Greedy Algorithm

Hayden Lee {s13215}
2021-03-06

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

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 Dynamic Programming, a greedy algorithm may not always give an optimal solution.

➔ Knowing when to use / you can use greedy algorithm is IMPORTANT

Greedy Algorithm vs Dynamic Programming

- 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
- What if locally optimal steps do not lead to **globally** optimal solution 😕 ?

- We probably need to consider all local steps and choose the best one by calculation
- Attend Dynamic Programming training sessions!🙂

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
- What if the problem is not reducible 😕 ?

- We will need to use other techniques/algorithms
- Attend ALL training sessions!🙂

Application of greedy algorithm in graph theory

Examples:

- Dijkstra's Algorithm on shortest-path
- Prim's Algorithm on minimum spanning tree
- Kruskal's Algorithm on minimum spanning tree

- Learning greedy algorithm builds you with a solid foundation for graph(II)
  (well not really....)

Illustration - Fractional Knapsack Problem

- 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?

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

What did we do

- Original problem: Maximize the sugar content for a cup with volume V, using the available drinks
- Reduced problem: Maximize the the sugar content for every 1 mL of drink for i = 1...V, using the available drinks
- Locally-optimal step: choose the remaining drink available with the highest amount of **sugar per mL**
- Does the locally-optimal steps lead to a globally-optimal solution?

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

What happened?

- Original problem: Maximize the sugar content for a cup with volume V, using the available drinks
  [but you have to use up the whole drink at once]
- Reduced problem: Maximize the the sugar content for every 1 mL of drink for i = 1...V, using the available drinks
- Locally-optimal step: choose the remaining drink available with the highest amount of **sugar per mL**
- Does the locally-optimal steps lead to a globally-optimal solution?

What happened?

- Does the locally-optimal steps lead to a globally-optimal solution?

- ❑ No😕
- ❑ The constraint of our remaining volume makes total volume & total sugar content matters as well
- ❑ Remaining volume = 400 mL
- ❑ drink X = 300 mL + 30g sugar
- ❑ drink Y = 400 mL + 31g sugar
- ❑ drink Z = 500 mL + 100g sugar

What should we do?

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

- Brute force (try all combinations of pouring the drinks)
- Dynamic Programming

J181 Wings and Nuggets

Subtask 3

- 4 nuggets: $a
- 6 nuggets: $b
- 9 nuggets: $c

Question:

- What is the minimum cost of buying at least X nuggets?

Original constraints: X <= 100
Let's consider X <= 100000 or even 1000000000

J181 Wings and Nuggets

- We can buy multiple of the same kind
- Therefore, we can model this task into a 0-1 Knapsack problem by making copies of the same item.
- The number of copies required = ceil(X / #pieces)
  - Item 1: $a (0 or 4)
  - Item 2: $a (0 or 4)
  - Item 3: $a (0 or 4)
  - ...
  - Item 25001: $b (0 or 6)
  - Item 25002: $b (0 or 6)
  - ...
  - Item 41668: $c (0 or 9)
  - Item 41669: $c (0 or 9)

J181 Wings and Nuggets

- As mentioned before, there is a dynamic programming solution for 0-1 knapsack problems which runs in O(nW) where
  - n = number of items
  - W = capacity (here, it's number of nuggets required)
- By making copies of the items, n also grows linearly with number of nuggets.
- Therefore, time complexity is $O(X^2)$
  - Cannot pass X <= 100000
- Can we do better? Can we apply some greedy algorithm?

Partially greedy algorithm

- If X is always a multiple of 36, will there exist a solution where we buy two kinds of nuggets?
  - No.
  - It is always optimal to buy one kind of nuggets.
- Then we can come up with the following greedy algorithm
  - Try to buy as many bundles of 36 (LCM of 4, 6 and 9)
  - Pay the minimum of (9 * a, 6 * b, 4 * c) per bundle
  - For the remaining nuggets (X % 36), we can use exhaustion because the number is small enough so it runs quickly.
- Is this algorithm correct?

