Sorting – 2

Mergesort

Quicksort

Mergesort

The sorting algorithms that we have studied so far have a quadratic complexity. In this section we shall look at two sorting methods which are subquadratic , i.e. the complexity is less than O(n^2).The *mergesort* sorting algorithm uses the divide and conquer strategy in which the original problem is split into two problems, with size about half the size of the original problem.

The basic idea is as follows. Suppose you have got a large number of integers to sort. Write each integer on a separate slip of paper. Make two piles of the slips. So the original problem has been reduced to sorting individually two piles of smaller size. Now reduce each pile to half of the existing size. There would be now 4 piles with a smaller set of integers to sort. Keep on increasing the number of piles by reducing their lengths by half every time. Continue with the process till you have got piles with maximum two slips in each pile. Sort the slips with smaller number on the top. Now take adjacent piles and sort them in the same manner. So now the number of piles go on reducing but piles are now sorted. Stop when all piles have been taken care of and there remains one single pile.

Thus the mergesort can be thought of as a recursive process:

mergesort

1. if the number of items to sort is 0 or 1, return.
2. recursively mergesort the first and second halves separately.
3. merge the two sorted halves into a single sorted group.

What would be the complexity of the process?
Since this algorithm uses the divide and conquer strategy and employs the halving principle, we can guess that the sorting process would have $O(\log_2 n)$ complexity. However, the merging operation would involve movement of all the n elements (linear time), and we shall show later that the overall complexity turns out to be $O(N \log_2 N)$.

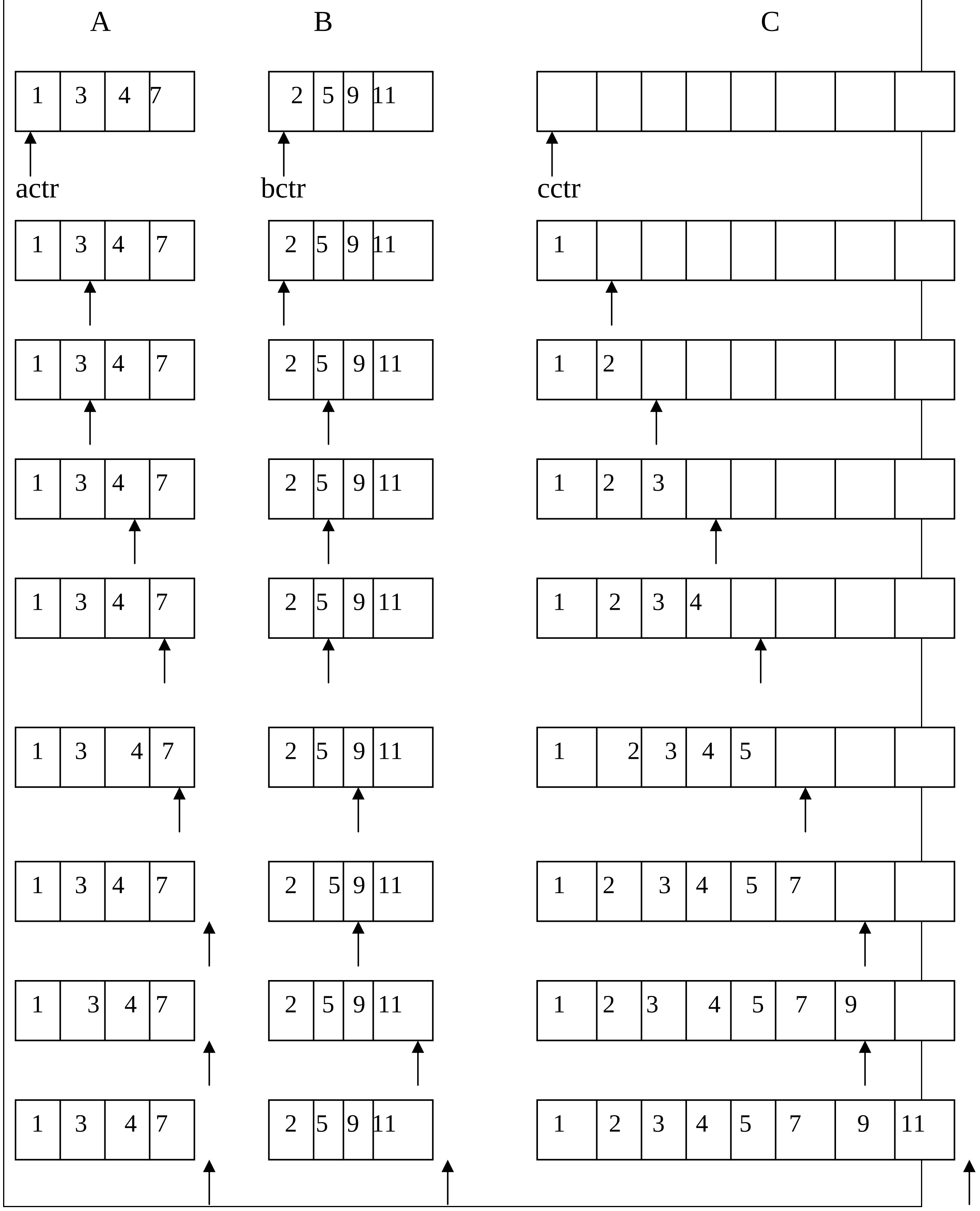
We can merge two input arrays A and B to result in a third array C. Let the index counter for the respective arrays be actr, bctr, and cctr. The index counters are initially set to the position of the first element. The smaller of the two elements A[actr] and B[bctr] is stored in C[cctr] as shown below:

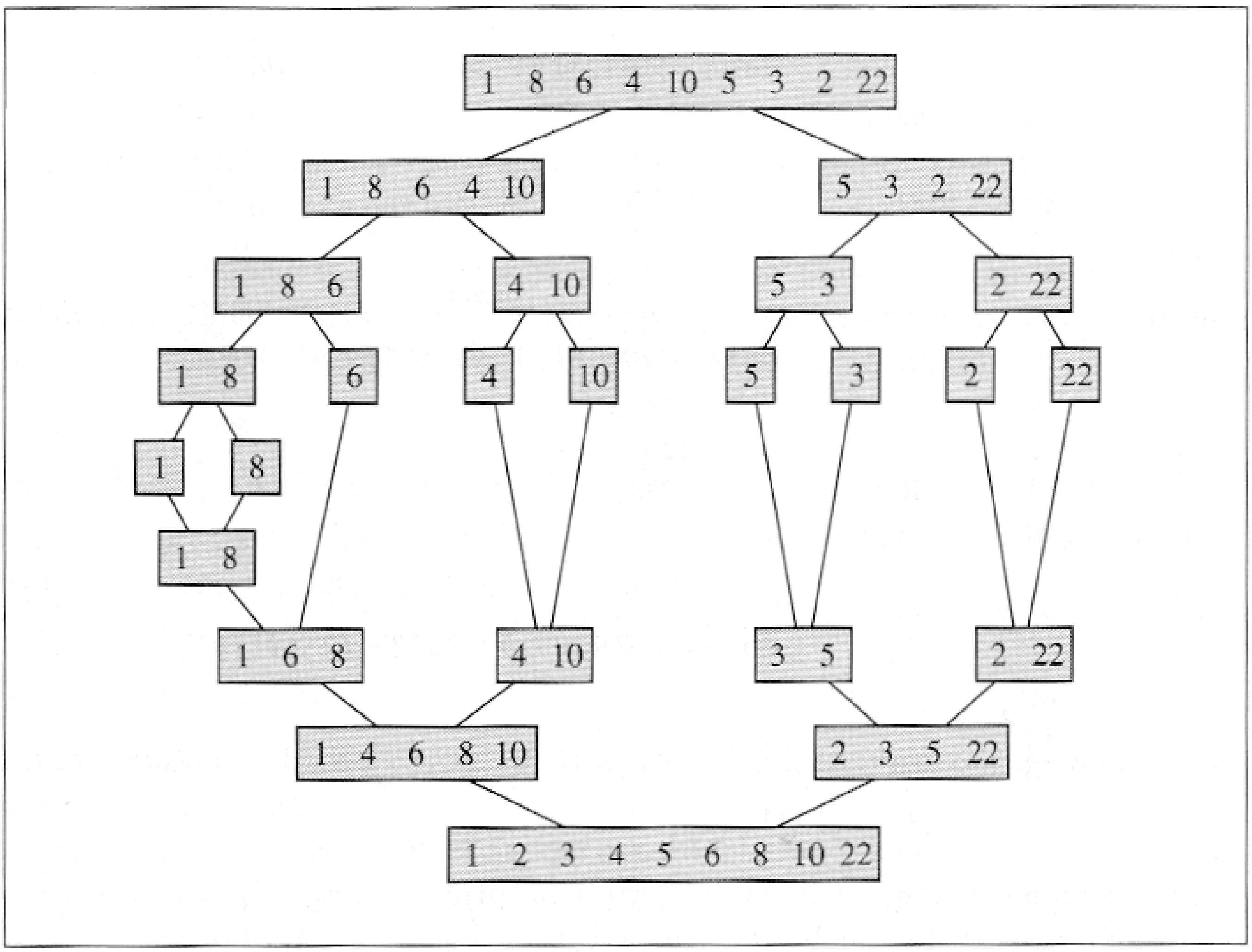
```
if A[actr] < B[bctr]
        C[cctr] = A[actr];
        cctr++;
        actr++;
} else      {
        C[cctr] = B[bctr];
        cctr++;
        bctr++;
    }
```

