**A+ Computer Science  
Sorting & Searching M/C Test**

**Directions ::** On your answer sheet, mark the letter of the best answer to each question.

1. **Which of the following sorts has a partition method that uses a pivot location?**

|  |  |
| --- | --- |
| a. | merge sort |
| b. | selection sort |
| c. | quick sort |
| d. | bubble sort |
| e. | insertion sort |

2. **What is the bigO of the code below?**

int n = //user input

for(int i=0; i<n; i++){

for(int j=0; j<n; j++){

System.out.println(i\*j);

}

}

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(1) |
| c. | O(N \* log2N) |
| d. | O(Log2N) |
| e. | O(N\*N) |

3. **What is the bigO of the code below?**

int n = //user input

for(int i=0; i<n; i++){

for(int j=1; j<n; j=j\*2){

System.out.println(i\*j);

}

}

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(1) |
| c. | O(N \* log2N) |
| d. | O(Log2N) |
| e. | O(N\*N) |

4. **What is output by the code below?**

String[] s = "one two dog".split(" ");

ArrayList<String> words;

words = new ArrayList<String>(Arrays.asList(s));

String big = Collections.min(words);

out.println(big);

|  |  |
| --- | --- |
| a. | one |
| b. | two |
| c. | dog |
| d. | one two dog |
| e. | one two |

5. **What is output by the code below?**

String[] s = "one two dog".split(" ");

ArrayList<String> list;

list = new ArrayList<String>(Arrays.asList(s));

String big = Collections.max(list);

out.println(big);

|  |  |
| --- | --- |
| a. | one |
| b. | two |
| c. | dog |
| d. | one two dog |
| e. | two dog |

6. **Which of the these algorithms has a O(1) best case runtime and a O(N) worst case runtime?**

|  |  |
| --- | --- |
| a. | Binary Search |
| b. | Linear Search |
| c. | Selection Sort |
| d. | Insertion Sort |
| e. | Quick Sort |

7. **Which of the these algorithms has a O(N) best case runtime and a O(N\*N) worst case runtime?**

|  |  |
| --- | --- |
| a. | Binary Search |
| b. | Linear Search |
| c. | Selection Sort |
| d. | Insertion Sort |
| e. | Quick Sort |

8. **Which of the these algorithms has a O(N\*Log2N) best case runtime and a O(N\*N) worst case runtime?**

|  |  |
| --- | --- |
| a. | Binary Search |
| b. | Linear Search |
| c. | Selection Sort |
| d. | Insertion Sort |
| e. | Quick Sort |

9. **Which of the following sorts selects an item and then moves items around to put the selected item in the correct location?**

|  |  |
| --- | --- |
| a. | merge sort |
| b. | insertion sort |
| c. | quick sort |
| d. | selection sort |
| e. | heap sort |

10. **Which of these is the correct BigO for searching a single linked linked-list?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N2) |

11. **Which of the these algorithms has a O(N\*N) best case runtime and a O(N\*N) worst case runtime?**

|  |  |
| --- | --- |
| a. | Binary Search |
| b. | Linear Search |
| c. | Selection Sort |
| d. | Insertion Sort |
| e. | Quick Sort |

12. **Which of the these algorithms has a O(1) best case runtime and a O(Log2N) worst case runtime?**

|  |  |
| --- | --- |
| a. | Binary Search |
| b. | Linear Search |
| c. | Selection Sort |
| d. | Insertion Sort |
| e. | Quick Sort |

13. **Which of these is the correct BigO for adding an item to a TreeSet?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

14. **Which of these is the correct BigO for adding an item to a HashMap?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

15. **Which of these is the correct BigO for adding an item to the front of an array?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

16. **Which of these is the correct BigO for adding an item to the end of an array?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

17. **Which of these is the correct BigO for adding an item to the front of a Java LinkedList?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

18. **Which of these is the correct BigO for deleting any item from an ArrayList?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

19. **Which of these is the correct BigO for searching for a binary search tree?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

20. **Which of these is the correct BigO for traversing all nodes in a binary search tree?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

