Dynamic Programming and Greedy Algorithms

A popular topic during interviews is dynamic programming, which simplifies a complicated problem by breaking it down into simpler subproblems by means of recursion. If an interview problem has optimal substructure and overlapping subproblems, it might be solved by dynamic programming.

The ingredient optimal substructure means that the solution to a given optimization problem can be obtained by a combination of optimal solutions to its subproblems. Therefore, the first step to utilize dynamic programming is to check whether the problem exhibits such optimal substructures.

For example, the problem to find the minimal number of coins to make change for a value t exhibits optimal substructures. If a set of coins that has the minimal number of coins to make change for the value t contains a coin v_i, the set of coins excluding v_i should have the minimal number of coins to make change for value t-v_i.

The other ingredient, overlapping subproblems, means a recursive algorithm solves subproblems over and over, rather than always generating new subproblems. Dynamic programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be retrieved when necessary.

Let's take the problem of making change as an example again, assuming we are going to make change for value 15 with a set of coins with value 1, 3, 9, 10. $f(t)$ is defined as the optimal solution to get the minimum number of coins to make change for the value t. There are four subproblems for the overall problem $f(15)$: $f(14)$, $f(12)$, $f(6)$, and $f(5)$. It is noticeable that $f(11)$ is a subproblem of both $f(14)$ and $f(12)$, and $f(5)$ is a subproblem of both $f(14)$ and $f(6)$, so there are many overlapping subproblems. Solutions of subproblems are stored in a 2-D table to avoid duplicated calculations. More details about the problem to make change are discussed in the section *Minimal Number of Coins for Change*.

A choice is made at each step in dynamic programming, and the choice usually depends on the solutions to subproblems. Consequently, dynamic programming problems are typically solved in a bottom-up manner, progressing from smaller subproblems to larger subproblems.

Greedy algorithms are different from dynamic programming solutions, in which a greedy choice is made that seems best at the moment and then you solve the subproblem arising after the choice is made. Therefore, a greedy strategy usually progresses in a top-down order, making one greedy choice after another and reducing each given problem instance to a smaller one.

For example, in order to minimize the number of key presses on a cell-phone keyboard, we choose the most frequent character at the first step and locate it as the first character on a key, and then choose the next most frequent character and locate it as the first character on another key. If the first locations of all keys are already occupied, a character is located as the second one on a key, and so on. Actually, it is a greedy choice to select the most frequent character of the remaining characters and locate it in a key with the lowest index. More details about this problem are discussed in the section *Minimal Times of Presses on Keyboards*.

Edit Distance

Question 32 Implement a function that finds the edit distance of two input strings. There are three types of edit operations: insertion, deletion, and substitution. Edit distance is the minimal number of edit operations to modify a string from one state to the other.

For example, the edit distance between "Saturday" and "Sunday" is 3 since the following three edit operations are required to modify one into another:

(1) Saturday → Sturday (deletion of 'a')

(2) Sturday → Surday (deletion of 't')

(3) Surday → Sunday (substitution of 'r' for 'n')

There is no way to achieve it with fewer than three operations.

If a function $f(i,j)$ is defined to indicate the edit distance between the substring of the first string ending with the j^{th} character and the substring of the second string ending with the i^{th} character, it is obvious that $f(i,0)=i$ because when we delete i characters from the substring of the first string ending with the i^{th} character, we get an empty string. (It is also the substring of the second string ending with the 0^{th} character.) Similarly, $f(0,j)=j$.

Let's analyze the cases when both i and j are greater than 0. If the i^{th} character of the second string is the same as the j^{th} character of the first string, no edit operations are necessary. Therefore, $f(i,j) = f(i\text{-}1,j\text{-}1)$ in such a case.

When the j^{th} character of the first string is different from the i^{th} character of the second string, there are three options available:

- Insert the i^{th} character of the second string into the first string. In this case, $f(i,j)=f(i\text{-}1,j)+1$.
- Delete the j^{th} character of the first string. In this case, $f(i,j)=f(i,j\text{-}1)+1$.
- Replace the j^{th} character of the first string with the i^{th} character of the second string. In this case, $f(i,j)=f(i\text{-}1,j\text{-}1)+1$.

