Heaps And Subarrays

[PROBLEM] Print the running median.

## Understanding the Median

* Median is the middle number in a sorted list of numbers.
* To determine the median value in a sequence of numbers, the numbers must first be sorted, or arranged, in value order from lowest to highest or highest to lowest.
  + If there is an odd number of numbers, the median value is the number that is in the middle, with the same number of numbers below and above.
  + If there is an even number of numbers in the list, the middle pair must be determined, added together, and divided by two to find the median value.

Example 1:

Given an array int myArray[] = { 3, 13, 2, 34, 11, 26, 47 };.

1. Sort the array.

{2, 3, 11, 13, 26, 34, 47}

1. Since there are 7 elements in the array, median is the middle number. In this case it is 13.

{2, 3, 11, **13**, 26, 34, 47}

Example 2:

Given an array int myArray[] = { 3, 13, 2, 34, 11, 17, 27, 47 };.

1. Sort the array.

{2, 3, 11, 13, 17, 27, 34, 47}

1. Since there are 8 elements in the array, median is the average of the two numbers in the middle.

{2, 3, 11, **13, 17**, 27, 34, 47}

1. So median is…

{(13 + 17) ÷ 2 = **15**}.

## Running Median

* Given that integers are being read from a data stream. Find the median of all the elements read so far starting from the first integer till the last integer. This is also called the **Median of Running Integers**.

Example:

After reading 1st element of *stream* - 5-> median - 5

After reading 2nd element of *stream* - 5, 15-> median - 10

After reading 3rd element of *stream* - 5, 15, 1-> median - 5

After reading 4th element of *stream* - 5, 15, 1, 3-> median - 4, so on...

**APPROACH 1**:

|  |
| --- |
| class RunningMedian {  void insert() {  }  float getMedian() {  }  }; |

1. Take a dynamic list of integers.
2. With every number coming in
   1. Add to end of list
   2. Sort the list
   3. Print Median

TC: n2logn

SC: 1 (not accounting for input numbers)

**APPROACH 2:**

1. Take minHeap
2. With every number coming in
   1. Add to minHeap
   2. Remove half elements (put in some temp storage)
   3. Calculate Median
   4. Insert half elements back
   5. Print Median

TC: n2logn

SC: n/2

**APPROACH 3:**

|  |
| --- |
| void RunningMedian::insert(int x)  {  if (left.size() == 0)  left.insert(x);  else  if (x <= left.get())  left.insert(x);  else  right.insert(x);  if (*abs*(left.size() - right.size()) > 1) {  if (left.size() > right.size()) {  right.insert(left.get());  left.remove();  }  else {  left.insert(right.get());  right.remove();  }  }  }  float RunningMedian::getMedian()  {  if (left.size() == right.size())  return (left.get() + right.get()) / 2.0;  else if (left.size() > right.size())  return left.get();  return right.get();  } |

TC: nlogn

SC: n (heaps cumulative size) — if we don’t consider storing of input 0(1)

## Subarray, Subsequence and Subset

* Consider an array…

int myArray[] = { 1,2,3,4 };

* + **Subarray**: contiguous sequence in an array i.e.

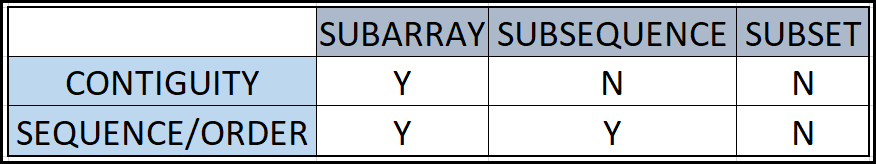
{1, 2}, { 1,2,3 }

* + **Subsequence**: Need not to be contiguous, but maintains order i.e.

{1, 2, 4}

* + **Subset**: contain any possible combinations of original set.

{ 1, 3 }, { 1, 4 }, {}



Example:

Consider an array

int A[] = { 2,6,1,9,0,5,4,8,7 };

* Subset which is not a subarray => {8, 2}, {2, 7, 8, 4}
* Subset which is not a subsequence => {2, 7, 8, 4}
* Subsequence which is not a subarray => {2, 0, 8}, {1,0, 7}
* Every subsequence is a SUBSET.
* Every subarray is a subsequence and a subset too.

