Outline of these notes

- Review of basic data structures
- Searching in a sorted array/binary search: the algorithm, analysis, proof of optimality
- Sorting, part 1: insertion sort, quicksort, mergesort

Basic Data structures

Prerequisite material. Review [GT Chapters 2–4, 6] as necessary)

- Arrays, dynamic arrays
- Linked lists
- Stacks, queues
- Dictionaries, hash tables
- Binary trees

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python
- Linked lists:
  - Collection of nodes that form a linear ordering.
    - The list has a first node and a last node
    - Each node has a next node and a previous node (possibly null)
  - Inserting or deleting an item in the middle of linked list is fast.
  - Accessing a cell given its index (i.e., finding the kth item in the list) is slow.

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.
  - Insert and remove are usually called `push` and `pop`
- Queues (often called FIFO Queues)
  - Container of objects that are inserted and removed according to First-In First-Out (FIFO) principle:
    - Only the element that has been in the queue the longest can be removed.
  - Insert and remove are usually called `enqueue` and `dequeue`
  - Elements are inserted at the rear of the queue and are removed from the front

Dictionaries/Maps

- Dictionaries
  - A Dictionary (or Map) stores `<key,value>` pairs, which are often referred to as items
  - There can be at most one item with a given key.
  - Examples:
    1. `<Student ID, Student data>`
    2. `<Object ID, Object data>`

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining
  - Open addressing: linear probing, quadratic probing, double hashing
  - Cuckoo hashing

Hashing is fast:

- $O(1)$ expected time for access, insertion
- Cuckoo hashing improves the access time to $O(1)$ worst-case time. Insertion time remains $O(1)$ expected time.

Disadvantages on next slide.

Hashing: Disadvantages

- Access time (except for cuckoo hashing) and insertion time (for all strategies) is expected time, not worst-case time. So there is no absolute guarantee on performance.
- Performance depends on the load factor

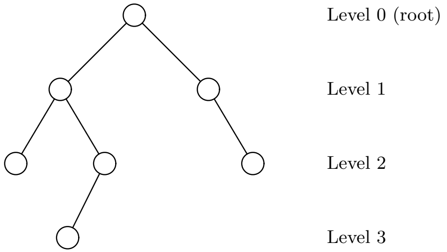
  $$\alpha = \frac{n}{N},$$

  where n is the number of items stored and N is the table size. As α gets larger, performance deteriorates.
- Hashing can tell us whether a given key is in the dictionary. It cannot tell us if nearby keys are in the dictionary.
  - Is the word cas stored in the dictionary? (Exact match query)
  - What is the first word in the dictionary that comes after cas? (Successor query)

Binary Trees: a quick review

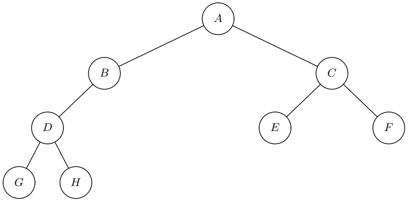
We will use as a data structure and as a tool for analyzing algorithms.

The depth or level of a node is the number of nodes above it on the path to the root. Hence the root is at depth 0.



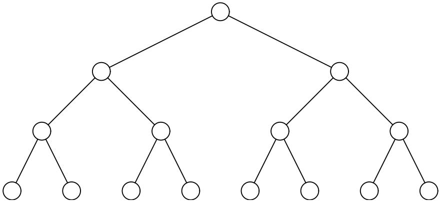
The depth of a binary tree is the maximum of the levels of all its leaves.

Traversing binary trees



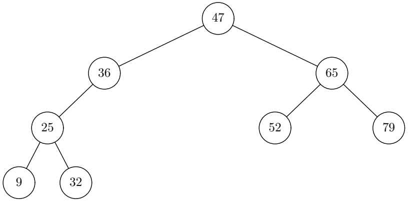
- Preorder: root, left subtree (in preorder), right subtree (in preorder): *ABDGHCEF*
- Inorder: left subtree (in inorder), root, right subtree (in inorder): *GDHBAECF*
- Postorder: left subtree (in postorder), right subtree (in postorder), root: *GHDBEFCA*
- Breadth-first order (level order): level 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. (Can find successors, predecessors)
- find, insert, and remove can all be done in $O(h)$ time ($h =$ tree height)
- AVL trees, Red-Black Trees, Weak AVL trees: $h = O(\log n)$, so find, insert, and remove can all be done in $O(\log n)$ time.
- Splay trees and Skip Lists: alternatives to balanced trees
- Can traverse the tree and list all items in $O(n)$ time.
- [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

```
def binarySearch(A,x,first,last)
if first > last:
  return (-1)
else:
  mid = ⌊(first+last)/2⌋
  if x == A[mid]:
    return mid
  else if x < A[mid]:
    return binarySearch(A,x,first,mid-1)
  else:
    return binarySearch(A,x,mid+1,last)

binarySearch(A,x,0,n-1)
```