What is the final value for $f(i,j)$? It should be the minimum value of the three cases.

A formal equation can be defined for this problem:

$$f(i,j) = \begin{cases} i \\ j \\ f(i-1, j-1) \\ min(f(i-1, j), f(i, j-1), f(i-1 \end{cases}$$

If we draw a table to show the edit distance values $f(i,j)$ between "Saturday" and "Sunday," it looks like Table 4-1. The edit distance of two strings is at the right-bottom corner of the table for edit distance values.

Table 4-1. *Edit Distance Value* f(i,j) *between "Saturday" and "Sunday"*

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

A table of edit distance values can be implemented as a 2-D array to simulate Table 4-1, which is the variable `distances` in the code found in Listing 4-17.

Listing 4-17. *C# Code for Moving Robots*

```
int GetEditDistance(String str1, String str2) {

int len1 = str1.Length;;

int len2 = str2.Length;

   int[,] distances = new int[len2 + 1, len1 + 1];

int editDistance = GetEditDistance(str1, str2, distances, len1, len2);

return editDistance;
```

```
}

int GetEditDistance(String str1, String str2, int[,] distances, int len1, int len2) {
    for (int i = 0; i < len2 + 1; ++i)
        distances[i, 0] = i;
    for (int j = 0; j < len1 + 1; ++j)
        distances[0, j] = j;

    for (int i = 1; i < len2 + 1; ++i) {
        for (int j = 1; j < len1 + 1; ++j) {
            if (str1[j - 1] == str2[i - 1])
                distances[i, j] = distances[i - 1, j - 1];
            else {
                int deletion = distances[i, j - 1] + 1;
                int insertion = distances[i - 1, j] + 1;
                int substitution = distances[i - 1, j - 1] + 1;
                distances[i, j] = Min(deletion, insertion, substitution);
            }
        }
    }

    return distances[len2, len1];
}

int Min(int num1, int num2, int num3) {
    int less = (num1 < num2) ? num1 : num2;
    return (less < num3) ? less : num3;
}
```

Source Code:

032_EditDistance.cs

Test Cases:

- Functional cases: Finding edit distances of normal strings, including two identical strings, or a string is a substring of the other
- Boundary cases: One or two strings are empty

Minimal Number of Coins for Change

Question 33 Please implement a function that gets the minimal number of coins with values $v_1, v_2, \ldots, v_n$, to make change for an amount of money with value t. There are an infinite number of coins for each value v_i.

For example, the minimum number of coins to make change for 15 out of a set of coins with values 1, 3, 9, 10 is 3. We can choose two coins with value 3 and a coin with value 9. The number of coins for other choices should be greater than 3.

First, let's define a function $f(t)$, which is the minimum number of coins to make change for the total value t. If there are n different coins, we have n choices to make change for the value t. We can add a coin with value v_1 to a set of coins whose total value is t-v_1. The minimum number of coins to get a value t-v_1 is $f(t$-$v_1)$. Similarly, we can add a coin with value v_2 into a set of coins whose total value is t-v_2. The minimal number of coins to get value t-v_2 is $f(t$-$v_2)$. It is similar for other coins.

Therefore, a problem to calculate $f(t)$ is divided into n sub-problems: $f(t$-$v_1)$, $f(t$-$v_2)$, …, $f(t$-$v_n)$. We can get a formal equation for $f(t)$ as the following accordingly:

$$f(t) = min(f(t - v_i)) + 1, \text{where} 0 < i \leq n$$

This equation can be implemented with recursion easily. However, the recursive solution may cause serious performance issues since there are overlaps when we divide this problem into n subproblems. A better solution is to utilize iteration and store the result of subproblems into a table (as in Table 4-2).

