, sometimes, it leads to an optimal one.

When applying this method, we construct a solution in stages. At each stage, we make a decision that appears to be the best \textit{at that time}, according to certain \textit{greedy criterion}. Such a decision will not be changed in later stages. Hence, each decision should assume the feasibility.

Let's look at a few simple examples....
A thirsty baby

Assume there is a thirsty, but smart, baby, who has access to a glass of water, a carton of milk, etc., a total of $n$ different kinds of liquids. Let $a_i$ be the amount of ounces in which the $i^{\text{th}}$ liquid is available.

Based on her experience of taste, and desire for nutrition 😊, she also assigns certain satisfying factor, $s_i$, to the $i^{\text{th}}$ liquid. If the baby needs to drink $t$ ounces of liquid, how much of each liquid should she drink?

Let $x_i, 1 \leq i \leq n$, be the amount of the $i^{\text{th}}$ liquid the baby will drink. The solution for this thirsty baby problem is obtained by finding real numbers $x_i, 1 \leq i \leq n$, that maximize $\sum_{i=1}^{n} s_i x_i$, subject to the constraints that $\sum_{i=1}^{n} x_i = t$ and for all $1 \leq i \leq n$, $0 \leq x_i \leq a_i$.

We notice that if $\sum_{i=1}^{n} a_i < t$, then this instance is not solvable. 😞
A specification

Input: $n, t, s_i, a_i, 1 \leq i \leq n$. $n$ is an integer, and the rest are positive reals.

Output: If $\sum_{i=1}^n a_i \geq t$, output is a set of real numbers $x_i, 1 \leq i \leq n$, such that $\sum_{i=1}^n s_i x_i$ is maximum, $\sum_{i=1}^n x_i = t$, and for all $1 \leq i \leq n$, $0 \leq x_i \leq a_i$.

In this case, the constraints are $\sum_{i=1}^n x_i = t$, and for all $1 \leq i \leq n$, $0 \leq x_i \leq a_i$, and the optimization function is $\sum_{i=1}^n s_i x_i$.

Every set of $x_i$ that satisfies the constraints is a feasible solution, and it is optimal if it further maximizes $\sum_{i=1}^n s_i x_i$. 
How should we feed her?

We certainly should feed her with what she likes most first.... thus assume \( s[n] \) is reversely sorted, in \( \Theta(n \log n) \).

\[
\begin{align*}
  s &= 0; \quad T = 0; \quad i = 1; \quad d = 0; \\
  &\text{for } (j = 1; j \leq n; j++) \\
  &\quad x[i] = 0; \\
  &\text{while } ((i \leq n) && (T < t)) \\
  &\quad \quad x[i] = a[i]; \quad \text{//Take it in order of } s[i] \\
  &\quad \quad T += x[i]; \quad \text{//total amount so far} \\
  &\quad \quad s += x[i] \times s[i]; \quad \text{//total satisfaction so far} \\
  &\quad \quad i++; \\
  &\text{if } (T < t) \text{ return } 1; \quad \text{//failure: no way, Jose} \\
  &\text{else } \\
  &\quad \quad i--; \quad d = T - t; \quad \text{//The most we can give} \\
  &\quad \quad x[i] -= d; \quad \text{//Adjust the last kind} \\
  &\quad \quad s -= d \times s[i]; \quad \text{//Adjust the satisfying number} \\
  &\quad \quad T = T - d; \quad \text{//Should be the same as } t \\
  &\text{return } 0; \quad \text{//x[n] contains the solution} \\
\end{align*}
\]

It clearly takes \( \Theta(n \log n) \) total time to finish.
Loading problem

A large ship is to be loaded with containers of cargos. Different containers, although of equal size, will have different weights.

Let $w_i$ be the weight of the $i^{th}$ container, $1 \leq i \leq n$, and the capacity of the ship is $c$, we want to find out a way to load the ship with the maximum number of containers, without tipping over the ship.

Let $x_i \in \{0, 1\}$. If $x_i = 1$, we will load the $i^{th}$ container, otherwise, we will not load it.

We wish to assign values to $x_i$’s such that $\sum_{i=1}^{n} w_i \leq c$, and $\sum_{i=1}^{n} x_i$ is maximized.

