Dynamic Programming Wonderland

Stanislav Ostapenko

February 26, 2024

Abstract

Your abstract goes here.

Contents

Listings

$$
\begin{aligned}
\mathcal{O}(1) &= \mathcal{O}(\text{yeah}) \\
\mathcal{O}(\log n) &= \mathcal{O}(\text{nice}) \\
\mathcal{O}(n) &= \mathcal{O}(\text{k}) \\
\mathcal{O}(n^2) &= \mathcal{O}(\text{my}) \\
\mathcal{O}(2^n) &= \mathcal{O}(\text{no}) \\
\mathcal{O}(n!) &= \mathcal{O}(\text{mg}) \\
\mathcal{O}(n^n) &= \mathcal{O}(\text{sh*t!})
\end{aligned}
$$

1 Longest Increasing Subsequence

1.1 Recursive Algorithm

```java
public class Solution {
 private int max;
 public int lengthOfLISHelper(int[] arr, int n) {
  // Base case: if there is only one element, the LIS length is 1
  if (n == 1) {
   return 1;
  }
  int currResult;
  int maxEnding = 1;

  for (int i = n - 1; i > 0; i--) {
   // Recursively calculate LIS for previous elements
   currResult = lengthOfLISHelper(arr, i);

   // Check if the current element can be included in the increasing
       subsequence
   if ((arr[i - 1] < arr[n - 1]) && (currResult + 1 > maxEnding)) {
    maxEnding = currResult + 1;
   }
  }
  max = Math.max(maxEnding, max);
  return maxEnding;
 }

 public int lengthOfLIS(int[] nums) {
  lengthOfLISHelper(nums, nums.length);
  return max;
 }
}
```

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

1.2 Backtracking Algorithm

```java
public class Solution {
 private List<List<Integer>> generateSubsequences(int[] arr) {
  List<List<Integer>> allSubsequences = new ArrayList<>();
  generateSubsequencesHelper(arr, 0, new ArrayList<>(), allSubsequences);
  return allSubsequences;
 }

 private void generateSubsequencesHelper(int[] arr, int index, List<Integer>
     current, List<List<Integer>> allSubsequences) {
  if (index == arr.length) {
   // Base case: add the current subsequence to the result
   allSubsequences.add(new ArrayList<>(current));
   return;
  }
  // Exclude the current element
  generateSubsequencesHelper(arr, index + 1, current, allSubsequences);
  // Include the current element
  current.add(arr[index]);
  generateSubsequencesHelper(arr, index + 1, current, allSubsequences);
  // Backtrack to exclude the current element
  current.removeLast();
 }

 private boolean isStrictlyIncreasing(List<Integer> list) {
  for (int i = 1; i < list.size(); i++) {
   if (list.get(i) <= list.get(i - 1)) {
    return false;
   }
  }
  return true; // Strictly increasing
 }

 public int lengthOfLIS(int[] nums) {
  List<List<Integer>> allSubsequences = generateSubsequences(nums);
  int max = 1;
  for (List<Integer> subsequence : allSubsequences) {
   if (isStrictlyIncreasing(subsequence)) {
    max = Math.max(max, subsequence.size());
   }
  }
  return max;
 }
}
```

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:

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

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 arr[j] is less than arr[i].

Complexity :

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

```java
class Solution {
 private int max(int[] L) {
  int maxLength = Integer.MIN_VALUE;
  for (final int length : L) {
   maxLength = Math.max(maxLength, length);
  }
  return maxLength;
 }

 public int lengthOfLIS(int[] nums) {
  int n = nums.length;
  int[] L = new int[n];
  // Initialize the array with minimum length 1 for each index
  Arrays.fill(L, 1);

  // Iterate to fill in the values of L(i) using the recurrence relation
  for (int i = 1; i < n; i++) {
   for (int j = 0; j < i; j++) {
    if (nums[i] > nums[j]) {
     L[i] = Math.max(L[i], L[j] + 1);
    }
   }
  }
  // Find the maximum value in the array L
  return max(L);
 }
}
```

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

1.4 DP with Binary Search

```
import java.util.Arrays;

public class Solution {
 public int lengthOfLIS(int[] nums) {
  if (nums == null || nums.length == 0) {
   return 0;
  }

  int[] dp = new int[nums.length];
  int len = 0;

  for (int num : nums) {
   int index = Arrays.binarySearch(dp, 0, len, num);
   if (index < 0) {
    index = -(index + 1);
   }
   dp[index] = num;
   if (index == len) {
    len++;
   }
  }

  return len;
 }
}
```

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.

