inzva Algorithm Program

Bundle 5

DP - 1

Editor
Halil Çetiner

Reviewer(s)
Onur Yıldız

Contents

1 Introduction

Next section is about the `Greedy Algorithms` and `Dynamic Programming`. It will be quite a generous introduction to the concepts and will be followed by some common problems.

2 Greedy Algorithms

A **greedy algorithm** is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. A greedy algorithm never takes back its choices, but directly constructs the final solution. For this reason, greedy algorithms are usually very efficient.
The difficulty in designing greedy algorithms is to find a greedy strategy that always produces an optimal solution to the problem. The locally optimal choices in a greedy algorithm should also be globally optimal. It is often difficult to argue that a greedy algorithm works.

2.1 Coin Problem

We are given a value V, if we want to make change for V cents, and we have an infinite supply of each of the coins = { C1, C2, .. , Cm } valued coins (sorted in descending order), what is the minimum number of coins to make the change?

2.1.1 Solution

Approach:

```
1- Initialize the result as empty.
2- Find the largest denomination that is
smaller than amount.
3- Add found denomination to the result. Subtract
the value of found denomination from amount.
4 - If amount becomes 0, then print the result.
Else repeat the steps 2 and 3 for the new value of amount
```

```
1    def min_coins(coins, amount):
2        n = len(coins)
3        for (i in range(0, n)):
4            while (amount >= coins[i]):
5                #while loop is needed since one coin can be used multiple times
6                amount = amount - coins[i]
7                print(coins[i])
```

For example, if the coins are the euro coins (in cents) 200,100,50,20,10,5,2,1 and the amount is 548. Then the optimal solution is to select coins 200+200+100+20+20+5+2+1 whose sum is 548.

200	100	50	20	10	5	2	1	remaining	current_total
X								348	200
X								148	400
	X							48	500
			X					28	520
			X					8	540
					X			3	545
						X		1	547
							X	0	548

In the general case, the coin set can contain any kind of coins and the greedy algorithm does not necessarily produce an optimal solution.
We can prove that a greedy algorithm does not work by showing a counterexample where the algorithm gives a wrong answer.
In this problem we can easily find a counterexample: if the coins are 6,5,2 and the target sum is 10, the greedy algorithm produces the solution 6+2+2 while the optimal solution is 5+5.

2.2 Scheduling

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows:
We are given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, thus the minimum possible deadline for any job is 1. How do we maximize total profit if only one job can be scheduled at a time.

2.2.1 Solution

A Simple Solution is to generate all subsets of the given set of jobs and check an individual subset for the feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential. This is a standard Greedy Algorithm problem. Following is the algorithm:

```
1- Sort all jobs in decreasing order of profit.
2- Initialize the result sequence as the first job in sorted jobs.
3- Do the following for the remaining n-1 jobs
      If the current job can fit in the current result sequence without missing the
         deadline, add the current job to the result.
      Else ignore the current job.
```

```
# sample job : ['x', 4, 25] -> [job_id, deadline, profit]
# jobs: array of 'job's
def print_job_scheduling(jobs, t):
    n = len(jobs)

    # Sort all jobs according to decreasing order of profit
    for (i in range(n)):
        for (j in range(n - 1 - i)):
            if (jobs[j][2] < jobs[j + 1][2]):
                jobs[j], jobs[j + 1] = jobs[j + 1], jobs[j]

    # To keep track of free time slots
    result = [False] * t
    # To store result (Sequence of jobs)
    job = ['-1'] * t

    # Iterate through all given jobs
    for (i in range(len(jobs))):

        # Find a free slot for this job
        # (Note that we start from the last possible slot)
        for (j in range(min(t - 1, jobs[i][1] - 1), -1, -1)):

            # Free slot found
            if (result[j] is False):
                result[j] = True
                job[j] = jobs[i][0]
                break
    print(job)
```

2.3 Tasks and Deadlines

Let us now consider a problem where we are given n tasks with the durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn d - x points where d is the task's deadline and x is the moment when we finish the task. What is the largest possible total score we can obtain?
For example, suppose that the tasks are as follows:

$$\{task, duration, deadline\}$$
$$\{A, 4, 2\}, \{B, 3, 5\}, \{C, 2, 7\}, \{D, 4, 5\}$$

In this case, an optimal schedule for the tasks is C, B, A, D.
In this solution, C yields 5 points, B yields 0 points, A yields -7 points and D yields -8 points, so the total score is -10.
Surprisingly, the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks sorted by their durations in increasing order. The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks.

