Start with a quick review of data structures

Basic Data structures (Prerequisite material; Review [GT] Chapter 2–4 as necessary)

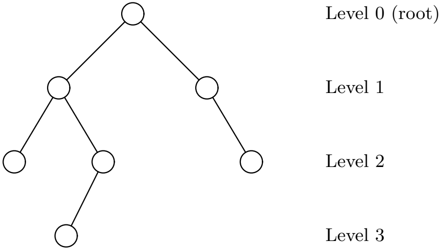
- Arrays
- Linked lists
- Hash Tables
- Trees

Abstract data types

- Vectors: support access by rank
- Lists: support access by current position
- Sequences: supports access by both rank and current position
- Dictionaries: supports access by key

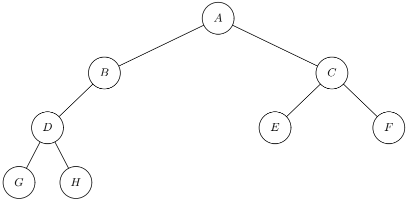
Binary Trees: a quick review

We will use as a data structure and as a tool for analyzing algorithms.



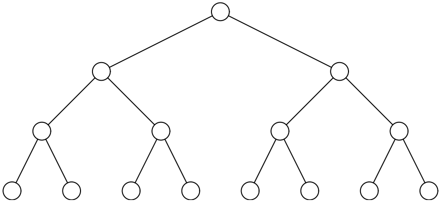
The depth of a binary tree is the maximum of the levels of all its leaves.

Traversing binary trees



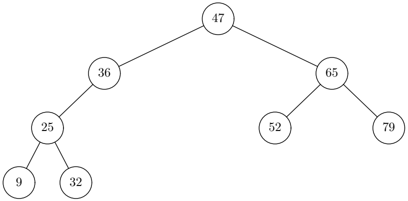
- Preorder: root, left subtree (in preorder), right subtree (in preorder): *ABDGHCEF*
- Inorder: left subtree (in inorder), root, right subtree (in inorder): *GDHBAECF*
- Postorder: left subtree (in postorder), right subtree (in postorder), root: *GHDBEFCA*
- Breadth-first order (level order): level 0 left-to-right, then level 1 left-to-right, . . . : *ABCDEFGH*

Facts about binary trees



1. There are at most 2^k nodes at level k.
2. A binary tree with depth d has:
   - At most 2^d leaves.
   - At most $2^{d+1} - 1$ nodes.
3. A binary tree with n leaves has depth $\geq \lceil \lg n \rceil$.
4. A binary tree with n nodes has depth $\geq \lfloor \lg n \rfloor$.

Binary search trees



- Function as ordered dictionaries
- find, insert, and remove can all be done in $O(h)$ time (h = tree height)
- AVL trees and Red-Black Trees: $h = O(\log n)$, so find, insert, and remove can all be done in $O(\log n)$ time.
- listAllItems in $O(n)$ time, where n = number of items in tree.
- Splay trees and Skip Lists: alternatives to balanced trees
- [GT] Chapters 3–4 for details

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.
- Problem is to locate x in the array.
- Several variants of the problem, for example...
  1. Determine whether x is stored in the array
  2. Find the largest i such that $A[i] \leq x$ (with a reasonable convention if $x < A[0]$).

  We will focus on the first variant.
- We will show that binary search is an optimal algorithm for solving this problem.

Binary Search: Searching in a sorted array

Input:	A:	Sorted array with n entries $[0..n-1]$
	x:	Item we are seeking
Output:	Location of x, if x found	
	-1, if x not found	

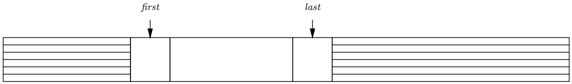
Top-level call is `binarySearch(`A`,`x`,0,`$n-1$`)`

```
int binarySearch($A$,$x$,first,last);
if first > last then
  return (-1);
else
  index = $\lfloor$(first+last)/2$\rfloor$;
  if x == A[index] then
    return index;
  else if x < A[index] then
    return binarySearch($A$,$x$,first,index-1);
  else
    return binarySearch($A$,$x$,index+1,last);
```