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

2.2 Memoization : improved recursion

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

```java
private static final Map<Integer, Integer> memo = new HashMap<>();
public static int fib_memo(int n) {
 // Base cases
 if (n <= 1) {
  return n;
 }

 // Check if the result for the given n is already in the memo map
 if (memo.containsKey(n)) {
  return memo.get(n);
 }

 // If not, calculate the Fibonacci number and store it in the memo map
 int result = fib_memo(n - 1) + fib_memo(n - 2);

 memo.put(n, result);

 return result;
}
```

2.3 Dynamic Programming (using array)

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

```java
public static int fib1(int n) {
 int[] fib = new int[n + 1];
 fib[1] = 1;
 fib[2] = 1;
 for (int i = 2; i <= n; i++) {
  fib[i] = fib[i - 1] + fib[i - 2];
 }
 return fib[n];
}
```

2.4 Dynamic Programming (improved memory)

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

```java
static int MAX_SAVE = 3;
static int[] fib = new int[MAX_SAVE];
public static int fib2(int n) {
 fib[0] = 0;
 fib[1] = 1;
 for (int i = 2; i <= n; i++){
  fib[i % MAX_SAVE] = fib[(i - 1) % MAX_SAVE] + fib[(i - 2) % MAX_SAVE];
 }
 return fib[n % MAX_SAVE];
}
```

2.5 Dynamic Programming (efficient)

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

```java
public static int fib3(int n) {
 int first = 1;
 int second = 1;
 int fib = 1;
 for (int i = 2; i < n; i++) {
  second = fib;
  // same as with array but changed the way of saving intermediate results
  fib = first + second;
  first = second;
 }
 return fib;
}
```

2.6 Binet's formula

The Binet formula for the Fibonacci sequence is given by:

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

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

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

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

2.7 Using matrix calculus

```java
// Method to raise a matrix to a power
static long[][] matrixPower(long[][] matrix, int n) {
 int row = matrix.length;
 int col = matrix[0].length;
 long[][] result = new long[row][col];

 // Initialize result matrix as the identity matrix
 for (int i = 0; i < row; i++) {
  result[i][i] = 1;
 }

 // Multiply matrix by itself n times
 while (n > 0) {
  if (n % 2 == 1) {
   result = matrixMultiply(result, matrix);
  }
  matrix = matrixMultiply(matrix, matrix);
  n /= 2;
 }

return result;
}

// Method to multiply two matrices
static long[][] matrixMultiply(long[][] a, long[][] b) {
 int rowA = a.length;
 int colA = a[0].length;
 int colB = b[0].length;
 long[][] result = new long[rowA][colB];

 for (int i = 0; i < rowA; i++) {
  for (int j = 0; j < colB; j++) {
   for (int k = 0; k < colA; k++) {
    result[i][j] += a[i][k] * b[k][j];
   }
  }
 }

 return result;
}

// Method to calculate the nth Fibonacci number using matrix exponentiation
static long fibonacci(int n) {
 if (n <= 0)
  return 0;

 long[][] matrix = {{1, 1}, {1, 0}};
 long[][] result = matrixPower(matrix, n - 1);
 return result[0][0];
}
```

2.8 Maple solution

```
with(combinat, fibonacci);
fibonacci_numbers := seq(fibonacci(i), i = 0 .. 100000):

```

```
f := fopen("fibonacci_numbers.txt", WRITE):

for num in fibonacci_numbers do
 fprintf(f, "%a\n", num);
end do:

fclose(f);
```

to run this code use this command :

```
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.