Partially greedy algorithm

3	7	Accepted	0.001 s	1.535 MB	100.000
3	8	Accepted	0.001 s	1.535 MB	100.000
3	9	Accepted	0.001 s	1.543 MB	100.000
3	10	Accepted	0.000 s	1.535 MB	100.000
3	11	Wrong Answer	0.001 s	1.535 MB	
3	12	Accepted	0.001 s	1.539 MB	100.000
3	13	Accepted	0.001 s	1.539 MB	100.000
3	14	Accepted	0.000 s	1.535 MB	100.000
3	15	Accepted	0.001 s	1.539 MB	100.000

Counter Example

It is easier to illustrate with just 4 and 9 nuggets by making 6 expensive.

Pack	Price	Price per piece
4	40	10.0
6	89	14.8
9	91	10.1

If X = 36 (a bundle), we would choose to buy packs of 4 pieces because the price per piece is the least expensive.

What about X = 37?

According to our algorithm, we would buy a pack of 4 pieces for just 1 piece. It seems to be wasteful.

Counter Example

10 packs of 4 pieces = $400.

Type	Price	Price per piece
4	40	10.0
6	89	14.8
9	91	10.1

The optimal way would be to buy 7 packs of 4 pieces + 1 pack of 9 pieces.

Total cost = 280 + 91 = $371.

So... can we still use greedy algorithm?

Partially greedy algorithm

- Let say there are 3 types: Q_1 cost P_1, Q_2 cost P_2, Q_3 cost P_3
- Bundle size B = LCM(Q_1, Q_2, Q_3)
- Let say type 1 is cheapest in terms of cost per item, then
  Cost per bundle = B / Q_1 * P_1
- The optimal solution may be to buy some of each type,
  but you cannot buy more than B / Q_2 type 2s not more than B / Q_3 of type 3s.
- If you buy more, it's cheaper to switch all of them to type 1.
- Therefore, as long as we buy not more than (floor(X / B) - 2) bundles, we can still find the optimal solution using exhaustion.

Partially greedy algorithm

- To generalize, if there are K different kinds of items, we can greedy up to floor(X / B) - K + 1 bundles.
- Solve the remaining items using exhaustion or dynamic programming.

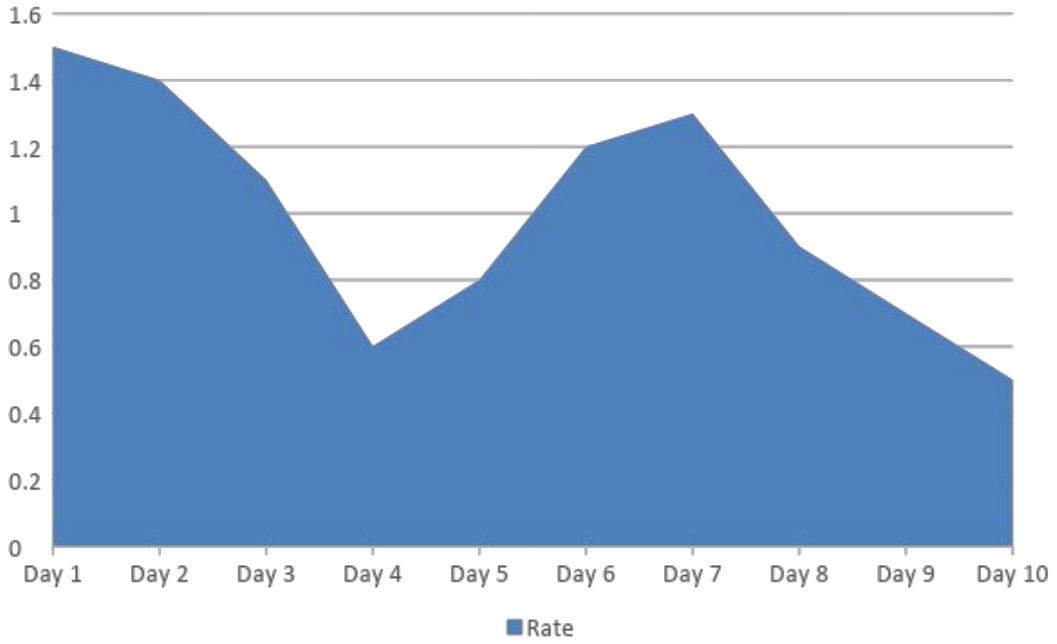
- Whether the algorithm can be applied depends on the problem statement. If bundle size B can be large, then it cannot be applied.
  - For example, if the sizes are co-prime, then the LCM of the sizes could be huge.
- If the sizes are given and are small, then the algorithm can be applied.