Correctness of Binary Search

We need to prove two things:

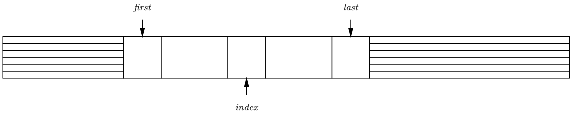
1. If x is in the array, its location (index) is between *first* and *last*, inclusive.
2. On each recursive call, the difference *last* − *first* gets strictly smaller.



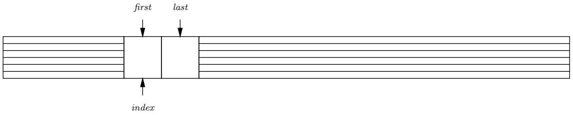
Correctness of Binary Search

To prove that the invariant continues to hold, we need to consider three cases.

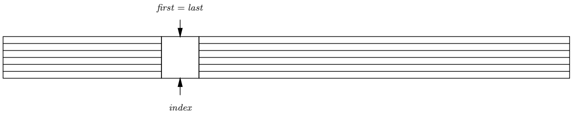
1. $last \geq first + 2$



2. $last = first + 1$



3. $last = first$



Binary Search: Analysis of Running Time

- We will count the number of 3-way comparisons of x against elements of A. (also known as decisions)
- This is the essentially the same as the number of recursive calls. Every recursive call, except for possibly the very last one, results in a 3-way comparison.

Binary Search: Analysis of Running Time (continued)

- Binary search in an array of size 1: 1 decision
- Binary search in an array of size $n > 1$: after 1 decision, either we are done, or the problem is reduced to binary search in a subarray of size $\leq \lfloor n/2 \rfloor$ (with equality possible).
- So the worst-case time to do binary search on an array of size n satisfies the inequality

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{otherwise} \end{cases}$$

- The solution to this equation is:

$$T(n) = \lfloor \lg n \rfloor + 1$$

  This can be proved by induction.
- So binary search does $\lfloor \lg n \rfloor + 1$ 3-way comparisons on an array of size n, in the worst case.

Optimality of binary search

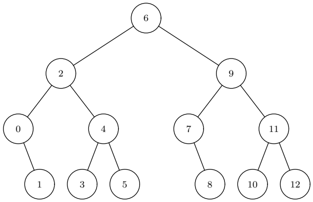
- We will establish a lower bound on the number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this bound, it is optimal.
- Our lower bound is established using a Decision Tree model.
- Note that the bound is exact (not just asymptotic).

The decision tree model for searching in an array

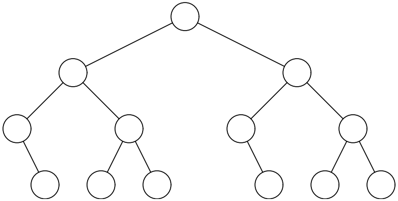
Consider any algorithm that searches for an item x in an array A of size n by comparing entries in A against x. Any such algorithm can be modeled as a decision tree:

- Each node is labeled with an integer $\in \{0 \ldots n-1\}$.
- A node labeled i represents a 3-way comparison between x and $A[i]$.
- The left subtree of a node labeled i describes the decision tree for what happens if $x < A[i]$.
- The right subtree of a node labeled i describes the decision tree for what happens if $x > A[i]$.

Example: Decision tree for binary search with $n = 13$:



Lower bound on locating an item in an array of size n



1. Any algorithm for searching an array of size n can be modeled by a decision tree with at least n nodes.
2. Since the decision tree is a binary tree with n nodes, the depth is at least $\lfloor \lg n \rfloor$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree $+1$. (Remember, root has depth 0).

Hence any algorithm for locating an item in an array of size n using only comparisons must perform at least $\lfloor \lg n \rfloor + 1$ comparisons.

So binary search is optimal.

