Programming exercises #2

Contents

1 Fractional Knapsack

1.1 Problem

Given the weights and profits of `N` items, in the form of `{profit, weight}` put these items in a knapsack of capacity `W` to get the maximum total profit in the knapsack.

In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

1.2 Input

```
Input: arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50
Output: 240

Explanation: By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.

Hence total price will be 60+100+(2/3)(120) = 240

Input:  arr[] = {{500, 30}}, W = 10
Output: 166.667
```

1.3 Naive Approach

Try all possible subsets with all different fractions.

Time Complexity: $O(2^N)$ Auxiliary Space: $O(N)$

1.4 Greedy approach

The basic idea of the greedy approach is to calculate the ratio `profit/weight` for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).

This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

1.4.1 Example

```
arr[] = {{100, 20}, {60, 10}, {120, 30}}, W = 50.

Sorting: Initially sort the array based on the profit/weight ratio. The sorted array w

Iteration:
- For i = 0, weight = 10 which is less than W. So add this element in the knapsack. pr
- For i = 1, weight = 20 which is less than W. So add this element too. profit = 60 +
- For i = 2, weight = 30 is greater than W. So add 20/30 fraction = 2/3 fraction of th

So the final profit becomes 240 for W = 50.
```

1.5 Example solution

```
// C++ program to solve fractional Knapsack Problem

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

// Structure for an item which stores weight and
// corresponding value of Item
struct Item {
    int profit, weight;

    // Constructor
```

```
    Item(int profit, int weight)
    {
        this->profit = profit;
        this->weight = weight;
    }
};

// Comparison function to sort Item
// according to profit/weight ratio
static bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.profit / (double)a.weight;
    double r2 = (double)b.profit / (double)b.weight;
    return r1 > r2;
}

// Main greedy function to solve problem
double fractionalKnapsack(int W, struct Item arr[], int N)
{
    // Sorting Item on basis of ratio
    sort(arr, arr + N, cmp);

    double finalvalue = 0.0;

    // Looping through all items
    for (int i = 0; i < N; i++) {

        // If adding Item won't overflow,
        // add it completely
        if (arr[i].weight <= W) {
            W -= arr[i].weight;
            finalvalue += arr[i].profit;
        }

        // If we can't add current Item,
        // add fractional part of it
        else {
            finalvalue
                += arr[i].profit
                   * ((double)W / (double)arr[i].weight);
```

```
            break;
        }
    }

    // Returning final value
    return finalvalue;
}

// Driver code
int main()
{
    int W = 50;
    Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    cout << fractionalKnapsack(W, arr, N);
    return 0;
}
```

2 Longest Palindrome

Given a string `s` which consists of lowercase or uppercase letters, return the length of the longest palindrome that can be built with those letters.

Letters are case sensitive, for example, `"Aa"` is not considered a palindrome here.

2.1 Examples

2.1.1 Example 1

```
Input: s = "abccccdd"
Output: 7
```

Explanation: One longest palindrome that can be built is `"dccaccd"`, whose length is 7.

2.1.2 Example 2

```
Input: s = "a"
Output: 1
```

Explanation: The longest palindrome that can be built is `"a"`, whose length is 1.

2.2 Constraints:

- $1 \leq s.length \leq 2000$
- `s` consists of lowercase and/or uppercase English letters only.

2.3 Solution

2.3.1 Greedy approach using hashmaps

To get a palindrome from a group of characters, the group of characters should have some constraints:

1. Every letter must appear **even** number of times like `"aabb"` or `"baba"` in any order it doesn't matter, what matters is that we have **2 a** and **2 b** which are even numbers and at the end it can be converted to a palindrome in our case `"abba"` or `"baab"`. The exception is that **one letter may appear an odd number of times** like `"abcba"` where we have `1 c` but it's not a problem as long as it's only one letter that appears an odd number of times.
2. If there is only one letter type then it must appear an odd number of times like `"a"` or `"aaa"` or `"aaaaa"` etc..

1. Algorithm:
   (a) Get the number of times each letter appears (letter frequency).
   (b) Get how many characters appear an odd number of times.
   (c) Return your the number of all characters – the number of characters that appear an odd number of times.

```
int longestPalindrome_array(string s) {
    //Get every character frequency in an array
    vector<int> freqCount(128); // working with the lower ASCII subset
    for (char c : s)
        freqCount[c]++;

    /* Get how many characters have an odd frequency.*/
```

```
    /* because of the exception that
one character may have odd frequency, we will set oddCount to -1 */
    int oddCount = -1;
    for (int v : freqCount) {
        if (v % 2) oddCount++;
    }
    /* Return your the number of all characters -
the number of characters that appear an odd number of times. */
    // if there is no letter with odd frequency return full length
    return (oddCount > 0) ? s.length() - oddCount: s.length();
}
```

Alternative solution using `unordered_map`:

```
int longestPalindrome_umap(string s) {
    std::unordered_map<char,int> mp;
    for(auto&i:s) mp[i]++;

    int evenLen=0, longestPalindrome=0 , oddLen=0;
    bool visited=0;
    for(auto&it:mp){
        int freq = it.second ;

        //take all freq if even number of times
        if(!(freq&1)){
            evenLen+=freq;
        }
        else if(freq&1){
            //a->3,b->5,c->7 --> take c (7) all and rest freq-1 times i.e a->2 and
            visited=1;
            oddLen+=freq-1;
        }
    }

    longestPalindrome = evenLen;

    if(visited){
        longestPalindrome+=oddLen+1; //+1 bcs max freq was taken freq-1 times abov
    }
```

```
        return longestPalindrome;

    }
```

3 Maximum Subarray

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

3.1 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.

