Submitted by: Anup Dahal

**Q. Explain Divide and conquer algorithm taking reference to merge sort**

Ans: Divide and conquer is a powerful tool for solving conceptually difficult problems:

1. All it requires is a way of breaking the problem into sub-problems of solving the trivial cases and of combining sub-problems to the original problem.
2. Similarly, decrease and conquer only requires reducing the problem to a single smaller problem, such as the classic Tower of Hanoi puzzle, which reduces moving a tower of height n to moving a tower of height n – 1 .

Divide and conquer algorithms can also be implemented by a non-recursive program that stores the partial sub-problems in some explicit data structure, such as a stack, queue, or priority queue. This approach allows more freedom in the choice of the sub-problem that is to be solved next, a feature that is important in some applications — e.g. in breadth-first recursion and the branch and bound method for function optimization. This approach is also the standard solution in programming languages that do not provide support for recursive procedures.

**Q. Write some paragraphs about External Sorting with some examples**.

Ans: External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. It is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

It typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into runs such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

**Q. Write down your analysis on complexities of each algorithm we’ve discussed in the class.**

**1. Bubble sort**: Bubble sort repetitively compares adjacent pairs of elements and swaps if necessary. Scan the array, swapping adjacent pair of elements if they are not in relative order. This bubbles up the largest element to the end. Scan the array again, bubbling up the second largest element. Repeat until all elements are in order.

The following bubbleSort() method implements bubble sort. It uses a nested loop to repetitively swap elements and bubble up the largest elements one by one.

public static void bubbleSort (int[] data)

{

 for (int i = data.length - 1; i >= 0; i--)

 {        // bubble up

 for (int j = 0; j <= i - 1; j++)

   {            if (data[j] > data[j + 1])

 swap(data, j, j + 1);

  }

 }

}

**2. Selection sort:** Selection sort is to repetitively pick up the smallest element and put it into the right position:

i) Find the smallest element, and put it to the first position.

ii) Find the next smallest element, and put it to the second position.

iii) Repeat until all elements are in the right positions.

A loop through the array finds the smallest element easily. After the smallest element is put in the first position, it is fixed and then we can deal with the rest of the array.

public static void selectionSort(int[] arr)

{    // find the smallest element starting from position i

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

 {        int min = i;  // record the position of the smallest

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

    {            // update min when finding a smaller element

    if (arr[j] < arr[min])                min = j;

 }    // put the smallest element at position i

 swap(arr, i, min);

}

}

public static void swap (int[] arr, int i, int j)

{    int temp = arr[i];

   arr[i] = arr[j];

  arr[j] = temp;

}

**3. Insertion sort:** Insertion sort maintains a sorted sub-array, and repetitively inserts new elements into it. The process is as following:

i) Take the first element as a sorted sub-array.

ii) Insert the second element into the sorted sub-array (shift elements if needed).

iii) Insert the third element into the sorted sub-array.

iv) Repeat until all elements are inserted.

The following insertionSort()method implements insertion sort. It uses a nested loop to repetitively insert elements into the sorted sub-array.

public static void insertionSort(int[] arr)

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

    {        // a temporary copy of the current element

 int tmp = arr[i];

 int j;         // find the position for insertion

   for (j = i; j > 0; j--)

   {            if (arr[j - 1] < tmp)

  break;            // shift the sorted part to right

  arr[j] = arr[j - 1];

 }         // insert the current element

 arr[j] = tmp;

 }

}

**4. Merge sort:** Merge sort is a sorting technique based on divide and conquers technique. With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

To understand merge sort, we take an unsorted array as depicted below:



We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



.We further divide these arrays and we achieve atomic value which can no more be divided. Now, we combine them in exactly same manner they were broken down. We first compare the element for each list and then combine them into another list in sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order 19 and 35. 42 and 44 are placed sequentially.



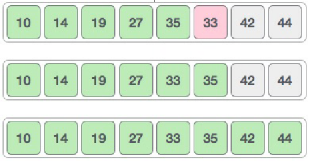
In next iteration of combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in sorted order.

