Introduction

Dynamic programming (DP) is a method for solving complex problems by breaking down the original problem into relatively simpler sub-problems. It is not a specific algorithm but a method to solve specific problems, it appears in a variety of data structures, and the types of questions related to it are more complicated. Common types of DP includes: **Linear DP**, **Knapsack DP**, **Interval DP**, **DAG DP** and **Tree DP**.

Without specified mentioned, assume `arr` is declared as a signed integer array. The following code is added to the beginning of all C++ programs.

```cpp
#include <bits/stdc++.h>
using namespace std;
```

In the whole topic of DP, we will use 1-based indexing for convenience.

1 Introduction to Dynamic Programming

Definition

Dynamic programming (DP) is a method for solving complex problems by breaking down the original problem into relatively **simpler sub-problems**. It is **not a specific algorithm** but a method to solve specific problems, it appears in a variety of data structures, and the types of questions related to it are more complicated.

The key of DP is to "**memoize**" something that have been calculated before. So it is also called "**Trading Space for Time**".

Terminologies

Base case(s):	The simplest secnario(s) where the answer is immendiately known.
State:	Store all essential information to define a subproblem.
Transitional Formula:	Defines the relationship between subproblems.

Identification of DP Problems

- Have **optimal substructure**
    - The optimal solution of a problem can be constructed efficiently from the optimal solutions of its subproblems.
- Have **overlapping subproblems**
    - The problem can be broken down into smaller pieces that are reused multiple times.
- No **aftereffect**
    - The optimal solution of all subproblems should be determined and fixed. Current decisions won't affect previous outcomes.

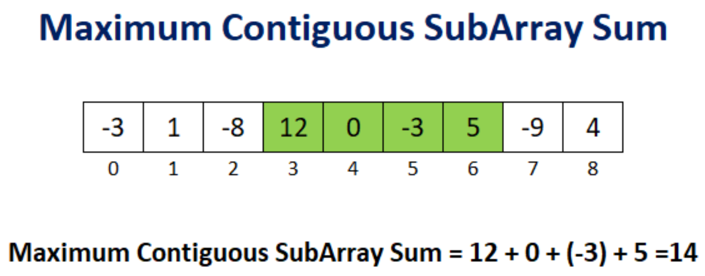
Steps of DP

1. Define the DP state
2. Find the transition formula between states
3. Determine the calculation order
4. Optimize the DP if necessary

2 Maximum Subarray Sum

Definition

Find the maximum subarray sum of an array.


Maximum Contiguous SubArray Sum
-3 1 -8 12 0 -3 5 -9 4
0 1 2 3 4 5 6 7 8
Maximum Contiguous SubArray Sum = 12 + 0 + (-3) + 5 =14

Observation

Define S(a : b) = sum of elements in arr[a : b]

Assume MSS of arr[i : j] is S(k : j)

MSS of arr[i : j+1] =

- S(k : j+1) if S(k : j), arr[j] > 0
- S(k : j) if arr[j] < 0 and arr[j] < S(k : j)
- arr[j] if S(k : j) < 0 and arr[j] > S(k : j)

We can easily transit from j to j+1 if we define state properly.

Derivation

1. Define State and target
    - Define dp[i] as the MSS of the subarrays which end with a[i].
    - Our target is max(dp[1], dp[2], …, dp[n]), where n is the length of the given array.
2. Transitional equation
    - ***dp[i] = max(dp[i - 1] + arr[i], arr[i])***
3. Initialize DP array
    - *dp[0]* = 0
4. Confirm the traversal order
    - i from 1 to n
5. Time complexity
    - O(n)
    - Good enough. No need further optimization.
    - Space complexity can be optimized to O(1).

C++ Code Implementation

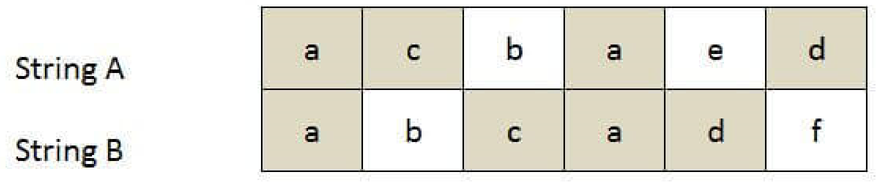
```
long long dp[200010];
long long N, num;
long long ans = -1e9;

int main(){
    dp[0] = 0;
    cin >> N;
    for (int i = 1; i <= N; i++)
    {
        cin >> num;
        dp[i] = max(dp[i-1] + num, num);
        ans = max(ans, dp[i]);
    }
    cout << ans;
    return 0;
}
```

3 Longest Common Subsequence

Definition

Find the longest common subsequence of two given sequences.