3.2 Constraints:

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

3.3 Solutions

3.3.1 Brute force

We can start with brute-force by trying out every possible sub-array in *nums* and choosing the one with maximum sum.

```
int maxSubArray_bf(vector<int> &nums) {
    int n = size(nums), ans = INT_MIN;
    for (int i = 0; i < n; i++)
```

```
        for (int j = i, curSum = 0; j < n; j++)
            curSum += nums[j],
                    ans = max(ans, curSum);
    return ans;
}
```

Time Complexity $O(N^2)$, where N is the number of elements in `nums`. There are $N*(N+1)/2$ total sub-arrays and trying out each one takes time of $O(N^2)$

Space Complexity $O(1)$

3.3.2 Divide & conquer

We can solve this using divide and conquer strategy. This is the follow-up question asked in this question. This involves modelling the problem by observing that the maximum subarray sum is such that it lies somewhere:

- entirely in the left-half of array `[L, mid-1]`, OR
- entirely in the right-half of array `[mid+1, R]`, OR
- in array consisting of mid element along with some part of left-half and some part of right-half such that these form contiguous subarray - `[L', R'] = [L', mid-1] + [mid] + [mid+1,R']`, where `L' >= L` and `R' <= R`.

With the above observation, we can recursively divide the array into sub-problems on the left and right halves and then combine these results on the way back up to find the maximum subarray sum.

```
int maxSubArray_dq(vector<int> &nums) {
    return maxSubArray_dq(nums, 0, size(nums) - 1);
}

int maxSubArray_dq(vector<int> &A, int L, int R) {
    if (L > R) return INT_MIN;
    int mid = (L + R) / 2, leftSum = 0, rightSum = 0;
    // leftSum = max subarray sum in [L, mid-1] and starting from mid-1
    for (int i = mid - 1, curSum = 0; i >= L; i--)
        curSum += A[i],
                leftSum = max(leftSum, curSum);
```

```
    // rightSum = max subarray sum in [mid+1, R] and starting from mid+1
    for (int i = mid + 1, curSum = 0; i <= R; i++)
        curSum += A[i],
                rightSum = max(rightSum, curSum);
    // return max of 3 cases
    return max({maxSubArray_dq(A, L, mid - 1), maxSubArray_dq(A, mid + 1, R), leftSum +
}
```

Time Complexity $O(NlogN)$, the recurrence relation can be written as $T(N) = 2T(N/2) + O(N)$ since we are recurring for left and right half $(2T(N/2))$ and also calculating maximal subarray including mid element which takes $O(N)$ to calculate. Solving this recurrence using master theorem, we can get the time complexity as $O(NlogN)$

Space Complexity $O(logN)$, required by recursive stack

We can further optimize the previous solution. The $O(N)$ term in the recurrence relation of previous solution was due to computation of max sum subarray involving $nums[mid]$ in each recursion.

But we can reduce that term to $O(1)$ if we precompute it. This can be done by precomputing two arrays *pre* and *suf* where pre[i] will denote maximum sum subarray ending at i and $suf[i]$ denotes the maximum subarray starting at i.

```
vector<int> pre, suf; //Global vectors

int maxSubArray_dq1(vector<int> &nums) {
    pre = suf = nums;
    for (int i = 1; i < size(nums); i++) pre[i] += max(0, pre[i - 1]);
    for (int i = size(nums) - 2; ~i; i--) suf[i] += max(0, suf[i + 1]);
    return maxSubArray_dq1(nums, 0, size(nums) - 1);
}

int maxSubArray_dq1(vector<int> &A, int L, int R) {
    if (L == R) return A[L];
    int mid = (L + R) / 2;
    return max({maxSubArray_dq1(A, L, mid), maxSubArray_dq1(A, mid + 1, R), pre[mid] +
}
```

Time Complexity $O(N)$, the recurrence relation can be written as $T(N) = 2T(N/2)+O(1)$ since we are recurring for left and right half $(2T(N/2))$

and calculating maximal subarray including mid element in $O(1)$. Solving this recurrence using master theorem, we can get the time complexity as $O(N)$

Space Complexity $O(N)$, required by `suf` and `pre`.

4 Create Maximum Number

You are given two integer arrays `nums1` and `nums2` of lengths `m` and `n` respectively. `nums1` and `nums2` represent the digits of two numbers. You are also given an integer `k`.