2.3.1 Solution

```
1     def order_tasks(task):
2       n = len(tasks)
3
4       # Sort all task according to increasing order of duration
5       for (i in range(n)):
6          for (j in range(n - 1 - i)):
7             if (tasks[j][1] > tasks[j + 1][1]):
8                tasks[j], tasks[j + 1] = tasks[j + 1], tasks[j]
9
10      point = 0
11      current_time = 0
12      # Iterate through all given tasks and calculate point
13      for (i in range(len(tasks))):
14         current_time = current_time + tasks[i][1]
15         point = point + (tasks[i][2] - current_time)
16
17      print(point)
```

2.4 Minimizing Sums

Where we are given n numbers and our task is to find a value x that minimizes the sum:

$$|a_1 - x|^c + |a_2 - x|^c + ... + |a_n - x|^c.$$

We focus on the cases c = 1 and c = 2.

- **Case c = 1**

In this case, we should minimize the sum

$$|a_1 - x| + |a_2 - x| + ... + |a_n - x|$$

For example, if the numbers are [1,2,9,2,6], the best solution is to select x = 2 which produces the sum

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

In the general case, the best choice for x is the median of the numbers, i.e., the middle number after sorting. For example, the list [1,2,9,2,6] becomes [1,2,2,6,9] after sorting, so the median is 2.
The median is an optimal choice, because if x is smaller than the median, the sum becomes smaller by increasing x, and if x is larger then the median, the sum becomes smaller by decreasing x. Hence, the optimal solution is that x is the median. If n is even and there are two medians, both medians and all values between them are optimal choices.

- **Case c = 2**

In this case, we should minimize the sum:

$$(a_1 - x)^2 + (a_2 - x)^2 + ... + (a_n - x)^2.$$

For example, if the numbers are [1,2,9,2,6], the best solution is to select x = 4 which produces the sum:

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46$$

In the general case, the best choice for x is the average of the numbers. In the example the average is $(1 + 2 + 9 + 2 + 6)/5 = 4$.
This result can be derived by presenting the sum as follows:

$$nx^2 - 2x(a_1 + a_2 + ... + a_n) + ({a_1}^2 + {a_2}^2 + ... + {a_n}^2)$$

The last part does not depend on x, so we can ignore it. The remaining parts form a function

$$nx^2 2xs$$

$$s = a_1 + a_2 + ... + a_n.$$

This is a parabola opening upwards with roots x = 0 and x = 2s/n, and the minimum value is the average of the roots x = s/n, i.e., the average of the numbers.

3 Dynamic Programming

Dynamic programming is a technique used to avoid computing multiple times the same sub-solution in a recursive algorithm. A sub-solution of the problem is constructed from the previously found ones. DP solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force etc.

3.1 Memoization - Top Down

Memoization ensures that a method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs (usually in a hash map).

To avoid the duplicate work caused by the branching, we can wrap the method in a class that stores an instance variable, cache, that maps inputs to outputs. Then we simply,

- check cache to see if we can avoid computing the answer for any given input, and
- save the results of any calculations to cache.

Memoization is a common strategy for dynamic programming problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the Fibonacci problem, above).
The other common strategy for dynamic programming problems is going bottom-up, which is usually cleaner and often more efficient.

3.2 Bottom Up

Going bottom-up is a way to avoid recursion, saving the memory cost that recursion incurs when it builds up the call stack.

Simply a bottom-up algorithm "starts from the beginning," while a recursive algorithm often "starts from the end and works backwards."

