Sorting

Merge Two Sorted Array and Print Merged Answer

TC: n + m

SC: 1

void mergeSorted(vector<int> a, vector<int> b){

}

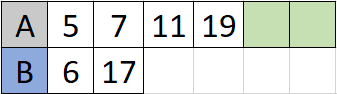
Merge Two Sorted Array and Return a new Merged Array

TC: n + m

SC: 1 (no input and output params counted)

|  |
| --- |
| *vector*<int> mergeSorted(*vector*<int> a, *vector*<int> b) {  int al = a.*size*();  int bl = b.*size*();  int ai = 0, bi = 0, ci = 0;  *vector*<int> c(al +bl ); // allocate c of size al+bl  while (ai < al && bi < bl) {  if (a[ai] < b[bi])  c[ci++] = a[ai++];  else  c[ci++] = b[bi++];  }  while (ai < al)  c[ci++] = a[ai++];  while (bi < bl)  c[ci++] = b[bi++];  return c;  } |
| *vector*<int> mergeSorted(*vector*<int> a, *vector*<int> b) {  int al = a.*size*();  int bl = b.*size*();  int ai = 0, bi = 0, ci = 0;  *vector*<int> c(al + bl);  while (ai < al || bi < bl) {  if (ai >= al)  c[ci++] = b[bi++];  else if (bi >= bl)  c[ci++] = a[ai++];  else if (a[ai] < b[bi])  c[ci++] = a[ai++];  else  c[ci++] = b[bi++];  }  return c;  } |

[**PROBLEM**] Given two sorted arrays A and B, A has enough space to accommodate B. Merge both into A and keep A sorted.



|  |
| --- |
| void merge(*vector*<int>& nums1, int m, *vector*<int>& nums2, int n) {  int p1 = m - 1;  int p2 = n - 1;  int i = nums1.*size*() - 1;  while (p2 >= 0) {  if (p1 >= 0 && nums1[p1] > nums2[p2])  nums1[i--] = nums1[p1--];  else  nums1[i--] = nums2[p2--];  }  } |

## Squares of Sorted Array

*vector*<int> getSqSortedArray(*vector*<int> a) {

}

**APPROACH 1**:

1. Square Each element Of A
2. Sort A
3. Return A.

TC: N + NlogN

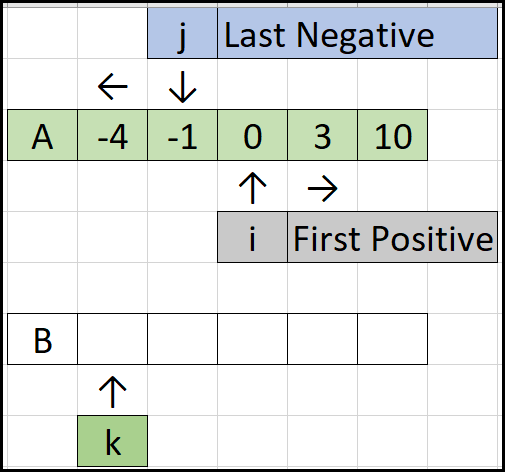
SC: 1

**APPROACH** **2:** (BRUTE FORCE)

1. Take Abs of each value of A
2. Sort A
3. Square of Each element of A
4. Return A

TC: N + NlogN + N

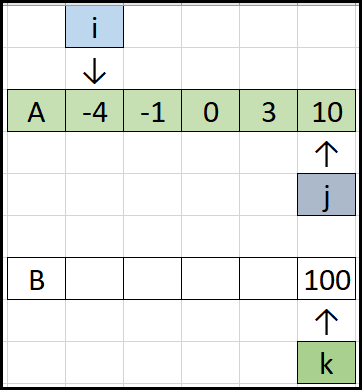
SC: 1

**APPROACH** **3:**

***TWO Pointers*** by finding first-Positive and last-Negative element.

Optimize for cases of ‘all Positive’ and ‘all Negative’ separately.

