Sorting – 1

Mergesort

Quicksort

The efficiency of handling data can be substantially improved if the data is sorted according to some criteria of order. In a telephone directory we are able to locate a phone number, only because the names are alphabetically ordered. Same thing holds true for listing of directories created by us on the computer. Retrieval of a data item would be very time consuming if we don't follow some order to store book indexes, payrolls, bank accounts, customer records, items inventory records, especially when the number of records is pretty large.

We want to keep information in a sensible order. It could be one of the following schemes:

- alphabetical order
- ascending/descending order
- order according to name, ID, year, department etc.

The aim of sorting algorithms is to organize the available information in an ordered form.

There are dozens of sorting algorithms. The more popular ones are listed below:

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort

As we have been doing throughout the course, we are interested in finding out as to which algorithms are best suited for a particular situation. The efficiency of a sorting algorithm can be worked out by counting the number of comparisons and the number of data movements involved in each of the algorithms. The order of magnitude can vary depending on the initial ordering of data.

How much time does a computer spend on data ordering if the data is already ordered? We often try to compute the data movements, and comparisons for the following three cases:

best case (often, data is already in order),
worst case(sometimes, the data is in reverse order),
and average case(data in random order).

Some sorting methods perform the same operations regardless of the initial ordering of data. Why should we consider both comparisons and data movements?

If simple keys are compared, such as integers or characters, then the comparisons are relatively fast and inexpensive. If strings or arrays of numbers are compared, then the cost of comparisons goes up substantially.

If on the other hand, the data items moved are large, such as structures, then the movement measure may stand out as the determining factor in efficiency considerations. In this section we shall discuss two efficient sorting algorithms – the merge sort and the quick sort procedures.

Mergesort

The *mergesort-sorting* algorithm uses the divide and conquer strategy of solving problems 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 merge them such that resulting pile is in sorted form. The piles would keep growing in size but now this time these are in sorted order. 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. Let us assume that the elements are stored in an array.

mergesort

1. if the number of items to sort is 0 or 1, return.
2. Divide the array into two halves and copy the items in two subarrays.
3. Recursively mergesort the first and second halves separately.
4. Merge the two-sorted halves into a single sorted array.

What would be the complexity of the process?

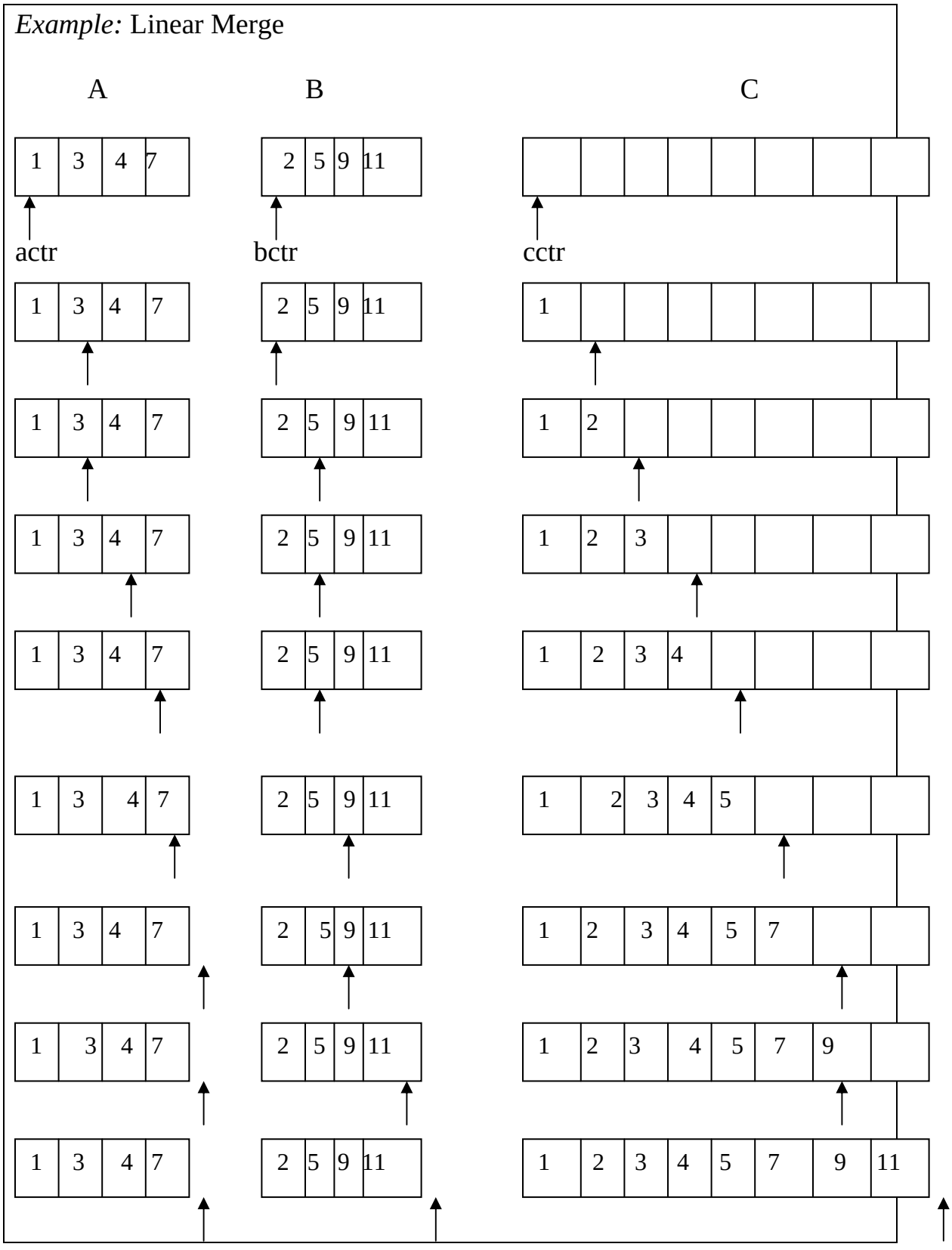
Since this algorithm uses the divide and conquer strategy and employs the halving principle, we can guess that the sorting process would have $O(\log_2 n)$ complexity. However, the merging operation would involve movement of all the n elements (linear time), and we shall show later that the overall complexity 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 and 2, 5, 9, 11

