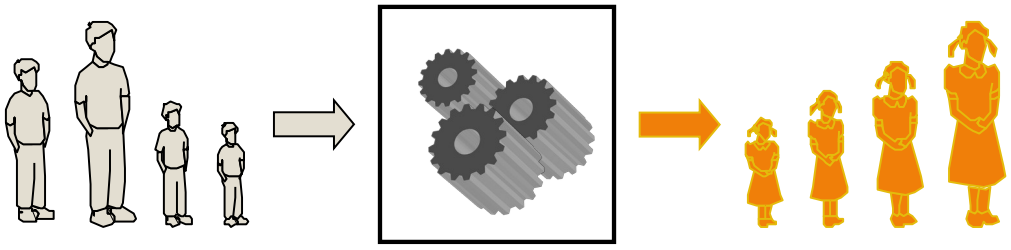
Design and Analysis of Algorithms

Lecture 02: Simple Searching and Sorting Algorithms



2.1. Searching

- Searching is a process of looking for a specific element in a list of items or determining that the item is not in the list.

- There are two simple searching algorithms:

  - Sequential Search, and
  - Binary Search

Linear Search (Sequential Search)

- A linear search looks down a list, one item at a time, without jumping.

- In complexity terms this is an O(n) search - the time taken to search the list gets bigger at the same rate as the list does.

Algorithm

- Loop through the array starting at the first element until the value of target matches one of the array elements.

- If a match is not found, return –1.

- Time is proportional to the size of input (n) and we call this time complexity O(n).

Linear Search (Sequential Search)

- **Example Implementation:**

```
int Linear_Search(int list[], int key)
{
int index=0;
int found=0;
do{
if(key==list[index])
     found=1;
else
    index++;
}while(found==0&&index<n);
if(found==0)
     index=-1;
return index;
}
```

Binary Search

- **searching start with the middle of a sorted list, and see whether that's greater than or less than the value you're looking for, which determines whether the value is in the first or second half of the list.**

- **Jump to the half way through the sub list, and compare again etc.**

- **This is pretty much how humans typically look up a word in a dictionary (although we use better heuristics, obviously - if you're looking for "cat" you don't start off at "M").**

- **In complexity terms this is an O(log n) search - the number of search operations grows more slowly than the list does, because you're halving the "search space" with each operation.**

- **This searching algorithm works only for an ordered list.**

Binary Search

The basic idea is:

- **Locate midpoint of array to search**

- **Determine if target is in lower half or upper half of an array.**
  - **If in lower half, make this half the array to search**
  - **If in the upper half, make this half the array to search**

- **Loop back to step 1 until the size of the array to search is one, and this element does not match, in which case return –1.**

- **The computational time for this algorithm is proportional to log2 n. Therefore the time complexity is O(log n)**

Example

Example Implementation:

```
int Binary_Search(int list[],int k)
{
int left=0, right=n-1, found=0;
do
{
mid=(left+right)/2;
if(key==list[mid])
                found=1;
else
{
                        if(key<list[mid])
                                right=mid-1;
                        else
left=mid+1;
              }
}
while(found==0&&left<right);
if(found==0)
  index=-1;
else
  index=mid;
return index;
}
```

Comparing the two:

- **Binary search requires the input data to be sorted; linear search doesn't**
- **Binary search requires an ordering comparison; linear search only requires equality comparisons**
- **Binary search has complexity O(log n); linear search has complexity O(n) as discussed earlier**
- **Binary search requires random access to the data; linear search only requires sequential access (this can be very important - it means a linear search can stream data of arbitrary size)**

Sorting Algorithms

- **Sorting is a process of reordering a list of items in either increasing or decreasing order.**
- **The following are simple sorting algorithms used to sort small-sized lists.**
- **Types:**
  - Insertion Sort
  - Selection Sort
  - Bubble Sort

Insertion Sort

- **inserts each item into its proper place in the final list.**
- **requires two list structures - the source list and the list into which sorted items are inserted.**

Insertion Sort

- **The process involved in insertion sort is as follows:**
  - **The left most value can be said to be sorted relative to itself. Thus, we don’t need to do anything.**
  - **Check to see if the second value is smaller than the first one. If it is, swap these two values. The first two values are now relatively sorted.**
  - **Next, we need to insert the third value in to the relatively sorted portion so that after insertion, the portion will still be relatively sorted.**
  - **Remove the third value first. Slide the second value to make room for insertion. Insert the value in the appropriate position.**
  - **Now the first three are relatively sorted.**
  - **Do the same for the remaining items in the list.**