21. **Which of these is the correct BigO for searching a binary search tree if all items are inserted in the tree in order?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

22. **Which of these is the correct BigO for adding an item to the end of an ArrayList?**

|  |  |
| --- | --- |
| a. | O(N) |
| b. | O(Log2N) |
| c. | O(N \* log2N) |
| d. | O(1) |
| e. | O(N\*N) |

23. **Which of the following would correctly fill < blank 1 > ?**

public static void sortOne( Comparable[] list )

{

for(int i=0; i<list.length-1; i++)

{

int min = i;

for(int j=i+1; j<list.length; j++)

{

if(list[j]. **< blank 1 >** (list[min])<0)

min = j;

}

if( min != i)

{

Comparable temp = list[min];

list[min] = list[i];

list[i] = temp;

}

}

}

|  |  |
| --- | --- |
| a. | compareTo |
| b. | int |
| c. | Comparable |
| d. | String |
| e. | Object |

24. **Assuming <blank 1> is filled correctly, what sort is sortOne()?**

public static void sortOne( Comparable[] list )

{

for(int i=0; i<list.length-1; i++)

{

int min = i;

for(int j=i+1; j<list.length; j++)

{

if(list[j]. **< blank 1 >** (list[min])<0)

min = j;

}

if( min != i)

{

Comparable temp = list[min];

list[min] = list[i];

list[i] = temp;

}

}

}

|  |  |
| --- | --- |
| a. | bubble sort |
| b. | selection sort |
| c. | insertion sort |
| d. | binary search |
| e. | merge sort |

25. **Which of the following code statements could fill <blank 1>?**

public static void sortTwo( Comparable[] x )

{

for(int i=1; i<x.length; i++)

{

Comparable val = x[i];

int j = i;

while( j>0 && **< blank 1 >** )

{

x[j] = x[j-1];

j--;

}

x[j] = val;

}

}

|  |  |
| --- | --- |
| a. | x[j-1].compareTo(val)>0 |
| b. | x[j-1].compareTo(val)<0 |
| c. | x[j-1].compareTo(val)==0 |
| d. | x[j-1].compareTo(val)<-1 |
| e. | x[j-1].compareTo(val)<1 |

26. **Assuming <blank 1> is filled correctly, what sort is** sortTwo()?

public static void sortTwo(Comparable[] x)

{

for(int i=1; i<x.length; i++)

{

Comparable val = x[i];

int j = i;

while(j > 0 && **< blank 1 >**  )

{

x[j] = x[j-1];

j--;

}

x[j] = val;

}

}

|  |  |
| --- | --- |
| a. | bubble sort |
| b. | selection sort |
| c. | insertion sort |
| d. | binary search |
| e. | merge sort |

27. **Which of the following sorts splits data into smaller lists, sorts the smaller lists, and then combines all of the sorted smaller lists back into one big list?**

|  |  |
| --- | --- |
| a. | merge sort |
| b. | radix sort |
| c. | quick sort |
| d. | selection sort |
| e. | heap sort |

28. **What is output by the code below?**

ArrayList<Integer> r;

r = new ArrayList<Integer>();

r.add(1);

r.add(2);

r.add(3);

out.println( r.indexOf(100) );

|  |  |
| --- | --- |
| a. | -1 |
| b. | 1 |
| c. | true |
| d. | False |
| e. | 0 |

29. **For the code at right to correctly implement a linear search, which of the following would correctly fill < blank 1 > ?**

public static int find( ArrayList list, Object param )

{

for(int i=0; i < list.size(); i++)

{

if( list.get(i).equals(param) )

return i;

}

**< blank 1 >**

}

|  |  |
| --- | --- |
| a. | return -1; |
| b. | return 0; |
| c. | return i; |
| d. | return list.size(); |
| e. | return 1; |

30. **What would be the value of list after four passes of the** help()  **method?**

public void sort( int[] list, int front, int back)

{

int mid = (front+back)/2;

if( mid==front) return;

sort(list, front, mid);

sort(list, mid, back);

help(list, front, back);

}

private void help(int[] list, int front, int back)

