Chapter 16
The Greedy Method

We have looked at the *divide and conquer* technique with, e.g., *MergeSort, QuickSort* algorithms, as well as in the BST related operations. We now discuss another general technique, *the greedy method,* on designing algorithms.

We will go over the basic scenarios, where it is appropriate to apply this technique, and several concrete applications.

When solving *optimization problems,* we are given a set of *constraints,* and an *optimization function.* Solutions to such a problem 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.*

We certainly *want* to find an optimal solution ☺, which could be costly, if possible. ☹

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 following a greedy approach, we construct a solution *in stages.* At each stage, we make a decision that appears to be *the best at that time,* according to a certain *greedy criterion.* Such a decision will not be changed in later stages. Hence, each decision should assume the feasibility.

Sometimes, such a *locally optimal* solution does lead to an overall optimal one. ☺

Let's start with a few simple examples, and will further make use of *it* in studying graph algorithm down the road such as in looking for shortest path, the backbone of the GPS App.

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. An optimal one is ... .

For example, if the desired change is 67 cents. Instead of giving back 67 pennies, 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, with a total of six coins.

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, $i \in [1, n]$, 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 $\sum_{i=1}^{n} x_i w_i \leq c$, and $\sum_{i=1}^{n} x_i$ is maximized.

Question: What is a feasible, and optimal, solution in this case? How could we maximize the number of containers?

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.

I want to see...

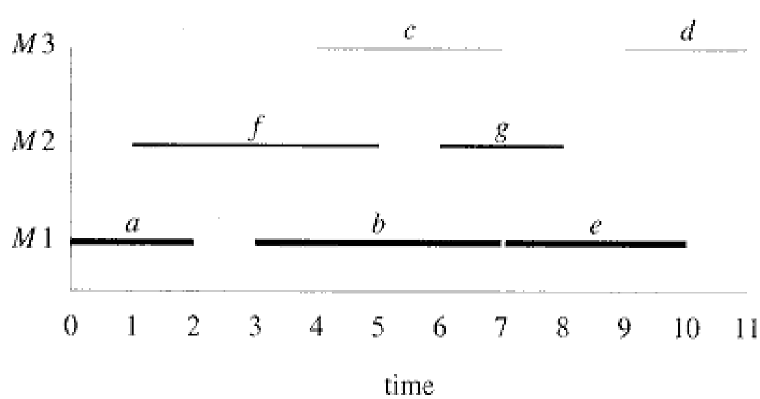
Given seven tasks, their start time, as well as their finish time as follow:

task	a	b	c	d	e	f	g
start	0	3	4	9	7	1	6
finish	2	7	7	11	10	5	8

We sort it out by the Start time:

task	a	f	b	c	g	e	d
start	0	1	3	4	6	7	9
finish	2	5	7	7	8	10	11

We can then assign the tasks to machines in the following way, taking $\Theta(n \log n)$, by using a minHeap:



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 the i^{th} liquid.

Based on her experience of taste, and desire for nutrition ☺, she also assigns certain *satisfying factor*, s_i, to the i^{th} liquid.

Question: How should we make the baby most satisfied if the baby needs to drink t ounces of liquid?

Let's set it up

Let $x_i, 1 \le i \le n$, be the amount of the i^{th} liquid the baby will drink. The solution to this *thirsty baby* problem is to find real numbers $x_i, 1 \le i \le n$, that is to *maximize* $\sum_{i=1}^{n} s_i x_i$ i.e., to make the baby most satisfied. ☺

Notice that, if $\sum_{i=1}^{n} a_i < t$, then this instance is not solvable (Not enough juice. ☹)

We certainly have to satisfy the *constraints* that

- $\sum_{i=1}^{n} x_i = t$ (*Thou shall not drink too much*. ☹), and

- for all $1 \le i \le n$, $0 \le x_i \le a_i$ (We have only this much. ☹)

A specification

Input: $n, t, s_i, a_i, 1 \leq i \leq n$, where 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, for all $1 \leq i \leq n$, $0 \leq x_i \leq a_i$, $\sum_{i=1}^{n} x_i = t$, and $\sum_{i=1}^{n} s_i x_i$ is maximized.