Insertion Sort

- **Example: for sorting the array the array 52314 First, 2 is inserted before 5, resulting in 25314 Then, 3 is inserted between 2 and 5, resulting in 23514 Next, one is inserted at the start, 12354 Finally, 4 is inserted between 3 and 5, 12345**

Implementation

```
void insertion_sort(int list[])
{
int key;
for(int i=1;i<n;i++){
                    key=list[i];
        for(int j=i; j>=0;j--)
            {
 // work backwards through the array finding where temp should go
if(key<list[j])
{
                                                    list[j+1]=list[j];
                                    list[j]=temp;
                                   }
                        }//end of inner loop
                    }//end of outer loop
}//end of insertion_sort
```

Selection Sort

- **Basic Idea:**
  - **Loop through the array from i=0 to n-1.**
  - **Select the smallest element in the array from i to n**
  - **Swap this value with value at position i.**

- **Selection sort is the most conceptually simple of all the sorting algorithms.**

- **It works by selecting the smallest (or largest, if you want to sort from big to small) element of the array and placing it at the head of the array.**

- **Then the process is repeated for the remainder of the array; the next largest element is selected and put into the next slot, and so on down the line.**

Selection Sort

- **Because a selection sort looks at progressively smaller parts of the array each time (as it knows to ignore the front of the array because it is already in order), a selection sort is slightly faster than bubble sort, and can be better than a modified bubble sort.**

- **Implementation:**

```
{           int index_of_min = x;
for(int y=x; y<n; y++)
{if(array[index_of_min]<array[y])
{
index_of_min = y;
}
}
int temp = array[x];
array[x] = array[index_of_min];
array[index_of_min] = temp;
}
```

Bubble Sort

- **Bubble sort is the simplest algorithm to implement and the slowest algorithm on very large inputs.**
- **The simplest sorting algorithm is bubble sort.**
- **The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary.**
- **This process is repeated as many times as necessary, until the array is sorted.**

Bubble Sort

- **Since the worst case scenario is that the array is in reverse order, and that the first element in sorted array is the last element in the starting array, the most exchanges that will be necessary is equal to the length of the array.**

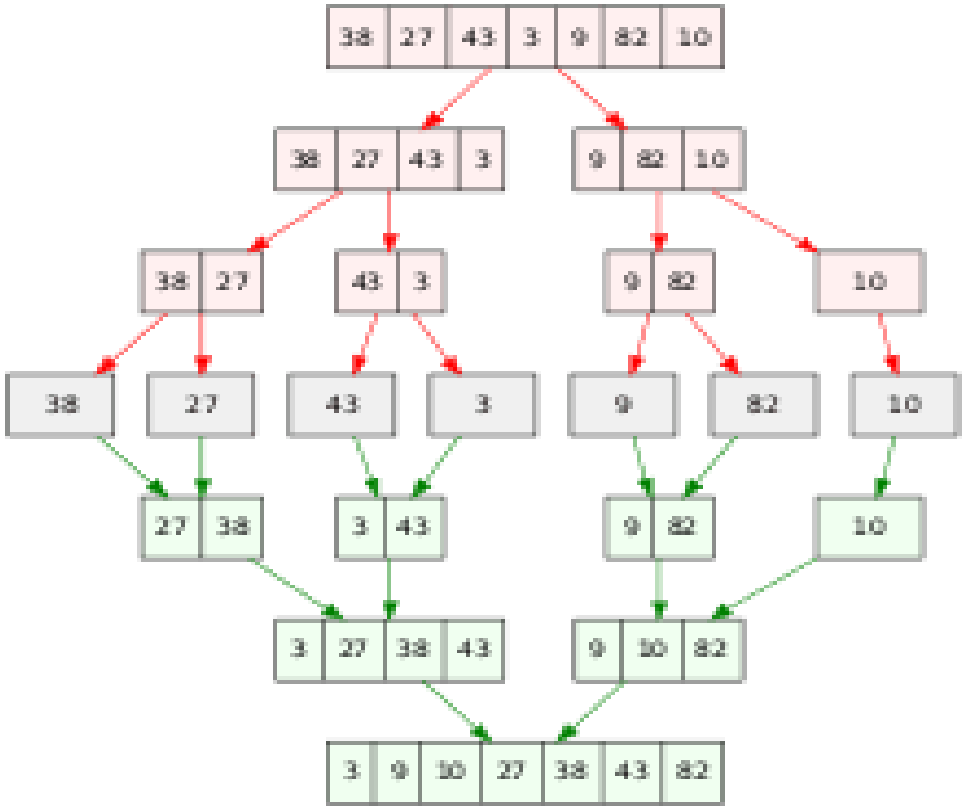
Bubble Sort

- **example:**
- **Given an array 23154 a bubble sort would lead to the following sequence of partially sorted arrays: 21354, 21345, 12345. First the 1 and 3 would be compared and switched, then the 4 and 5. On the next pass, the 1 and 2 would switch, and the array would be in order.**