Sorting

- Rearranging a list of items in nondescending order.
- Useful preprocessing step (e.g., for binary search)
- Important step in other algorithms
- Illustrates more general algorithmic techniques

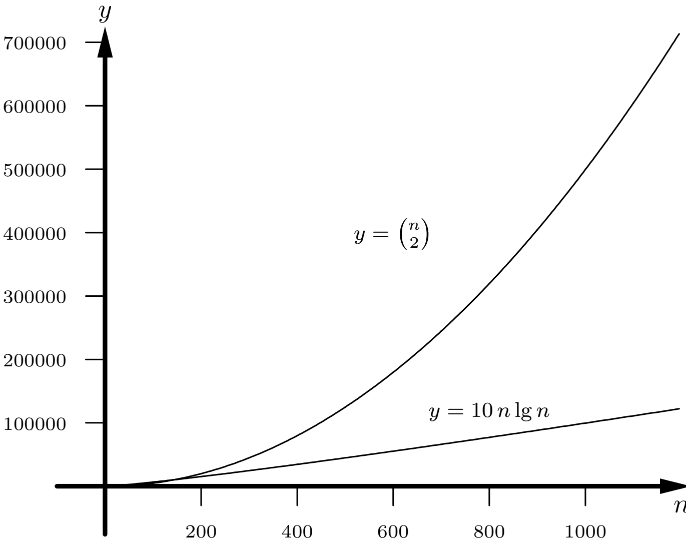
We will discuss

- Comparison-based sorting algorithms (Insertion sort, Selection Sort, Quicksort, Mergesort, Heapsort)
- Bucket-based sorting methods
- Disk-based sorting

Comparison-based sorting

- Basic operation: compare two items.
- Abstract model.
- Advantage: doesn't use specific properties of the data items. So same algorithm can be used for sorting integers, strings, etc.
- Disadvantage: under certain circumstances, specific properties of the data item can speed up the sorting process.
- Measure of time: number of comparisons
  - Consistent with philosophy of counting basic operations, discussed earlier.
  - Could be misleading if other operations dominate. For example, what if there are more assignment statements than comparisons? a lot of data movement.
- Comparison-based sorting requires $\Omega(n \log n)$ comparisons. (We will prove this.)

$\Theta(n \log n)$ work vs. quadratic ($\Theta(n^2)$) work



Some terminology

- A permutation of a sequence of items is a reordering of the sequence. A sequence of n items has $n!$ distinct permutations.
- Note: Sorting is the problem of finding a particular distinguished permutation of a list.
- An inversion in a sequence or list is a pair of items such that the larger one precedes the smaller one.

Example: The list

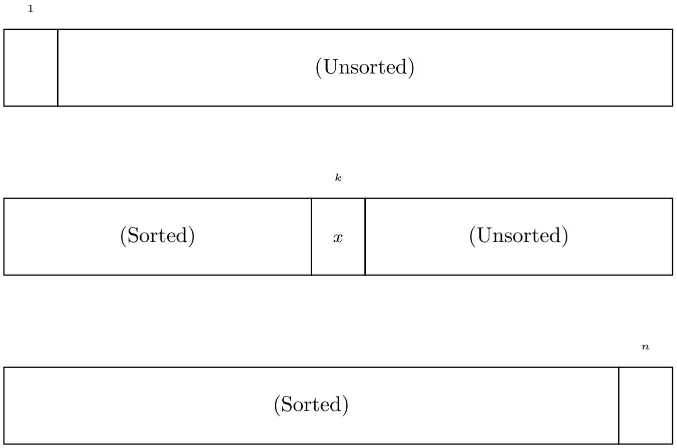
$$18 \quad 29 \quad 12 \quad 15 \quad 32 \quad 10$$

