Dynamic Programming - exercises

Contents

1 Maximum Subarray

Given an integer array `nums`, find the subarray with the largest sum, and return its sum.

1.1 Examples

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
```

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

```
Input: nums = [1]
Output: 1
```

Explanation: The subarray [1] has the largest sum 1.

```
Input: nums = [5,4,-1,7,8]
Output: 23
```

Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

1.2 Constraints:

- $1 <= nums.length <= 10^5$
- $-10^4 <= nums[i] <= 10^4$

1.3 Solutions

1.3.1 Recursive Approach Dynamic Programming - Top-down (Memoization)

We can devise a recursive approach to solve the problem. Here we can state the approach as:

- At each index i, we can either pick that element or not pick it.
- If we pick current element, then all future element must also be picked since our array needs to be contiguous.
- If we had picked any elements till now, we can either end further recursion at any time by returning sum formed till now or we can choose current element and recurse further. This denotes two choices of either choosing the subarray formed from 1st picked element till now or expanding the subarray by choosing current element respectively.

In the code below, we will use `mustPick` to denote whether we must compulsorily pick current element. When `mustPick` is true, we must either return `0` or pick current element and recurse further. If `pickCur` is false, we have both choices of not picking current element and moving on to next element, or picking the current one.

```
int maxSubArray_recursive(vector<int>& nums) {
    return solve(nums, 0, false);
}
int solve(vector<int>& A, int i, bool mustPick) {
// Our subarray must contain atleast 1 element.
  // If mustPick is false at end means no element
  //    is picked and this is not valid case
  if(i >= size(A)) return mustPick ? 0 : -1e5;

  if(mustPick)
    // either stop here or choose current element and recurse
    return max(0, A[i] + solve(A, i+1, true));
```

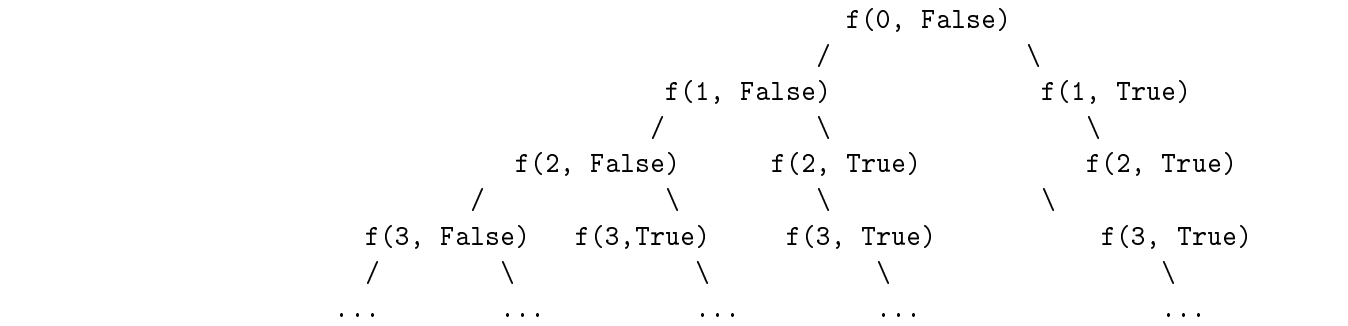
```
  // try both choosing current element or not choosing
  return max(solve(A, i+1, false), A[i] + solve(A, i+1, true));
}
```

Time Complexity $O(N^2)$, we are basically considering every subarray sum and choosing maximum of it.

Space Complexity $O(N)$, for recursive space

1.3.2 Dynamic Programming - Top-down (Memoization)

We can observe a lot of repeated calculations if we draw out the recursive tree for above solution.



These redundant calculations can be eliminated if we store the results for a given state and reuse them later whenever required rather than recalculating them over and over again.

Thus, we can use *memoization* technique here to make our solution more efficient. Here, we use a `dp` array where `dp[mustPick][i]` denotes the maximum sum subarray starting from `i` and `mustPick` denotes whether the current element must be picked compulsorily or not.

