Design and Analysis of Algorithms

Module I: Part B: Merge Sort, Insertion Sort, Quick Sort, Selection

Prof. PV Nageswara Rao
Department of Computer Science and Engineering
GITAM Institute of Technology, GITAM, VISAKHAPATNAM

Merging: 2 way merge

Merging Two sorted lists:

Array A : m =6 elements

Index	1	2	3	4	5	6	7	8	9
Element	7	9	11	21	23	44			

Array B : n= 9 elements

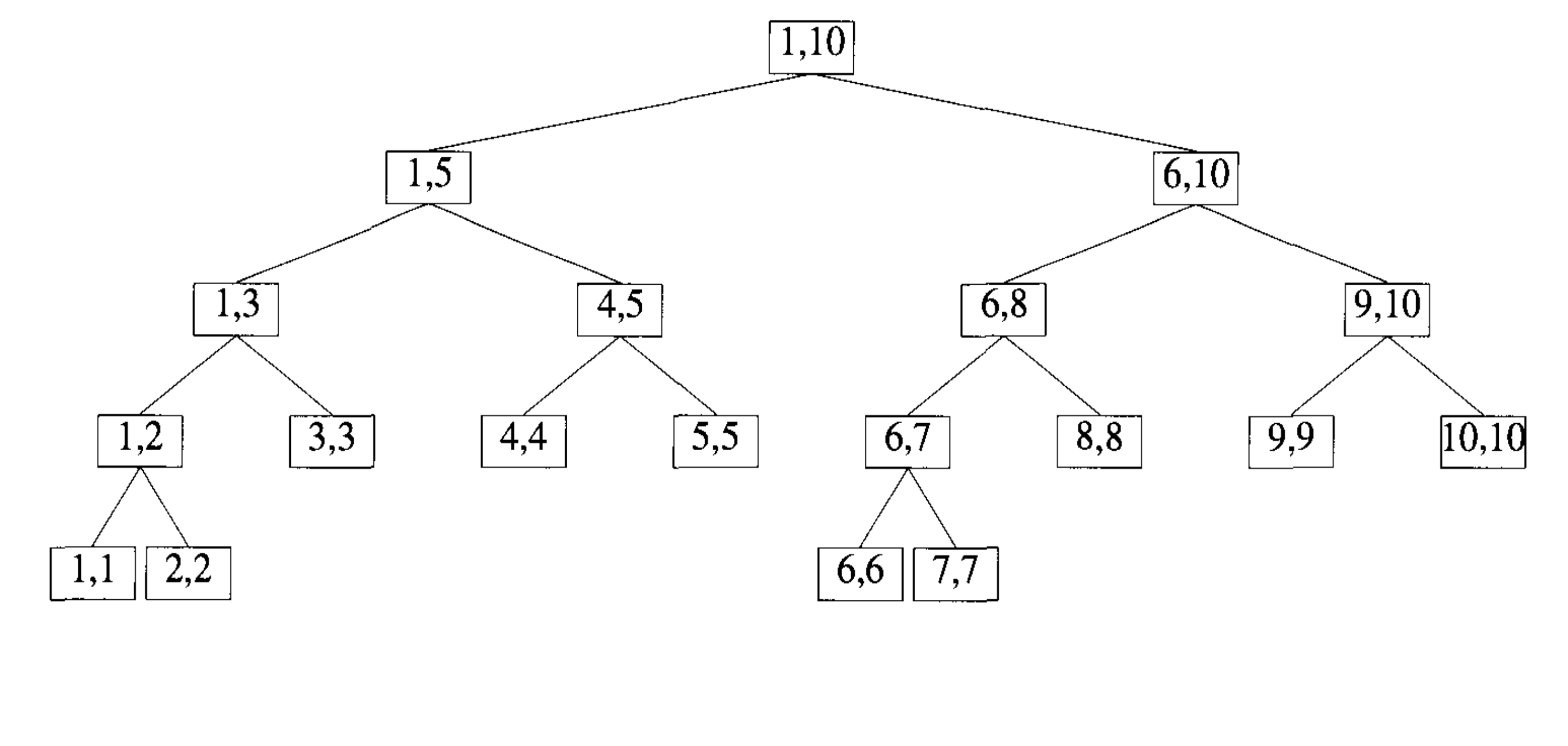
Index	1	2	3	4	5	6	7	8	9
Element	4	6	10	19	22	32	77	79	81

Array C:
p= m+n=6+9=15 elements

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Element	4	6	7	9	10	11	19	21	22	23	32	44	77	79	81	

