Lecture 6
Divide and conquer, mergesort

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

Divide and Conquer

Divide and conquer paradigm

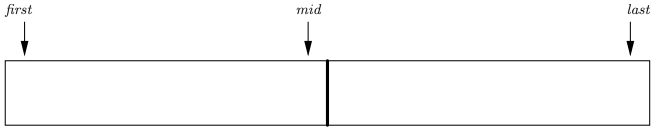
1. Split problem into subproblem(s)
2. Solve each subproblem (usually via recursive call)
3. Combine solution of subproblem(s) into solution of original problem

We will discuss two sorting algorithms based on this paradigm:

- Quicksort (done)
- Mergesort

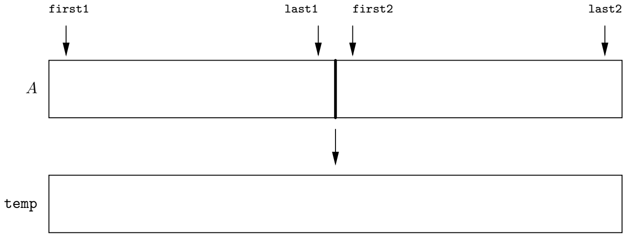
MergeSort

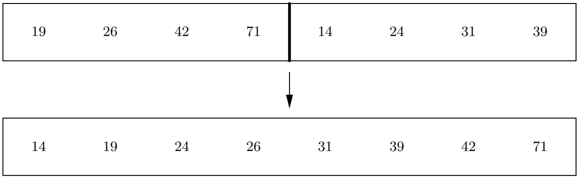
- Split array into two equal subarrays
- Sort both subarrays (recursively)
- Merge two sorted subarrays



```
def mergeSort(A,first,last):
    if first < last:
        mid = $\lfloor$(first + last)/2$\rfloor$
        mergeSort(A,first,mid)
        mergeSort(A,mid+1,last)
        merge(A,first,mid,mid+1,last)
```

The merge step





Merging two lists of total size n requires at most $n - 1$ comparisons.

Code for the merge step

```
def merge(A,first1,last1,first2,last2):
    index1 = first1; index2 = first2; tempIndex = 0
   // Merge into temp array until one input array is exhausted
    while (index1 <= last1) and (index2 <= last2)
        if A[index1] <= A[index2]:
            temp[tempIndex++] = A[index1++]
        else:
            temp[tempIndex++] = A[index2++]
   // Copy appropriate trailer portion
    while (index1 <= last1):  temp[tempIndex++] = A[index1++]
    while (index2 <= last2):  temp[tempIndex++] = A[index2++]
   // Copy temp array back to A array
    tempIndex = 0; index = first1
    while (index <= last2):  A[index++] = temp[tempIndex++]
```

Analysis of Mergesort

$T(n) =$ number of comparisons required to sort n items in the worst case

$$T(n) = \begin{cases} T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1, & n > 1 \\ 0, & n = 1 \end{cases}$$

The asymptotic solution of this recurrence equation is

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

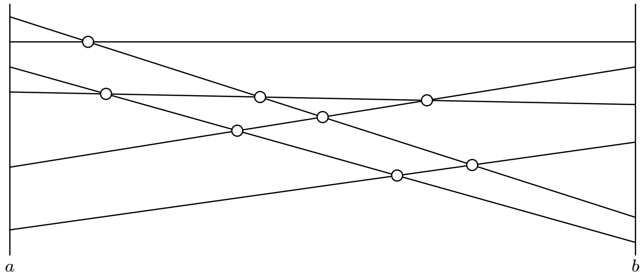
The exact solution of this recurrence equation is

$$T(n) = n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$$

Geometrical Application: Counting line intersections

- Input: n lines in the plane, none of which are vertical; two vertical lines $x = a$ and $x = b$ (with $a < b$).
- Problem: Count/report all pairs of lines that intersect between the two vertical lines $x = a$ and $x = b$.

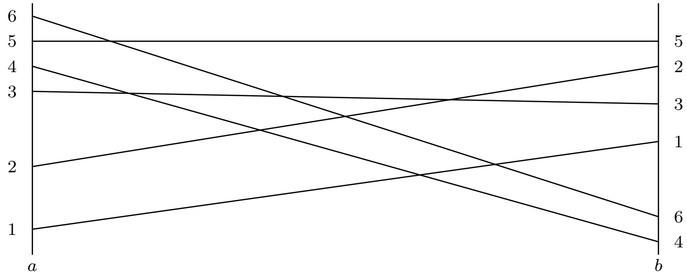
Example: $n = 6$ 8 intersections



Checking every pair of lines takes $\Theta(n^2)$ time. We can do better.

Geometrical Application: Counting line intersections

1. Sort the lines according to the y-coordinate of their intersection with the line $x = a$. Number the lines in sorted order. [$O(n \log n)$ time]
2. Produce the sequence of line numbers sorted according to the y-coordinate of their intersection with the line $x = b$ [$O(n \log n)$ time]
3. Count/report inversions in the sequence produced in step 2.