**Homework:** Come up with a solution as how we should load into the ship....
Change making

A child buys a candy bar at less than one buck and gives a $1 bill to the cashier, who wants to make a change using the smallest number of coins. The cashier constructs the change in stages, in each of which a coin is added to the change.

The greedy criterion is as follows: At each stage, increase the total amount as much as possible.

A feasible solution is one such that in no stage the amount paid out so far exceeds the desired change.

For example, if the desired change is 67 cents. The first two stages will add in two quarters. The next one adds a dime, and following one will add a nickel, and the last two will finish off with two pennies.
Machine scheduling

We are given an infinite supply of machines, and \( n \) tasks to be performed in those machines. Each task has a start time, \( s_i \), and finish time, \( t_i \). The period \([s_i, t_i]\) is called the *processing interval* of task \( i \). Two tasks \( i \) and \( j \) might overlap, e.g., \([1, 4]\) overlaps with \([2, 4]\), but not with \([4, 7]\).

A *feasible* assignment is one in which no machine is given two overlapped tasks. An *optimal assignment* is a feasible one that uses fewest number of machines.

We line up tasks in nondecreasing order of \( s_i \)'s, and call a machine *old*, if it has been assigned a task, otherwise, call it *new*. A greedy strategy could be the following: At each stage, if an old machine becomes available by the start time of a task, assign the task to this machine; otherwise, assign it to a new one.
An example

Given seven tasks, their start time, as well as their finish time as follow:

<table>
<thead>
<tr>
<th>task</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>finish</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>11</td>
<td>10</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Then, by the aforementioned strategy, we can assigned the tasks to machines in the following way:

It may be implemented in $\Theta(n \log n)$, by using a minHeap of availability times for the old machines.
It does not always work!

Given the following digraph:

we want to find out the shortest path from $v_1$ to $v_5$. An intuitive way is to find it in stages. At a certain stage, if the path built so far ends at vertex $q$, we can select the nearest vertex that is adjacent to $q$, but not on the path yet.

For our example, this strategy will lead to $v_1, v_3, v_4, v_2, v_5$ of length 10, which is certainly not the shortest one. 😞

This is certainly not how WAZE works. We will study this shortest path problem a lot later on.
The Knapsack problem

We want to pack a knapsack with a capacity of $c$ by selecting items from a list of $n$. Each item has both a weight of $w_i$ and a profit of $p_i$ ($i \in [1, n]$). In a feasible solution, the sum of the weights must not exceed $c$, and an optimal solution is both feasible and reaches the maximum profit.

This problem generalizes the container loading one, in the sense that in the loading problem, the profit of every container is the same.

For example, if you win the first-prize in a grocery store contest, and the prize is a free cartful of groceries. Your goal is to fit the cart with the maximum value: Bounty or Rolex watch?

This can certainly be modeled as a knapsack problem.
Strategies

As this 0/1 knapsack problem, each item is either in or out, is \textit{NP-complete}, which we will get to later, we don’t expect to find an “easy” solution. \textit{Will being greedy help?}

An obvious greedy criterion is to pick up the one with the most profit first. For example, if \( n = 3, \ w = [100, 10, 10], \ p = [20, 15, 15], \) and, \( c = 105. \)

This \textit{profit first} strategy will bring in a piece worth 20, i.e., the first one; even though we could bring in 30 by picking up the two less profitable pieces.

Thus, this strategy will not always lead to an optimal solution. 😊
More strategies

Another idea is to be greedy on weight, i.e., among the remaining objects, always pick up the one with minimum weight first.

For example, when $n = 2, w = [10, 20], p = [5, 100]$, and $c = 25$. If we pick up first piece with less weight, we will only rake in a profit of 5.... Thus, this weight-first strategy will not work in general, either.

Yet another one is to be greedy on the profit density, i.e., $p_i/w_i$. It considers both factors, thus more considerate. For example, when $w = [20, 15, 15], p = [40, 25, 25]$, and $c = 30$. The respective densities are 2, $5/3$ and $5/3$, instead of 50, we will only bring in 40.... Apparently, this one also fails in this case.

**Homework:** Read through the subsection on the Knapsack problems in pp.425, then finish Exercise 16.2-4(*).
Which one is the best?