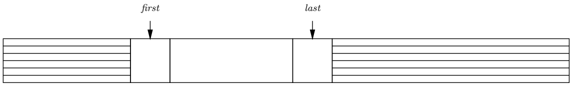
Correctness of Binary Search

We need to prove two things:

1. If x is in the array, its location in the array (its index) is between *first* and *last*, inclusive.
   Note that this is equivalent to:

   > *Either x is not in the array, or its location is between first and last, inclusive.*

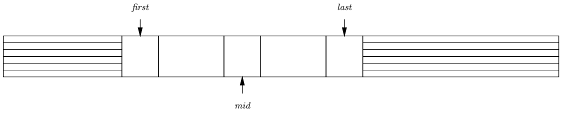
2. On each recursive call, the difference *last* − *first* gets strictly smaller.



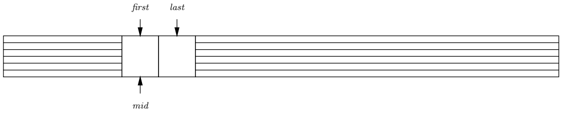
Correctness of Binary Search

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

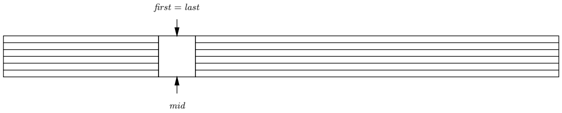
1. $last \geq first + 2$



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)
- Rationale:
  1. 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.
  2. Gives us a way to compare binary search against other algorithms that solve the same problem: searching for an item in an array by comparing the item against array entries.

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 with a worst-case size of $\lfloor n/2 \rfloor$
- So the worst-case time to do binary search on an array of size n is $T(n)$, where $T(n)$ satisfies the equation

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

- 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

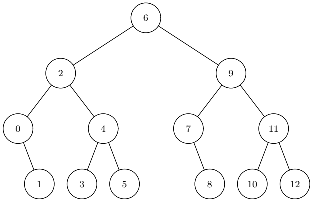
- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this bound, it is optimal.
- Our lower bound is established using a Decision Tree model.
- Note that the bound is exact (not just asymptotic)
- Our lower bound is on the worst case
  - It says: for every algorithm for finding an item in an array of size n, there is some input that forces it to perform $\lfloor \lg n \rfloor + 1$ comparisons.
  - It does not say: for every algorithm for finding an item in an array of size n, every input forces it to perform $\lfloor \lg n \rfloor + 1$ comparisons.

The decision tree model for searching in an array

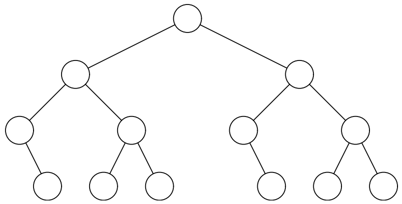
Consider any algorithm that searches for an item x in an array A of size n by comparing entries in A against x. Any such algorithm can be modeled as a decision tree:

- 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 in the worst case.

So binary search is optimal with respect to worst-case performance.

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

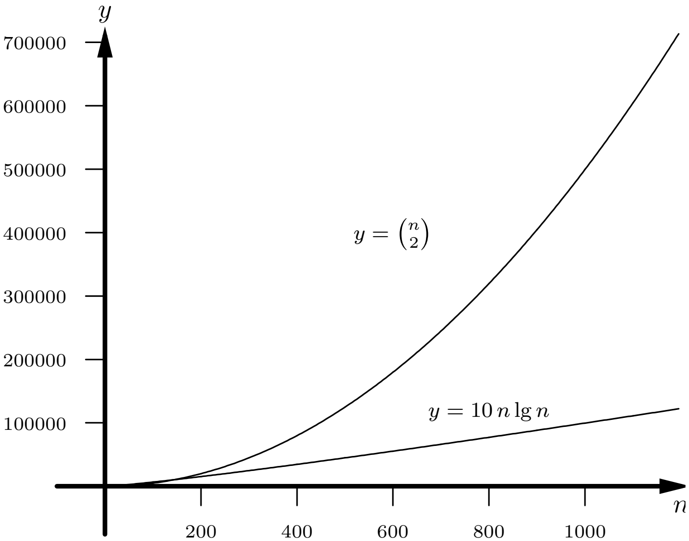
We will discuss

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

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.
  - Misleading if other operations dominate (e.g., if we sort by moving items around without comparing them)
- Comparison-based sorting has lower bound of $\Omega(n \log n)$ 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