TC: n+n=2n =O(n)



**APPROACH** **4:**

***TWO Pointers*** (i from Beginning of A, j from End Of A, choose max squared value to be kept at the end of B)

## Learning about Two Pointer Technique

1. Two pointers in one array
   1. Can start from left.
   2. Can start from right.
   3. One can start from left MOVES RIGHT, another can start from right and MOVES LEFT.
   4. One starts from somewhere middle and MOVES LEFT, another starts from somewhere middle and MOVES RIGHT.
2. Two pointers in two arrays
   1. Both arrays from left side.
   2. Both arrays from right side.
   3. One from left of one array and Another from right of second yesterday.

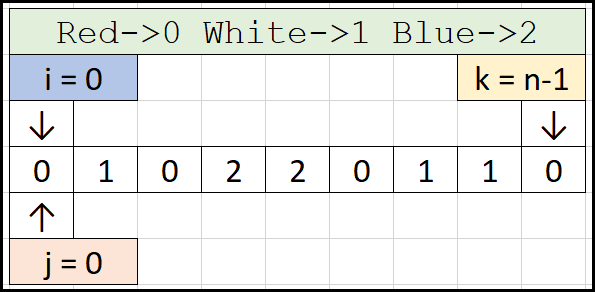
## Three Pointers

[**PROBLEM**] Given an array of red, green, and blue balls arrange them in groups of all red together, greens together and blue together. Do in a single scan of the array.

// Here Red->0, White->1, Blue->2

int arr[] = { 0, 1, 0, 2, 2, 0, 1, 1, 0 };

|  |
| --- |
| Sudo Code:  i = 0, j = 0, k = n-1 (Where n is number of balls)  while(j <= k)  if pointer j point to red ball than  swap a[i] and a[j];  and perform i++; j++;  if pointer j point to white ball than  just perform j++;  if pointer j point to blue ball than  swap a[j] and a[k];  and perform k--; |



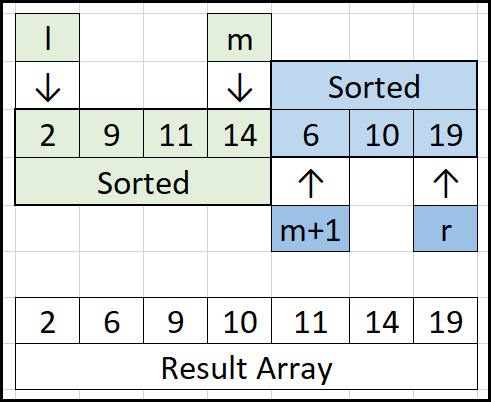
|  |
| --- |
| void dutchflag(*vector*<int>& arr) {  int i = 0, j = 0, k = arr.*size*() - 1;  while (j <= k) {  switch (arr[j]) {  case 0:  *swap*(arr[i++], arr[j++]);  break;  case 1:  j++;  break;  case 2:  *swap*(arr[j], arr[k--]);  break;  }  }  }  int main() {  // Here red->0, white->1, blue->2  *vector*<int> arr = { 0,1,0,2,2,0,1,1,0 };  dutchflag(arr);  for (auto i : arr)  *cout* << i << " ";  *cout* << "\n";  return 0;  } |

[**PROBLEM**] Given a string containing uppercase, lowercase and digits. Arrange the string as LOWERCASE - DIGITS – UPPERCASE. Order among all lowercase can be anything, Digits order among themselves can be anything.

|  |
| --- |
| *string* dutchflag(*string* s) {  int i = 0, j = 0, k = s.*length*() - 1;  while (j <= k) {  if (s[j] >= '0' && s[j] <= '9') {  j++;  }  else if (s[j] >= 'a' && s[j] <= 'z') {  *swap*(s[j++], s[i++]);  }  else {  *swap*(s[j], s[k--]);  }  }  return s;  }  int main() {  *string* s = "a1BA2CA5nishithDADDAB5";  *string* ans = dutchflag(s);    *cout* << ans << "\n";  return 0;  } |

[**PROBLEM**]

Merge two sorted parts of A, first part A[l, m] and second part A[m+1, r]. And keep the sorted answer back in A. Extra space of [r-l+1] is allowed.