By and large, the profit density strategy, although not guaranteed to work in all the cases, is a pretty good one, as compared with the others.

In an experiment involved with 600 randomly generated instances, the profit density strategy generated optimal solutions 239 out of the 600 cases.

Moreover, with 583 of these 600 cases, the solution generated with this strategy had a value within 10% of the optimal, and all 600 solutions fell within 25% of the optimal.

Thus, it is the best of the three, and takes $\Theta(n \log n)$ as it sorts out the density sequence.
Variable-length code

In ASCII code, every character is coded in 8 bits. So, if we have a text file with 1,000 characters, we have to use 1,000 bytes.

In reality, some characters are used more often than the others (Think about *Wheel of Fortune*). It makes sense to assign shorter codes to those used more often, and longer codes to those used less often.

The question is how? One approach is to find out the *frequencies* of the letters, then assign shorter codes to the more frequently occurring one, and longer codes to the less frequently occurring ones.

*Look back at the histogram related stuff at the end of last chapter.*
An example

With the string “aaxuaxz”, the frequency of ‘a’, ‘x’, ‘u’ and ‘z’ are 3, 2, 1 and 1. We can then assign 0 to ‘a’, 10 to ‘x’, 110 to ‘u’, and 111 to ‘z’.

Hence, “aaxuaxz” is coded as 0010110010111, 13 bits, compared with 14, if we give each of them two bits (?). No big deal for this case.

On the other hand, if the file contains 1,000 letters, and the frequency of these four symbols, ‘a’, ‘x’, ‘u’ and ‘z’, are (996, 2, 1, 1), then the “two-bit per symbol” method leads to 2,000 bits long, while our code will lead to a file of only 1,006 bits, almost a 50% saving 😊.
The other side...

To decode “0010110010111”, since no code starts with “00”, “00”, gives “aa”. Similarly, no code starts with “10” other than that of ‘x’, we read off an 'x', etc..

In general, we always read off the longest possible piece from the remaining code string, since this coding is a prefix code, i.e., no code is a prefix of another one.

**Question:** How to generate such a nice coding for a given text file?

**Answer:** Huffman tree.

This is what you will do with Project 7, MOAP.
Below is a buggy Huffman tree

An optimal code is always represented by a full binary tree, which is constructed as follows:

When we construct a Huffman tree, we always want to add a node with the smallest weight, an minHeap, \( Q \), of binary trees, is an obvious choice of the data structure.
Huffman’s algorithm

Assume that \( C \) is a set of \( n \) characters and for each \( c \in C \), \( f(c) \) stands for its frequency. Huffman designed a greedy algorithm, back in 1952, that builds up the tree corresponds to the optimal coding for \( C \) by carrying out a \(|C| - 1\) tree merging, starting with \(|C|\) leaves.

Huffman(C)
1. \( n <- |C| \) //\( C \) is the collection of symbols
2. \( Q <- C \) //Build a miniHeap of trees
3. for \( i <- 1 \) to \( n-1 \)
4. \( \text{do allocate a new node } z \)
5. \( \text{left}[z] <- x <- \text{Extract-Min}(Q) \)
6. \( \text{right}[z] <- y <- \text{Extract-Min}(Q) \)
7. \( f[z] <- f[x] + f[y] \)
8. \( \text{Insert}(Q, z) \)
9. return Extract-Min(Q) //Only one tree stands.

Notice that if we apply the above to the example, we would get the code for ‘a’, ‘b’ and ‘c’ are “01”, “000” and “001”, respectively.
Algorithm analysis

The algorithm is rather straightforward: We initialize the priority queue with the character set $C$, then, repeatedly merge two trees with the smallest frequencies kept in their roots into a new tree with its frequency being the sum of those two, until we have only one tree left, which is returned as the resulting Huffman tree.

Line 2 takes $\Theta(n)$. Line 3 takes $\Theta(n)$, while lines 5, 6 and 8 all take $\log n$. Thus, it takes $O(n \log n)$ to construct a Huffman tree for $n$ characters. 😊

It has been proved that Huffman tree leads to the shortest code overall.

You are now ready to take on Project 7, the mother of all the projects.

**Homework:** Exercise 16.3-3(*).