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


Example: Linear Merge
A
B
C
actr
bctr
cctr

The array is recursively split into two halves and mergesort function is applied on the two arrays separately. The arrays get sorted. Then these two arrays are merged. The mergesort and merge functions are shown below:

```
void mergesort(int array[], int n)
{
        int j,n1,n2,arr1[n],arr2[n];
        if (n<=1)return;
        n1=n/2;
        n2 = n - n1;
        for ( j = 0; j<n1; j++ )
           arr1[j]= array[ j];
        for ( j = 0; j<n2; j++ )
           arr2[j]= array[ j+n1];

        mergesort(arr1, n1);
        mergesort(arr2, n2);
        merge(array, arr1, n1, arr2, n2);
}

void merge ( int array[], int arr1[], int n1,int arr2[],
int n2)
{
        int j, p=0, p1=0,p2=0;

        printf("\n After merging  [");
        for(j=0; j<n1; j++)
                printf("%d ",arr1[j] );
        printf("]  [");
        for(j=0; j<n2; j++)
                printf("%d ",arr2[j] );
        printf("]");

        while ( p1 < n1 && p2 < n2 )
        {
              if( arr1[p1] < arr2[p2 ])
                  array [p++] = arr1[p1++];
              else
                  array[p++] = arr2[p2++];
        }
        while ( p1 < n1 )
                array [p++] = arr1[p1++];
        while ( p2 < n2 )
                array[p++] = arr2[p2++];

        printf("       merged array is  [");
```

```
        for(j=0; j<n1+n2; j++)
                printf("%d ",array[j] );
         printf("]\n");

}
```

To sort an array A of size n, the call from the main program would be **mergesort(A, n).**
Here is a typical run of the algorithm, for an array of size 8.
Unsorted array A = [31 45 24 15 23 92 30 77]

The original array is kept on splitting in two halves till it reduces to array of one element. Then these are merged.

After merging [31] &[45] merged array is [31 45]

After merging [24] & [15] merged array is [15 24]

After merging [31 45] & [15 24] merged array is [15 24 31 45]

After merging [23] & [92] merged array is [23 92]

After merging [30] & [77] merged array is [30 77]

After merging [23 92] & [30 77] merged array is [23 30 77 92]

After merging [15 24 31 45] & [23 30 77 92]

merged array is [15 23 24 30 31 45 77 92]

Computational Complexity:

Intuitively we can see that as the **mergesort** routine 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 the number of operations involved using the 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

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

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

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

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

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

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

The next substitution would lead us to

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

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

$T(n) = 2^k \; T(n/ 2^k) + k n$

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 not preferred for sorting data in main memory. The main problem is that merging two sorted lists uses linear extra memory (as you need to copy the original array into two arrays of half the size), 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 the Quicksort, which we shall be studying next.

Quicksort

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. The steps involved in the quicksort algorithm are explained through an example.

Example array:

[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. In this example it already points to value smaller than the pivot.

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
```

move lh to right till it finds an element larger than 56

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

Now exchange

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

move rh to left

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

No exchange now, as both lh and rh point to the same element

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.

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

8. Repeat steps 1 to 6 for the two sub arrays recursively.

Here is the code for the quick sort method:

```
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);
  }


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;

      }
    if(values[left]>= pivot)
          return 0;

    values[0] = values[left];
```

```
    values[left]= pivot;
    return left;
}
```

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.

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

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

leads to

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

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:

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

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

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

…..
…..

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

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

$= N(N+1)/2$

$= O(N^2)$

This implies that whenever a wrong pivot is selected it leads to unbalanced partitioning.