Lecture 5
Binary search (cont.), insertion/selection sort, analysis of quick sort

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

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)
```

Optimality of binary search

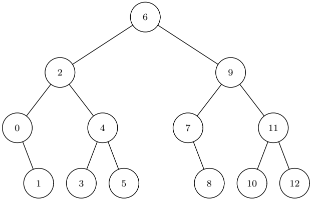
- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this 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

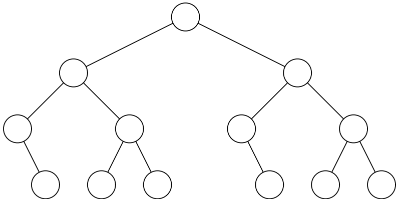
Consider any algorithm that searches for an item x in an array A of size n by comparing entries in A against x. Any such algorithm can be modeled as a decision tree:

- 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

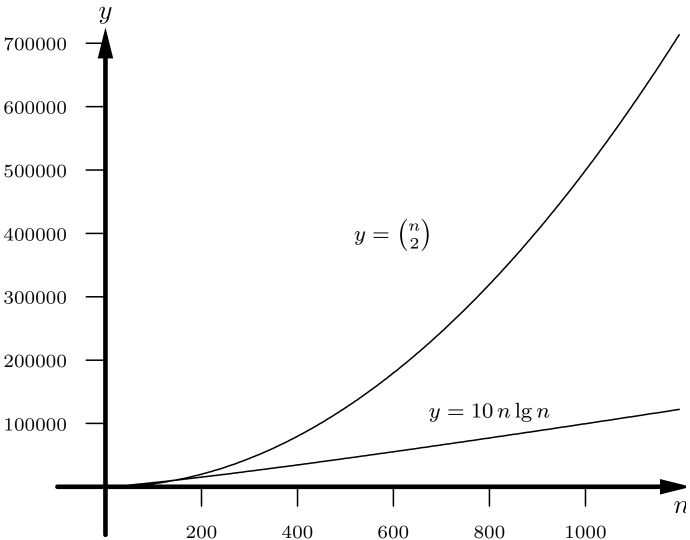
We will discuss in the class

- 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

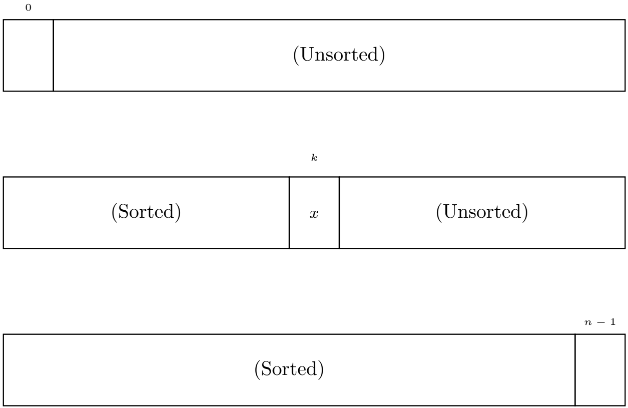
18 29 12 15 32 10

has 9 inversions:

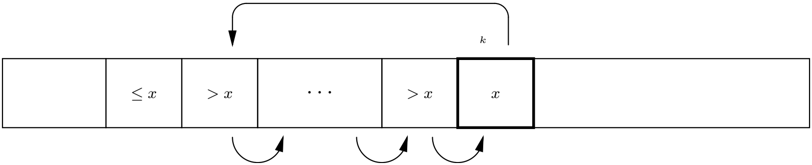
$$\{(18,12), (18,15), (18,10), (29,12), (29,15), (29,10), (12,10), (15,10), (32,10)\}$$

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

23	19	42	17	85	38

23	19	42	17	85	38

19	23	42	17	85	38

19	23	42	17	85	38

17	19	23	42	85	38

17	19	23	42	85	38

17	19	23	38	42	85

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

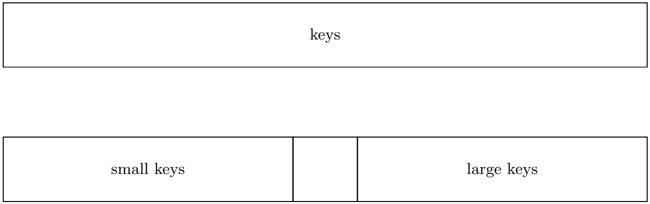
- Two variants:
  1. Repeatedly (for i from 0 to $n-1$) find the minimum value, output it, delete it.
     - Values are output in sorted order
  2. Repeatedly (for i from $n-1$ down to 1)
     - Find the maximum of $A[0]$,$A[1]$,...,$A[i]$.