```
int solve(vector<int> &A, int i, bool mustPick, vector<vector<int>> &dp) {
    if (i >= size(A)) return mustPick ? 0 : -1e5;
    if (dp[mustPick][i] != -1) return dp[mustPick][i];
    if (mustPick)
        return dp[mustPick][i] = max(0, A[i] + solve(A, i + 1, true, dp));
    return dp[mustPick][i] = max(solve(A, i + 1, false, dp), A[i] + solve(A, i + 1, tr
}

int maxSubArray_dp_memo(vector<int> &nums) {
```

```
    vector<vector<int>> dp(2, vector<int>(size(nums), -1));
    return solve(nums, 0, false, dp);
}
```

Time Complexity $O(N)$, we are calculating each state of the dp just once and memoizing the result. Thus, we are calculating results for $2 * N$ states and returning them directly in future recursive calls.

Space Complexity $O(N)$, for recursive space.

1.3.3 Dynamic programming - Bottom-up (Tabulation)

We can employ similar logic in iterative version as well. Here, we again use `dp` array and use *bottom-up* approach. Here `dp[1][i]` denotes maximum subarray sum ending at `i` (including `nums[i]`) and `dp[0][i]` denotes maximum subarray sum up to `i` (may or may not include `nums[i]`).

At each index, we update `dp[1][i]` as `max` between either only choosing current element - `nums[i]` or extending from previous subarray and choosing current element as well - `dp[1][i-1] + nums[i]`.

Similarly, `dp[0][1]` can be updated as max between maximum sum subarray found till last index - `dp[0][i-1]` or max subarray sum found ending at current index `dp[1][i]`.

```
int maxSubArray_dp_tab(vector<int>& nums) {
    vector<vector<int>> dp(2, vector<int>(size(nums)));
    dp[0][0] = dp[1][0] = nums[0];
    for(int i = 1; i < size(nums); i++) {
        dp[1][i] = max(nums[i], nums[i] + dp[1][i-1]);
        dp[0][i] = max(dp[0][i-1], dp[1][i]);
    }
    return dp[0].back();
}
```

We can actually do away with just 1 row as well. We denoted `dp[1][i]` as the maximum subarray sum ending at `i`. We can just store that row and calculate the overall maximum subarray sum at the end by choosing the maximum of all max subarray sum ending at `i`.

```
int maxSubArray_dp_tab_v2(vector<int>& nums) {
    vector<int> dp(nums);
    for(int i = 1; i < size(nums); i++)
```

```
        dp[i] = max(nums[i], nums[i] + dp[i-1]);
    return *max_element(begin(dp), end(dp));
}
```

Time Complexity $O(N)$, we are just iterating over the nums array once to compute the `dp` array and once more over the `dp` at the end to find maximum subarray sum. Thus overall time complexity is $O(N) + O(N) = O(N)$

Space Complexity $O(N)$, for maintaining `dp`.

1.3.4 Kadane's algorithm

We can observe that in the previous approach, `dp[i]` only depended on `dp[i-1]`. So do we really need to maintain the whole `dp` array of `N` elements? One might see the last line of previous solution and say that we needed all elements of `dp` at the end to find the maximum sum subarray. But we can simply optimize that by storing the max at each iteration instead of separately calculating it at the end.

Thus, we only need to maintain `curMax` which is the maximum subarray sum ending at `i` and `maxTillNow` which is the maximum sum we have seen till now. And this way of solving this problem is what we popularly know as *Kadane's Algorithm.*

```
int maxSubArray_kadane(vector<int>& nums) {
    int curMax = 0, maxTillNow = INT_MIN;
    for(auto c : nums)
        curMax = max(c, curMax + c),
        maxTillNow = max(maxTillNow, curMax);
    return maxTillNow;
}
```

2 Maximize value of expression

Given an array A, maximize value of expression $(A[s] - A[r] + A[q] - A[p])$, where p, q, r, and s are indices of the array and $s > r > q > p$.

2.1 Examples

2.1.1 Example 1