Let us take an example. Say at some point in the sorting process we have to merge two lists 1, 3, 4, 7 and2,5, 9, 11

We store the first set in Array A and the second set in Array B. The merging goes in following fashion:

Example: Linear Merge





Let us now examine the overall mergesort process by considering a specific example:

Example of a mergesort.

In fact, instead of using separate input arrays, we can as well concatenate the two input arrays in array C , and use a temporary array to carry out the operations. The two sub arrays can be identified by means of the indexes for the beginning and end of the subarrays.

The merging code now takes the following form:

```
merge ( list[], first, last)
mid = ( first +  last ) / 2 ;
ia = 0 ;
ib = first ;
ic = mid + 1 ;
while ( ib < mid && ic < last )
   {
      if list[ib] < list [ic ]
          temp [ia++] = list[ib++];
      else
          temp[ia++] = list[ic++];
    }
while ( ib < mid )
    {
       temp [ia++] = list[ib++];
    }
while ( ic < last )
    {
       temp[ia ++] = list[ic++];
    }
for( ia = 0; ia < last ; ia++)
       list[ia] = temp [ia];
```

With this background, we are now in a position to write the pseudocode for mergesort.

```
mergesort ( list, first, last)
```

```
if first < last
   mid = (first + last)/2 ;
mergesort ( list, first, mid);
mergesort ( list, mid+1 , last );
merge ( list, first, last) ;
```

Computational Complexity:

Intuitively we can see that **mergesort** reduces the problem to half its size every time, (done twice), it can be viewed as creating a tree of calls, where each level of recursion is a level in the tree. Effectively, all n elements are processed by the **merge** routine the same number of times as there are levels in the recursion tree. Since the number of elements is divided in half each time, the tree is a balanced binary tee. The height of such a tree tends to be log n.

The **merge** routine steps along the elements in both halves, comparing the elements. For n elements, this operation performs n assignments, using at most n – 1 comparisons, and hence it is O(n). So we may conclude that [log n . O(n)] time merges are performed by the algorithm.

The same conclusion can be drawn more formally using the method of recurrence relations. Let us assume that n is a power of 2, so that we always split into even halves. The time to mergesort n numbers is equal to the time to do two recursive mergesorts of size n/2, plus the time to merge, which is linear.

For n = 1 , the time to mergesort is constant.