{

int[] temp = new int[back-front];

int i = front, j = (front+back)/2, k =0;

int mid =j;

while( i<mid && j<back)

{

if(list[i]< list [j])

temp[k++]= list[i++];

else

temp[k++]= list[j++];

}

while(i<mid)

temp[k++]= list[i++];

while(j<back)

temp[k++]= list[j++];

for(i = 0; i<back-front; ++i)

list[front+i]=temp[i];

}

**//code in the main**

int[] list = {39,6,11,23,18,3,20,5,57};

sort(list, 0, list.length);

|  |  |
| --- | --- |
| a. | 6 39 11 23 18 3 20 5 57 |
| b. | 6 11 23 39 3 18 20 5 57 |
| c. | 6 11 23 39 3 18 5 20 57 |
| d. | 6 11 23 39 18 3 20 5 57 |
| e. | 6 39 11 23 18 3 20 5 57 |

31. **How many times would** help() **be called before list was sorted?**

public void sort( int[] list, int front, int back)

{

int mid = (front+back)/2;

if( mid==front) return;

sort(list, front, mid);

sort(list, mid, back);

help(list, front, back);

}

private void help(int[] list, int front, int back)

{

int[] temp = new int[back-front];

int i = front, j = (front+back)/2, k =0;

int mid =j;

while( i<mid && j<back)

{

if(list[i]< list [j])

temp[k++]= list[i++];

else

temp[k++]= list[j++];

}

while(i<mid)

temp[k++]= list[i++];

while(j<back)

temp[k++]= list[j++];

for(i = 0; i<back-front; ++i)

list[front+i]=temp[i];

}

**//code in the main**

int[] list = {39,6,11,23,18,3,20,5,57};

sort(list, 0, list.length);

|  |  |
| --- | --- |
| a. | **5** |
| b. | **6** |
| c. | **7** |
| d. | **8** |
| e. | **9** |

32. **Which sort is shown below?**

public void sort( int[] list, int front, int back)

{

int mid = (front+back)/2;

if( mid==front) return;

sort(list, front, mid);

sort(list, mid, back);

help(list, front, back);

}

private void help(int[] list, int front, int back)

{

int[] temp = new int[back-front];

int i = front, j = (front+back)/2, k =0;

int mid =j;

while( i<mid && j<back)

{

if(list[i]< list [j])

temp[k++]= list[i++];

else

temp[k++]= list[j++];

}

while(i<mid)

temp[k++]= list[i++];

while(j<back)

temp[k++]= list[j++];

for(i = 0; i<back-front; ++i)

list[front+i]=temp[i];

}

|  |  |
| --- | --- |
| a. | bubble sort |
| b. | selection sort |
| c. | insertion sort |
| d. | merge sort |
| e. | heap sort |

33. **What type of sort is funSort()?**

public static void funSort( int[] ray )

{

for(int i=0; i< ray.length-1; i++)

{

int max = i;

for(int j = i+1; j<ray.length; j++)

{

if( ray[j] > ray[max] )

max = j;

}

if( max != i){

int temp = ray[max];

ray[max] = ray[i];

ray[i] = temp;

}

}

}

|  |  |
| --- | --- |
| a. | bubbleSort |
| b. | insertionSort |
| c. | selectionSort |
| d. | quickSort |
| e. | heap sort |

34. funSort()  **will put the items in ray in what type of order?**

public static void funSort( int[] ray )

{

for(int i=0; i< ray.length-1; i++)

{

int max = i;

for(int j = i+1; j < ray.length; j++)

{

if( ray[j] > ray[max] )

max = j;

}

if( max != i){

int temp = ray[max];

ray[max] = ray[i];

ray[i] = temp;

}

}

}

|  |  |
| --- | --- |
| a. | ascending |
| b. | descending |
| c. | random |
| d. | median |
| e. | what a hooptie order |

35. **If ray is storing 10 elements, what is the maximum number of swaps** funSort()  **could make in order to put the elements in order?**

public static void funSort( int[] ray )

{

for(int i=0; i< ray.length-1; i++)

