CS1020E Tutorial + Lab 11

Mark NG

a0116298@u.nus.edu
http://mollymr305.github.io

November 11, 2016

Tutorial Solutions

"Tutorial 11 – The Last Tutorial"

Question 1: Longest Sub-Array

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 */ }
```

Question 1: Longest Sub-Array

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
         }
      }
   }
}
```

Question 1 (a)

Design and implement an $\mathcal{O}(N^2)$ algorithm, which is much better than the $\mathcal{O}(N^3)$ brute force algorithm.

Question 1 (a)

Design and implement an $\mathcal{O}(N^2)$ algorithm, which is much better than the $\mathcal{O}(N^3)$ brute force algorithm.

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]

Question 1 (a)

Design and implement an $\mathcal{O}(N^2)$ algorithm, which is much better than the $\mathcal{O}(N^3)$ brute force algorithm.

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]

	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

Question 1 (a)

Design and implement an $\mathcal{O}(N^2)$ algorithm, which is much better than the $\mathcal{O}(N^3)$ brute force algorithm.

```
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 $O(N^2) == O(N \times N + N)$

Question 1 (b)

The $\mathcal{O}(N^2)$ algorithm can be **optimized**. Design and implement an $\mathcal{O}(N \log N)$ algorithm.

Question 1 (b)

The $\mathcal{O}(N^2)$ algorithm can be **optimized**. Design and implement an $\mathcal{O}(N \log N)$ algorithm.

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.

Question 1 (b)

The $\mathcal{O}(N^2)$ algorithm can be **optimized**. Design and implement an $\mathcal{O}(N \log N)$ algorithm.

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

Question 1 (b)

The $\mathcal{O}(N^2)$ algorithm can be **optimized**. Design and implement an $\mathcal{O}(N \log N)$ algorithm.

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.

Question 1 (b)

The $\mathcal{O}(N^2)$ algorithm can be **optimized**. Design and implement an $\mathcal{O}(N \log N)$ algorithm.

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

Question 1 (b)

```
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;
   }
}
```

Question 1 (c)

If the array only contains positive integers, implement an $\mathcal{O}(N)$ algorithm that does the job.

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.

As this O(N) solution is better than the solution in part (B), part (A), and the naïve $O(N^3)$ solution, one should use this solution instead...

```
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++];
      }
   }
}
```

Question 2:

...

Question 3: VisuAlgo Online Quiz

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

End of Tutorial Discussion

Note: Detailed solutions (i.e. the file `T11_ans.pdf`) will be released soon at

`http://www.comp.nus.edu.sg/~stevenha/cs1020e.html`

Take Home Lab

No Comments...

Let's take a short break!

Kattis Problem

Simple application of `unordered_map <string, string>`?

`https://open.kattis.com/problems/whatdoesthefoxsay`

Algorithm Problem

Suppose you have an array of numbers $[x_1, x_2, x_3, \ldots, x_N]$. Your task is to find the largest sum which can be formed by taking a contiguous subarray, i.e. of the form $[x_t, x_{t+1}, \ldots, x_{t+k}]$.

Algorithm Problem

Suppose you have an array of numbers $[x_1, x_2, x_3, \ldots, x_N]$. Your task is to find the largest sum which can be formed by taking a contiguous subarray, i.e. of the form $[x_t, x_{t+1}, \ldots, x_{t+k}]$.

Example:

Algorithm Problem

Suppose you have an array of numbers $[x_1, x_2, x_3, \ldots, x_N]$. Your task is to find the largest sum which can be formed by taking a contiguous subarray, i.e. of the form $[x_t, x_{t+1}, \ldots, x_{t+k}]$.

Example:

$\Longrightarrow$ Given $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$.

Algorithm Problem

Suppose you have an array of numbers $[x_1, x_2, x_3, \ldots, x_N]$. Your task is to find the largest sum which can be formed by taking a contiguous subarray, i.e. of the form $[x_t, x_{t+1}, \ldots, x_{t+k}]$.

Example:

$\Longrightarrow$ Given $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$.

$\Longrightarrow$ Take the subarray $[4, -1, 2, 1]$ with largest sum 6.

Algorithm Problem

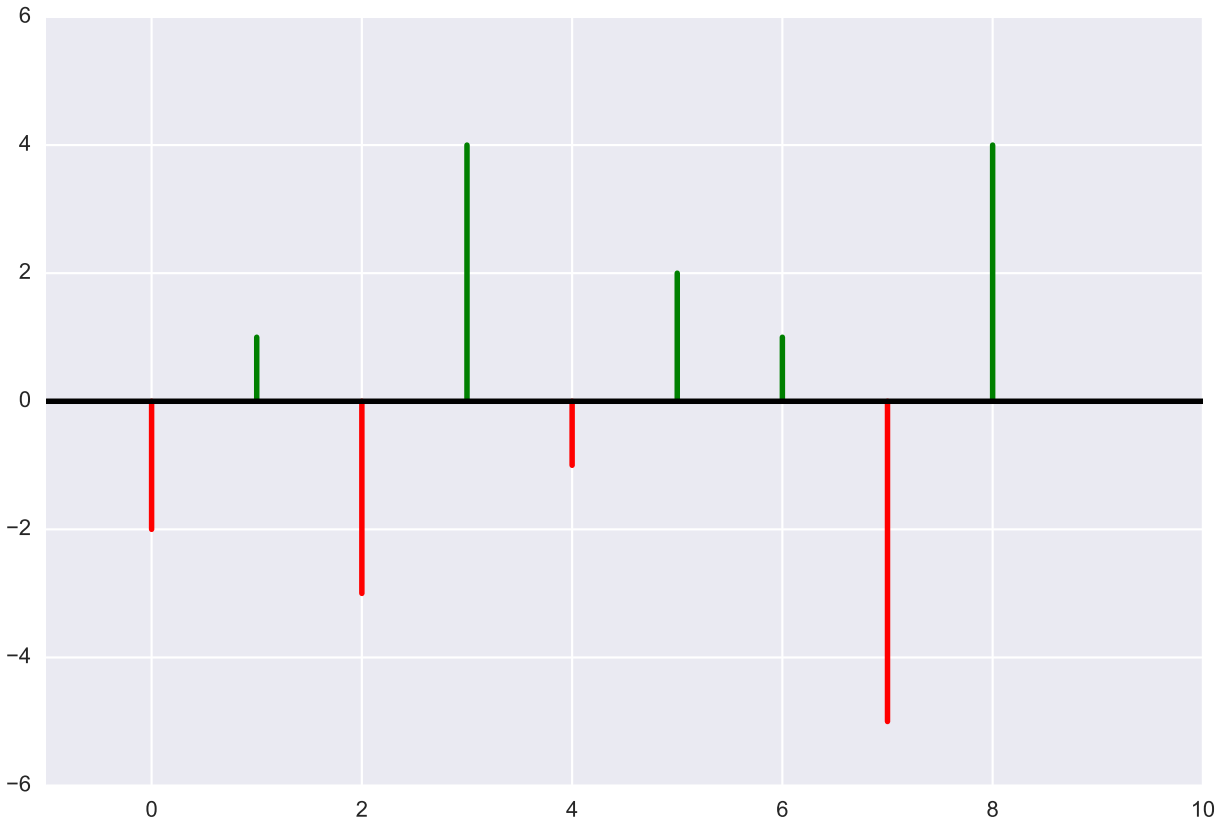

Figure : Kadane's Algorithm

Algorithm Problem

https://open.kattis.com/problems/commercials

Any Questions?

Good luck for your finals!