Greedy Algorithms

Charlie Li

Coin Problem

- Suppose you are given infinitely many coins of the following values:
- \$10, \$5, \$2, \$1, \$0.5, \$0.2, \$0.1

- What is the minimum number of coins needed to get \$M where
  - (i) M = 1.9
  - (ii) M = 7.8
  - (iii) M = 16.1

Coin Problem

- Answers:
  - (i) 4
  - (ii) 5
  - (iii) 4

- Why can we get the answer so quickly?

Coin Problem

- Cashier's algorithm
  - While there are outstanding amount, we take one of the coin with largest value but not exceeding the outstanding amount
- () means the outstanding amount
- (\$1.9) = \$1 + (\$0.9) = \$1 + \$0.5 + (\$0.4) = \$1 + \$0.5 + \$0.2 + \$0.2
- (\$7.8) = \$5 + (\$2.8) = \$5 + \$2 + (\$0.8) = \$5 + \$2 + \$0.5 + (\$0.3) = \$5 + \$2 + \$0.5 + \$0.2 + \$0.1
- (\$16.1) = \$10 + (\$6.1) = \$10 + \$5 + (\$1.1) = \$10 + \$5 + \$1 + \$0.1

Coin Problem

- Does Cashier’s algorithm always give us the optimal solution for HK’s coins ?
  - {$10, $5, $2, $1, $0.5, $0.2, $0.1}

- Is there any counter example? If no, why is it optimal?

- If you cannot come up with the answer, never mind, let’s look at another example

Coin Problem

- Consider the system of stamps
- \$5, \$3.7, \$3.1, \$2.9, \$2.3, \$2.2, \$2, \$1.7, \$1, \$0.5, \$0.2, \$0.1
- What is the minimum number of stamps needed to get \$M where
  - (i) M = 1.9
  - (ii) M = 7.8
  - (iii) M = 16.1

Coin Problem

- Using the same algorithm,
- (i) \$1.9 = \$1.7 + \$0.2
- (ii) \$7.8 = \$5 + \$2.3 + \$0.5
- (iii) \$16.1 = \$5 + \$5 + \$5 + \$1 + \$0.1

- The algorithm gives optimal solution for (i) and (ii) but not (iii)
- \$16.1 = \$5 + \$3.7 + \$3.7 + \$3.7
- We can use 4 stamps only but greedy told us 5

Coin Problem

- Go back to the original problem

- Why the Cashier's algorithm gives us the optimal solution for the coins with value {\$10, \$5, \$2, \$1, \$0.5, \$0.2, \$0.1}

- Let's see if there are any properties for the optimal solution

Coin Problem

- Property 1
  - The optimal solution contains at most one $0.1 coin.
    - If there are more than one $0.1 coin, we can change two $0.1 into one $0.2
  - Similarly, the optimal solution contains at most one $5, $1, $0.5
- Property 2
  - The optimal solution contains at most two $0.2 coins.
    - If there are more than two $0.2 coins, we can change three $0.2 into $0.5 and $0.1
  - Similarly, the optimal solution contains at most two $2
- Property 3
  - The optimal solution contains no $0.1 coins if it contains two $0.2 coins
    - If there are one $0.1 and two $0.2, we can change them to $0.5
  - Similarly, optimal solution contains no $1 coins if it contains two $2 coins

Coin Problem

Value of coin	Constraint	Max. value if we only use coins with less value
\$0.1	At most 1	\$0
\$0.2	At most 2 no \$0.1 when 2	\$0.1
\$0.5	At most 1	\$0.4
\$1	At most 1	\$0.9
\$2	At most 2 no \$1 when 2	\$1.9
\$5	At most 1	\$4.9
\$10	No limit	\$9.9

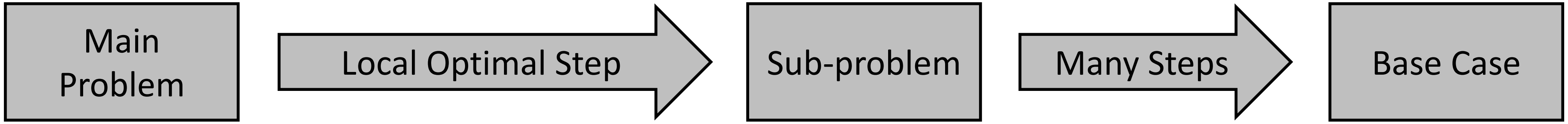
Coin Problem

- From the table, we cannot get \$10 just using coins less than \$10, so we must use \$10 coin
- This is also true for \$5, \$2, \$1, \$0.5, \$0.2 and \$0.1