{

int max = i;

for(int j = i+1; j < ray.length; j++)

{

if( ray[j] > ray[max] )

max = j;

}

if( max != i){

int temp = ray[max];

ray[max] = ray[i];

ray[i] = temp;

}

}

}

|  |  |
| --- | --- |
| a. | 8 |
| b. | 9 |
| c. | 10 |
| d. | 11 |
| e. | 7 |

36. **If** funSearch() **will only work on sorted data, what type of sort is** funSearch()?

public static int funSearch( int[] list, int val )

{

int len=list.length;

int bot=0, top=len-1;

int middle=0;

while( bot <= top )

{

middle=(bot+top)/2;

if (list[middle] == val)

return middle;

else

if(list[middle]>val)

top=middle-1;

else

bot=middle+1;

}

return -1;

}

|  |  |
| --- | --- |
| a. | linearSearch |
| b. | selectionSearch |
| c. | binarySearch |
| d. | quickSearch |
| e. | whatASearch |

37. **If** funSearch() **is passed a list that contains 100 integers in sorted order, how many checks will** funSearch()  **have to make before it determines that the item it is searching for is NOT in the list ?**

public static int funSearch( int[] list, int val )

{

int len=list.length;

int bot=0, top=len-1;

int middle=0;

while( bot <= top )

{

middle = (bot+top)/2;

if( list[middle] == val )

return middle;

else

if( list[middle] > val )

top=middle-1;

else

bot=middle+1;

}

return -1;

}

|  |  |
| --- | --- |
| a. | 5 |
| b. | 6 |
| c. | 7 |
| d. | 8 |
| e. | 9 |

38. Consider the following instance variable and incomplete method.

The method getBiggest should return the largest value in tRay.

private int[] tRay; //assume the tRay contains values

public int getBiggest()

{

int big = Integer.MIN\_VALUE;

**/\* code \*/**

return big;

}

Which of the following code segments shown below could be used to replace

**/\* code \*/** so that getBiggest will work as intended?

I. for( int i=0; i<tRay.length; i++ )

if( tRay[i] > big )

big = tRay[i];

II. for( int item : tRay )

if( tRay[item] > big )

big = tRay[item];

III. for( int item : tRay )

if( item > big )

big = item;

|  |  |
| --- | --- |
| a. | I only |
| b. | II only |
| c. | II and III only |
| d. | I and II only |
| e. | I and III only |

39. Consider the following instance variable and incomplete method.

The method findInsertLocation should return the position in the ArrayList where

the new word should be added in order to maintain the sorted ascending order of the list.

private ArrayList<String> list;

public int findInsertLocation( String word )

{

int loc = 0;

**/\* code \*/**

return loc;

}

Which of the following code segments shown below could be used to replace

**/\* code \*/** so that findInsertLocation will work as intended?

I.

for( String s : list )

{

if( s > word )

return loc;

loc++;

}

II.

for( String s : list )

{

if( s.compareTo( word ) > 0 )

return loc;

loc++;

}

III.

for( String s : list )

{

if( s.compareTo( word ) < 0 )

return loc;

loc++;

}

|  |  |
| --- | --- |
| a. | I only |
| b. | II only |
| c. | II and III only |
| d. | I and II only |
| e. | I and III only |

40. Consider the following instance variable and incomplete method.

The method findInsertLocation should return the position in the ArrayList where

the new word should be added in order to maintain the sorted descending order of the list.

private ArrayList<String> list;

public int findInsertLocation( String word )

{

int loc = 0;

**/\* code \*/**

return loc;

}

Which of the following code segments shown below could be used to replace

**/\* code \*/** so that findInsertLocation will work as intended?

I.

for( String s : list )

{

if( s > word )

return loc;

loc++;

}

II.

for( String s : list )

{

if( s.compareTo( word ) > 0 )

return loc;

loc++;

}

III.

for( String s : list ){

if( s.compareTo( word ) < 0 )

return loc;

loc++;

}

|  |  |
| --- | --- |
| a. | I only |
| b. | II only |
| c. | III only |
| d. | I and II only |
| e. | I and III only |