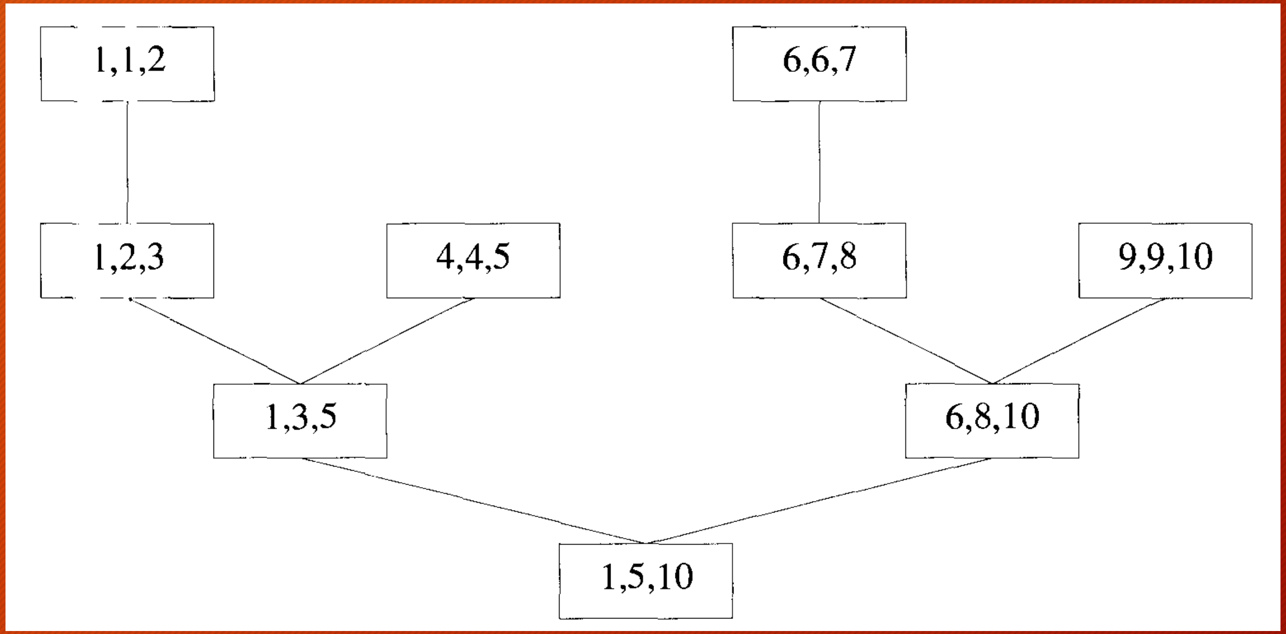
Algorithms: Merge Sort, Merge

```
Algorithm MergeSort(low, high)
// a[low : high]is a global array to be sorted.
// Small(P)is true if there is only one element
 // to sort.  In this case the list is already sorted.
{  if (low< high) then //there are more than one element
       {
           // Divide P into sub problems.
           // Find where to split the set.
         mid:=(low + high)/2;
          // Solve the sub-problems.
          MergeSort(low, mid);
          MergeSort(mid+ 1, high);
          // Combine the solutions.
          Merge(low, mid, high);
           } // end of if
   }// end of MergeSort
```



Algorithm: Merge

```
Algorithm Merge(low, mid, high)
// a[low, high] is a global array containing two
//sorted subsets in a[low : mid] and
// a[mid+1 : high].  The goal is to merge these
//two sets into a single set residing
// in a[low : high].  b[ ]is an auxiliary global array.
{ h:= low;  i:= low; j := mid+1;
  while ((h<=mid) and (j<=high) do
    { if a[h] <= a[j] then { b[i] := a[h] ; h++;}
       else {b[i]:=a[j]; j++;}
       i++;
     } // end of while
if (h>mid) then for k:= j to high do { b[i]:=a[k]; i++}
else for k:= h to mid do { b[i] := a[k]; i++}
for k:= low to high do a[k]:= b[k]
}
```



Merge Sort: Step by Step

Index:

1	2	3	4	5	6	7	8	9	10
310	285	179	652	35	423	861	254	450	520
310	285	179	652	35	423	861	254	450	520
310	285	179	652	35	423	861	254	450	520
310	285	179	652	35	423	861	254	450	520
285	310								
285	310	179	652	35	423	861	254	450	520
179	285	310							
179	285	310	652	35	423	861	254	450	520
			35	652					
179	285	310	35	652	423	861	254	450	520
35	179	285	310	652					
35	179	285	310	652	423	861	254	450	520