For this problem, the constraints are, for all $1 \leq i \leq n$, $0 \leq x_i \leq a_i$; and $\sum_{i=1}^{n} x_i = t$; 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.*

Furthermore, such a solution is *optimal* if it 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 (*being greedy*)..., thus assume `s[n]` is reversely sorted, in $\Theta(n \log n)$.

```
s=0; T=0; i=1; d=0;
for (j=1;j<=n;j++)
  x[j]=0; //Start with nothing
//Not done yet, and we still have juices left
while ((i<=n)&&(T<t)){
  x[i]=a[i]; //Take it in order of s[i]
  T+=x[i];//total amount so far
  s+=x[i]*s[i];//total satisfaction so far
  i++; }
if(T<t) return 1;//failure: no way, Jose
else {
  i--; d=T-t; //The most we can give
  x[i]-=d;//Adjust the last kind
  s-=d*s[i]; //Adjust the satisfying number
  T=T-d; //Should be the same as T
  return 0;//x[n] contains the solution
}
```

It clearly takes $\Theta(n \log n)$.

Question: How does this work?

I want to see...

	1	2	3	n=4
	Water	Apple Juice	Milk	Lemonade
s	2	4	3	1
a	10	3	7	8

t $\boxed{18}$

Sort by s, the satisfaction factors, in O(n log n).

	1	2	3	4
	Apple Juice	Milk	Water	Lemonade
s	4	3	2	1
a	3	7	10	8

	1	2	3	4	T	S	i
X	0	0	0	0	0	0	1
X	3	0	0	0	3	12	2
X	3	7	0	0	10	33	3
X	3	7	10	0	20	53	4

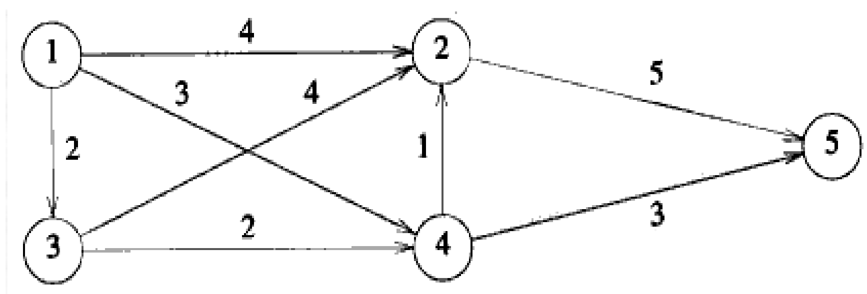
O(n)

	1	2	3	4	i	d	T	S
X	3	7	8	0	3	2	18	49

So, we will feed the kid with three ounces of Apple Juice, seven ounces of Milk, and eight ounces of water to make her happy. ☺

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 leads 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.

It could be challenging... .

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.

The *Thirsty baby problem* is certainly a direct instance of this general problem.

If you win the first-prize in a grocery store contest, and the prize is a free cart full of groceries. Your goal is to fit the cart with the maximum value.

Question: *Bounty* towels or *Rolex* watches?

Strategies

As this *0/1* knapsack problem, each item is either *in* or *out,* is *NP-complete,* which we will get to later, we don't expect to find an "easy" solution. *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 *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 *profit first* strategy will not always lead to an optimal solution. ☹

What else?

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

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. But, 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 ☹.

Assignment: Read through the subsection on the Knapsack problems in the textbook.

Which one is the best?

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

In an experiment involved with 600 randomly generated instances, the profit density strategy generated an optimal solution 239 out of the 600 cases, about 40% of the time. ☺

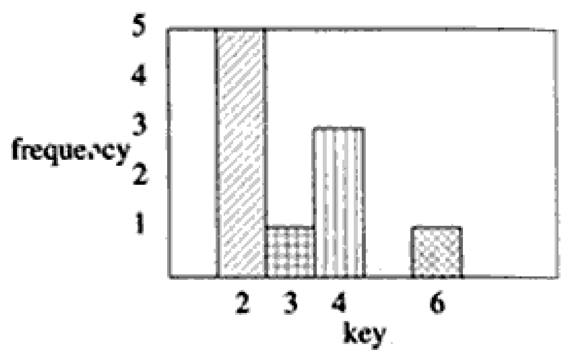
Moreover, with 583 of these 600 cases, the solution generated with this strategy had a value within 10% of the optimal, and all 600 solutions fell within 25% of the optimal.

It does not provide the optimal solution, but a pretty good approximate one, taking $\Theta(n \log n)$, as it sorts out the density sequence.

