H2.Peinan Weng

problem 1 *FFT*

Answer

Step 1 do $\{3.0, 3.0\}$

1. $A(\omega_1) = A(1) = 3.0 + 3.0 = 6.0$.
2. $A(\omega_2) = A(-1) = 3.0 - 3.0 = 0.0$.

Result = $\{6.0, 0.0\}$

Step 2 do $\{7.2, 8.3\}$

1. $A(\omega_1) = A(1) = 7.2 + 8.3 = 15.5$.
2. $A(\omega_2) = A(-1) = 7.2 - 8.3 = -1.1$.

Result = $\{15.5, -1.1\}$

Step 3 do $\{3.0, 7.2, 3.0, 8.3\}$

1. $A(\omega_1) = A(1) = A_e(1) + A_o(1) = 6.0 + 15.5 = 21.5$.
2. $A(\omega_2) = A(i) = A_e(-1) + i \cdot A_o(-1) = 0.0 - 1.1i = -1.1i$.
3. $A(\omega_3) = A(-1) = A_e(1) - A_o(1) = 6.0 - 15.5 = -9.5$.
4. $A(\omega_4) = A(-i) = A_e(-1) - i \cdot A_o(-1) = 0.0 + 1.1i = 1.1i$.

Result = $\{21.5, -1.1i, -9.5, 1.1i\}$

Step 4 do $\{1.6, 5.2\}$

1. $A(\omega_1) = A(1) = 1.6 + 5.2 = 6.8$.
2. $A(\omega_2) = A(-1) = 1.6 - 5.2 = -3.6$.

Result = $\{6.8, -3.6\}$

Step 5 do $\{4.8, 2.2\}$

1. $A(\omega_1) = A(1) = 4.8 + 2.2 = 7.0$.
2. $A(\omega_2) = A(-1) = 4.8 - 2.2 = 2.6$.

Result = $\{7.0, 2.6\}$

Step 6 do $\{1.6, 4.8, 5.2, 2.2\}$

1. $A(\omega_1) = A(1) = A_e(1) + A_o(1) = 6.8 + 7.0 = 13.8.$
2. $A(\omega_2) = A(i) = A_e(-1) + i \cdot A_o(-1) = -3.6 + 2.6i = -3.6 + 2.6i.$
3. $A(\omega_3) = A(-1) = A_e(1) - A_o(1) = 6.8 - 7.0 = -0.2.$
4. $A(\omega_4) = A(-i) = A_e(-1) - i \cdot A_o(-1) = -3.6 - 2.6i = -3.6 - 2.6i.$

Result = $\{13.8, -3.6 + 2.6i, -0.2, -3.6 - 2.6i\}$

Step 7 do $\{3.0, 1.6, 7.2, 4.8, 3.0, 5.2, 8.3, 2.2\}$

1. $A(\omega_1) = A(1) = A_e(1) + A_o(1) = 21.5 + 13.8 = 35.3.$
2. $A(\omega_2) = A_e(i) + \omega_2 \cdot A_o(i) = -1.1i + \omega_2(-3.6 + 2.6i).$
3. $A(\omega_3) = A(i) = A_e(-1) + iA_o(-1) = -9.5 - 0.2i.$
4. $A(\omega_4) = A_e(-i) + \omega_4 \cdot A_o(-i) = 1.1i + \omega_4(3.6 - 2.6i).$
5. $A(\omega_5) = A(-1) = A_e(1) - A_o(1) = 21.5 - 13.8 = 7.7.$
6. $A(\omega_6) = A_e(i) + \omega_6 \cdot A_o(i) = -1.1i + \omega_6(-3.6 + 2.6i).$
7. $A(\omega_7) = A(-i) = A_e(-1) - iA_o(-1) = -9.5 + 0.2i.$
8. $A(\omega_8) = A_e(-i) + \omega_8 \cdot A_o(-i) = 1.1i + \omega_8(3.6 - 2.6i).$

Result = $\{35.3, -1.1i + \omega_2(-3.6 + 2.6i), -9.5 - 0.2i, 1.1i + \omega_4(3.6 - 2.6i), 7.7, -1.1i + \omega_6(-3.6 + 2.6i), -9.5 + 0.2i, 1.1i + \omega_8(3.6 - 2.6i)\}$

PROBLEM 2 *Operating Cost Analysis*

Answer

The key to solve this problem is that for a certain time [n], there are only 2 potential path.

Path 1. The [n] city is same as [n-1] city.

Path 2. The [n] city is different from [n-1] city, and we need to add 1 ”move cost”.

