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

Greedy Algorithms

David Wai {wjx}
2023-03-04

Greedy Algorithm

Flow of a greedy algorithm:

- Solve the problem by choosing **locally optimal step**
- Reduce the original problem into another sub-problem
- Solve the sub-problem using the same strategy

It is just a **concept**, but not a specific algorithm

It is important to know that the previous choices should not affect the latter choices in a greedy strategy

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

Attend graph sessions for details
We will only talk about basic greedy strategies today
And we will focus on how to come up different greedy strategies when doing different problems

Coin Problem

Suppose you have infinite 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:

- M = 1.7
- M = 9.2
- M = 14.1

Coin Problem

Answers:

- 3
- 4
- 4

Why can you find it so fast?

Is that optimal? Why?

Solution

- We try to use $10
- If M >= 10, then pay $10; otherwise we try $5
- etc.

Solution

- We try to use $10
- If M >= 10, then pay $10; otherwise we try $5
- etc.

Seems trivial, right?

Another Example

The same question but having coins with following values:
$0.2, $1, $1.7, $3

Try to pay $3.4

Solution

Coin values: $0.2, $1, $1.7, $3, target: $3.4

If we use the strategy before:

- $3.4 = $3 + $0.2 + $0.2 (3 coins)

Solution

Coin values: $0.2, $1, $1.7, $3, target: $3.4

If we use the strategy before:

- $3.4 = $3 + $0.2 + $0.2 (3 coins)

Is this the optimal solution?

Solution

Coin values: $0.2, $1, $1.7, $3, target: $3.4

If we use the strategy before:

- $3.4 = $3 + $0.2 + $0.2 (3 coins)

Is this the optimal solution?

How about:

- $3.4 = $1.7 + $1.7 (2 coins only)

Why?

Why our strategy is not working for this case
What's wrong?

Let's back to the first example

Coin Problem

Suppose you have infinite 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:

- M = 1.7
- M = 9.2
- M = 14.1

Observation

For optimal solution:

1. No more than 1 $0.1 is needed
   a. If 2 $0.1 is used, we can change it to $0.2
   b. Same for $5, $1, $0.5
2. No more than 2 $0.2 is needed
   a. if 3 $0.2 is used ($0.6), we can change it to $0.5 + $0.1
   b. Same for $2
3. No $0.1 if 2 $0.2 is used
   a. If 1 $0.1 and 2 $0.2 is used, we can change it to $0.5
   b. Also, no $1 if 2 $2 is used

Observation

Value of coin	Constraints	Max. value if we only use coins with less value
$0.1	<= 1	$0
$0.2	<= 2 No $0.1 when 2	$0.1
$0.5	<= 1	$0.4
$1	<= 1	$0.9
$2	<= 2 No $1 when 2	$1.9
$5	<= 1	$4.9

Greedy Algorithm

What if optimal local steps do not lead to optimal global solution?

Try to use other algorithms

- For example, we may need to consider all local steps and choose the best one by calculation, which can be done by dynamic programming
- Attend others sessions for more!

Unlike brute force and dynamic programming, a greedy algorithm may not always give an optimal solution

So knowing when / whether to use a greedy algorithm is important

Cashier Algorithm

The strategy we use is called Cashier Algorithm

Cashier Algorithm can give optimal solution to Hong Kong's coin system, but does not work for all coin systems

If you want to ensure it works, you need to prove by yourself

Cashier Algorithm

Problem: optimal solution of paying $M

Sub-problem: optimal solution of paying $M - $d

- We want to minimize $M - $d, thus we need to maximize $d

Base case: optimal solution of paying $0 → 0

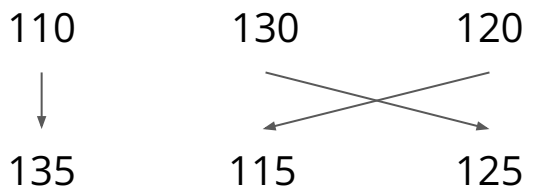
Sample Problem

Please read the problems:

- J054 - Competition
- M2202 - Poodle
- J042 - Currency Exchange

J054 Competition

- Direct battle of 2 schools
- Each with N students and each with different strength
- Strength of your team: a_i
- Strength of opponents: b_i
- You know your opponent's sequence
- Try to win as many rounds as possible



J054 Competition

Subtask 1 (50%): 1 <= *N* <= 10

Brute Force

- Try every combination of 1, 2, 3, ... ,*N*
  - 1, 2, 3, 4, 5
  - 1, 2, 3, 5, 4
  - ...
  - 5, 4, 3, 2, 1
- Check the number of wins and determine the best permutation

Time complexity: *O(N!)*

J054 Competition

Full solution: $N <= 5000$

For the i^{th} student:

- Find the strongest opponent such that $a_i > b_i$
  - Win, increase the number of wins by 1
- Otherwise, if there is no such opponent, we find the strongest opponents left, and lose to it. (sacrifice)

Time complexity: $O(N^2)$

M2202 Poodle

- There are *N* stacks of money on a number line, each with different amount of coins
- Pick up whole stack when arrive position *i*
- Walk to next position takes 1 second
- Money will come back after 2 seconds
- Start at position 1
- Find the maximum number of coins after *T* seconds

M2202 Poodle

Once we walk to two positions that have maximum sum, we are likely to walk back and forth between the two positions