Merge Sort: Step by Step

35	179	285	310	652	423	861	254	450	520
35	179	285	310	652	423	861	254	450	520
35	179	285	310	652	423	861	254	450	520
					423	861			
35	179	285	310	652	423	861	254	450	520
					254	423	861		
35	179	285	310	652	254	423	861	450	520
35	179	285	310	652	254	423	861	450	520
								450	520
35	179	285	310	652	254	423	861	450	520
					254	423	450	520	861
35	179	285	310	652	254	423	450	520	861
35	179	254	285	310	423	450	520	652	861
35	179	254	285	310	423	450	520	652	861

Space requirement of Merge Sort

- Additional n locations are required to merge the two sorted portions of max size n/2 and n/2 i.e., in the last phase of exiting from the recursion.
- Every time merged elements from the auxiliary array should be copied back to the original array.

- Stable Sorting:
- A sorting method is said to be stable if at the end of the method, identical elements occur in the same order as in the original unsorted set.

- Is Merge Sort a Stable Sorting Method?
- Is Bubble Sort a Stable Sorting Method?

Time Complexity:

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\ \vdots \\ &= 2^k T(1) + kcn \\ &= an + cn\log n \end{aligned}$$

It is easy to see that if $2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$

Algorithm: Mergesort with Internal Insertion Sort

```
Algorithm MergeSort1(low, high)
// The global array a[low : high]is sorted in non-decreasing order
// using the auxiliary array link[low : high]. The values in link
// represent a list of the indices low through high giving a[ ] in
// sorted order.  A pointer to the beginning of the list is returned.
{  if {(high-low) < 15)then
        return InsertionSort(a, link, low, high);
   else
     {mid := (low+high)/2;
       q := MergeSort1(low, mid);
       r := MergeSort1(mid+1, high);
       return Mergel(q, r);
      }
}
```

Insertion Sort:

// 1st element is already in sorted order

for j :=2 to n do {

Place a[j] in its correct position in the sorted set a[l:j-1];

}

Algorithm: Insertion Sort

```
Algorithm InsertionSort(a, n)
// Sort the array a[1 : n] into
non-decreasing order, n >= 1.
{   for j := 2 to n do
      { // a[1: j-1] is already sorted.
          item := a[j]; i := j -1;
          while ((i >= 1) and (item<a[i]))do
           { a[i+ 1]:=a[i]; i :=i - 1;
             }
          a[i+ 1] := item;
         } //end of for
  }  // end of algorithm
```

The statements within the while loop can be executed zero up to a maximum of j times. Since j goes from 2 to n, the worst-case time of this Procedure is bounded by

$$\sum_{2 \le j \le n} j = n(n+1)/2 - 1 = \Theta(n^2)$$

Its best-case computing time is 0(n) under the assumption that the body of the while loop is never entered. This will be true when the data is already in sorted order

Quick Sort

11

Quick Sort : Partitioning Example: 1st Iteration

Pivot	Left							Right
54	26	93	17	77	31	44	55	20

Initial List

Pivot		Left						Right
54	26	93	17	77	31	44	55	20

Compared left with pivot till arr[left] < pivot and right with pivot till arr[right] > pivot.

Pivot		Left						Right
54	26	20	17	77	31	44	55	93

Exchanged left and right elements

Pivot				Left		Right		
54	26	20	17	77	31	44	55	93

Compared left with pivot till arr[left] < pivot and right with pivot till arr[right] > pivot.

Pivot				Left		Right		
54	26	20	17	44	31	77	55	93

Exchanged left and right elements

					Right	Left		
54	26	20	17	44	31	77	55	93

Right and left crossed over so stop. The right index is the split point and is the place where pivot element should be placed and we need to swap pivot with element at right.

