The key observation about mergesort is that no matter

what the input array looks like, mergesort will divide

the array recursively into the same subproblems.

The way the input influences the way the algorithm works

in is the merge phase of the algorithm. A merge of

two N-element sorted arrays requires at least N

comparisons (this happens if one array's elements are

all less than the other array's smallest element)

and at most 2N-1 comparisons. In any case, merge

requires 2N assignments.

So, merge requires Theta(N) time regardless of the input.

This implies that mergesort takes Theta(N log N) time

for sorted, reverse-sorted, and random inputs.

**METHOD 3 (Compare in Pairs)**  
If n is odd then initialize min and max as first element.  
If n is even then initialize min and max as minimum and maximum of the first two elements respectively.  
For rest of the elements, pick them in pairs and compare their  
maximum and minimum with max and min respectively.

|  |
| --- |
| #include<stdio.h>    /\* structure is used to return two values from minMax() \*/  struct pair  {    int min;    int max;  };    struct pair getMinMax(int arr[], int n)  {    struct pair minmax;    int i;      /\* If array has even number of elements then      initialize the first two elements as minimum and      maximum \*/    if (n%2 == 0)    {      if (arr[0] > arr[1])      {        minmax.max = arr[0];        minmax.min = arr[1];      }      else      {        minmax.min = arr[0];        minmax.max = arr[1];      }      i = 2;  /\* set the startung index for loop \*/    }       /\* If array has odd number of elements then      initialize the first element as minimum and      maximum \*/    else    {      minmax.min = arr[0];      minmax.max = arr[0];      i = 1;  /\* set the startung index for loop \*/    }      /\* In the while loop, pick elements in pair and       compare the pair with max and min so far \*/    while (i < n-1)    {      if (arr[i] > arr[i+1])      {        if(arr[i] > minmax.max)          minmax.max = arr[i];        if(arr[i+1] < minmax.min)          minmax.min = arr[i+1];      }      else      {        if (arr[i+1] > minmax.max)          minmax.max = arr[i+1];        if (arr[i] < minmax.min)          minmax.min = arr[i];      }      i += 2; /\* Increment the index by 2 as two                 elements are processed in loop \*/    }      return minmax;  }    /\* Driver program to test above function \*/  int main()  {    int arr[] = {1000, 11, 445, 1, 330, 3000};    int arr\_size = 6;    struct pair minmax = getMinMax (arr, arr\_size);    printf("\nMinimum element is %d", minmax.min);    printf("\nMaximum element is %d", minmax.max);    getchar();  } |

Time Complexity: O(n)

Total number of comparisons: Different for even and odd n, see below:

If n is odd: 3\*(n-1)/2

If n is even: 1 Initial comparison for initializing min and max,

and 3(n-2)/2 comparisons for rest of the elements

= 1 + 3\*(n-2)/2 = 3n/2 -2

Second and third approaches make equal number of comparisons when n is a power of 2.

In general, method 3 seems to be the best.