Given an array/sequence of size n, possible

* Subarray = n\*(n+1)/2
* Subsequence = 2n -1 (non-empty subsequences)
* Subset = 2n
* Given array A of size N, how many non-empty subarrays it can have of size K. (K <= N).
  + N-K+1

[**PROBLEM**] Given an array A of size N. Find the maximum subarray sum out of all subarrays.

int A[] = { 3,2,-10,4,1,-3,**5,7,8**,-11,6,1 };

N = 12

ANS: 22

**APPROACH 1:**

TC: O(n3)

SC: 1

|  |
| --- |
| long long getMaxSubarraySum(*vector*<int> a) {  long long mxSum = a[0];  for (auto startldx = 0; startldx < a.*size*(); startldx++) {  for (auto endldx = startldx; endldx < a.*size*(); endldx++) {  long long curSum = 0;  for (auto k = startldx; k <= endldx; k++)  curSum += a[k];  mxSum = *max*(mxSum, curSum);  }  }  return mxSum;  } |

**APPROACH 2**:

TC: O(n2)

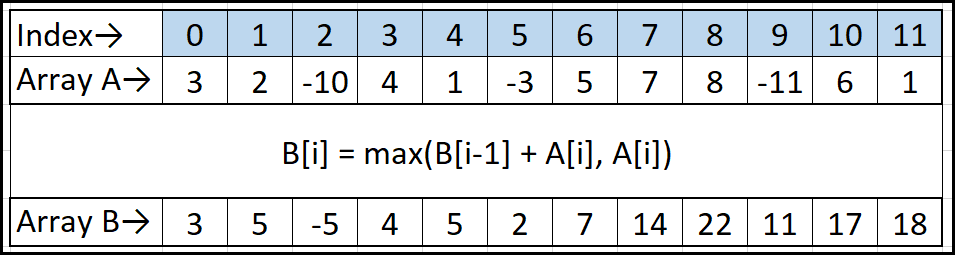
SC: 1

|  |
| --- |
| long long getMaxSubarraySum(*vector*<int> a) {  long long mxSum = a[0];  for (auto startldx = 0; startldx < a.*size*(); startldx++) {  long long curSum = 0;  for (auto endldx = startldx; endldx < a.*size*(); endldx++) {  curSum += a[endldx];  mxSum = *max*(mxSum, curSum);  }  }  return mxSum;  } |

**APPROACH 3**:

TC: N

SC: N



|  |
| --- |
| long long getMaxSubarraySum(*vector*<int> a) {  long long mxSum = a[0];  *vector*<long long> b(a.*size*());  b[0] = a[0];  mxSum = *max*(mxSum, b[0]);  for (auto i = 1; i < a.*size*(); i++) {  b[i] = *max*((long long)a[i], b[i - 1] + a[i]);  mxSum = *max*(mxSum, b[i]);  }  return mxSum;  } |

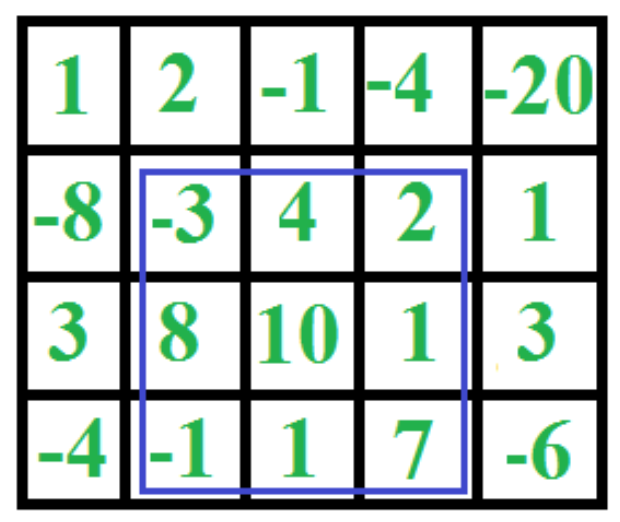
## KADANE'S ALGORITHM

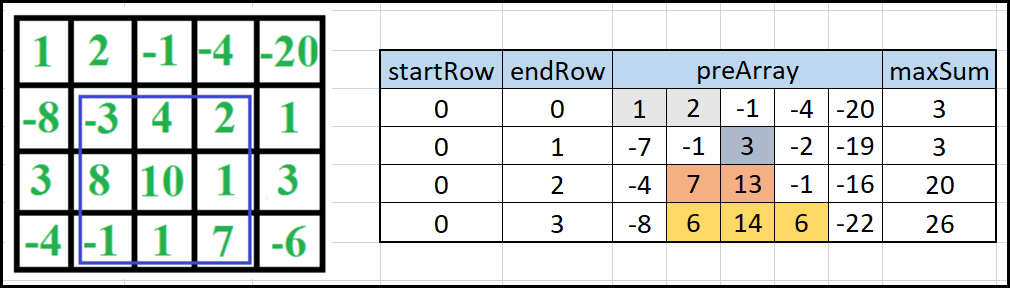
TC: N

SC: 1

|  |
| --- |
| long long getMaxSubarraySum(*vector*<int> a) {  long long mxSum = a[0];  long long preSum = a[0];  for (auto i = 1; i < a.*size*(); i++) {  mxSum = *max*((long long)a[i], preSum + a[i]);  preSum = mxSum;  }  return mxSum;  } |