Histogramming

We start with a collection of n keys and want to output a list of r distinct keys and their *frequencies.* Assume that $n = 10$, and $keys =$ [2, 4, 2 , 2, 3, 4, 2, 6, 4, 2].

key	frequency
2	5
3	1
4	3
6	1



Question: How to do *it*?

When the range of those keys is quite small, this problem can be solved vary easily, in $\Theta(n)$, by using an array `h[0..r]`, where r is the maximum key value, and `h[i]` stores the frequency of i.

Question: How to do it?

More general cases

When the key values are not integers, we can sort out the n keys, in $\Theta(n \log n)$, then scan the sorted list from left to right to find the frequencies of respective keys. For example, $keys =$ [2.1, 4.2, 2.1, 2.1, 3.2, 4.2, 2.1, 6.3, 4.2, 2.1].

This solution can be further improved, when m, the number of distinct keys, is quite small. We can use a (balanced) *BST* tree, each of whose nodes contains two fields, `key` and `frequency`. We insert all the n keys into such a tree, when a key is already there, we simply increment its frequency by 1, as we mentioned in the *BST* chapter.

This procedure takes an expected complexity of $\Theta(n \log m)$, better than $\Theta(n \log n)$, when $m << n$.

When a balanced tree is used, this complexity is guaranteed, even in the worst case.

Variable-length code

With ASCII, 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. Unicode is also fixed length of 8, 16, or 32 bits.

In reality, some characters are used more often than the others (" *Wheel of Fortune* "). To save space, it makes sense to assign *shorter codes to those used more often, and longer codes to those used less often,* thus the notion of variable-length code.

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

Questions: How to find frequencies of letters in a file?

Answer: Histogramming

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`'.

Question: Why this code?

With this coding, "`aaxuaxz`" is coded with 13 bits: 0010110010111, 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 direction...

To decode "0010110010111", since the only code that starts with "0" is 'a', "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 anything else.* Thus, decoding is easy. ☺

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

Answer: Huffman tree (Check out the link on the course page).

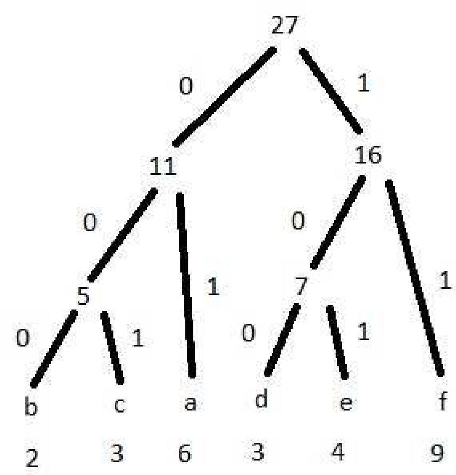
In fact, we will go through the whole nine yards in Project 7 on this topic: Given a text file, find out the frequency of all the letters, construct a Huffman tree, encode the file, and then get back the original file, via a decoding.

An example

Given the following letters and their frequencies:

Letter	a	b	c	d	e	f
Freq.	6	2	3	3	4	9

We can construct the following Huffman tree:



Then, the code of these six letters will be the following:

Letter	a	b	c	d	e	f
Code	01	000	001	100	101	11

Huffman's algorithm

Let C be a set of n characters, and, for each $c \in C$, $f(c)$ the frequency of c. Huffman designed a greedy algorithm, back in 1951, that builds up the tree corresponds to the optimal coding for C.

```
Huffman(C)
n <- |C| //C is the collection of symbols
Q <- C //Build a miniHeap of trees, all leaves
for i<-1 to n-1
  do allocate a new node z
    //Get x, y, two minimum trees out of Q
    //merge them to z, and put it back to Q
    left[z]<-x<-Extract-Min(Q)
    right[z]<-y<-Extract-Min(Q)
    f[z]<-f[x]+f[y]
    Insert(Q, z)
//Only one tree stands now
return Extract-Min(Q)
```

Since we must keep track of subtrees with small root values, an *minHeap, Q, of binary trees,* is an obvious choice of the data structure, which we worked with in Project 4.

Question: Remember this stuff in Test 1?

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 7, 8 and 10 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.

Check out the course page for more discussion and examples related to Huffman tree.

It is time to work on Project 7.