We can express all of it through following recurrence relations:

$T(1) = 1$
$T(n) = 2\,T(n/2) + n$

Using same logic and going further down

$T(n/2) = 2\,T(n/4) + n/2$

Substituting for T(n/2) in the equation for T(n),we get

$T(n) = 2[\,2\,T(n/4) + n/2\,] + n$

$\quad = 4\,T(n/4) + 2n$

Again by rewriting T(n/4) in terms of T(n/8), we have

$T(n) = 4\,[\,2\,T(n/8) + n/4\,] + 2n$

$\quad = 8\,T(n/8) + 3\,n$

$\quad = 2^3\,T(n/2^3) + 3\,n$

The next substitution would lead us to

$T(n) = 2^4 T(n/ 2^4) + 4n$

Continuing in this manner, we can write for any k,

$T(n) = 2^k T(n/ 2^k) + kn$

This should be valid for any value of k. Suppose we choose $k = \log n$, i.e. $2^k = n$. Then we get a very neat solution:

$T(n) = n T(1) + n \log n$

$= n \log n + n$

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

This analysis can be refined to handle cases when n is not a power of 2. The answer turns out to be almost identical.

Although mergesort's running time is very attractive, it is hardly ever used for sorting data in main memory. The main problem is that merging two sorted lists uses linear extra memory, and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably.

The copying can be avoided by judiciously switching the roles of list and temp arrays at alternate levels of the recursion. For serious internal sorting applications, the algorithm of choice is Quicksort, which we shall be studying next.

Quick-Sort

Quick-Sort

As the name implies the quicksort is the fastest known sorting algorithm in practice. It has the best average time performance. Like merge sort, Quicksort is also based on the *divide-and-conquer* paradigm.

- But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls.
- It works by partitioning an array into two parts, then sorting the parts independently, and finally combining the sorted subsequences by a simple concatenation.

In particular, the quick-sort algorithm consists of the following three steps:

1. ***Choosing a pivot:***:.
   To partition the list, we first choose some element from the list which is expected to divide the list evenly in two sublists. This element is called a ***pivot***.

2. ***Partitioning:***
Then we partition the elements so that all those with values less than the pivot are placed in one sublist and all those with greater values are placed in the other sublist.

3. Recur:
Recursively sort the sublists separately. Repeat the partition process for both the sublists. Choose again two pivots for the two sublists and make 4 sublists now. Keep on partitioning till there are only one cell arrays that do not need to be sorted at all. By dividing the task of sorting a large array into two simpler tasks and then dividing those tasks into even simpler task, it turns out that in the process of getting prepared to sort , the data have already been sorted. This is the core part of the quicksort.

Example:

```
56    25    37    58    95    19    73    30
      lh                                  rh
```

1. Choose first element 56 as pivot.
2. Move rh index to left until it coincides with lh or points to value smaller than pivot. Here it already points to small value.
3. Move lh index to right until it coincides with rh or points to value equal to or greater than pivot.

```
56    25    37    58    95    19    73    30
                  lh                      rh
```

4. If lh and rh not pointing to same element , exchange the elements.

56	25	37	30	95	19	73	58
			lh				rh

5. Repeat steps 2 to 4 until lh and rh coincide.

56	25	37	30	95	19	73	58
			lh		rh		

56	25	37	30	95	19	73	58
				lh	rh		

56	25	37	30	19	95	73	58
				lh	rh		

56	25	37	30	19	95	73	58
				lh			
				rh			

6. This will be the smallest value. Exchange this with the pivot..

19	25	37	30	56	95	73	58

7. This will result in two sub arrays, one to left of pivot and one to right of pivot. Repeat steps 1 to 6.

19	**25**	**37**	**30**	*56*	**95**	**73**	**58**

The code for the quick sort function can take the following form:

```
//Precondition:A valid integer type array with its
maximum size that has to be sorted using Quick sort
//Postcondition: Sorts the array corresponding to
items
void sorting(int values[] , int n)
{
        int mid;
        if(n < 2 )
        return;

        mid= partitioning( values,  n);
        sorting(values,   mid);
        sorting(values+ mid+1 ,   n-mid-1);
  }


//Precondition: Called by sorting function with two
arrays - the item numbers in values array
//Postcondition: Retruns the partition element to
the sorting function after checking through the
arrays entered.
int partitioning( int values[],int n)
{
        int  pivot,left,right,temp;
        pivot= values[0];
        left=1 ;
        right= n-1;

        while(1)
          {
       while(left<right && values[right]>= pivot)
       right--;
       while(left<right && values[left]< pivot)
                left++;
                if( left == right)
                break;
                temp= values[left];
                values[left]= values[right];
                values[right]=temp;

        }
```