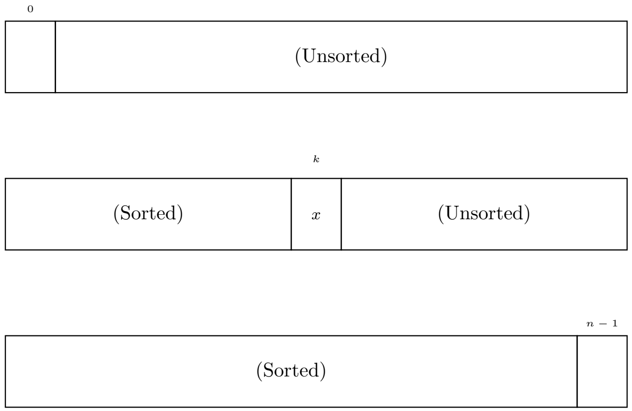
$$18 \quad 29 \quad 12 \quad 15 \quad 32 \quad 10$$

has 9 inversions:

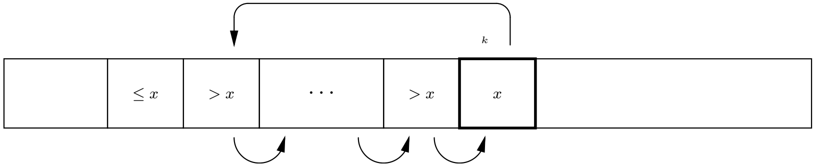
$$\{(18,12), (18,15), (18,10), (29,12), (29,15), (29,10), (12,10), (15,10), (32,10)\}$$

Insertion sort

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

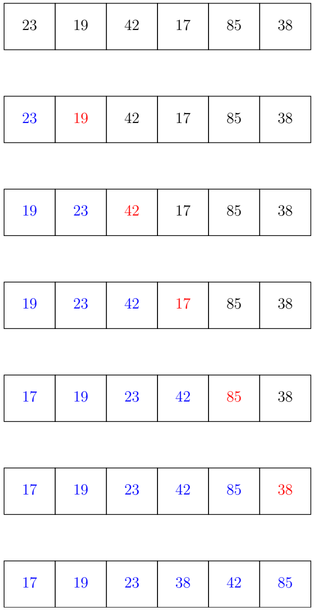


Insertion sort pseudocode



```
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

- Worst-case running time:
  - On kth iteration of outer loop, element $A[k]$ is compared with at most k elements:
    $A[k-1], A[k-2], \ldots, A[0]$.
  - Total number comparisons over all iterations is at most:

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

  - Insertion Sort is a bad choice when n is large. ($O(n^2)$ vs. $O(n \log n)$).
  - Insertion Sort is a good choice when n is small. (Constant hidden in the "big oh" is small).
  - Insertion Sort is efficient if the input is "almost sorted":

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

- Storage: in place: $O(1)$ extra storage

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 $A[i]$ (no-op if it is already $A[i]$).
- Both variants run in $O(n^2)$ time if we use the straightforward approach to finding the maximum/minimum.
- They can be improved by treating the items $A[0]$,$A[1]$,...,$A[i]$ as items in an appropriately designed priority queue. (Next set of notes)

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

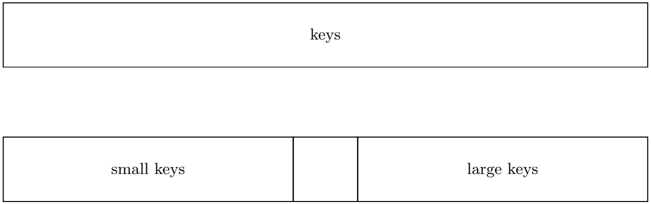
We will discuss two sorting algorithms based on this paradigm:

- Quicksort
- Mergesort

Quicksort

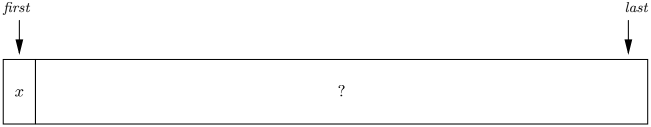
Basic idea

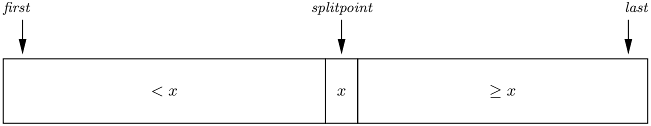
- Classify keys as small keys or large keys. All small keys are less than all large keys
- Rearrange keys so small keys precede all large keys.
- Recursively sort small keys, recursively sort large keys.



Quicksort: One specific implementation

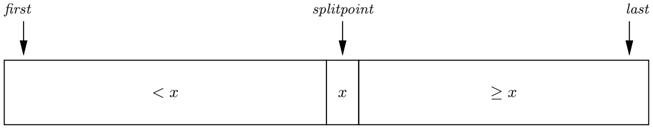
- Let the first item in the array be the pivot value x (also call the split value).
  - Small keys are the keys $< x$.
  - Large keys are the keys $\geq x$.





Pseudocode for Quicksort