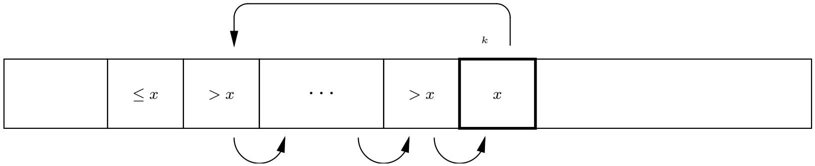
has 9 inversions:

$$\{(18,12), (18,15), (18,10), (29,12), (29,15),$$
$$(29,10), (12,10), (15,10), (32,10)\}$$

Insertion sort

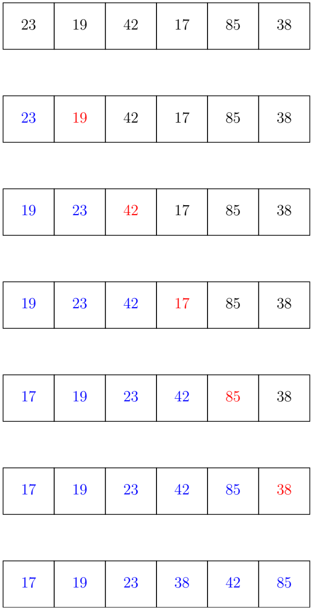
- Work from left to right across array
- Insert each item in correct position with respect to (sorted) elements to its left





```
def insertionSort(n, A):
    for k = 1 to n-1:
        x = A[k]
        j = k-1
        while (j >= 0) and (A[j] > x):
            A[j+1] = A[j]
            j = j-1
        A[j+1] = x
```

Insertion sort example



Analysis of Insertion Sort

- Storage: in place: $O(1)$ extra storage
- Worst-case running time:
  - On kth iteration of outer loop, element $A[k]$ is compared with at most $k-1$ elements:
    $A[k-1]$, $A[k-2]$, $\ldots$, $A[1]$.
  - Total number comparisons over all iterations is at most:

$$\sum_{k=1}^{n}(k-1) = \frac{n(n-1)}{2} = \Theta(n^2).$$

- Average-case Time: Approximately $\frac{n^2}{4} = \Theta(n^2)$
- Insertion Sort is efficient if the input is "almost sorted":

$$\text{Time} \leq n - 1 + (\#\text{ inversions}) \qquad \text{(Why?)}$$

Selection Sort

- Two variants:
  1. Repeatedly (for i from 0 to $n-1$) find the minimum value, output it, delete it.
     - Values are output in sorted order
  2. Repeatedly (for i from $n-1$ down to 1)
     - Find the maximum of $A[0]$,$A[1]$,...,$A[i]$.
     - Swap this value with $A[i]$ (if necessary).
- Both variants run in $O(n^2)$ time and are in place.

Sorting algorithms based on 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

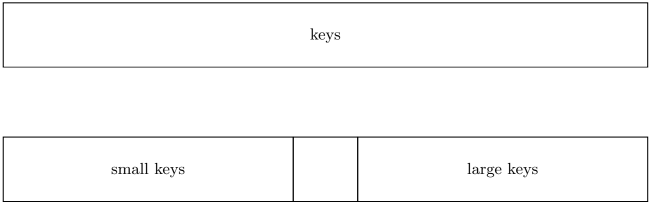
We will discuss two sorting algorithms based on this paradigm:

- Quicksort
- Mergesort

Quicksort

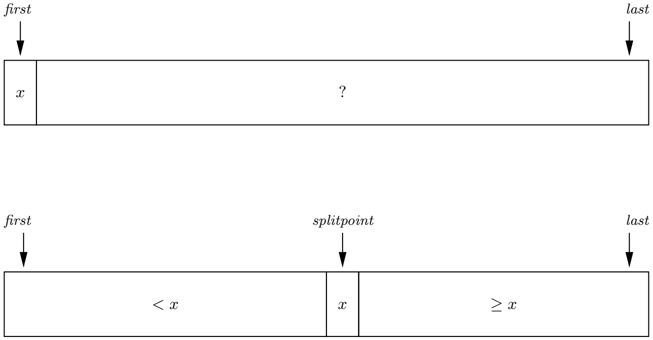
Basic idea

- Classify keys as “small” keys or “large” keys. All small keys are less than all large keys
- Rearrange keys so small keys precede all large keys.
- Recursively sort ”small keys”, recursively sort ”large” keys.