31	26	20	17	44	54	77	55	93

List after swapping

Algorithm: QuickSort

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], …., a[q] which reside in the global array a[1:n] into
// ascending order; a[n+1] is considered to be defined and must be >= all the
// elements in a[1:n]
{ if (p<q) then
    { j := partition(a,p, q+1);   // divide P into two subproblems; j is partitioning element
      // Solve the subproblems
      QuickSort(p, j-1)
       QuickSort(j+1, q)
       // There is no need for combining solutions
      }  end of if
  } // end of algorithm
```

Algorithms: partition and interchange

Algorithm partition(a, m, p)

// within a[m], ……a[p-1] the elements are rearranged in such a manner that if

// initially t=a[m], then after completion a[q]=t for some q between m and p-1,

// a[k]<=t for m<=k<q and a[k] >=t for q<k,p. q is returned. Set a[p]=∞

```
{ v:=a[m]; i:=m; j:=p;
   do {  do   i:=i+1;  while (a[i] < v);
         do   j:=j-1;   while a[j] > v);
              if (i<j) then interchange(a,i,j);
             } while (i<j)
   a[m]:=a[j]; a[j]:= v; return j;
```

Algorithm interchange(a,i,j)

// Exchange a[i] with a[j]

{p:=a[i]; a[i]:=a[j]; a[j]:=p; }

Time Complexity: Worst Case

- Consider the element comparisons only C(n)
- Assume n elements are distinct and the input distribution is such that the partitioning element v=a[m] in the call Partition(a, m, p) has an equal probability of being the i^{th} smallest element $1 \le i \le p-m$ in a[m:p-1]
- Worst Case Complexity: $C_W(n)$ of C(n)
- Number of element comparisons in each call of partition is at most : p- m +1
- Let r be the total number of elements in all the calls to Partition at any level of recursion.
- At level one only one call : Partition(a, 1, n+1) is made and r=n;
- At level two at most two calls are made r=n-1 and so on ..
- At each level of recursion O(n) element comparisons are made.
- $C_W(n)$ = sum of r, varies from 2 to n or $O(n^2)$
- Therefore, Quick sort uses $\Omega(n^2)$ comparisons

Average Time Complexity:

- Average Case: $C_A(n)$ of $C(n)$
- The partitioning element r has an equal probability of being the i^{th} smallest element $1 \le i \le p-m$ in a[m:p-1]
- Hence the two sub arrays remaining to be sorted are a[m:j] and a[j+1:p-1]

  with probability $1/(p-m)$, $m \le j < p$
- From this obtain the following recurrence:
- $C_A(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n} C_A\,(k-1) + C_A(n-k))$ ------ Eq. 1
- The number of element comparisons required by Partition on its first call is n+1
- $C_A(0)=C_A(1)=0$
- Multiplying Eq.1 by n
- $nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots. + C_A(n-1)$ --- Eq. 2

Time Complexity: Average Case Contd…

Replacing n by n-1 in Eq.2

$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + C_A(1) + \cdots .. + C_A(n-2)]$ ---- Eq.3

Eq. 3 – Eq.2 :

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

Or

$$nC_A(n) = (n+1)C_A(n-1) + 2n$$

Multiplying by 1/(n*(n+1))

$$\frac{C_A(n)}{n+1} = \frac{C_A(n-1)}{n} + \frac{2}{n+1}$$

Repeatedly using this equation to substitute for CA(n-1), CA(n-2) … we get:

$$\frac{C_A(n)}{n+1} = \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\frac{C_A(n)}{n+1} = \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

..

$$= 2\sum_{k=3}^{n+1}\frac{1}{k}$$

$$\sum_{k=3}^{n+1}\frac{1}{k} \le \int_{2}^{n+1}\frac{1}{x}dx = log_e(n+1) - log_e 2$$

$C_A(n) \le 2(n+1)[log_e(n+2) - log_e 2 = O(n \log n)$

The average time is O(n log n)

Space requirement: Stack Space

Maximum depth of recursion in the worst case is n-1

The amount of stack space needed can be reduced to O(log n) by using an iterative version of qucksort in which the smaller of the two subarrays a[p:j-1] and a[j+1:q] is always sorted first.

Maximum stack space needed:

Let S(n) $\leq \begin{cases} 2 + S(\lfloor \frac{n-1}{2} \rfloor & n > 1 \\ 0 & n \leq 1 \end{cases}$

Algorithm Selection – To find the kth smallest element

```
Algorithm Select1(a,n,k)
// Selects the kth smallest element in a[1:n] and places it in the kth position of a[].
// The remaining elements are rearranged such that a[m]<=a[k] for 1<=m<k
//  and a[m]>=a[k] for k<m<=n.
{ low:=1; up:=n+1; a[n+1]= infinity;
   do {
         // each time the loop is entered, 1<=low<=k<=up<=n+1.
         j=Partition(a, low, up);  // j is such that a[j] is the jth smallest value in a[].
         if (k==j) then return a[j];
         else if (k<j) then up=j; // j is the new upper limit
               else low=j+1;       // j+1 is the new lower limit
         } while(true);
}
```