3.3 An example - Fibonacci

Let's start with an example which most of us are familiar, fibonacci numbers: finding the n^{th} fibonacci number defined by

$$F_n = F_{n-1} + F_{n-2} \quad \text{and} \quad F_0 = 0, F_1 = 1$$

There are a few approaches and all of them work. Let's go through the codes and see the cons and pros:

3.3.1 Recursion

```
def fibonacci(n):
    if (n == 0):
        return 0
    if (n == 1):
        return 1

    return fibonacci(n - 1) + fibonacci(n - 2)
```

3.3.2 Dynamic Programming

- **Top Down - Memoization**

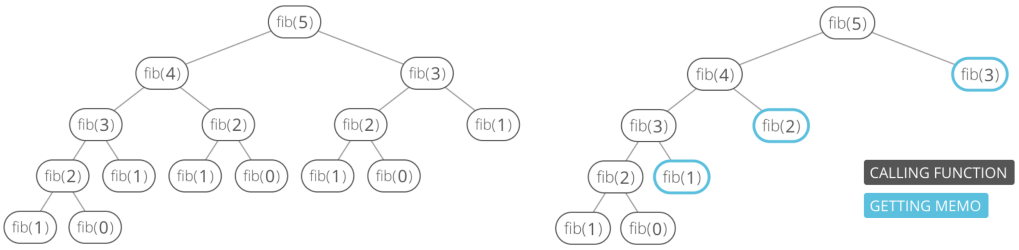
  The recursion does a lot of unnecessary calculation because a given fibonacci number will be calculated multiple times. An easy way to improve this is to cache the results (memoization):

```
cache = {}

def fibonacci(n):
    if (n == 0):
        return 0
    if (n == 1):
        return 1
    if (n in cache):
        return cache[n]

    cache[n] = fibonacci(n - 1) + fibonacci(n - 2)

    return cache[n]
```



Recursive Memoization

- **Bottom-Up**

  A better way to do this is to get rid of the recursion all-together by evaluating the results in the right order:

```
cache = {}

def fibonacci(n):
    cache[0] = 0
```

```
5           cache[1] = 1
6
7           for (i in range(2, n + 1)):
8              cache[i] = cache[i - 1] + cache[i - 2]
9
10          return cache[n]
```

We can even use constant space, and store only the necessary partial results along the way:

```
1       def fibonacci(n):
2         fib_minus_2 = 0
3         fib_minus_1 = 1
4
5         for (i in range(2, n + 1)):
6            fib = fib_minus_1 + fib_minus_2
7            fib_minus_1, fib_minus_2 = fib, fib_minus_1
8
9         return fib
```

3.4 How to Apply Dynamic Programming?

- Find the recursion in the problem.
- **Top-down:** store the answer for each subproblem in a table to avoid having to recompute them.
- **Bottom-up:** Find the right order to evaluate the results so that partial results are available when needed.

Dynamic programming generally works for problems that have an inherent left to right order such as strings, trees or integer sequences. If the naive recursive algorithm does not compute the same subproblem multiple time, dynamic programming won't help.

4 Common DP Problems

4.1 Coin Problem

As we discussed above Greedy approach doesn't work all the time for the coin problem.
For example: if the coins are 4,3,1 and the target sum is 6, the greedy algorithm produces the solution 4+1+1 while the optimal solution is 3+3.
This is where Dynamic Programming helps us:

4.1.1 Solution

Approach:

```
1- If V == 0, then 0 coins required.
2- If V > 0, minCoins(coins[0..m-1], V) = min {1 + minCoins(V-coin[i])}
           where i varies from 0 to m-1
           and coins[i] <= V
```

```
1   def minCoins(coins, target):
2       # base case
3       if (V == 0):
4           return 0
5
6       n = len(coins)
7       # Initialize result
8       res = sys.maxsize
9
10      # Try every coin that has smaller value than V
11      for i in range(0, n):
12          if (coins[i] <= target):
13              sub_res = minCoins(coins, target-coins[i])
14
15              # Check for INT_MAX to avoid overflow and see if
16              # result can minimized
17              if (sub_res != sys.maxsize and sub_res + 1 < res):
18                  res = sub_res + 1
19
20      return res
```