After final merging, the list should look like this –



Now we should learn some programming aspects of merge sorting.





**Q. Explain partition strategies of merge sort and quick sort.**

**Partition strategy of Merge sort:**

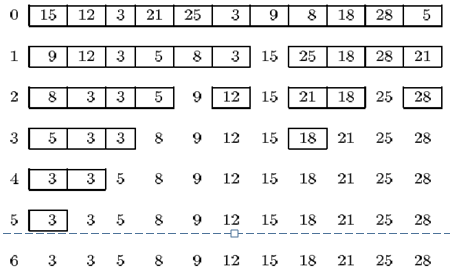
* Split array A[1..n] in two and make copies of each half in arrays B[1.. n/2 ] and C[1.. n/2 ]
* Sort arrays B and C
* Merge sorted arrays B and C into array A as follows:
* Repeat the following until no elements remain in one of the arrays:
  + compare the first elements in the remaining unprocessed portions of the arrays
  + copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array

Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array and

* Merge sort always partitions the array equally. Thus, the recursive depth is always O(log n) .
* The amount of work done at each level is O(n) .
* Intuitively, the complexity should be O(n logn).
* The amount of extra memory used is O(n).

**Partition strategy of Quick sort:**

* Select a pivot (partitioning element).
* Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than or equal to the pivot .
* Exchange the pivot with the last element in the first (i.e., ≤) sub-list – the pivot is now in its final position.
* Sort the two sub lists recursively.
* Recursive implementation with the left most array entry selected as the pivot element.



**Q. Why is merge-sort preferred over quick sort for sorting linked list?**

Ans: Merge sort works better than quick sort when data cannot be loaded to memory at once. Quicksort uses worst-case O(log n) memory (if implemented correctly), while merge sort requires O(n) memory due to the overhead of merging. Merge sort is a superior algorithm for sorting linked lists, since it makes fewer total comparisons and isn't susceptible to a poor pivot choice. In addition, merge sort is ordinarily an in-place sort, useful when sorting by column headers.

Merge sort has a guaranteed upper limit of O(N log2N). Quick sort has such limit, too, but it is much higher - it is O(N2). When you need a guaranteed upper bound on the timing of your code, use merge sort over quick sort. Most important advantage of merge sort over quick sort is its stability: the elements compared equal retain their original order. Merge Sort uses (about 30%) fewer comparisons than Quicksort. Thus, merge sort is preferred over quick sort for sorting linked list.

**Write something about binary tree sort and analyze the algorithm. WAP a program and submit repo link.**

**BINARY SEARCH TREE**: A binary tree is a rooted tree that is also an ordered tree in which every node has at most two children. A rooted tree naturally imparts a notion of levels (distance from the root), thus for every node a notion of children may be defined as the nodes connected to it a level below.

In simple words, binary search tree (BST) is a tree in which all nodes follow the below mentioned properties:

* The left sub-tree of a node has key less than or equal to its parent node's key.
* The right sub-tree of a node has key greater than or equal to its parent node's key.

Thus, a binary search tree (BST) divides all its sub-trees into two segments; left sub-tree and right sub-tree.

**Program for the search operation:**

struct node\* search(int data)

{ struct node \*current = root;

printf("Visiting elements: ");

while(current->data != data)

{ if(current != NULL)

{ printf("%d ",current->data); //go to left tree

if(current->data > data)

{ current = current->leftChild;

} //else go to right tree

else

{ current = current->rightChild;

} //not found

if(current == NULL)

{

return NULL;

}

}

}

return current;

}

**Program for the insertion operation:**

void insert(int data)

{ struct node \*tempNode = (struct node\*)

malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL; //if tree is empty

if(root == NULL)

{ root = tempNode;

}

else

{ current = root; parent = NULL;

while(1)

{ parent = current; //go to left of the tree

if(data < parent->data)

{ current = current->leftChild; //insert to the left if(current == NULL)

{ parent->leftChild = tempNode;

return;

}

}//go to right of the tree

else

{ current = current->rightChild; //insert to the right

if(current == NULL)

{ parent->rightChild = tempNode;

return;

}

}

}

}

}