```
A[] = [3, 9, 10, 1, 30, 40]
```

46

2.1.2 Explanation:

- The expression $(40\check{}1 + 10\check{}3)$ will result in the maximum value.

2.2 Approach

A *naive* solution would be to generate all combinations of such numbers. The time complexity of this solution would be $O(n^4)$, where `n` is the size of the input.

We can use dynamic programming to solve this problem. The idea is to create four lookup tables, `first[]`, `second[]`, `third[]`, and `fourth[]`, where:

first[] stores the maximum value of `A[s]`.

second[] stores the maximum value of `A[s] - A[r]`.

third[] stores the maximum value of `A[s] - A[r] + A[q]`.

fourth[] stores the maximum value of `A[s] - A[r] + A[q] - A[p]`.

The maximum value would then be present in index `0` of `fourth[]`

2.3 Example Solution

```
#include <climits>
#include <iostream>
#include <vector>
using namespace std;

// Function to print an array
void printVector(vector<int> const &A) {
  for (int i : A) {
    cout << i << " ";
  }
  cout << endl;
}
void printArray(int * A, uint n) {
    for (int i=0; i < n; i++) {
        cout << A[i] << " ";
```

```
    }
    cout << endl;
}
// Function to find the maximum value of the expression
// (A[s] - A[r] + A[q] - A[p]), where s > r > q > p
int maximizeExpression(vector<int> const &A) {
  int n = A.size();

  // input should have at least 4 elements
  if (n < 4) {
    exit(-1);
  }

  // create 4 lookup tables and initialize them to `INT_MIN`
  int first[n + 1], second[n], third[n - 1], fourth[n - 2];

  for (int i = 0; i <= n - 3; i++) {
    first[i] = second[i] = third[i] = fourth[i] = INT_MIN;
  }

  first[n - 2] = second[n - 2] = third[n - 2] = INT_MIN;
  first[n - 1] = second[n - 1] = first[n] = INT_MIN;

  // `first[]` stores the maximum value of `A[l]`
  for (int i = n - 1; i >= 0; i--) {
    first[i] = max(first[i + 1], A[i]);
  }

    printArray(first, n+1);

  // `second[]` stores the maximum value of `A[l] - A[k]`
  for (int i = n - 2; i >= 0; i--) {
    second[i] = max(second[i + 1], first[i + 1] - A[i]);
  }

    printArray(second, n);
  // `third[]` stores the maximum value of `A[l] - A[k] + A[j]`
  for (int i = n - 3; i >= 0; i--) {
    third[i] = max(third[i + 1], second[i + 1] + A[i]);
  }
```

```
    printArray(third, n-1);
  // ‘fourth[]‘ stores the maximum value of ‘A[l] - A[k] + A[j] - A[i]‘
  for (int i = n - 4; i >= 0; i--) {
    fourth[i] = max(fourth[i + 1], third[i + 1] - A[i]);
  }

  // maximum value would be present at ‘fourth[0]‘
  return fourth[0];
}

int main() {
  vector<int> A = {3, 9, 10, 1, 30, 40};

  cout << maximizeExpression(A);

  return 0;
}
```

2.3.1 Metrics

Time complexity $O(n)$

Space complexity $O(n)$ extra space.

3 Implement `Diff` utility

Implement your diff utility, i.e., given two similar strings, efficiently list out all differences between them.

The diff utility is a data comparison tool that calculates and displays the differences between the two texts. It tries to determine the smallest set of deletions and insertions and create one text from the other. Diff is **line-oriented** rather than **character-oriented**, unlike edit distance.

3.1 Examples

3.1.1 Example 1

```
string X = "XMJYAUZ"
string Y = "XMJAATZ"
```

```
X M J -Y A -U +A +T Z
```

3.1.2 Example 2

```
string X = "ABCDFGHJQZ";
string Y = "ABCDEFGIJKRXYZ";