So, if we stored those 2 potential minimum cost in each step, this problem can be easily solved. C++ code with comments is in next page.

```cpp
vector<int> homeWork2_22
    (vector<int>& a, vector<int>& b, int& cost, int& minCost) {
    // city1 stores each step cost with our last step at city 1.
    // city2 stores each step cost with our last step at city 2.
    // change city cost is not in those 2 arrays.
    vector<int> city1;
    vector<int> city2;
    // Use dp to store latest result.
    pair<int, int> dp;
    pair<int, int> cur;
    dp.first = a[0];
    dp.second = b[0];
    city1.push_back(1);
    city2.push_back(2);
    for (int i = 1; i < a.size(); ++i) {
        vector<int> tmpCity1 = dp.first <= dp.second + cost ? city1 : city2;
        vector<int> tmpCity2 = dp.second <= dp.first + cost ? city2 : city1;
        tmpCity1.push_back(1);
        tmpCity2.push_back(2);
        city1 = move(tmpCity1);
        city2 = move(tmpCity2);
        cur.first = a[i] + min(dp.first, dp.second + cost);
        cur.second = b[i] + min(dp.second, dp.first + cost);
        dp = cur;
    }
    // minCost is the minimum cost and will be returned.
    minCost = min(cur.first, cur.second);
    // Return value is like ("112212122211")
    // Which indicates the city we choose of each step.
    return cur.first <= cur.second ? city1 : city2;
}
```

Time Complexity = $O(n)$.

PROBLEM 3 *New Skyline*

Answer (Time Complexity = $O(n^2)$, Space Complexity = $O(n)$)

First of all, this problem can be converted to find a longest increasing sub array of a given array.

$dp[n]$ denotes the length of the longest increasing sub array of the last elements start from $nums[n]$ to $nums[nums.size() - 1]$.

In each outer loop with n, we need to traversal elements from the end back to $n + 1$, if element $nums[i] \le nums[j] (with\ i < j)$, means we could add $nums[i]$ to current longest increasing sub array starting from $nums[j]$, and update longest length starting from $nums[i]$(that is $dp[i]$), choose the maximum between original $dp[i]$ and $dp[j] + 1$.

We also need to calculate the maximum length, that is the maximum value in array dp. After that, we need to reconstruct the longest increasing sub array and do output. We need to find the first element i that satisfies $dp[i] = maxLength$, and push it in our output array. Then we need to find first elements with $dp[i] = maxLength - 1$, $dp[i] = maxLength - 2$, $dp[i] = maxLength - 3$, etc., and push them to our output array, then we will get the final result. C++ code is as follows,

```
vector<int> lengthOfLIS_DP_Output(vector<int>& nums) {
    if (nums.empty())
        return nums;
    int maxLength = 1;
    vector<int> dp(nums.size(), 1);
    dp[nums.size() - 1] = 1;
    for (int i = nums.size() - 2; i >= 0; --i) {
        for (int j = nums.size() - 1; j > i; --j)
            if (nums[i] <= nums[j])
                dp[i] = max(dp[i], dp[j] + 1);
        maxLength = max(maxLength, dp[i]);
    }
    vector<int> result;
    int ptr = 0;
    while (maxLength) {
        while (dp[ptr] != maxLength)
            ++ptr;
        result.push_back(nums[ptr]);
        --maxLength;
        ++ptr;
    }
    return result;  // maxLength = result.size().
}
```

PROBLEM 4 *Divide People*

Answer

The Solution of this problem is quite complicated.

Time Complexity = $O(n^2 \cdot sum(Array))$ = $O(n^2 \cdot nM)$ = $O(n^3 M)$.

Space Complexity = $O(n \cdot sum(Array))$ = $O(n \cdot nM)$ = $O(n^2 M)$.

At the beginning, I would like to divide this problem into 2 sub-problems.

1. Find out the smallest difference between 2 sub-arrays.

2. Reconstruct one sub-array.

The solution is based on 01 knapsack problem, with constrained element number.

The key to solve this problem is that we need to build a memory $dpPath[i][j]$, which means the last number added into a sub-array, with $sum(subarray) = i$ and $sizeof(subarray) = j$, then we could convert the problem to find the maximum $i \leq n/2$, with $dpPath[i][j] \neq 0$, that means this path exists.

Firstly, we begin a 3-level traversal:

Level - 1, traversal from $n = 0$ to $n = array.size() - 1$, that means add each element in original input array one by one.

Level - 2, traversal from $k = sum$ to $k = array[n]$, that means updating each potential path.

Level - 3, traversal from $eNumber = 1$ to $eNumber = array.size()$, that means updating path with our current element be the eNumber-th element in that potential subarray.

Secondly, we find the minimum difference between 2 subarrays.

