CS1020E: DATA STRUCTURES AND ALGORITHMS I

Tutorial 11 – 

(Week 13, starting 7 November 2016)

1. *Longest Sub-Array*

You are given `arr` of integers, its size (which is very large), and a non-zero integer `sum`. **For each** index `rightIdx`, in increasing order, **print out** the pair containing the **leftmost** index `leftIdx` and that `rightIdx` of the longest consecutive sub-array arr[leftIdx..rightIdx], such that the sum of all numbers in arr[leftIdx..rightIdx] is equal to `sum`, provided such a left index exists.

```
void solve(int arr[], int size, int sum) { /* leftIdx rightIdx */ }
```

(a) Design and implement a **O(N^2)** algorithm, which is much better than the O(N^3) brute force algorithm
(b) The O(N^2) algorithm can be **optimized**. Design and implement an **O(N log N)** algorithm
(c) Now if the array only contains positive integers, implement an **O(N)** algorithm that does the job

Answer

Naïve O(N^3) algorithm[1]

```
void solve(int arr[], int size, int sum) {
   for (int rightIdx = 0; rightIdx < size; rightIdx++) { // O(N)
      for (int leftIdx = 0; leftIdx <= rightIdx; leftIdx++) { // O(N)
         int subArrSum = 0;
         for (int subIdx = leftIdx; subIdx <= rightIdx; subIdx++) //dep.
            subArrSum += arr[subIdx];
         if (subArrSum == sum){
            cout << "[" << leftIdx << "," << rightIdx << "]" << endl;
            break; // solution found, reset leftIdx, next rightIdx
         }
      }
   }
}
```

(a)

The sum of all elements in arr[left..right] is generally arr[right] - arr[left - 1]. However, there is the boundary case when left is 0, i.e. the start of the array. Therefore, we can create a cumulative sum array `arrSum`, starting with the sum of 0 before the first element is added.

arrSum[idx+1] = arrSum[idx] + arr[idx]
arr[left..right] = arrSum[right + 1] - arrSum[left]

[1] O(N^3) time complexity – This can be seen by drawing a 3D graph

	0	1	2	3	4	5	6	7	8	size 9
arr	2	6	-3	-5	-4	8	12	-20	1	
arrSum	0	2	8	5	0	-4	4	16	-4	-3

```
void solve(int arr[], int size, int sum) {
   int arrSum [size + 1];
   arrSum[0] = 0;
   for (int idx = 0; idx < size; idx++) // O(N)
      arrSum[idx + 1] = arrSum[idx] + arr[idx];

   for (int rightIdx = 0; rightIdx < size; rightIdx++) { // O(N)
      for (int leftIdx = 0; leftIdx <= rightIdx; leftIdx++) { // O(N)
         if (arrSum[rightIdx + 1] - arrSum[leftIdx] == sum){
            cout << "[" << leftIdx << "," << rightIdx << "]" << endl;
            break; // next rightIdx, reset leftIdx
         }
      }
   }
}
```

The time complexity is $O(N^2)$ == O(N x N + N)

Another possible $O(N^2)$ solution is to try all right index, and then do O(N) cumulative sum from that index backwards to the left to find cumulative sum that equals to target sum.

(b)

In part (a), we have successfully transformed the problem such that we no longer bother about the original array `arr`. For every right index, how can sorting help us efficiently find a matching left index that meets a certain condition?

If we have a **sorted** array, for every right index, we can perform binary search to find the left index. Binary search can be applied because the **outcome** of our evaluation is either '<' or '==' or '>', and we are able to eliminate half the array based on the outcome.

Original problem

	0	1	2	3	4	5	6	7	8	size 9
arr	2	6	-3	-5	-4	8	12	-20	1	
arrSum	0	2	8	5	0	-4	4	16	-4	-3

Problem converted to difference of cumulative sums, so original array is ignored

	0	1	2	3	4	5	6	7	8	9
sumIdx	0	1	2	3	4	5	6	7	8	9
arrSum	0	2	8	5	0	-4	4	16	-4	-3

Sorted by cumulative sum ascending; elements with same sum should appear sorted by sumIdx ascending

Sorted	0	1	2	3	4	5	6	7	8	9
sumIdx	5	8	9	0	4	1	6	3	2	7
arrSum	-4	-4	-3	0	0	2	4	5	8	16

Since we have exactly N values of j, we need to find each i in O(log N) time, to achieve total O(N log N) time. How does binary search accomplish this? Let's take the above example, where we want to find `sum` of -12.