Picking the Pivot:

The algorithm would work, no matter which element is chosen as a pivot. However, some choices are going to be obviously better than the other ones. A popular choice would be to use the first element as a pivot. This may work if the input is random.

But, if the list is presorted or in reverse order, then what would be the result of choosing such a pivot? This may consistently happen throughout the recursive calls and turn the whole process into a quadratic time algorithm.

If the data is presorted, the first element is not chosen as the pivot element. A good strategy would be to take the first value and swap it with the center value, and then choose the first value as a pivot. A better strategy is to take the median of the first, last and the center elements. This works pretty well in many cases.

Analysis of Quicksort:

To do the analysis let us use the recurrence type relation used for analyzing mergesort. We can drop the steps involved in finding the pivot as it involves only constant time.
We can take $T(0) = T(1) = 1$.

The running time of quicksort is equal to the running time of the two recursive calls, plus the linear time spent in the partition. This gives the basic quicksort relation

$T(N)$ the time for Quicksort on array of N elements, can be given by

$$T(N) = T(j) + T(N - j - 1) + N,$$

Where j is the number of elements in the first sublist.

Worst Case Analysis:

The partitions are very lopsided, meaning that either left partition $|L| = 0$ or $N - 1$ or right partition $|R| = N - 1$ or 0, at each recursive step. Suppose that Left contains no elements, Right contains all of the elements except the pivot element (this means that the pivot element is always chosen to be the smallest element in the partition),

1 time unit is required to sort 0 or 1 elements, and N time units are required to partition a set containing N elements.

Then if $N > 1$ we have:

$$T(N) = T(N-1) + N$$

This means that the time required to quicksort N elements is equal to the time required to recursively sort the N-1 elements in the Right subset plus the time required to partition the N elements. By telescoping the above equation we have:

$T(N) = T(N-1) + N$

$T(N-1) = T(N-2) + (N-1)$

$T(N-2) = T(N-3) + (N-2)$

…..
…..

$T(2) = T(1) + 2$

$$T(N) = T(1) + 2 + 3 + 4 + \dots + N$$

$$= N(N+1)/2$$

$= O(N^2)$

Therefore, you never want to select a pivot element that leads to an unbalanced paritioning.

Best case analysis:
The best case analysis assumes that the pivot is always in the middle. To simplify the math, we assume that the two sublists are each exactly half the size of the original. Then we can follow the same analysis as in merge sort and can show that

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

Average Case analysis:

If each partition is equally likely to contain 0, 1, 2, …, N-1 elements, then the average running time of the quicksort algorithm is $O(N \log_2 N)$. More formally this is stated as:

$T(\text{Left})_{\text{average}} = T(\text{Right})_{\text{average}} = [T(0) + T(1) + T(2) + \ldots + T(N-1)]/N$

$T(N)_{\text{average}} = T(\text{Left})_{\text{average}} + T(\text{Right})_{\text{average}} + N$

$= 2[T(\text{Left})_{\text{average}}] + N$

$= 2[[T(0) + T(1) + T(2) + \ldots + T(N-1)]/N] + N$

with manipulation you arrive at:

$T(N)/(N+1) = T(N-1)/N + 2/(N+1)$

Telescoping yields

$T(N)/(N+1) = 2[1 + ½ + 1/3 + \ldots + 1/(N+1)) – 5/2$ which is $O(\log_2 N)$.

Therefore, multiplying both side by N+1 gives: $T(N) = O(N \log_2 N)$