Greedy Algorithms: Problems

Problems That Have Greedy Optimal Solutions

Problem 1: Making Change

Problem. A cashier at a store accepts money in payment for goods and needs to make change. The money comes in a finite number of fixed coin and bank note denominations. Assuming that the cashier has access to unlimited quantities of bank notes and coins of every denomination, the cashier needs to give change *using the smallest number of coins/bank notes* given a specific amount of money.

Input.

- $M[1..K]$: array of coin/bank note denominations sorted in descending order: $M[1] > M[2] \ldots > M[K]$.
- K: number of coins/bank notes.
- $Amount$: the amount of change to be given. (e.g., 27 for 27 cents, 234 for \$2.34, etc.)

Output. An array $Change[1..K]$. $Change[i]$ is the number of coins/bank notes of denomination $M[i]$.

Problem 2: Egyptian Fractions

Problem. Given a simple fraction $\frac{n}{m}$ $(n < m)$, represent it as the sum:

$$\frac{n}{m} = \sum_{i=1}^{k} \frac{1}{a_i}.$$

Historic Note. Ancient Egyptians represented fractions as sums of unit fractions (i.e., fractions of the form $\frac{1}{s}$. For example,

$$\frac{8}{9} = \frac{1}{2} + \frac{1}{3} + \frac{1}{18}.$$

Theorem. Any rational number $0 < x < 1$ can be represented in the *Egyptian fraction* form. That is, for any $0 < x < 1$, there exists such $k > 0$, and such $a_1 < a_2 < \ldots a_k$, that

$$x = \sum_{i=1} k \frac{1}{a_i}.$$

Input. Two numbers: n (numerator) and m (denominator), s.t., $n < m$.

Output. An array $EF[0..K]$, where $EF[0] = K$ and $EF[i]$ for $1 \leq i \leq K$ is the denominator of the ith largest unit fraction, such that:

$$\frac{n}{n} = \sum_{i=1}^{EF[0]} \frac{1}{EF[i]}.$$

Problem 3: Activity Selection

Problem. You are trying to schedule a list of activities to take place in a conference room. A number of activity requests are presented to you. For each activity, the request contains the start and the end time. Your goal is to produce the activity schedule for the conference room that would accommodate as many activities as possible.

Notes.

- Each activity A_i from the input list $A_1, \ldots A_N$, has the duration

  $$[start(A_i), end(A_i)),$$

  a semi-open interval.
- Only **one activity per time** can be scheduled.
- Two activities, whose intervals overlap are called conflicting.
- From each set of conflicting activities, no more than one activity can be scheduled.
- The **best** solution has **the largest possible number** of scheduled activities.

  (Note, that, generally speaking, other criteria are also possible, e.g., maximizing the time the room is in use).

Input.

- N: number of activities.
- $A[1..N][1..2]$: array of start ($A[i][1]$) and end ($A[i][2]$) times of the activities.

Output. $S[0..N]$. $S[0]$ contains the number of scheduled activities[1]. For $1 \leq i \leq N$, $S[i] = 1$ if $A[i]$ is scheduled and $S[i] = 0$ is $A[i]$ is NOT scheduled.

Problem 4: Optimal Announcement Schedule

Problem. Consider a schedule of activities just like in the **Activity Selection** problem. In this problem, the activities happen in various location on your campus. Each room is equipped with a PA system. You need to make a really important announcement so that *every person participating in any activity* would hear it. Find the times at which you should make the announcement assuming that you want to use the PA system as few times as possible.

Note. This is also known as the minimal stab set problem:

- A time point x stabs an activity $A(start(A), end(A))$ iff $x \in [start(A), end(A))$.
- Given a set of activities $A_1, \ldots A_N$, a stab set S is the set of time points $S = \{s_1, \ldots s_K\}$, such that every activity A_i is stabbed by at least on time point.
- The optimal announcement schedule problem can now be rephrased as follows:

  > Given a set of activities, find a stab set of smallest cardinality for it.

Input.

- N: number of activities.
- $A[1..N][1..2]$: array of start ($A[i][1]$) and end ($A[i][2]$) times of the activities.

Output. An array $S[0..K]$, where $S[0] = K$ is the number of stab points and $S[1], \ldots, S[K]$ is the list of stab points. (K must be the smallest possible)

Problem 5: Room Assignment

Problem. This time you have control over a large number of rooms. As in the Activity Selection problem, you receive requests for activities to schedule. Your goal is to schedule **all** activities using *as few rooms as possible.*

Notes. Another name for this problem is interval graph coloring. It is an instance of a *more general* graph coloring problem: given a graph $G = \langle V, E \rangle$, assign colors to all nodes such that every edge has end-points of different colors.

Each activity can be viewed as an node in a graph. Two activities are connected by an edge if they conflict. Each color is matched to a room number. The graph obtained this way is called an interval graph.

[1] Because we can.

Input.

- N: number of activities.
- $A[1..N][1..2]$: array of start ($A[i][1]$) and end ($A[i][2]$) times of the activities.

Output. Array $R[1..N]$, where $R[i]$ is the room number (color) of activity $A[i]$.

Fractional Knapsack

Knapsack Packing problems come in a number of varieties and usually involve selecting the right items/quantities to optimize the weight/value of a load that can be carried in a *knapsack* of a limited capacity.

The Fractional Knapsack is one of the simpler versions of the problem.

Problem. You have a limited-capacity knapsack and want to pack it tightly with a variety of goods from a warehouse. A finite selection of goods is available for choosing from the warehouse. With each type of good we associate the available quantity and the cost of a unit measure. The task is to determine the quantity of each type of good to put in the knapsack, such that the total value of the goods in it is maximized.

Input.

- C: capacity of the knapsack.
- K: number of different goods.
- $G[1..K]$: available quantities of goods.
- $V[1..K]$: value of goods.

Output. Array $Sack[1..N]$, such that,

- $Sack[i]$ represents the amount of good with index i to put in the knapsack.
- For all $1 \leq i \leq N$, $0 \leq Sack[i] \leq G[i]$.
- $\sum_{i=1}^{K} Sack[i] = C$ (knapsack capacity reached).
- $\sum_{i=1}^{K}(Sack[i] \cdot V[i])$ is maximized.