TC: R-L+1=O(N)

SC: R-L+1=O(N)

Space can be optimized to (R-L)/2. Copy second half of A to another array and merge two arrays back to A starting from RIGHT SIDE in all three (first half, second half, Overall array A last index)

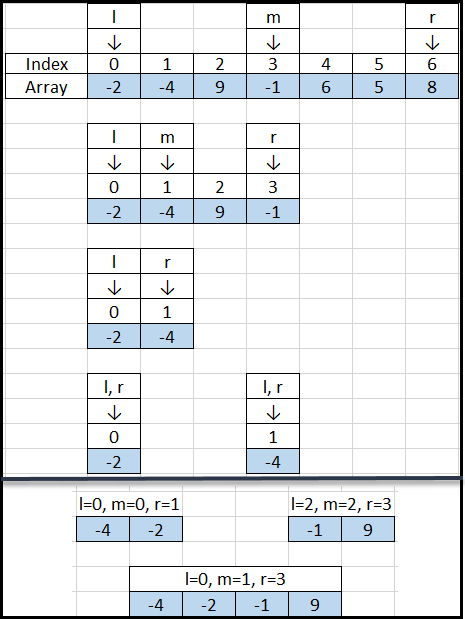
|  |
| --- |
| void mergeTwoSortedPartsAndUpdateA(*vector*<int>& a, int l, int m, int r) {  int i = l, j = m + 1;  *vector*<int> b(r - l + 1);  int k = 0;  while(i <= m && j <= r) {  if (a[i] < a[j])  b[k++] = a[i++];  else  b[k++] = a[j++];  }  while(i <= m)  b[k++] = a[i++];  while(j <= r)  b[k++] = a[j++];  for (int i = l; i <= r; i++)  a[i] = b[i-l];  }  int main(void) {  *vector*<int> arr = { 2, 9, 11, 14, 6, 10, 19 };  int l = 0, m = 3, r = arr.*size*()-1;  mergeTwoSortedPartsAndUpdateA(arr, l, m, r);  for (auto i : arr)  *cout* << i << " ";  *cout* << "\n";  return 0;  } |

## MERGE SORT

T(N) = T(N/2) + T(N/2) + N

T(N) = 2\*T(N/2) + N => NlogN

SC: N



|  |
| --- |
| int mergeTwoSortedPartsAndUpdateA(*vector*<int> a, int l, int m, int r) {  int i = l, j = m + 1;  *vector*<int> b(r - l + 1);  int k = 0;  int c = 0;  while (i <= m && j <= r) {  if (a[i] <= a[j]) // For equal elements  b[k++] = a[i++];  else {  c += (m - i + 1);  b[k++] = a[i++];  }  }  while (i <= m)  b[k++] = a[i++];  while (j <= r)  b[k++] = a[i++];  for (int i = l; i <= r; i++) // Copying Back b to a  a[i] = b[i - l];  return c;  } |

|  |
| --- |
| void mergeSortHelper(*vector*<int>& a, int l, int r) {  if (l < r) {  int m = l + (r - l) / 2;  mergeSortHelper(a, l, m);  mergeSortHelper(a, m + 1, r);  mergeTwoSortedPartsAndUpdateA(a, l, m, r);  }  }  void mergeSort(*vector*<int>& a) {  mergeSortHelper(a, 0, a.*size*() - 1);  }  int main(void) {  *vector*<int> arr = { 2, 9, 11, 14, 6, 10, 19 };  mergeSort(arr);  for (auto i : arr)  *cout* << i << " ";  *cout* << "\n";  return 0;  } |