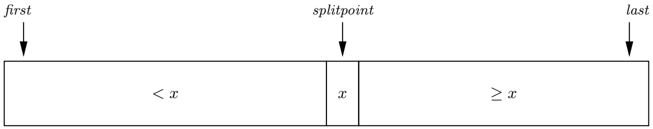
Quicksort: One specific implementation

- Let the first element in the array be the pivot value x.
- Small keys are the keys $< x$.
- Large keys are the keys $\geq x$.



Top level pseudocode

```
def quickSort(A,first,last):
    if first < last:
        splitpoint = split(A,first,last)
        quickSort(A,first,splitpoint-1)
        quickSort(A,splitpoint+1,last)
```



The split step

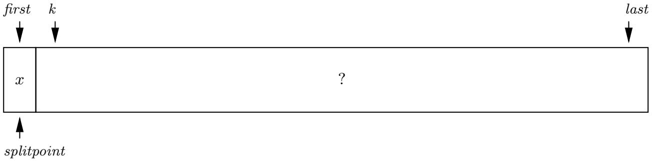
```
def split(A,first,last):
    splitpoint = first
    x = A[first]
    for k = first+1 to last do:
        if A[k] < x:
            A[splitpoint+1] ↔ A[k]
            splitpoint = splitpoint + 1
    A[first] ↔ A[kwsplitpoint]
    return splitpoint
```

Loop invariants:

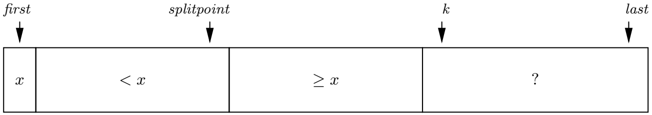
- $A[\mathtt{first}+1..\mathtt{splitpoint}]$ contains keys $< x$.
- $A[\mathtt{splitpoint}+1..k-1]$ contains keys $\geq x$.
- $A[k..\mathtt{last}]$ contains unprocessed keys.

The split step

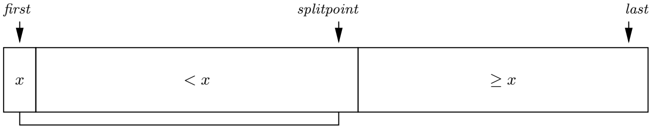
At start:



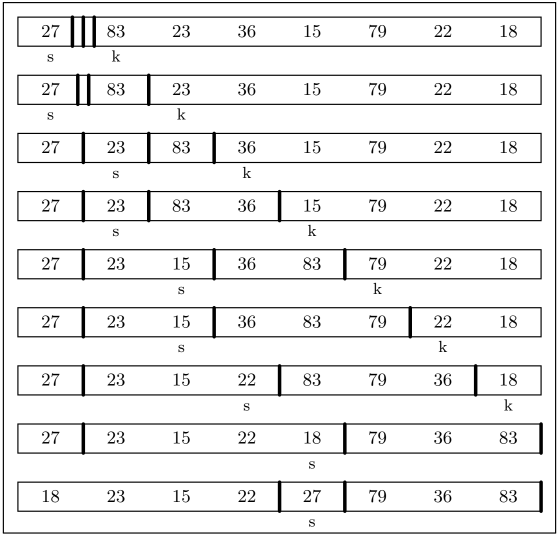
In middle:



At end:



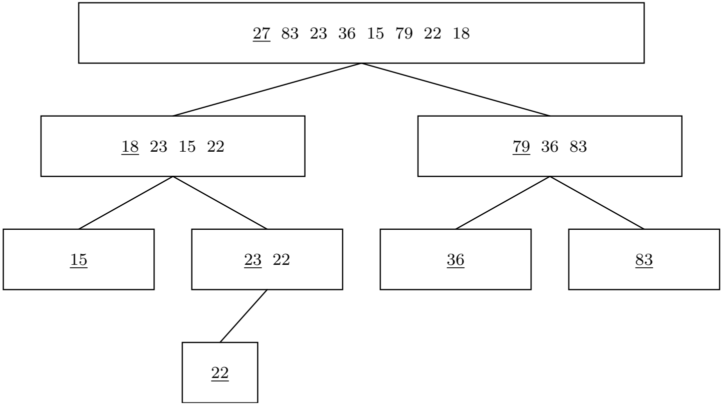
Example of split step



Analysis of Quicksort

We can visualize the lists sorted by quicksort as a binary tree.

- The root is the top-level list (of all elements to be sorted)
- Identify each list with its split value.
- The children of a node are the two sublists to be sorted.



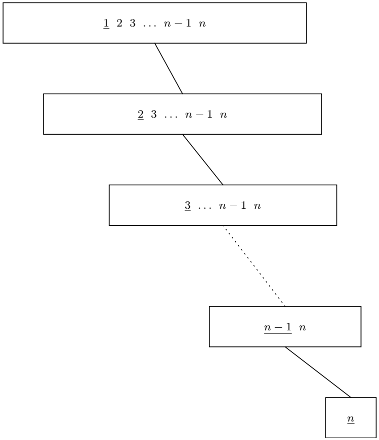
Worst-case Analysis of Quicksort

- Any pair of values x and y gets compared at most once during the entire run of Quicksort.
- The number of possible comparisons is

$$\binom{n}{2} = \Theta(n^2)$$

- Hence the worst-case number of comparisons performed by Quicksort when sorting n numbers is $O(n^2)$.
- Can we be more precise? Is it $o(n^2)$? Is it $\Theta(n^2)$?
- It is $\Theta(n^2)$...

A bad case case for Quicksort: $1, 2, 3, \ldots, n-1, n$


$\underline{1}$ 2 3 ... $n-1$ n
$\underline{2}$ 3 ... $n-1$ n
$\underline{3}$ ... $n-1$ n
$\underline{n-1}$ n
$\underline{n}$

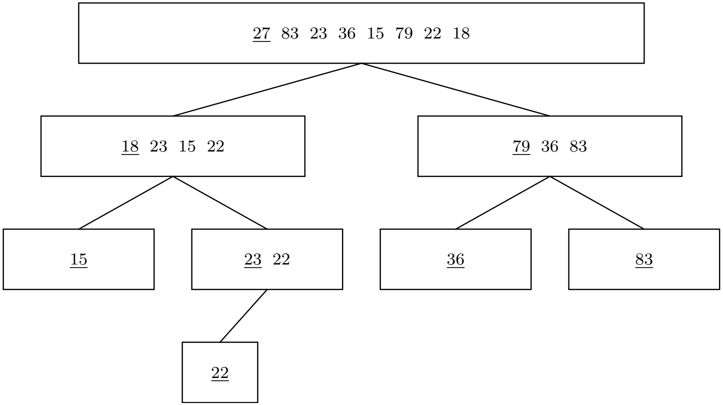
$\binom{n}{2}$ comparisons required. So the worst-case time for Quicksort is $\Theta(n^2)$. But what about the average case ...?

Average-case analysis of Quicksort:

Our approach:

1. Use the binary tree of sorted lists
2. Number the elements in sorted order
3. Calculate the probability that two elements get compared
4. Use this to compute the expected number of comparisons performed by Quicksort.

Average-case analysis of Quicksort:



Sorted order: $\boxed{15\ \ 18\ \ 22\ \ 23\ \ 27\ \ 36\ \ 79\ \ 83}$

Key Fact

During the run of Quicksort, two values x and y get compared if and only if the first key in the range $[x..y]$ to be chosen as a pivot is either x or y.

- Examples where both statements are true: $(18, 23)$, $(27, 83)$
- Examples where both statements are false: $(15, 23)$, $(18, 83)$

Average-case analysis of Quicksort

Assume:

- All permutations are equally likely
- All n values are distinct
- The values in sorted order are $S_1 < S_2 < \cdots < S_n$.