- Going to other positions won't let you get more coins

But we start at position 1, so we need to walk to that position first

We can calculate the value for every prefix and find the maximum among them

Time Complexity: $O(N)$

J042 Currency Exchange

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

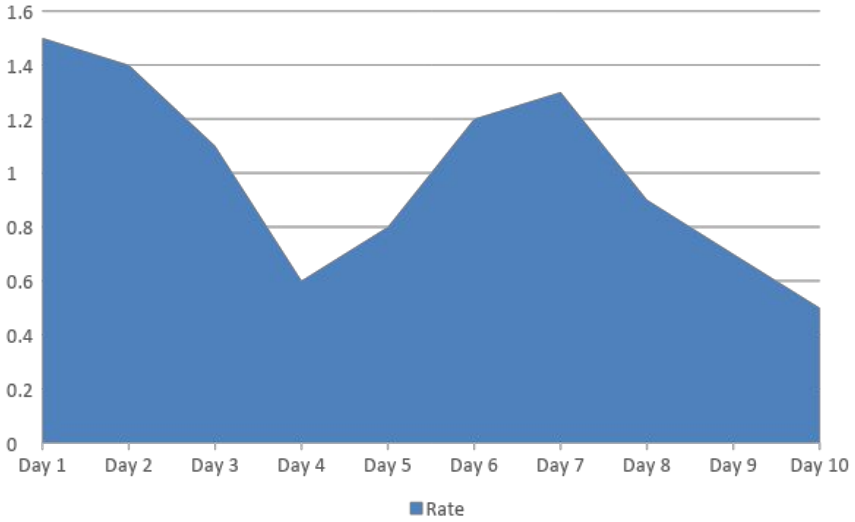
Explanation:

- Rate = 1.78: 1 USD to 1.78 EUR
- Rate = 0.9: 1 USD to 0.9 EUR

J042 Currency Exchange

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

- USD is worthy when increasing
- So holding USD when it's worthy
- Holding EUR when USD is not worthy



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 ...

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

Flavour	Volume	Total sugar
A	320	35
B	220	0
C	240	40
D	200	30

Fractional Knapsack

Any idea?

Try to think Greedily

Fractional Knapsack

We can calculate a new column: sugar per volume

Flavour	Volume	Total sugar	Sugar per volume
A	320	35	0.109
B	220	0	0
C	240	40	0.167
D	200	30	0.15

Fractional Knapsack

Then sort by it

Flavour	Volume	Total sugar	Sugar per volume
C	240	40	0.167
D	200	30	0.15
A	320	35	0.109
B	220	0	0

0-1 Knapsack

Let's change the problem a bit:

- If you used some flavor X, you must use up all flavor X

Is the strategy above still giving an optimal solution?

0-1 Knapsack

Let's change the problem a bit:

- If you used some flavor X, you must use up all flavor X

Is the strategy above still giving an optimal solution?

- Sadly, no
- This is actually a standard problem for dynamic programming
- Go to attend dynamic programming sessions for more details!

Sample Problem

Please read the problem J181 - Wings and Nuggets

You can focus on subtask 3 as we will discuss it later

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 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

Pack	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.

M1222 Longest Increasing Subsequence

Given an integer array, find the length of its longest strictly increasing subsequence.

Sample input: 8 1 2 7 5 6 3 4

Answer: 4

M1222 Longest Increasing Subsequence

For an array a[1 .. n] to have an increasing subsequence of length *k* which ends with a[n], we need the following two statements to be true:

- The array[1 .. n - 1] has an increasing subsequence of length *k* - 1
- One of such increasing subsequence has the $(k - 1)^{th}$ element smaller than a[n]

M1222 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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth element in any increasing subsequence

For []:

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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth element in any increasing subsequence

For [8]:

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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth element in any increasing subsequence

For [8 1]:

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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth element in any increasing subsequence

For [8 1 2]:

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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth element in any increasing subsequence

For [8 1 2 7]:

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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth element in any increasing subsequence

For [8 1 2 7 5]:

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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth element in any increasing subsequence

For [8 1 2 7 5 6]:

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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth element in any increasing subsequence

For [8 1 2 7 5 6 3]:

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

M1222 Longest Increasing Subsequence

For example, a[] = 8 1 2 7 5 6 3 4

We record the smallest kth 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

M1222 Longest Increasing Subsequence

Finally, we just need to check the length of v, it will be the answer.
Note that 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

Sample Problem

Please read the problem S011 - Activities

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 if and only if they do not overlap
- Find the maximum number of activities you can join

S011 Activities

4 ways to ‘greedy’:

- Attend the activity with smallest starting time
- Attend the activity with smallest ending time
- Attend the activity with smallest conflicts
- 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: attend the activity with smallest ending time

Join the one which ends earlier, this can help us reserve more time to join other activities

Solution:

- Sort the activities with ascending end time.
- Attend the first one. (local optimal step)
- Eliminate activities that we cannot join. (Reduce to sub-problem)
- Repeat until there are no activities left. (base case)

Sample Problem

Please read the problem M1713 - Biscuit Clicker

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 one upgrade i:

- We will buy upgrade i if $C_i + K / P_i < K$

If we consider only two upgrades i and j:

- We will buy upgrade i before j if $C_i + C_j / P_i < C_j + C_i / P_j$

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

M1713 Biscuit Clicker

The most important part is to see the transitive property of the first inequality
$C_i + C_j / P_i < C_j + C_i / P_j$

- If we buy upgrade i before upgrade j, and 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 tells 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

Conclusion

It is not realistic to prove the correctness during the competition

- Just prove in mind for a little bit is enough, trust yourself at that time
- Try to prove it completely afterwards, you may learn more from it

Suggested Tasks

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

References

https://assets.hkoi.org/training2021/greedy.pdf
https://assets.hkoi.org/training2022/greedy.pdf