Traversal from $i = array.size()/2$ to $i = 1$, find maximum potential sum *result* of a half number subarray, then break;

At last, do reconstruction. Because we already build up a path graph, we are able to output 1 case of any potential subarray with $sum(subarray) = k$ and $sizeof(subarray) = n$, then we just need to reconstruction 1 subarray with the sum we get before, and elements number $\frac{sizeof(input)}{2}$. C++ code is in next page.

```cpp
vector<int> homework4_DP(vector<int>& input) {
    int sum = 0;
    for (int&i : input)
        sum += i;
    // When input.size() is odd, add a '0' in it to make it even.
    // Note that only positive number will be add into our result.
    if (input.size() & 1)
        input.push_back(0);
    // dpPath[i][j] denotes the last number added in a subarray
    // Which satisfies sum(subarray) = i and sizeof(subarray) = j.
    vector<vector<int>> dpPath(sum + 1, vector<int>(input.size() + 1, 0));
    dpPath[0][0] = 1;
    for (int i = 0; i < input.size(); ++i)
        for (int j = sum; j >= input[i]; --j)
            for (int k = 1; k <= input.size(); ++k)
                if (!dpPath[j][k] && dpPath[j - input[i]][k - 1])
                    dpPath[j][k] = input[i];
    int result = 0;
    for (int i = sum / 2; i >= 0; --i)
        if (dpPath[i][input.size() / 2]) {
            result = i;
            break;
        }
    // We find the minimum difference, stored in "result".
    cout << "Array Input : ";
    for (int&i : input)
        cout << i << ' ';
    cout << endl;
    cout << "Sum of Array = " << sum << endl;
    cout << "Sum of Subarray 1 = " << result << endl;
    cout << "Sum of Subarray 2 = " << sum - result << endl;
    cout << "Difference Between Subarrays = " << sum - 2 * result << endl;
    cout << "Elements in Subarray 1 = ";
    vector<int> oneResult;
    // Do subarray reconstruction based on path data
    for (int i = input.size() / 2, step = result; i > 0; --i) {
        oneResult.push_back(dpPath[step][i]);
        step -= dpPath[step][i];
        cout << oneResult.back() << ' ';
    }
    cout << endl;
    return oneResult;   // oneResult = one of the two divided subarrays.
}
```

A sample output is as follows, with marks on how to do reconstruction,

```
  0    1    0    0    0    0    0    0
  1    0    1    0    0    0    0    0
  2    0    0    0    0    0    0    0
  3    0    0    0    0    0    0    0
  4    0    4    0    0    0    0    0
  5    0    0    4    0    0    0    0
  6    0    0    0    0    0    0    0
  7    0    7    0    0    0    0    0
  8    0    0    7    0    0    0    0
  9    0    0    0    0    0    0    0
 10    0   10    0    0    0    0    0
 11    0    0    7    0    0    0    0
 12    0    0    0    7    0    0    0
 13    0   13    0    0    0    0    0
 14    0    0   10    0    0    0    0
 15    0    0    0   10    0    0    0
 16    0   16    0    0    0    0    0
 17    0    0   10    0    0    0    0
 18    0    0    0   10    0    0    0
 19    0    0    0    0    0    0    0
 20    0    0   13    0    0    0    0
 21    0    0    0   10    0    0    0
 22    0    0    0    0   10    0    0
 23    0    0   13    0    0    0    0
 24    0    0    0   13    0    0    0
 25    0    0    0    0   13    0    0
 26    0    0   16    0    0    0    0
 27    0    0    0   13    0    0    0
 28    0    0    0    0   13    0    0
 29    0    0   16    0    0    0    0
 30    0    0    0   13    0    0    0
 31    0    0    0    0   13    0    0
 32    0    0    0    0    0    0    0
 33    0    0    0   16    0    0    0
 34    0    0    0    0   13    0    0
 35    0    0    0    0    0   13    0
 36    0    0    0   16    0    0    0
 37    0    0    0    0   16    0    0
 38    0    0    0    0    0   16    0
 39    0    0    0   16    0    0    0
 40    0    0    0    0   16    0    0
 41    0    0    0    0    0   16    0
 42    0    0    0    0    0    0    0
 43    0    0    0    0   16    0    0
 44    0    0    0    0    0   16    0
 45    0    0    0    0    0    0    0
 46    0    0    0    0   16    0    0
 47    0    0    0    0    0   16    0
 48    0    0    0    0    0    0    0
 49    0    0    0    0    0    0    0
 50    0    0    0    0    0   16    0
 51    0    0    0    0    0    0   16
Array Input : 1 4 7 10 13 16
Sum of Array = 51
Sum of Subarray 1 = 24
Sum of Subarray 2 = 27
Difference Between Subarrays = 3
Elements in Subarray 1 = 13 7 4
```

find 4, end of resconstruction

output array : 13 7 4

find 7, move on to 11-7 = 4

find 13, add it into the output array, then move on to 24-13=11

one sub array is 24, find it.