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:

```cpp
#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?

Longest Increasing Subsequence

- Given an integer array, find the length of its longest strictly increasing subsequence.
- For example, the length of LIS of the following array is 4

8 1 2 7 5 6 3 4

Longest Increasing Subsequence

- For an array a[1..n] to have an increasing subsequence of length k, we need the following two statement to be true
  - The array[1..n-1] has an increasing subsequence of length k-1
  - One of such increasing subsequence has the k-1th element smaller than a[n]

Longest Increasing Subsequence

- So it maybe useful if we find the length of LIS of the whole array first by finding the one of its prefixes'
- However, instead of recording the LIS, it would be more useful if we record the smallest k^{th} element in any increasing subsequence
  - We can check which increasing subsequence a[n] can be appended to
- It is easy to show that if we find the answer in this way, it will be correct

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For []:

i	1	2	3	4
v[i]	+inf	+inf	+inf	+inf

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For [8]:

i	1	2	3	4
v[i]	8	+inf	+inf	+inf

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For [8 1]:

i	1	2	3	4
v[i]	1	+inf	+inf	+inf

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For [8 1 2]:

i	1	2	3	4
v[i]	1	2	+inf	+inf

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For [8 1 2 7]:

i	1	2	3	4
v[i]	1	2	7	+inf

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For [8 1 2 7 5]:

i	1	2	3	4
v[i]	1	2	5	+inf

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For [8 1 2 7 5 6]:

i	1	2	3	4
v[i]	1	2	5	6

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For [8 1 2 7 5 6 3]:

i	1	2	3	4
v[i]	1	2	3	6

Longest Increasing Subsequence

- For example, use a[] = 8 1 2 7 5 6 3 4
- We record the smallest k^{th} element in any increasing subsequence

For [8 1 2 7 5 6 3 4]:

i	1	2	3	4
v[i]	1	2	3	4

Longest Increasing Subsequence

- Finally, we just need to check the length of v, it will be the answer.
- Note, the array v may not be an LIS (see the second last step in the example), so if you are asked to output an LIS, don't just output v directly
- How can we find which cell to update quickly?
  - Use binary search

What did we do?

- Greedily use a number as part of the LIS whenever it's possible to
- Greedily update our **potential** answer with the smallest number possible whenever we found one
- In other words, we have considered the best possible cases -> the length of LIS we found will be maximized too

Longest Increasing Subsequence

- In C++, it will be very handy using std::vector and std::lower_bound or std::upper_bound (try and look for the differences yourself)
- The code can be as short as follows:

```
for (int i = 0; i < n; i++) {
    auto it = lower_bound(v.begin(), v.end(), a[i]);
    if (it == v.end()) {
        v.push_back(a[i]);
    } else {
        *it = a[i];
    }
}
printf("%d\n", v.size());
```

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
- Only one of them is correct (which one?)
- 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:

```cpp
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, because 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:

```
10 bool cmp(int x, int y) {
11     return c[x] + 1. * c[y] / p[x] < c[y] + 1.* c[x] / p[y];
12 }
13
14 int main() {
15     cin >> n >> k;
16     for (int i = 0; i < n; i++) f[i] = i;
17     for (int i = 0; i < n; i++) cin >> c[i] >> p[i];
18     for (int i = 0; i < n; i++)
19         if (c[i] + 1. * k / p[i] < k) {
20             u[i] = true;
21             tt++;
22         }
23     sort(f, f + n, cmp);
24     cout << tt << endl;
25     for (int i = 0; i < n; i++) if (u[f[i]]) cout << f[i] + 1 << " ";
26     cout << endl;
27 }
```

M1713 Biscuit Clicker

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

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

- 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;
}
```