- So Cashier’s Algorithm can give optimal solution for HK’s coin system

Greedy Algorithm

- A greedy algorithm attempt to solve the problem by choosing locally optimal step and reduce the original problem into another sub-problem then solve the sub-problem using the same strategy
- It is not a specific algorithm, it is just an idea
- It is important to know that for a greedy algorithm, the previous choices should not affect the decision
- Unlike brute force and DP(will be taught in April), a greedy algorithm may not always give an optimal solution, eg. The stamp system



Coin Problem – How Greedy?

- Answer to main problem =
  - Optimal step's effect to answer + Answer to sub-problem

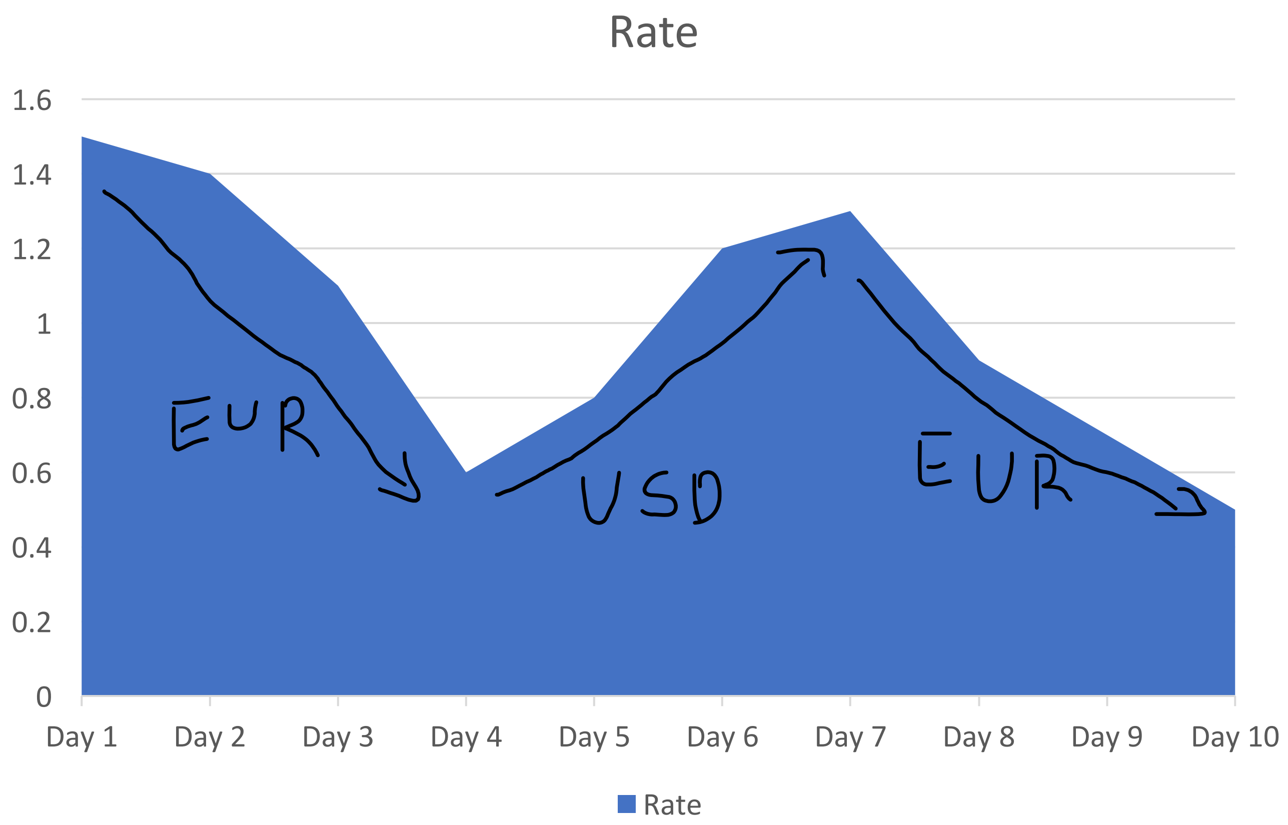
- Problem: Remaining amount K
  - We will greedily take a coin with largest value but not exceeding K
  - We will take a coin d where $d = \max\{D_i | D_i \leq K\}$
  - The sub-problem is remaining amount K – d
  - Answer for remaining amount K: Ans(K) = 1 + Ans(K - d)
  - Answer to main problem: Ans(N)
  - Base case: Ans(0) = 0