Sorted	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Res**
sumIdx	5	8	9	0	4	1	6	3	2	7	
arrSum	-4	-4	-3	0	0	2	4	5	8	16	-4-0 > -12
arrSum	-4	-4	-3	0	0	2	4	5	8	16	-4-5 > -12
arrSum	-4	-4	-3	0	0	2	4	5	8	16	-4-8 ==-12
arrSum	-4	-4	-3	0	0	2	4	5	8	16	low==hi

Notice that we cannot simply use the binary search covered in lectures. When we find a match, we cannot stop there, because there may exist another index with the same cumulative sum to the left of the current location. For example, if I am at sorted index 1 (arrSum[8]), I should eventually take leftmost sorted index 0 (arrSum[5]) instead.

Finally, we still have to check that the result of the "binary search" is a match, and that the matching number is to the left of the right number in the cumulative sum array.

```
bool hasLowerSumThan(const pair<int, int>& left,
      const pair<int, int>& right) { // order by cumuluative sum ONLY
   return left.first < right.first;
}

void solve(int arr[], int size, int sum) {
   pair<int, int> arrSum [size + 1], sorted [size + 1]; // <sum, sumIdx>
   arrSum[0] = sorted[0] = make_pair(0, 0);
   for (int idx = 0; idx < size; idx++) { // O(N)
      arrSum[idx+1] = make_pair(arrSum[idx].first + arr[idx], idx + 1);
      sorted[idx+1] = arrSum[idx+1];
   }

   sort(sorted, sorted + size + 1); // use natural ordering

   for (int sumIdx = 1; sumIdx <= size; sumIdx++) { // O(N)
      int sortedLeftIdx = lower_bound( // first match by cumul. sum ONLY
         sorted, sorted + size + 1,
         make_pair(arrSum[sumIdx].first - sum, -1), // dummy pair
         hasLowerSumThan) - sorted;
      if (sortedLeftIdx <= size && // if exists exact match, and (L, R)
         sorted[sortedLeftIdx].first == arrSum[sumIdx].first - sum &&
         sorted[sortedLeftIdx].second < arrSum[sumIdx].second)
         cout << "[" <<
            sorted[sortedLeftIdx].second << "," <<
            sumIdx - 1 << "]" << endl;
   }
}
```

(c)

Implement a sliding window. As every number is positive, once we match or exceed the given sum, the left end of the window should slide as there are no other possible solutions containing the given left index.

This is **O(N)**.

```
void solve(int arr[], int size, int sum) {
   if (sum <= 0) return; // prevents window from shrinking when empty
   int windowSum = 0, leftIdx = 0;
   for (int rightIdx = 0; rightIdx < size; rightIdx++) { // O(N)
      windowSum += arr[rightIdx];
      while (windowSum > sum) // exceeds given sum
         windowSum -= arr[leftIdx++];
      if (windowSum == sum) {
         cout << "[" << leftIdx << "," << rightIdx << "]" << endl;
         windowSum -= arr[leftIdx++];
      }
   }
}
```

Alternative and better **O(N)** solution that uses hash table (PS: Not written in function format).

```
#include <iostream>
#include <unordered_map>
using namespace std;

#define MAX_N 10000

int main() {
  int i, ri, n, target, check, arr[MAX_N], arrSum[MAX_N];
  unordered_map<int, int> mapper;
  mapper[0] = -1;
  cin >> n;
  for (i = 0; i < n; i++) {
    cin >> arr[i];
    arrSum[i] = (i == 0 ? arr[i] : arrSum[i-1]+arr[i]);
    if (mapper.find(arrSum[i]) == mapper.end())
      mapper[arrSum[i]] = i;
  }
  cin >> target;
  for (ri = 0; ri < n; ri++) {
    check = arrSum[ri]-target;
    if ((mapper.find(check) != mapper.end()) && mapper[check] < ri)
      cout << mapper[check]+1 << " " << ri << endl;
  }
  return 0;
}
```

2. *Next Problem*

Have you completed question 1(a) - (c)?

3. *VisuAlgo Online Quiz*

This semester, we will not do VisuAlgo Online Quiz formally as the PE2 setting is already too stressful. Therefore this part is currently optional. That is, not graded (0%).

However, Lab TA will instruct you to try the following exercise (20 minutes) as it is still very useful:

https://visualgo.net/training.html?diff=Hard&n=15&tl=20&module=list,recursion,sorting,hashtable

See how a machine (VisuAlgo) creates questions and auto grade them… instantly… :O.

TA will also spend some time discussing the solution of some random questions in VisuAlgo online quiz.

Note to TA: Just pick any question in VisuAlgo Online Quiz that you find interesting and discuss how you will solve them (quickly).