Create the maximum number of length `k <= m + n` from digits of the two numbers. The relative order of the digits from the same array must be preserved.

Return an array of the `k` digits representing the answer.

4.1 Examples

```
Input: nums1 = [3,4,6,5], nums2 = [9,1,2,5,8,3], k = 5
Output: [9,8,6,5,3]

Input: nums1 = [6,7], nums2 = [6,0,4], k = 5
Output: [6,7,6,0,4]

Input: nums1 = [3,9], nums2 = [8,9], k = 3
Output: [9,8,9]
```

4.2 Constraints:

- $m == nums1.length$
- $n == nums2.length$
- $1 \leq m, n \leq 500$
- $0 \leq nums1[i], nums2[i] \leq 9$
- $1 \leq k \leq m + n$

4.3 Solution

4.3.1 Greedy approach

To create max number of length `k` from two arrays, you need to create max number of length `i` from array one and max number of length `k-i` from array two, then combine them together. After trying all possible `i`, you will get the max number created from two arrays.

Optimization steps:

1. Suppose `nums1 = [3, 4, 6, 5]`, `nums2 = [9, 1, 2, 5, 8, 3]`, the maximum number you can create from `nums1` is `[6, 5]` with length `2`. For `nums2`, it's `[9, 8, 3]` with length 3. Merging the two sequence, we have `[9, 8, 6, 5, 3]`, which is the max number we can create from two arrays without length constraint. If the required length $k \leq 5$, we can simply trim the result to required length from front. For instance, if $k = 3$, then `[9, 8, 6]` is the result.

2. Suppose we need to create max number with `length 2` from `num = [4, 5, 3, 2, 1, 6, 0, 8]`. The simple way is to use a stack, first we push `4` and have `stack [4]`, then comes `5 > 4`, we pop `4` and push `5`, stack becomes `[5]`, `3 < 5`, we push `3`, stack becomes `[5, 3]`. Now we have the required length 2, but we need to keep going through the array in case a larger number comes, $2 < 3$, we discard it instead of pushing it because the stack already grows to required size `2`. `1 < 3`, we discard it. `6 > 3`, we pop `3`, since `6 > 5` and there are still elements left, we can continue to pop `5` and push `6`, the stack becomes `[6]`, since `0 < 6`, we push `0`, the stack becomes `[6, 0]`, the stack grows to required length again. Since `8 > 0`, we pop `0`, although $8 > 6$, we can't continue to pop 6 since there is only one number, which is `8`, left, if we pop `6` and push `8`, we can't get to length 2, so we push `8` directly, the stack becomes `[6, 8]`.

3. In the basic idea, we mentioned trying all possible lengths `i`. If we create max number for different `i` from scratch each time, that would be a waste of time. Suppose `num = [4, 9, 3, 2, 1, 8, 7, 6]`, we need to create max number with length from `1` to `8`. For `i==8`, result is the original array. For `i==7`, we need to drop 1 number from array, since `9 > 4`, we drop `4`, the result is `[9, 3, 2, 1, 8, 7, 6]`. For `i==6`, we need to drop `1` more number, `3 < 9`, skip, `2 < 3`, skip, `1 < 2`, skip, `8 > 1`, we drop `1`, the result is `[9, 3, 2, 8, 7, 6]`. For `i==5`, we need to drop `1` more, but this time, we needn't check from beginning, during

last scan, we already know `[9, 3, 2]` is monotonically non-increasing, so we check `8` directly, since `8 > 2`, we drop `2`, the result is `[9, 3, 8, 7, 6]`. For `i==4`, we start with `8`, `8 > 3`, we drop `3`, the result is `[9, 8, 7, 6]`. For `i==3`, we start with `8`, `8 < 9`, skip, `7 < 8`, skip, `6 < 7`, skip, by now, we've got maximum number we can create from `num` without length constraint. So from now on, we can drop a number from the end each time. The result is `[9, 8, 7]`, For `i==2`, we drop last number `7` and have `[9, 8]`. For `i==1`, we drop last number `8` and have `[9]`.

```
vector<int> maxNumber(vector<int> &nums1, vector<int> &nums2, int k) {
    int n1 = nums1.size(), n2 = nums2.size();
    vector<int> best;
    for (int k1 = max(k - n2, 0); k1 <= min(k, n1); ++k1)
        best = max(best, maxNumber(maxNumber(nums1, k1),
                                   maxNumber(nums2, k - k1)));
    return best;
}

vector<int> maxNumber(vector<int> nums, int k) {
    int drop = nums.size() - k;
    vector<int> out;
    for (int num: nums) {
        while (drop && out.size() && out.back() < num) {
            out.pop_back();
            drop--;
        }
        out.push_back(num);
    }
    out.resize(k);
    return out;
}

vector<int> maxNumber(vector<int> nums1, vector<int> nums2) {
    vector<int> out;
    while (nums1.size() + nums2.size()) {
        vector<int> &now = nums1 > nums2 ? nums1 : nums2;
        out.push_back(now[0]);
        now.erase(now.begin());
    }
```

```
    return out;
}
```