Table 4-2. *The Iterative Process to Calculate the Minimal Number of Coins to Make Changes for 15*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	*1*	*2*	3	*2*	*3*	4	*3*	*4*	5	*2*	*2*	3	3	4	5
3	-	-	*1*	*2*	*3*	*2*	*3*	*4*	3	4	5	*2*	*2*	3	3
9	-	-	-	-	-	-	-	-	*1*	2	3	*2*	3	4	3
10	-	-	-	-	-	-	-	-	-	*1*	*2*	3	*2*	3	4

In Table 4-2, each column denotes the number of coins to make change for a specific value. We can calculate the numbers in Table 4-2 from left to right to simulate the iterative process to get the result of $f(15)$.

For instance, there are two numbers 4 and 2 under the column titled "6". We have two alternatives to make change for 6: the first one is to add a coin with value 1 to a set of coins whose total value is 5. Since the minimal number of coins to get value 5 is 3 (*highlighted* number under the column titled "5"), the number in the first cell under column titled "6" is 4 (4=3+1). The second choice is to add a coin with value 3 to a set of coins whose total value is 3. Since the minimal number of coins to get a value of 3 is 1 (the *highlighted* number under the column tile "3"), the number in the second cell under column title "6" is 2 (2=1+1). We highlight the number 2 in the column under title "6" because 2 is less than 4.

Even though we have a 2-D matrix to show the iterative process, it only requires a 1-D array for coding because it is only necessary to store the minimum number of coins to make change for each total value. The sample code is shown in Listing 4-18.

Listing 4-18. *C# Code for Coin Changes*

```
int GetMinCount(int total, int[] coins) {
    int[] counts = new int[total + 1];
    counts[0] = 0;

    const int MAX = Int32.MaxValue;

    for (int i = 1; i <= total; ++i) {
        int count = MAX;
        for (int j = 0; j < coins.Length; ++j) {
            if (i - coins[j] >= 0 && count > counts[i - coins[j]])
                count = counts[i - coins[j]];
        }

        if (count < MAX)
            counts[i] = count + 1;
        else
            counts[i] = MAX;
    }

    return counts[total];
}
```

Source Code:

033_CoinChanges.cs

Test Cases:

- There are multiple choices to make change for a total value
- There is only one choice to make change for a total value
- It is not possible to make change for a total value

Minimal Times of Presses on Keyboards

Question 34 Please design an efficient algorithm to lay out cells on phone keyboards in order to minimize key presses.

The current phone keyboard looks like the following layout:

```
            key 2: abc   key 3: def
key 4: ghi   key 5: jkl   key 6: mno
key 7: pqrs  key 8: tuv   key 9: wxyz
```

The first press of a key types the first letter. Each subsequent press advances to the next letter. For example, to type the word "snow", we have to press the key 7 four times, followed by the 6 key twice, followed by the 6 key three times, followed by the 9 key once. The total number of key presses is 10.

This typical solution is not an efficient one. The letter 'i' is used much more frequently than the letter 'w' in English. However, we have to press the 4 key three times to get an 'i', and press the 9 key only once to get a 'w'. If these two letters are interchanged on the keyboard, it will reduce the overall press times.

Let's place letters on a keyboard one by one. It is a greedy choice at each step to select the most frequent letter from the remaining letters and locate it on a key with the lowest index. That is to say, we choose the most frequent letter at the first step and place it as the first letter on a key, and then choose the next most frequent letter and place it as the first letter on another key. If the first locations of all keys are already occupied, a letter is placed as the second one on a key, and so on.

When given the number of keys on a keyboard, as well the frequency of every letter in an alphabet (which is not necessarily English), we may implement the greedy algorithm with the C# code shown in Listing 4-19.

Listing 4-19. *C# Code to Get the Minimal Number Of Key Touches*

```
int MinKeyPress(int keys, int[] frequencies) {
    Array.Sort(frequencies);

    int letters = frequencies.Length;
    int presses = 0;

    // The last element has the highest frequency in
    // an increasingly sorted array
    for(int i = letters - 1; i >= 0; --i) {
        int j = letters - 1 - i;
        presses += frequencies[i] * (j / keys + 1);
    }

    return presses;
}
```

Letters are sorted based on their frequencies in this code. The most frequent letter is the last one in the sorted array, so we begin to select and place letters from the end of the array.

Source Code:

034_MinimalPresses.cs

Test Cases:

- Functional Tests: Given different sets of characters and various numbers of keys