A B C D +E F G -H +I J -Q +K +R +X +Y Z
```

3.1.3 Explanation:

- `'-'` indicates that character is **deleted** from `Y` but it was present in `X`.
- `'+'` indicates that character is **inserted** in `Y` but it was not present in `X`.

3.2 Approach

We can use the `Longest Common Subsequence (LCS)` to solve this problem. The idea is to find the longest sequence of characters present in both original sequences in the same order. From the longest common subsequence, it is only a small step to get the diff-like output:

- If a character is absent in the subsequence but present in the first original sequence, it must have been deleted (indicated by the - marks).
- If it is absent in the subsequence but present in the second original sequence, it must have been inserted (indicated by the + marks).

3.2.1 Longest Common Subsequence problem

Let two sequences be defined as follows: $X = (x_1 x_2 \cdots x_m)$ and $Y = (y_1 y_2 \cdots y_n)$. The prefixes of X are $X_0, X_1, X_2, \ldots, X_m$; the prefixes of Y are $Y_0, Y_1, Y_2, \ldots, Y_n$. Let $LCS(X_i, Y_j)$ represent the set of longest common subsequence of prefixes X_i and Y_j. This set of sequences is given by the following:

$$LCS(X_i, Y_j) = \begin{cases} \epsilon, & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \frown x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{1}$$

3.3 Example Solution

Let see a solution to the LCS problem.

```
// Function to fill the lookup table by finding the length of LCS of substring
vector<vector<int>> findLCS(string X, string Y, uint m, uint n)
{
    // lookup[i][-j] stores the length of LCS of substring X[0...i-1] and Y[0...j-1]
    vector<vector<int>> lookup(m + 1, vector<int>(n + 1));

    // first column -of the lookup table will be all 0
    for (int i = 0; i <= m; i++) {
        lookup[i][0] = 0;
    }

    // first row of the lookup table will be all 0
    for (int j = 0; j <= n; j++) {
        lookup[0][j] = 0;
    }

    // fill the lookup table in a bottom-up manner
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            // if current character of ‘X‘ and ‘Y‘ matches
            if (X[i - 1] == Y[j - 1]) {
                lookup[i][j] = lookup[i - 1][j - 1] + 1;
            }
                // otherwise, if the current character of ‘X‘ and ‘Y‘ don’t match
            else {
                lookup[i][j] = max(lookup[i - 1][j], lookup[i][j - 1]);
            }
        }
    }

    return lookup;
}
```

This will generate the following lookup table for our example:

		A	B	C	D	E	F	G	I	J	K	R	X	Y	Z
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2
C	0	1	2	3	3	3	3	3	3	3	3	3	3	3	3
D	0	1	2	3	4	4	4	4	4	4	4	4	4	4	4
F	0	1	2	3	4	4	5	5	5	5	5	5	5	5	5
G	0	1	2	3	4	4	5	6	6	6	6	6	6	6	6
H	0	1	2	3	4	4	5	6	6	6	6	6	6	6	6
J	0	1	2	3	4	4	5	6	6	7	7	7	7	7	7
Q	0	1	2	3	4	4	5	6	6	7	7	7	7	7	7
Z	0	1	2	3	4	4	5	6	6	7	7	7	7	7	8

Each character of the second string (`Y`) is represented by the columns, while every character from X is represented in the rows. Also note that the *0-th* row and column do not represent a character in the strings.

After generation of the lookup table, we search for the differences by iterating the `lookup` table from the bottom-right corner (something like a *backtrack*).

1. If we find identical character (`X[i] == Y[j]`) we **recursively** repeat the search by using the upper-left neighbor element as the new starting point.

2. If the current character of `Y` is not present in `X`, then we found a character that needs to be added and search **recursively** starting from the left neighbor.

3. If the current character of `X` is not present in `Y`, we found a char that needs to be deleted and we should continue to search **recursively** starting from the top neighbor.