```
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

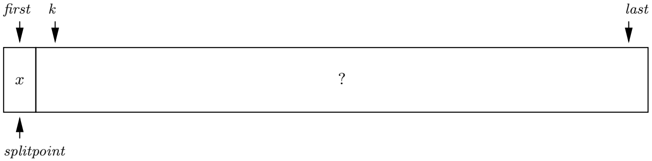
```
def split(A,first,last):
    splitpoint = first
    x = A[first]
    for k = first+1 to last do:
        if A[k] < x:
            A[splitpoint+1] ↔ A[k]
            splitpoint = splitpoint + 1
    A[first] ↔ A[splitpoint]
    return splitpoint
```

Loop invariants:

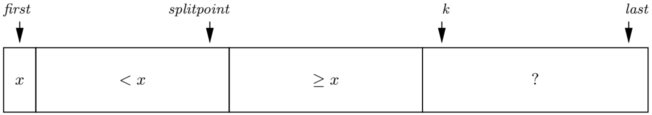
- `A[first+1..splitpoint]` contains keys $< x$.
- `A[splitpoint+1..k-1]` contains keys $\geq x$.
- `A[k..last]` contains unprocessed keys.

The split step

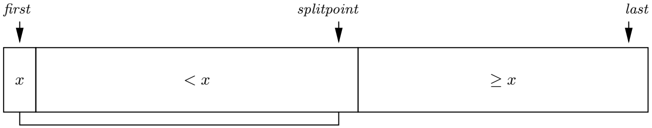
At start:



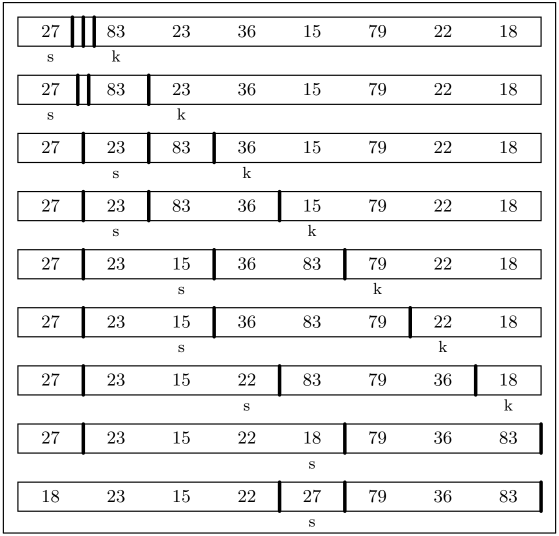
In middle:



At end:



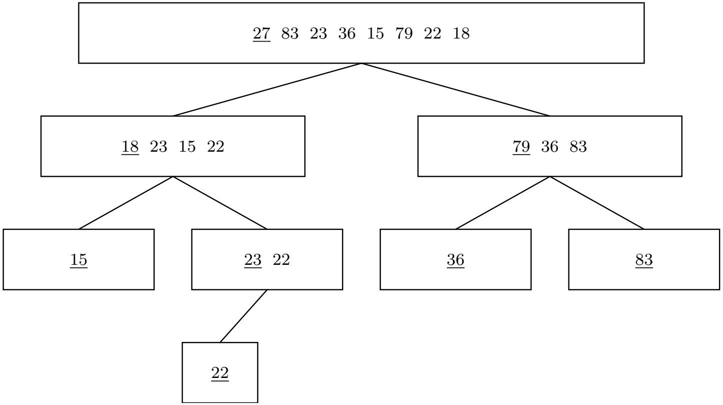
Example of split step



Analysis of Quicksort

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

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



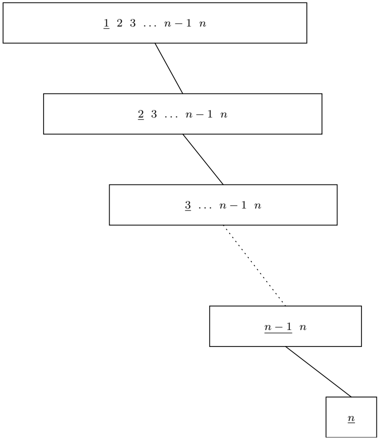
Worst-case Analysis of Quicksort

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

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

- Hence the worst-case number of comparisons performed by Quicksort when sorting n items is $O(n^2)$.
- Question: Is there a better bound? Is it $o(n^2)$? Or is it $\Theta(n^2)$?
- Answer: The bound is tight. It is $\Theta(n^2)$. We will see why on the next slide.

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
$\underline{n}$

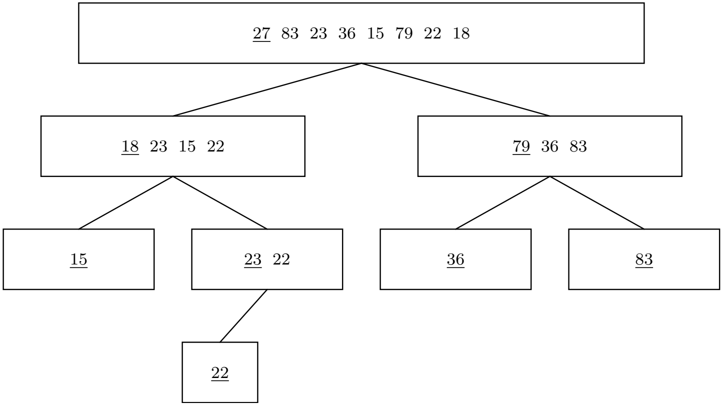
$\binom{n}{2}$ comparisons required. So the worst-case running time for Quicksort is $\Theta(n^2)$. But what about the average case . . . ?

Average-case analysis of Quicksort:

Our approach:

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

Average-case analysis of Quicksort:



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

Average-case analysis of Quicksort

- Number the keys in sorted order: $S_1 < S_2 < \cdots < S_n$.
- Fact about comparisons: During the run of Quicksort, two keys S_i and S_j get compared if and only if the first key from the set of keys $\{S_i, S_{i+1}, \ldots, S_j\}$ to be chosen as a pivot is either S_i or S_j.
  - If some key S_k is chosen first with $S_i < S_k < S_j$, then S_i goes in the left half, S_j goes in the right half, and S_i and S_j never get compared.
  - If S_i is chosen first, it is compared against all the other keys in the set in the split step (including S_j).
  - Similar if S_j is chosen first.

Examples:

- 23 and 22 (both statements true)
- 36 and 83 (both statements false)

Average-case analysis of Quicksort

Assume:

- All n keys are distinct
- All permutations are equally likely
- The keys 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 Fact about comparisons on previous slide:

$$\begin{aligned} P_{i,j} &= \text{The probability that the first key from } \{S_i, S_{i+1}, \ldots, S_j\} \text{ to be chosen as a pivot value is either } S_i \text{ or } S_j \\ &= \frac{2}{j-i+1} \end{aligned}$$

Average-case analysis of Quicksort

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

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

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

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

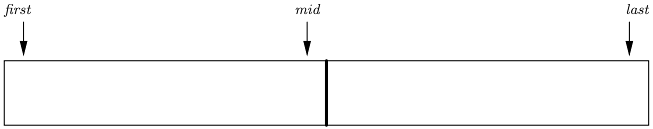
So the average time for Quicksort is $O(n \lg n)$.