So the problem reduces to counting/reporting inversions.

Counting Inversions: An Application of Mergesort

An inversion in a sequence or list is a pair of items such that the larger one precedes the smaller one.

Example: The list [18, 29, 12, 15, 32, 10] has 9 inversions:

$(18, 12), (18, 15), (18, 10), (29, 12), (29, 15), (29, 10), (12, 10), (15, 10), (32, 10)$

In a list of size n, there can be as many as $\binom{n}{2}$ inversions.

Problem: Given a list, compute the number of inversions.

Brute force solution: Check each pair i, j with $i < j$ to see if $L[i] > L[j]$. This gives a $\Theta(n^2)$ algorithm. We can do better.

Inversion Counting

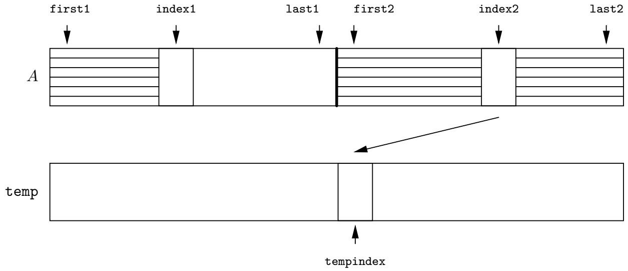
Sorting is the process of removing inversions. So to count inversions:

- Run a sorting algorithm
- Every time data is rearranged, keep track of how many inversions are being removed.

In principle, we can use any sorting algorithm to count inversions. Mergesort works particularly nicely.

Inversion Counting with MergeSort

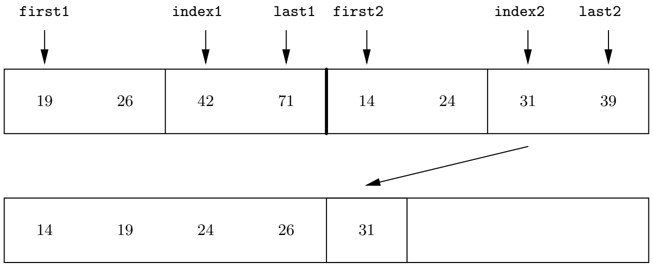
In Mergesort, the only time we rearrange data is during the merge step.



The number of inversions removed is:

$$\texttt{last1} - \texttt{index1} + 1$$

Example



2 inversions removed: $(42, 31)$ and $(71, 31)$

Pseudocode for the merge step with inversion counting

```
def merge(A,first1,last1,first2,last2):
    index1 = first1; index2 = first2; tempIndex = 0
    invCount = 0
   // Merge into temp array until one input array is exhausted
    while (index1 <= last1) and (index2 <= last2)
        if A[index1] <= A[index2]:
            temp[tempIndex++] = A[index1++]
        else:
            temp[tempIndex++] = A[index2++]
            invCount += last1 - index1 + 1;
   // Copy appropriate trailer portion
    while (index1 <= last1):  temp[tempIndex++] = A[index1++]
    while (index2 <= last2):  temp[tempIndex++] = A[index2++]
   // Copy temp array back to A array
    tempIndex = 0; index = first1
    while (index <= last2):  A[index++] = temp[tempIndex++]
    return invCount
```

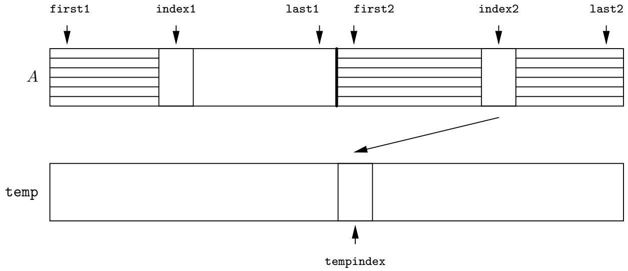
Pseudocode for MergeSort with inversion counting

```
def mergeSort(A,first,last):
    invCount = 0
    if first < last:
        mid = $\lfloor$(first + last)/2$\rfloor$
        invCount += mergeSort(A,first,mid)
        invCount += mergeSort(A,mid+1,last)
        invCount += merge(A,first,mid,mid+1,last)
    return invCount
```

Running time is the same as standard mergeSort: $O(n \log n)$

Listing inversions

We have just seen that we can count inversions without increasing the asymptotic running time of Mergesort. Suppose we want to list inversions. When we remove inversions, we list all inversions removed:



(A[index1], A[index2]), (A[index1+1], A[index2]), . . . , (A[last1], A[index2]).

The extra work to do the reporting is proportional to the number of inversions reported.

Inversion counting summary

Using a slight modification of Mergesort, we can . . .

- Count inversions in $O(n \log n)$ time.
- Report inversions in $O(n \log n + k)$ time, where k is the number of inversions.

The same results hold for the line-intersection counting problem.

The reporting algorithm is an example of an output-sensitive algorithm. The performance of the algorithm depends on the size of the output as well as the size of the input.