J042 Currency Exchange

- You know the exact exchange rate from USD to EUR for the following N days, you have US$ M at day 1, how much USD will you have if you trade optimally?

J042 Currency Exchange

- When thinking greedily, we should be holding EUR when the rate decrease and holding USD when the rate increases



J042 Currency Exchange

- Try this input:

10 100

1.5 1.4 1.1 0.6 0.8 1.2 1.3 0.9 0.7 0.5

- Between day 1 and day 2, we will change to EUR at day 1 and change back at day 2 since the rate decreases
- Between day 2 and day 3, we will change to EUR at day 2 and change back at day 3 since the rate decreases
- …
- Between day 4 and day 5, we will do nothing since the rate increases …

J042 Currency Exchange

- The code looks like this:

```
#include<cstdio>

int n;
double m, c[10004];

int main() {
    scanf("%d %lf", &n, &m);
    for (int i = 0; i < n; i++) scanf("%lf", &c[i]);
    for (int i = 1; i < n; i++)
        if (c[i - 1] > c[i])
            m = m/c[i]*c[i - 1];
    printf("%.2lf\n", m);
}
```

J042 Currency Exchange

- Why is this optimal?
- What if we hold EUR when the rate increase?
- What if we hold USD when the rate decrease?
- How are these compare to our greedy solution?

Fractional Knapsack

- Suppose you have a cup with volume V ml.
- There are N flavors of drinks, for each flavor, there are only a_i ml available and contain a total of s_i g sugar
- At most how many grams of sugar can you consume if you fill up the cup optimally?

Fractional Knapsack

Flavor	Volume available (ml)	Total sugar (g)
A	320	35
B	220	0
C	240	40
D	200	30

Fractional Knapsack

- We can just sort the flavors according to sugar per unit volume, and then fill up our cup according to the order

Flavor	Volume available (ml)	Total sugar (g)	Sugar per volume	Used
C	240	40	0.167	240
D	200	30	0.15	200
A	320	35	0.109	160
B	220	0	0	0

Fractional Knapsack

- Why is this optimal?
- Try to answer seeing what will happen if we change some of C into D, A or B

Flavor	Volume available (ml)	Total sugar (g)	Sugar per volume	Used
C	240	40	0.167	240
D	200	30	0.15	200
A	320	35	0.109	160
B	220	0	0	0

0-1 Knapsack

- What will happen if the question is changed to “If you used some flavor X, you must use up all flavor X”
- Can we still use greedy? Can you think of counter example?

Flavor	Volume available (ml)	Total sugar (g)	Sugar per volume	Choice
C	240	40	0.167	0 or 240
D	200	30	0.15	0 or 200
A	320	35	0.109	0 or 320
B	220	0	0	0 or 220

S011 Activities

- You have N activities to join.
- The i^{th} activity start at S_i and end at E_i
- For any two activities, you can join both of them iff they do not overlap
- Find the max. no. activities you can join

S011 Activities

- 4 ways to greedy:
  - (1) Attend the activity with smallest starting time
  - (2) Attend the activity with smallest ending time
  - (3) Attend the activity with smallest conflicts
  - (4) Attend the activity with shortest interval
- Three of them are wrong, do you know which three?
- What are the counter examples?