- **Implementation:**

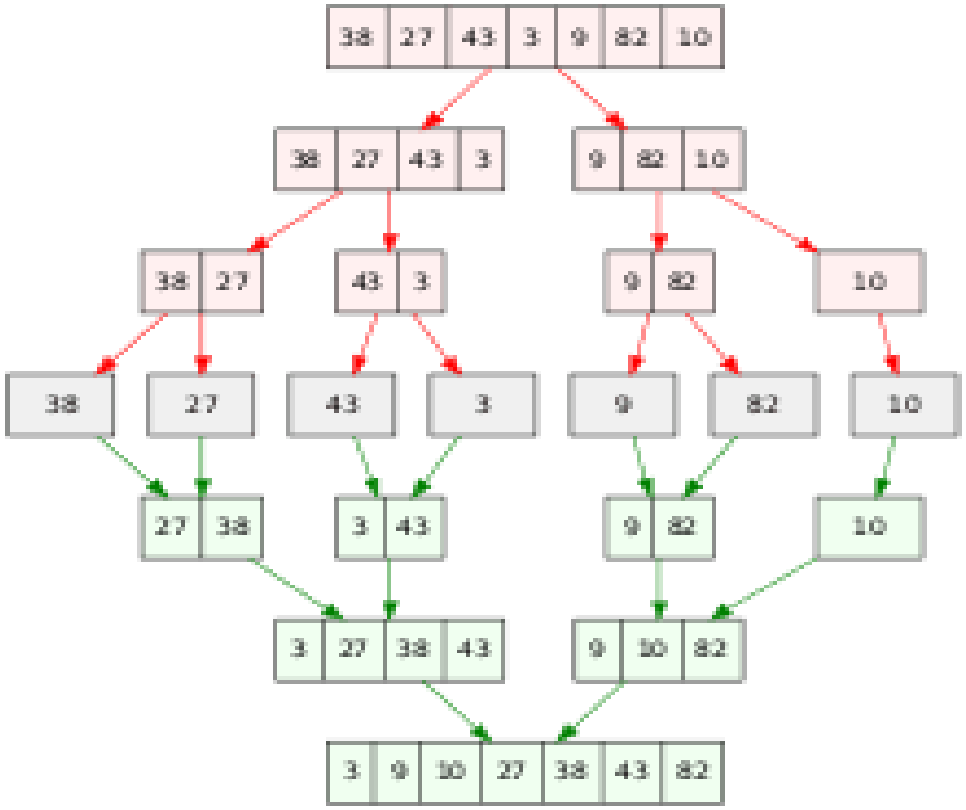
```
for(int x=0; x<n; x++)
        {for(int y=0; y<n-1; y++)
                {
                        if(array[y]>array[y+1])
                        {
                                int temp = array[y+1];
                                array[y+1] = array[y];
                                array[y] = temp;
                }
                }
        }
```

A divide and conquer



- is a strategy of solving a large problem by:
  1) breaking the problem into smaller sub-problems
  2) solving the sub-problems, and
  3) combining them to get the desired output.

A divide and conquer



How **Divide and Conquer Algorithms Work?**

- **Divide**: Divide the given problem into sub-problems using recursion.
- **Conquer**: Solve the smaller sub-problems recursively. If the sub problem is small enough, then solve it directly.

- **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

A divide and conquer

Merge Sort, Quick Sort, and Binary Search follows the strategy of Divide and Conquer algorithm

Reading Assignment: **Merge Sort and Quick Sort**

Merge Sort

The merge sort algorithm divides the given array into two halves (N/2).

And then, it recursively divides the set of two halves array elements into the single or individual elements or we can say that until no more division can take place.

After that, it compares the corresponding element to sort the element and finally, all sub elements are combined to form the final sorted elements.

Merge Sort

Steps to sort an array using the Merge sort algorithm:

- Suppose we have a given array, then first we need to divide the array into sub array.
- gave the first sub array name as A1 and divide into next two subarray as B1 and B2.
- Similarly, the right sub array name as A2 and divide it into next two sub array as B3 and B4.