Implementation tricks for improving Quicksort

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

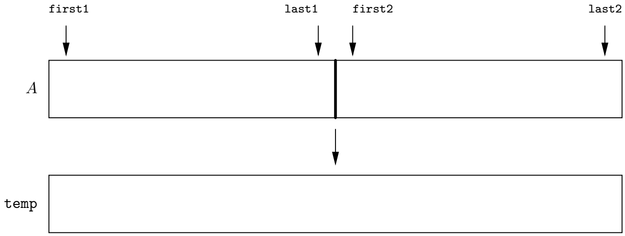
MergeSort

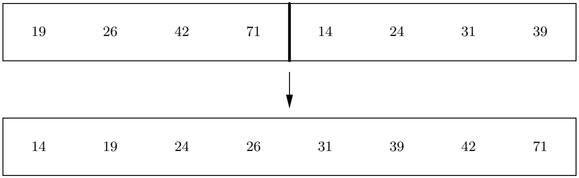
- Split array into two equal subarrays
- Sort both subarrays (recursively)
- Merge two sorted subarrays



```
def mergeSort(A,first,last):
    if first < last:
        mid = ⌊(first + last)/2⌋
        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

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

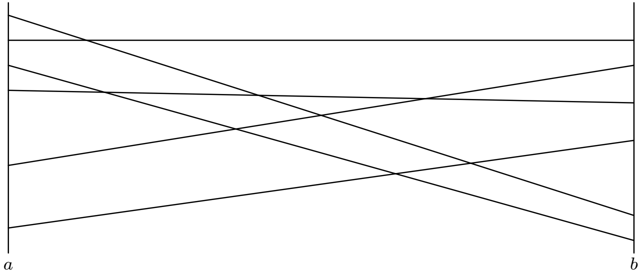
The exact solution of this recurrence equation is

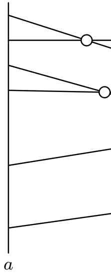
$$T(n) = n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$$

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

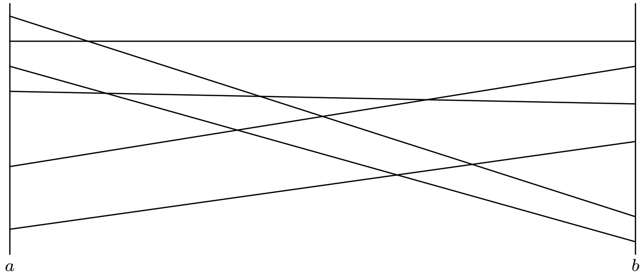


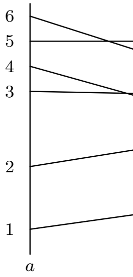


Checking every pair of lines takes $\Theta(n^2)$ time. We can do better.

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

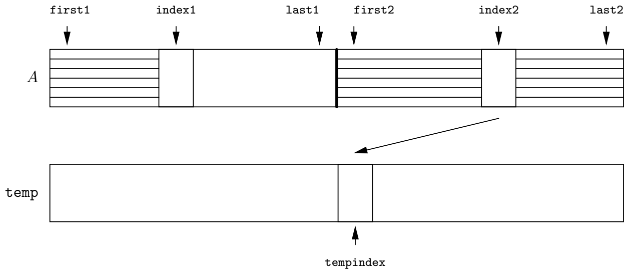
Sorting is the process of removing inversions. So to count inversions:

- 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

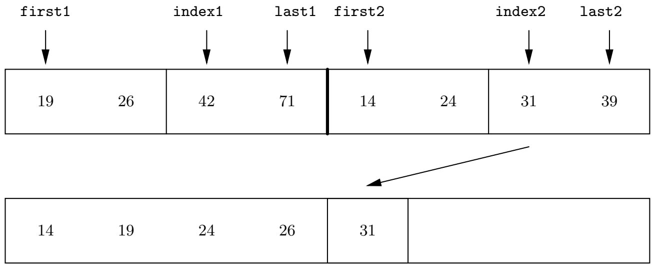
In Mergesort, the only time we rearrange data is during the merge step.



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
```

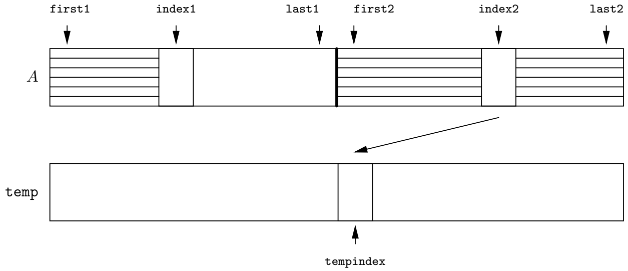
Pseudocode for MergeSort with inversion counting

```
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.