S011 Activities

- The correct answer is (2) attend the activity with smallest ending time
- We first sort the activities with ascending end time.
- Then we will attend the first one. (local optimal step)
- So we cannot join any other activities before the one end, we just eliminate those we cannot join. (Reduce to sub-problem)
- Repeat until there are no activities left. (base case)

S011 Activities

- The code looks like this:

```
bool cmp(activity x, activity y) {
    return x.e < y.e;
}

bool cmp2(activity x, activity y) {
    return x.i < y.i;
}

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i].s >> a[i].e;
        a[i].i = i;
    }
    sort(a + 1, a + n + 1, cmp);
    for (int i = 1, j = 0; i <= n; i++) {
        if (a[i].s < j) continue;
        j = a[i].e;
        a[i].u = true;
        c++;
    }
    sort(a + 1, a + n + 1, cmp2);
    cout << c << endl;
    for (int i = 1; i <= n; i++) if (a[i].u) cout << i << endl;
}
```

S011 Activities

- Why is this optimal?

- Consider only 2 activities, if we cannot join both, which should we join?
- We should join the one which ends earlier, be cause this can help us reserve more time to join other activities
- By applying the same argument to all pairs of activities, we know that always attending the one with smallest ending time is the best.

M1713 Biscuit Clicker

- When the production rate is P, Alice will get a biscuit every 1/P seconds
- The basic production rate is 1
- There are N upgrades where every upgrade can be bought at most once
- For the i^{th} upgrade, the cost is C_i and it multiply the production rate by P_i
- Find **a** fastest way to collect K biscuits in the bank

M1713 Biscuit Clicker

- If we consider only two upgrades
- We will buy upgrade i before j if

$$C_i + \frac{C_j}{P_i} \leq C_j + \frac{C_i}{P_j}$$

- If we consider only one upgrade
- We will buy upgrade i if

$$C_i + \frac{K}{P_i} \leq K$$

- Actually, if we sort according to the first inequality and pick upgrades according to the second inequality, this will gives us the full solution

M1713 Biscuit Clicker

- The code looks like this:

```
bool cmp(int x, int y) {
    return c[x] + 1. * c[y] / p[x] < c[y] + 1.* c[x] / p[y];
}

int main() {
    cin >> n >> k;
    for (int i = 0; i < n; i++) f[i] = i;
    for (int i = 0; i < n; i++) cin >> c[i] >> p[i];
    for (int i = 0; i < n; i++)
        if (c[i] + 1. * k / p[i] < k) {
            u[i] = true;
            tt++;
        }
    sort(f, f + n, cmp);
    cout << tt << endl;
    for (int i = 0; i < n; i++) if (u[f[i]]) cout << f[i] + 1 << " ";
    cout << endl;
}
```

M1713 Biscuit Clicker

- The most important part is to see the transverse property of the first inequality

$$C_i + \frac{C_j}{P_i} \le C_j + \frac{C_i}{P_j}$$

- If we buy upgrade i before upgrade j, and if we buy upgrade j before upgrade k, then we should buy upgrade i before upgrade k
- If this property holds, then we may try to use greedy to solve the problem
  - First, this tell us that we can sort the upgrades
  - Second, we can easily show if there are any inversions, it will not be the optimal solution

Conclusion

- When can we use greedy?
  - First, when you think you can (TRUE!!!)
  - Second, you cannot find counter example (You should try to do so)
  - The system gives you full feedback (it worth trying)
    - As you can see, greedy algorithm is not long, so it worth trying if you can code fastly
  - points are given per subtask and you have no idea (it worth trying)
    - Greedy solution sometimes give extremely good approximation for the optimal solution
- It is not realistic to prove the correctness during the competition
  - Just prove in mind for a little bit is enough

Suggested Tasks

- 01014 Stamps
- D109 Giving changes
- M0632 Machine Scheduling
- M0633 Children’s Game
- J044 Amazing Robot
- J064 Cave Adventures
- J154 Father’s Will

Extra: M0633 Children's Game

- Idea similar to M1713 Biscuit Clicker, think of what will happen if N = 2
- The code looks like this:

```cpp
#include<iostream>
#include<string>
#include<algorithm>

using namespace std;

int n;
string a[55];

bool cmp(string x, string y) {
	return [redacted]
}

int main() {
	cin >> n;
	for (int i = 0; i < n; i++) cin >> a[i];
	sort(a, a + n, cmp);
	for (int i = 0; i < n; i++) cout << a[i];
	cout << endl;
}
```