Let $P_{i,j}$ = The probability that keys S_i and S_j are compared with each other during the invocation of quicksort

Then by Key Fact on previous slide:

$P_{i,j}$ = The probability that the first key from $\{S_i, S_{i+1}, \ldots, S_j\}$ to be chosen as a pivot value is either S_i or S_j

$$= \frac{2}{j-i+1}$$

Average-case analysis of Quicksort

Define indicator random variables $\{X_{i,j} : 1 \leq i < j \leq n\}$

$$X_{i,j} = \begin{cases} 1 & \text{if keys } S_i \text{ and } S_j \text{ get compared} \\ 0 & \text{if keys } S_i \text{ and } S_j \text{ do \underline{not} get compared} \end{cases}$$

1. The total number of comparisons is:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}$$

2. The expected (average) total number of comparisons is:

$$E\left(\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}\right) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E\left(X_{i,j}\right)$$

3. The expected value of $X_{i,j}$ is:

$$E(X_{i,j}) = P_{i,j} = \frac{2}{j-i+1}$$

Average-case analysis of Quicksort

Hence the expected number of comparisons is

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j=i+1}^{n} E\left(X_{i,j}\right) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \quad (k = j-i+1) \\
&< \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} \\
&= 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \\
&= 2 \sum_{i=1}^{n} H_n = 2nH_n \in O(n \lg n).
\end{aligned}$$

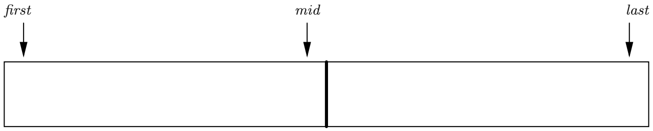
So the average time for Quicksort is $O(n \lg n)$.

Implementation tricks for improving Quicksort

1. Better choice of "pivot" element:
   - Instead of a single element, choose median of 3 (or 5, or 7, . . .)
   - Choose a random element (or randomly reorder the list as a preprocessing step)
   - Combine
2. Reduce procedure call overhead
   - Explicitly manipulate the stack in the program (rather than making recursive calls)
   - For small lists, use some other nonrecursive sort (e.g., insertion sort or selection sort, or a minimum-comparison sort)
3. Reduce stack space
   - Push the larger sublist (the one with more elements) and immediately working on the smaller sublist.
   - Reduces worst-case stack usage from $O(n)$ to $O(\lg n)$.

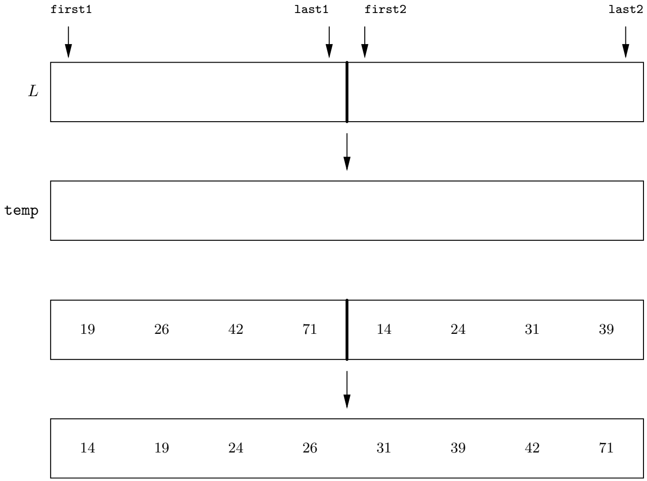
Mergesort

- 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 = first
    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

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

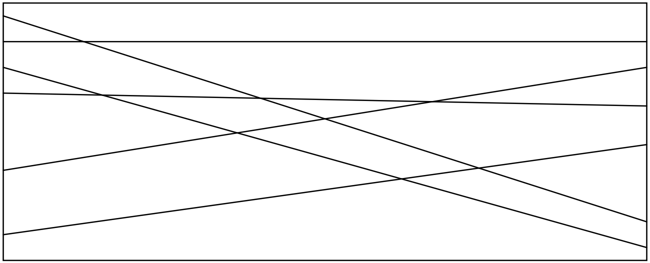
The exact solution of this recurrence equation is

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

Geometrical Application: Counting line intersections

- Given:A rectangle, R; n lines that enter R at the left edge and leave R at the right edge
- Problem: Count/report the intersection points of the various lines that occur <u>inside</u> R.

