

Dynamic Programming Wonderland

Stanislav Ostapenko

February 26, 2024

Abstract

Your abstract goes here.

Contents

1	Longest Increasing Subsequence	4
1.1	Recursive Algorithm	4
1.2	Backtracking Algorithm	5
1.3	Bottom-up Dynamic Programming solution	6
1.4	DP with Binary Search	7
2	Fibonacci numbers	7
2.1	Recursive solution	7
2.2	Memoization : improved recursion	7
2.3	Dynamic Programming (using array)	8
2.4	Dynamic Programming (improved memory)	8
2.5	Dynamic Programming (efficient)	9
2.6	Binet's formula	9
2.7	Maple solution	9
3	Climbing stairs	10
4	Coins change	10
5	Knapsack	10
6	Longest Common Subsequence	10
7	Shortest Graph Path	10
8	Sort integers by the power value - memoization	10
9	N-th derivative	10

Listings

1	$\mathcal{O}(2^n)$ Recursive Algorithm	4
2	$\mathcal{O}(2^n)$ Backtracking Algorithm	5
3	$\mathcal{O}(n^2)$ DP solution	6
4	$\mathcal{O}(n \log n)$ DP with Binary Search	7

$$\mathcal{O}(1) = \mathcal{O}(\text{yeah})$$

$$\mathcal{O}(\log n) = \mathcal{O}(\text{nice})$$

$$\mathcal{O}(n) = \mathcal{O}(\text{k})$$

$$\mathcal{O}(n^2) = \mathcal{O}(\text{my})$$

$$\mathcal{O}(2^n) = \mathcal{O}(\text{no})$$

$$\mathcal{O}(n!) = \mathcal{O}(\text{mg})$$

$$\mathcal{O}(n^n) = \mathcal{O}(\text{sh}^*\text{t!})$$

1 Longest Increasing Subsequence

1.1 Recursive Algorithm

```
1 public class Solution {
2     private int max;
3     public int lengthOfLISSHelper(int[] arr, int n) {
4         // Base case: if there is only one element, the LIS length is 1
5         if (n == 1) {
6             return 1;
7         }
8         int currResult;
9         int maxEnding = 1;
10
11         for (int i = n - 1; i > 0; i--) {
12             // Recursively calculate LIS for previous elements
13             currResult = lengthOfLISSHelper(arr, i);
14
15             // Check if the current element can be included in the increasing
16             // subsequence
17             if ((arr[i - 1] < arr[n - 1]) && (currResult + 1 > maxEnding)) {
18                 maxEnding = currResult + 1;
19             }
20             max = Math.max(maxEnding, max);
21             return maxEnding;
22         }
23
24         public int lengthOfLIS(int[] nums) {
25             lengthOfLISSHelper(nums, nums.length);
26             return max;
27         }
28     }
```

Listing 1: $\mathcal{O}(2^n)$ Recursive Algorithm

1.2 Backtracking Algorithm

```
1 public class Solution {
2     private List<List<Integer>> generateSubsequences(int[] arr) {
3         List<List<Integer>> allSubsequences = new ArrayList<>();
4         generateSubsequencesHelper(arr, 0, new ArrayList<>(), allSubsequences);
5         return allSubsequences;
6     }
7
8     private void generateSubsequencesHelper(int[] arr, int index, List<Integer>
9         current, List<List<Integer>> allSubsequences) {
10         if (index == arr.length) {
11             // Base case: add the current subsequence to the result
12             allSubsequences.add(new ArrayList<>(current));
13             return;
14         }
15         // Exclude the current element
16         generateSubsequencesHelper(arr, index + 1, current, allSubsequences);
17         // Include the current element
18         current.add(arr[index]);
19         generateSubsequencesHelper(arr, index + 1, current, allSubsequences);
20         // Backtrack to exclude the current element
21         current.removeLast();
22     }
23
24     private boolean isStrictlyIncreasing(List<Integer> list) {
25         for (int i = 1; i < list.size(); i++) {
26             if (list.get(i) <= list.get(i - 1)) {
27                 return false;
28             }
29         }
30         return true; // Strictly increasing
31     }
32
33     public int lengthOfLIS(int[] nums) {
34         List<List<Integer>> allSubsequences = generateSubsequences(nums);
35         int max = 1;
36         for (List<Integer> subsequence : allSubsequences) {
37             if (isStrictlyIncreasing(subsequence)) {
38                 max = Math.max(max, subsequence.size());
39             }
40         }
41         return max;
42     }
43 }
```

Listing 2: $\mathcal{O}(2^n)$ Backtracking Algorithm

1.3 Bottom-up Dynamic Programming solution

Let's define $L(i)$ as the length of the longest strictly increasing subsequence ending at index i . The recurrence formula for the longest strictly increasing subsequence is given by:

$$L(i) = 1 + \max_{\substack{j < i \\ \text{arr}[j] < \text{arr}[i]}} L(j)$$

This equation states that the length of the longest increasing subsequence ending at index i is 1 plus the maximum length obtained by considering all indices j less than i , where the corresponding element $\text{arr}[j]$ is less than $\text{arr}[i]$.