[PROBLEM] INVCNT - Inversion Count

Let A[0...n - 1] be an array of n distinct positive integers. If i < j and A[i] > A[j] then the pair (i, j) is called an inversion of A. Given n and an array A your task is to find the number of inversions of A.

|  |
| --- |
| int mergeTwoSortedPartsAndUpdateA(*vector*<int> a, int l, int m, int r) {  int i = l, j = m + 1;  *vector*<int> b(r - l + 1);  int k = 0;  int c = 0;  while (i <= m && j <= r) {  if (a[i] <= a[j]) // For equal elements  b[k++] = a[i++];  else {  c += (m - i + 1);  b[k++] = a[j++];  }  }  while (i <= m)  b[k++] = a[i++];  while (j <= r)  b[k++] = a[j++];  for (int i = l; i <= r; i++) // Copying Back b to a  a[i] = b[i - l];  return c;  }  int mergeSortHelper(*vector*<int>& a, int l, int r) {  int c = 0;  if (l < r) {  int m = l + (r - l) / 2;  c += mergeSortHelper(a, l, m);  c += mergeSortHelper(a, m + 1, r);  c += mergeTwoSortedPartsAndUpdateA(a, l, m, r);  }  return c;  }  int mergeSortCountInversions(*vector*<int>& a) {  return mergeSortHelper(a, 0, a.*size*() - 1);  }  int main(void) {  *vector*<int> arr = { 2, 9, 11, 14, 6, 10, 19 };  *cout* << mergeSortCountInversions(arr) << "\n";  return 0;  } |

# Searching

[PROBLEM] Given an array A of size N. and Q queries containing an element K.

For each query, print TRUE or FALSE (TRUE if item is present in the array, FALSE otherwise).

1 <= N <= 105

1 <= Q <= 105

-109 <= A[i] <= 109

|  |  |
| --- | --- |
| INPUT | OUTPUT |
| 5O  32 10 -20 -34 54 32 98 5555 6666 221 8 3 4  5 1 5 3 5 3 3 22 3322232 2 2 2 3 2 2 4 45 5  444 1 2 3 45 56 78 90 12 34 43 54 65 76 87  98 21 32 54  7  43  -100  999  22  -34 | TRUE  FALSE  FALSE  TRUE  TRUE |

|  |  |
| --- | --- |
| **APPROACH 1**  TC: Q\*N  SC: 1  **VERDICT**: TLE | Read and Store Elements in Array  For each Query  Do a linear Search |
| **APPROACH 2**  TC: NlogN + N + QIogN = (N+O)IogN  SC: 1  VERDICT: AC | Read and Store in Array  Sort the Array  Remove Duplicates from Array  For each query  Do a binary search |
| **APPROACH 3**  TC: NlogN + QlogN (map to be BBST)  SC: 2N = O(N)  VERDICT: AC | Read and store all elements in a Map (K:array element, V: freq)  For each Query  Check in Map and Print the Answer |
| **APPROACH 4**  TC: NlogN + QlogN (map to be BBST)  SC: N = O(N)  VERDICT: AC | Read and store all elements in a Set  For Each query  Check in Set and Print the Answer |
| **APPROACH 5**  **VERDICT: MLE** | Can we use frequency tables? |
| **APPROACH 6**  TC: NIogN + Q\*N/2 = O(Q\*N)  SC: 1  VERDICT: TLE | Read and Sort the Array  For Each Query  Using Two Pointers Find and Print the Answer |
| **APPROACH 7** | Use Binary Search Tree (to search efficiently) |

Binary Search - Searches for an element Which can also be done using SET/Map (because they provide a function to check a key is there or not)

We Will talk about certain search problems that can't be solved easily With SET/MAP.

## What is BINARY SEARCH?

[T(N) = T(N/2) + c] => logN

***Given an ordered RANGE, if we need to find an answer in ordered range, at each step if we can discard one half of the range. That is the idea of binary search.***