4.2 Knapsack Problem

We are given the weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent the values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item; or don't pick it (0-1 property).
Approach:

```
To consider all subsets of items, there can be two cases for every item: (1) the item
    is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of the
   following two values.
1- Maximum value obtained by n-1 items and W weight (excluding nth item).
2- Value of the nth item plus the maximum value obtained by n-1 items and W minus the
    weight of the nth item (including the nth item).

If the weight of nth item is greater than W, then the nth item cannot be included and
    case 1 is the only possibility.
```

For example:

```
   Let,
   Knapsack Max weight: W = 8 (units)
   Weight of items: wt = {3, 1, 4, 5}
   Values of items: val = {10, 40, 30, 50}
   Total items: n = 4

   Then,
   The following sums are possible:
   1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13

   The sum 8 is possible with 2 combinations
   {3, 5} and {1, 3, 4}

   for {3, 5} total value is 60
   and for {1, 3, 4} total value is 80

   But,
   There is a better solution with less weight:
   {1, 5} has total weight of 6 which is less than 8 and has total value 90.

   Hence,
   Our answer for maximum total value in the knapsack with given items and knapsack
      will be 80.
```

4.2.1 Recursion

```
def knapSack(W , wt , val , n):

    # Base Case
    if (n == 0 or W == 0):
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W , wt , val , n - 1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W - wt[n - 1] , wt , val , n - 1), knapSack(W
            , wt , val , n - 1))
```

4.2.2 DP

It should be noted that the above function computes the same subproblems again and again. Time complexity of this naive recursive solution is exponential (2^n).
Since suproblems are evaluated again, this problem has Overlapping Subproblems property. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]

    # Build table K[][] in bottom up manner
    for (i in range(n + 1)):
        for (w in range(W + 1)):
            if (i == 0 or w == 0):
                K[i][w] = 0
            elif (wt[i - 1] <= w):
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w])
            else:
                K[i][w] = K[i - 1][w]

    return K[n][W]
```

4.3 Longest Common Substring (LCS) Problem

We are given two strings 'X' and 'Y', find the length of the longest common substring.
Sample Case:

```
Input: X="inzvahackerspace", Y="spoilerspoiler"
Output: 4
The longest common substring is "ersp" and is of length 4.
```

Let m and n be the lengths of the first and second strings respectively.

A simple solution is to one by one consider all substrings of the first string and for every substring check if it is a substring in the second string. Keep track of the maximum length substring. There will be $O(m^2)$ substrings and we can find whether a string is subsring on another string in O(n) time (See this). So overall time complexity of this method would be $O(n * m^2)$.

Dynamic Programming can be used to find the longest common substring in O(m*n) time. The idea is to find the length of the longest common suffix for all substrings of both strings and store these lengths in a table.

```
The longest common suffix has following optimal substructure property
   LCSuff(X, Y, m, n) = LCSuff(X, Y, m-1, n-1) + 1
   if X[m-1] = Y[n-1] 0 Otherwise (if X[m-1] != Y[n-1])

The maximum length Longest Common Suffix is the longest common substring.
   LCSubStr(X, Y, m, n) = Max(LCSuff(X, Y, i, j)) where 1 <= i <= m and 1 <= j <= n
```

4.3.1 DP - Iterative

```
def LCSubStr(X, Y):
    m = len(X)
    n = len(Y)

    # Create a table to store lengths of
    # longest common suffixes of substrings.
    # Note that LCSuff[i][j] contains the
    # length of longest common suffix of
    # X[0...i-1] and Y[0...j-1]. The first
    # row and first column entries have no
    # logical meaning, they are used only
    # for simplicity of the program.