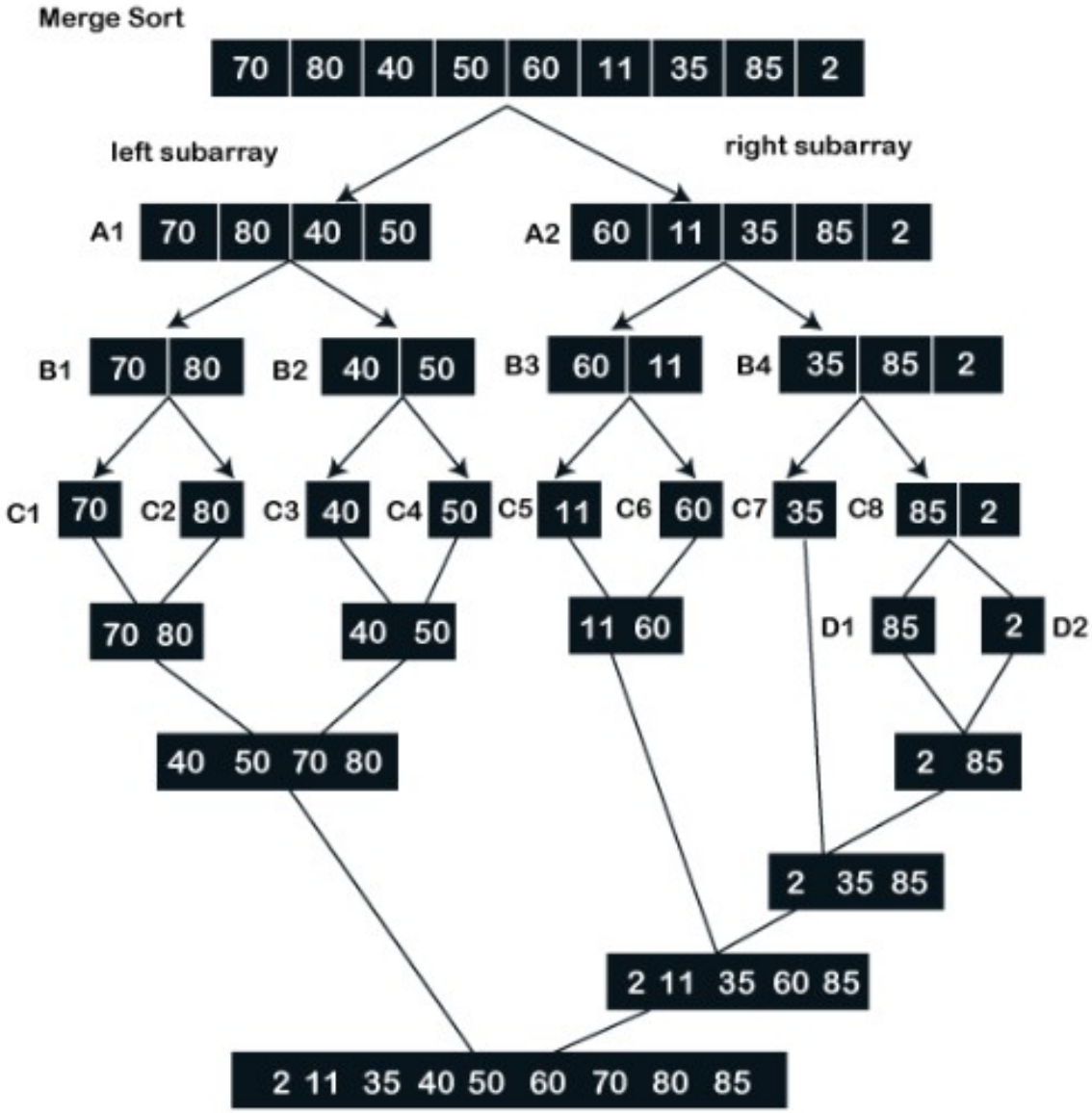
Merge Sort

- This process is repeated continuously until the sub array is divided into a single element and no more partitions may be possible.
- After that, compare each element with the corresponding one and then start the process of merging to arrange each element in such a way that they are placed in ascending order.
- The merging process continues until all the elements are merged in ascending order.

Merge Sort

Example: Consider an array of 9 elements. Sort the array using the merge sort.

arr[] = {70, 80, 40, 50, 60, 11, 35, 85, 2}



Quick Sort---Reading Asmt and its complexity analysis

Quicksort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

There are many different versions of quicksort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot
- Pick a random element as pivot.
- Pick median as pivot.

Thank You