Complexity :

$$T(n) = \mathcal{O}(n^2)$$

$$M(n) = \mathcal{O}(n)$$

```
1 class Solution {
2   private int max(int[] L) {
3     int maxLength = Integer.MIN_VALUE;
4     for (final int length : L) {
5       maxLength = Math.max(maxLength, length);
6     }
7     return maxLength;
8   }
9
10  public int lengthOfLIS(int[] nums) {
11    int n = nums.length;
12    int[] L = new int[n];
13    // Initialize the array with minimum length 1 for each index
14    Arrays.fill(L, 1);
15
16    // Iterate to fill in the values of L(i) using the recurrence relation
17    for (int i = 1; i < n; i++) {
18      for (int j = 0; j < i; j++) {
19        if (nums[i] > nums[j]) {
20          L[i] = Math.max(L[i], L[j] + 1);
21        }
22      }
23    }
24    // Find the maximum value in the array L
25    return max(L);
26  }
27 }
```

Listing 3: $\mathcal{O}(n^2)$ DP solution

1.4 DP with Binary Search

```
1  import java.util.Arrays;
2
3  public class Solution {
4      public int lengthOfLIS(int[] nums) {
5          if (nums == null || nums.length == 0) {
6              return 0;
7          }
8
9          int[] dp = new int[nums.length];
10         int len = 0;
11
12         for (int num : nums) {
13             int index = Arrays.binarySearch(dp, 0, len, num);
14             if (index < 0) {
15                 index = -(index + 1);
16             }
17             dp[index] = num;
18             if (index == len) {
19                 len++;
20             }
21         }
22
23         return len;
24     }
25 }
```

Listing 4: $\mathcal{O}(n \log n)$ DP with Binary Search

2 Fibonacci numbers

2.1 Recursive solution

$$T(n) = \mathcal{O}(2^n)$$

$$M(n) = \mathcal{O}(2^n)$$

The space complexity is determined by the maximum depth of the recursive call stack.

Since each function call adds a new frame to the call stack, and there are 2^n calls, the space complexity is also exponential.

```
1  public static int fib0(int n) {
2      if (n == 0) return 0;
3      if (n == 1) return 1;
4      return fib0(n - 1) + fib0(n - 2);
5  }
```

2.2 Memoization : improved recursion

$$T(n) = \mathcal{O}(n)$$

$$M(n) = \mathcal{O}(n)$$

```

1 private static final Map<Integer, Integer> memo = new HashMap<>();
2 public static int fib_memo(int n) {
3     // Base cases
4     if (n <= 1) {
5         return n;
6     }
7
8     // Check if the result for the given n is already in the memo map
9     if (memo.containsKey(n)) {
10        return memo.get(n);
11    }
12
13    // If not, calculate the Fibonacci number and store it in the memo map
14    int result = fib_memo(n - 1) + fib_memo(n - 2);
15
16    memo.put(n, result);
17
18    return result;
19 }

```

2.3 Dynamic Programming (using array)

$$T(n) = \mathcal{O}(n)$$

$$M(n) = \mathcal{O}(n)$$

```

1 public static int fib1(int n) {
2     int[] fib = new int[n + 1];
3     fib[1] = 1;
4     fib[2] = 1;
5     for (int i = 2; i <= n; i++) {
6         fib[i] = fib[i - 1] + fib[i - 2];
7     }
8     return fib[n];
9 }

```

2.4 Dynamic Programming (improved memory)

$$T(n) = \mathcal{O}(n)$$

$$M(n) = \mathcal{O}(1)$$

```

1 static int MAX_SAVE = 3;
2 static int[] fib = new int[MAX_SAVE];
3 public static int fib2(int n) {
4     fib[0] = 0;
5     fib[1] = 1;
6     for (int i = 2; i <= n; i++){
7         fib[i % MAX_SAVE] = fib[(i - 1) % MAX_SAVE] + fib[(i - 2) % MAX_SAVE];
8     }
9     return fib[n % MAX_SAVE];
10 }

```


2.5 Dynamic Programming (efficient)

$$T(n) = \mathcal{O}(n)$$

$$M(n) = \mathcal{O}(1)$$

```
1 public static int fib3(int n) {
2     int first = 1;
3     int second = 1;
4     int fib = 1;
5     for (int i = 2; i < n; i++) {
6         second = fib;
7         // same as with array but changed the way of saving intermediate results
8         fib = first + second;
9         first = second;
10    }
11    return fib;
12 }
```

2.6 Binet's formula

The Binet formula for the Fibonacci sequence is given by:

$$F_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}} \quad (1)$$

where F_n is the n -th Fibonacci number and ϕ is the golden ratio, defined as $\phi = \frac{1+\sqrt{5}}{2}$.

$$T(n) = \mathcal{O}(1)$$

$$M(n) = \mathcal{O}(1)$$

```
1 public static int fib4(int n) {
2     double sqrt5 = Math.sqrt(5);
3     double result = Math.pow((1 + sqrt5) / 2, n) - Math.pow((1 - sqrt5) / 2, n);
4     return (int) (result / sqrt5);
5 }
```

2.7 Maple solution

```
1 with(combinat, fibonacci);
2 fibonacci_numbers := seq(fibonacci(i), i = 0 .. 100000):
3
4 f := fopen("fibonacci_numbers.txt", WRITE):
5
6 for num in fibonacci_numbers do
7     fprintf(f, "%a\n", num);
8 end do:
9
10 fclose(f);
```

to run this code use this command :

```
1 cmaple -q fibonacci.mpl >out.log
```

- 3 Climbing stairs
- 4 Coins change
- 5 Knapsack
- 6 Longest Common Subsequence
- 7 Shortest Graph Path
- 8 Sort integers by the power value - memoization
- 9 N-th derivative

References

- [1] Author. (Year). Title. *Journal*, Volume(Issue), Page numbers.
- [2] Another author. (Year). Title. *Conference*, Location, Page numbers.