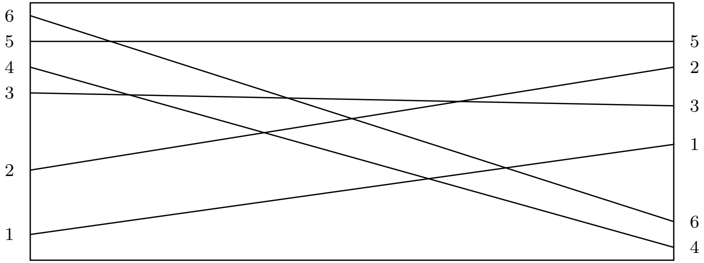
Example: ($n = 6$, 8 intersections)



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*-coordinates of their left-hand endpoints (in increasing order).
2. Examine the sequence at which the lines emerge (on the right), ordered by *y*-coordinate of right-hand-endpoint.
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

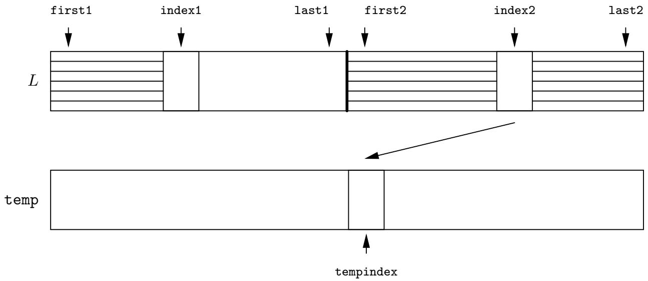
Sorting is the process of removing inversions. So to count inversions:

- 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

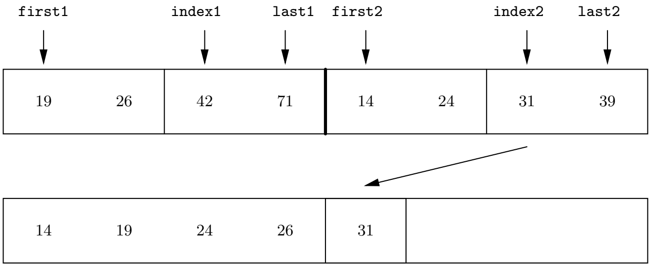
In Mergesort, the only time we rearrange data is during the merge step.



The number of inversions removed is:

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

Example



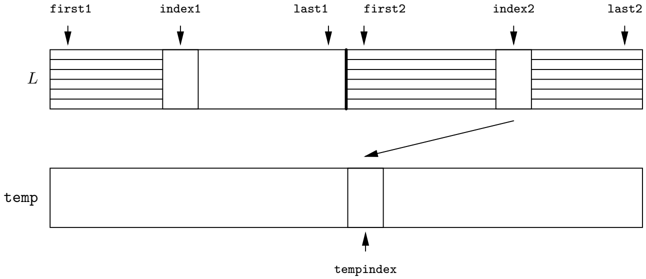
2 inversions removed: $(42, 31)$ and $(71, 31)$

Code for the merge step with inversion counting

```
def merge(A,first1,last1,first2,last2):
     index1 = first1; index2 = first2; tempIndex = 0
     int 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 = 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 = first
     while (index <= last2):  A[index++] = temp[tempIndex++]
```

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:



$(L[\texttt{index1}], L[\texttt{index2}])$, $(L[\texttt{index1+1}], L[\texttt{index2}])$, …, $(L[\texttt{last1}], L[\texttt{index2}])$.

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.