String A	a	c	b	a	e	d
String B	a	b	c	a	d	f

Observation

Assume we know the length of LCS of A[1 : i] and B[1 : j] already. Define it to be LCS[i][j].

Considering A[i + 1] and B[j + 1]:

If A[i + 1] ≠ B[j + 1], it is meaningless to include both of them in LCS.

In this case, LCS[i + 1][j + 1] =

- LCS[i + 1][j], if we only include A[i + 1] in LCS but not B[j].
- LCS[i][j + 1], if we only include B[j + 1] in LCS but not A[i].

If A[i + 1] = B[j + 1], we can count them as a "pair" and append them into the LCS.

In this case, LCS[i + 1][j + 1] =

- LCS[i + 1][j], if we only include A[i + 1] in LCS but not B[j].
- LCS[i][j + 1], if we only include B[j + 1] in LCS but not A[i].
- LCS[i][j] + 1, if we include both A[i + 1] and B[j + 1] in LCS.

Derivation

1. Define State and target
   - Define dp[i][j] as the length of LCS of A[1 : i] and B[1 : j].
   - Our target is dp[|S|][|T|], where |S|, |T| are lengths of sequence A and B.
2. Transitional equation
   - ***dp[i][j] = max(dp[i][j-1], dp[i-1][j], [dp[i-1][j-1]+1] × [A[i] == B[j]])***
3. Initialize DP array
   - *dp[i][0]* = *dp[0][j]* = 0 for any *i,j*
4. Confirm the traversal order
   - i from 1 to |S|, j from 1 to |T|. Doesn't matter which is in inner loop.
5. Time complexity
   - O(|S| * |T|)

C++ Code Implementation

```
string S, T;
int dp[1010][1010];

int main(){
    cin >> S >> T;
    int s = S.size();
    int t = T.size();
    for (int i = 1; i <= s; i++)
    {
        for (int j = 1; j <= t; j++)
        {
            dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
            if (S[i-1] == T[j-1]) dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
        }
    }
    cout << dp[s][t];
    return 0;
}
```

4 Longest Increasing Subsequence

Definition

Find the longest increasing subsequence of a given sequence.

Input Sequence 6 9 8 2 3 5 1 4 7

LIS 1 2 3 4 7

LIS 2 2 3 5 7

Observation

Define LIS[i] = length of LIS of arr[1 : i] that ends with arr[i].

Considering arr[i + 1],

- If arr[i + 1] > arr[i], trivially LIS[i + 1] = LIS[i] + 1.
- If arr[i + 1] ≤ arr[i], we need to consider all LIS[j] where 1 ≤ j < i + 1.

We can easily transit to i + 1 by considering all j less than it.

Derivation

1. Define State and target
    - Define dp[i] as the length of LIS of arr[1 : i] which ends with arr[i].
    - Our target is dp[n], where n is the length of the given sequence.
2. Transitional equation

$$dp[i] = \max_{0 \le j < i}\left(dp[j] \times [a[j] < a[i]]\right) + 1$$

3. Initialize DP array
    - *dp[0] = 0*
4. Confirm the traversal order
    - i from 1 to n
5. Time complexity
    - O(n^2)
    - Too slow. Need Optimize.

C++ Code Implementation

```
int n;
int num[5010];
long long dp[5010];
long long ans;
int main(){
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        cin >> num[i];
        dp[i] = 1;
        for (int k = 0; k < i; k++)
        {
            if (num[k] < num[i]) dp[i] = max(dp[i], dp[k] + 1);
        }
        ans = max(ans, dp[i]);
    }
    cout << ans;
    return 0;
}
```

Optimization

Trivially, a time of O(n) on searching for *a*[j] < *a*[i] is wasted.

- Note that *a*[i] can only append to the subsequence that the last element of that is smaller than *a*[i]. If it is possible, the last element of the new subsequence become *a*[i].
- What the required last element actually is doesn't really matter. We just care about if it is smaller than *a*[*i*]. Therefore, we can just find the **smallest last element**.

arr/key	10	5	9	2	3	7	101	8
$f/value$	1	1	2	1	2	3	4	?

$arr/elem$	10, 5, 2	<9, 3	< 7	<101
$f/index$	1	2	3	4

- Listing out the required elements, we can observe that it must be strictly increasing → Monotonicity
- Binary search can be performed

C++ Code Implementation

```
int n;
int main(){
    cin >> n;
    vector<int> lis(n, INT_MAX);
    vector<int> num(n);
    for (int i = 0; i < n; i++) cin >> num[i];
    for (auto x : num) lis[lower_bound(lis.begin(), lis.end(), x) - lis.begin()] = x;
    cout << lower_bound(lis.begin(), lis.end(), INT_MAX) - lis.begin();
    return 0;
}
```

Time Complexity: O(nlogn)