```
// Function to display the differences between two strings
void diff(string X, string Y, uint m, uint n, vector<vector<int>> &lookup)
{
    // if the last character of ‘X‘ and ‘Y‘ matches
    if (m > 0 && n > 0 && X[m - 1] == Y[n - 1])
    {
        diff(X, Y, m - 1, n - 1, lookup);
        cout << " " << X[m - 1];
    }
```

```
    // if the current character of ‘Y‘ is not present in ‘X‘
    else if (n > 0 && (m == 0 || lookup[m][n - 1] >= lookup[m - 1][n]))
    {
        diff(X, Y, m, n - 1, lookup);
        cout << " +" << Y[n - 1];
    }

    // if the current character of ‘X‘ is not present in ‘Y‘
    else if (m > 0 && (n == 0 || lookup[m][n - 1] < lookup[m - 1][n]))
    {
        diff(X, Y, m - 1, n, lookup);
        cout << " -" << X[m - 1];
    }
}
```

Link to the full working example.

3.3.1 Metrics

Time complexity $O(m * n)$

Space complexity $O(m*n)$ extra space, where m is the length of the first string and n is the length of the second string.

4 Word Break

Given a string and a dictionary of words, determine if the string can be segmented into a space-separated sequence of one or more dictionary words.

4.1 Examples

4.1.1 Example 1

```
dict[] = { this, th, is, famous, Word, break, b, r, e, a, k, br, bre, brea, ak, proble

word = "Wordbreakproblem"

Word b r e a k problem
Word b r e ak problem
Word br e a k problem
```

```
Word br e ak problem
Word bre a k problem
Word bre ak problem
Word brea k problem
Word break problem
```

4.2 Approach

The idea is to use recursion to solve this problem. We consider all prefixes of the current string one by one and check if the current prefix is present in the dictionary or not. If the prefix is a valid word, add it to the output string and recur for the remaining string. The recursion word base case is when the string becomes empty, and we print the output string.

4.3 Example Solution

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Function to segment a given string into a space-separated
// sequence of one or more dictionary words
void wordBreak(vector<string> const &dict, string word, string out)
{
    // if the end of the string is reached,
    // print the output string

    if (word.size() == 0)
    {
        cout << out << endl;
        return;
    }

    for (int i = 1; i <= word.size(); i++)
    {
        // consider all prefixes of the current string
        string prefix = word.substr(0, i);

        // if the prefix is present in the dictionary, add it to the
```

```
        // output string and recur for the remaining string

        if (find(dict.begin(), dict.end(), prefix) != dict.end()) {
            wordBreak(dict, word.substr(i), out + " " + prefix);
        }
    }
}

// Word Break Problem Implementation in C++
int main()
{
    // vector of strings to represent a dictionary
    // we can also use a Trie or a set to store a dictionary

    vector<string> dict = { "this", "th", "is", "famous", "Word", "break",
            "b", "r", "e", "a", "k", "br", "bre", "brea", "ak", "problem" };

    // input string
    string word = "Wordbreakproblem";

    wordBreak(dict, word, "");

    return 0;
}
```

4.3.1 Metrics

Time complexity ?

Space complexity ?

4.4 Alternate version of the problem

There is a very famous alternate version of the above problem in which we only have to determine if a string can be segmented into a space-separated sequence of one or more dictionary words or not, and not actually print all sequences.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

// Function to determine if a string can be segmented into space-separated
// sequence of one or more dictionary words
bool wordBreak(vector<string> const &dict, string word)
{
    // return true if the end of the string is reached
    if (word.size() == 0) {
        return true;
    }

    for (int i = 1; i <= word.size(); i++)
    {
        // consider all prefixes of the current string
        string prefix = word.substr(0, i);

        // return true if the prefix is present in the dictionary and the remaining
        // string also forms a space-separated sequence of one or more
        // dictionary words

        if (find(dict.begin(), dict.end(), prefix) != dict.end() &&
                wordBreak(dict, word.substr(i))) {
            return true;
        }
    }

    // return false if the string can't be segmented
    return false;
}

// Word Break Problem Implementation in C++
int main()
{
    // vector of strings to represent a dictionary
    // we can also use a Trie or a set to store a dictionary

    vector<string> dict = { "this", "th", "is", "famous", "Word", "break",
            "b", "r", "e", "a", "k", "br", "bre", "brea", "ak", "problem" };

    // input string
```

```
    string word = "Wordbreakproblem";

    if (wordBreak(dict, word)) {
        cout << "The string can be segmented";
    }
    else {
        cout << "The string can't be segmented";
    }

    return 0;
}
```

The time complexity of the above solution is exponential and occupies space in the call stack.

The word-break problem has **optimal substructure**. We have seen that the problem can be broken down into smaller subproblem, which can further be broken down into yet smaller subproblem, and so on. The word-break problem also exhibits **overlapping subproblems**, so we will end up solving the same subproblem over and over again. If we draw the recursion tree, we can see that the same subproblems are getting computed repeatedly.

The problems having optimal substructure and overlapping subproblem can be solved by dynamic programming, in which subproblem solutions are **memoized** rather than computed repeatedly.