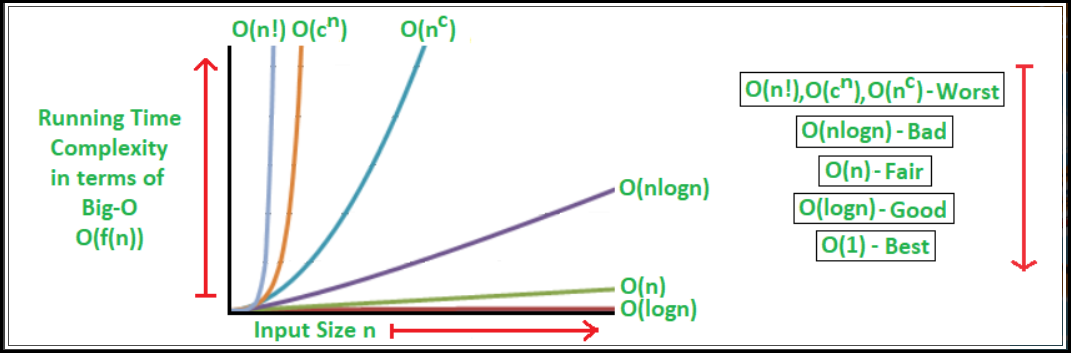
[**PROBLEM**] Find Maximum sum rectangle in a 2D matrix.





|  |
| --- |
| long long getMaxSubarraySum(*vector*<int> a) {  long long mxSum = a[0];  *vector*<long long> b(a.*size*());  b[0] = a[0];  mxSum = *max*(mxSum, b[0]);  for (auto i = 1; i < a.*size*(); i++) {  b[i] = *max*((long long)a[i], b[i - 1] + a[i]);  mxSum = *max*(mxSum, b[i]);  }  return mxSum;  }  void add2Array(const *vector*<int>& er, *vector*<int>& p) {  for (auto i = 0; i < p.*size*(); i++)  p[i] += er[i];  }  long long getMaxSumRectangle(*vector*<*vector*<int>> a) {  long long mxSum = a[0][0];  long long curSum = a[0][0];  int rows = a.*size*();  *vector*<int> p(a[0].*size*());  for (auto sr = 0; sr < rows; sr++) {  *std*::*copy*(a[sr].*begin*(), a[sr].*end*(), p.*begin*());  for (auto er = sr + 1; er < rows; er++) {  add2Array(a[er], p);  curSum = getMaxSubarraySum(p);  mxSum = *max*(curSum, mxSum);  }  }  return mxSum;  }  int main(void) {  *vector*<*vector*<int>> arr = {  {1,2,-1,-4,-20},  {-8,-3,4,2,1},  {3,8,10,1,3},  {-4,-1,1,7,-6}  };  for (auto i = 0; i < arr.*size*(); i++)  *cout* << getMaxSubarraySum(arr[i]) << *endl*;  *cout* << getMaxSumRectangle(arr) << *endl*;  return 0;  } |

## Analysis of Algorithms



|  |  |
| --- | --- |
| CATEGORY | ALGORITHMIC PROBLEMS |
| O(1) | * isPowerOf2 * Count Multiples of 3 and 5 Up to N * Natural Numbers Range Sum [M, N] [12, 19] |
| O(sqrtN) | * isPrimeCheck * countFactorsOfANumber * checkANumberlsPerfectSquare |
| O(logN) | * binarySearch * binExponentiation to do X power N * matrixExponentiation to calculate Nth fib * Counting Digits in a number |
| O(N) | * Kadane’s Algorithm |
| O(NlogN) |  |
| O(N2) |  |
| O(N2logN) |  |
| O(N3) |  |
| O(2N) |  |
| O(N\*2N) | * Subsets generation using bit manipulation * Meet in the middle |
| O(N!) |  |
| O(N\*N!) |  |