    # LCSuff is the table with zero
    # value initially in each cell
    LCSuff = [[0 for k in range(n+1)] for l in range(m + 1)]

    # To store the length of
    # longest common substring
    result = 0

    # Following steps to build
    # LCSuff[m+1][n+1] in bottom up fashion
    for (i in range(m + 1)):
        for (j in range(n + 1)):
            if (i == 0 or j == 0):
                LCSuff[i][j] = 0
            elif (X[i - 1] == Y[j - 1]):
                LCSuff[i][j] = LCSuff[i - 1][j - 1] + 1
                result = max(result, LCSuff[i][j])
            else:
                LCSuff[i][j] = 0
    return result
```

4.3.2 DP - Recursive

```
def lcs(int i, int j, int count):
    if (i == 0 or j == 0):
        return count

    if (X[i - 1] == Y[j - 1]):
        count = lcs(i - 1, j - 1, count + 1)

    count = max(count, max(lcs(i, j - 1, 0), lcs(i - 1, j, 0)))
    return count
```

4.4 Longest Increasing Subsequence (LIS) Problem

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in theincreasing order.
For example, given the array [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15], the longest increasing subsequence has the length of 6: it is {0, 2, 6, 9, 11, 15}.

4.4.1 Solution

The naive, brute force way to solve this is to generate each possible subsequence, testing each one for monotonicity and keeping track of the longest one. That would be prohibitively expensive: generating each subsequence already takes $O(2^N)$!
Instead, let's try to tackle this problem using recursion and then optimize it with dynamic programming.
Assume that we already have a function that gives us the length of the longest increasing subsequence. Then we'll try to feed some part of our input array back to it and try to extend the result.
Our base cases are: the empty list, returning 0, and an array with one element, returning 1.
Then,

- For every index i up until the second to last element, calculate longest_increasing_subsequence up to there.
- We can only extend the result with the last element if our last element is greater than arr[i] (since otherwise it's not increasing).
- Keep track of the largest result.

```
1   def longest_increasing_subsequence(arr):
2       if (not arr):
3           return 0
4       if (len(arr) == 1):
5           return 1
6
7       max_ending_here = 0
8       for (i in range(len(arr))):
9           ending_at_i = longest_increasing_subsequence(arr[:i])
10          if (arr[-1] > arr[i - 1] and ending_at_i + 1 > max_ending_here):
11              max_ending_here = ending_at_i + 1
12      return max_ending_here
```

This is really slow due to repeated subcomputations (exponential in time). So, let's use dynamic programming to store values to recompute them for later.

We'll keep an array A of length N, and A[i] will contain the length of the longest increasing subsequence ending at i. We can then use the same recurrence but look it up in the array instead:

```
1    def longest_increasing_subsequence(arr):
2        if (not arr):
3            return 0
4        cache = [1] * len(arr)
5        for (i in range(1, len(arr))):
6            for (j in range(i)):
7                if (arr[i] > arr[j]):
8                    cache[i] = max(cache[i], cache[j] + 1)
9        return max(cache)
```

This now runs in $O(N^2)$ time and $O(N)$ space.

5 Conclusion

As we discussed,
Dynamic programming is a technique that combines the correctness of complete search and the efficiency of greedy algorithms. Dynamic programming can be applied if the problem can be divided into overlapping subproblems that can be solved independently.
There are two uses for dynamic programming:

- **Finding an optimal solution:** We want to find a solution that is as large as possible or as small as possible.
- **Counting the number of solutions:** We want to calculate the total number of possible solutions.

References

[1] "Competitive Programmer's Handbook" by Antti Laaksonen - Draft July 3, 2018 https://cses.fi/book/book.pdf

[2] Wikipedia - Dynamic Programming https://en.wikipedia.org/wiki/Dynamic_programming

[3] Topcoder - Competitive Programming Community / Dynamic Programming from Novice to Advanced https://www.topcoder.com/community/competitive-programming/tutorials/dynamic-programming-from-novice-to-advanced/

[4] Hacker Earth - Dynamic Programming https://www.hackerearth.com/practice/algorithms/dynamic-programming/

[5] Geeks for Geeks - Dynamic Programming https://www.geeksforgeeks.org/dynamic-programming/