```
#include <iostream>
#include <string>
#include <unordered_set>
#include <algorithm>
using namespace std;

// Function to determine if a string can be segmented into space-separated
// sequence of one or more dictionary words
bool wordBreak(unordered_set<string> const &dict, string word, vector<int> &lookup)
{
    // `n` stores length of the current substring
    int n = word.size();

    // return true if the end of the string is reached
    if (n == 0) {
        return true;
```

```
    }

    // if the subproblem is seen for the first time
    if (lookup[n] == -1)
    {
        // mark subproblem as seen (0 initially assuming string
        // can't be segmented)
        lookup[n] = 0;

        for (int i = 1; i <= n; i++)
        {
            // consider all prefixes of the current string
            string prefix = word.substr(0, i);

            // if the prefix is found in the dictionary, then recur for the suffix
            if (find(dict.begin(), dict.end(), prefix) != dict.end() &&
                wordBreak(dict, word.substr(i), lookup))
            {
                // return true if the string can be segmented
                return lookup[n] = 1;
            }
        }
    }

    // return solution to the current subproblem
    return lookup[n];
}

// Word Break Problem Implementation in C++
int main()
{
    // set of strings to represent a dictionary
    // we can also use a Trie or a vector to store a dictionary
    unordered_set<string> dict = { "this", "th", "is", "famous", "Word", "break",
            "b", "r", "e", "a", "k", "br", "bre", "brea", "ak", "problem" };

    // input string
    string word = "Wordbreakproblem";

    // lookup array to store solutions to subproblems
```

```
    // lookup[i] stores if substring word[n-i...n) can be segmented or not
    vector<int> lookup(word.length() + 1, -1);

    if (wordBreak(dict, word, lookup)) {
        cout << "The string can be segmented";
    }
    else {
        cout << "The string can't be segmented";
    }

    return 0;
}
```

4.4.1 Metrics

Time complexity $O(n^2)$

Space complexity $O(n)$

5 Wildcard Pattern Matching

Given a string and a pattern containing wildcard characters, i.e., `'*'` and `'?'`, where `'?'` can match to any single character in the string and `'*'` can match to any number of characters including zero characters, design an efficient algorithm to check if the pattern matches with the complete string or not.

5.1 Examples

5.1.1 Example 1

```
string = "xyxzzxy"
pattern = "x***y"

Match
```

5.1.2 Example 2

```
string = "xyxzzxy"
pattern = "x***x"

No Match
```

5.1.3 Example 3

```
string = "xyxzzxy"
pattern = "x***x?"

Match
```

5.1.4 Example 4

```
string = "xyxzzxy"
pattern = "*"

Match
```

5.2 Approach

The idea is to use dynamic programming to solve this problem. If we carefully analyze the problem, we can see that it can easily be further divided into subproblems.

Let's take the top-bottom approach to solve this problem.
For a given `pattern[0...m]` and `word[0...n]`,

- If `pattern[m] == '*'`, if `*` matches the current character in the input string, move to the next character in the string; otherwise, ignore $*$ and move to the next character in the pattern.
- If `pattern[m] == '?'`, ignore current characters of both string and pattern and check if $pattern[0...m-1]$ matches $word[0...n-1]$.
- If the current character in the pattern is not a wildcard character, it should match the current character in the input string.

Special care has to be taken to handle base conditions:

- If both the input string and pattern reach their end, return `true`.
- If the only pattern reaches its end, return `false`.
- If only the input string reaches its end, return `true` only if the remaining characters in the pattern are all `*`.