     - Swap this value with $A[i]$ (no-op if it is already $A[i]$).
- Both variants run in $O(n^2)$ time if we use the straightforward approach to finding the maximum/minimum.

Quicksort

Quicksort

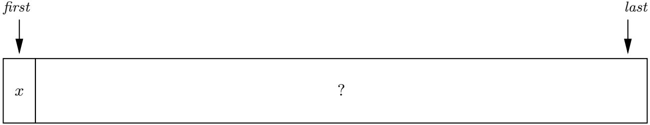
Basic idea

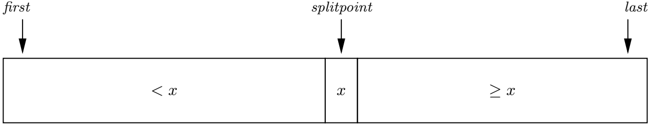
- Classify keys as small keys or large keys. All small keys are less than all large keys
- Rearrange keys so small keys precede all large keys.
- Recursively sort small keys, recursively sort large keys.



Quicksort: One specific implementation

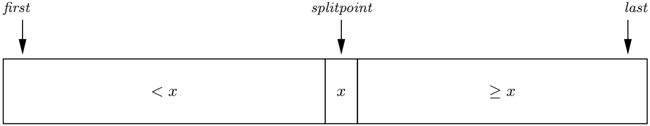
- Let the first item in the array be the pivot value x (also call the split value).
  - Small keys are the keys $< x$.
  - Large keys are the keys $\geq x$.





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

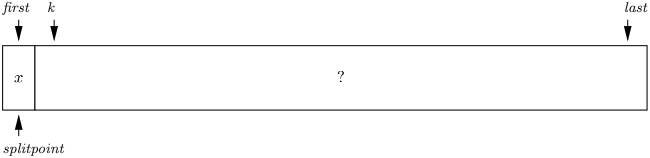
```
def split(A,first,last):
    splitpoint = first
    x = A[first]
    for k = first+1 to last do:
        if A[k] < x:
            A[splitpoint+1] ↔ A[k]
            splitpoint = splitpoint + 1
    A[first] ↔ A[splitpoint]
    return splitpoint
```

Loop invariants:

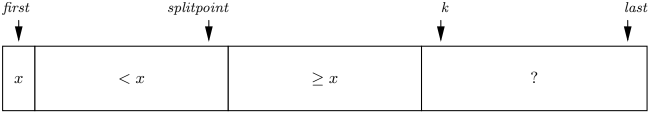
- `A[first+1..splitpoint]` contains keys $< x$.
- `A[splitpoint+1..k-1]` contains keys $\geq x$.
- `A[k..last]` contains unprocessed keys.

The split step

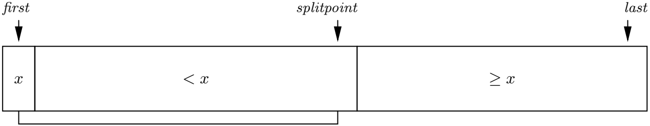
At start:



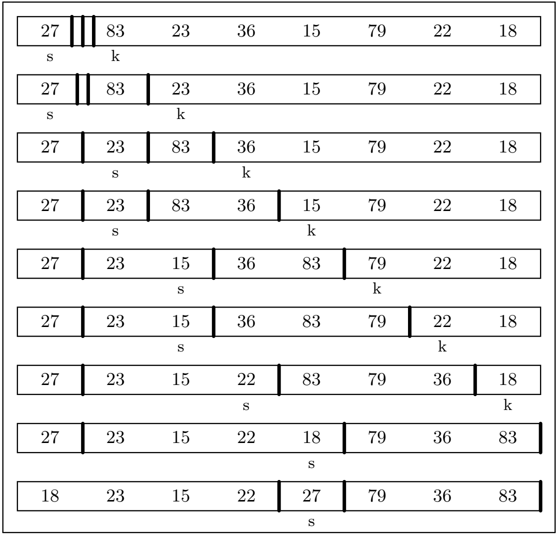
In middle:



At end:



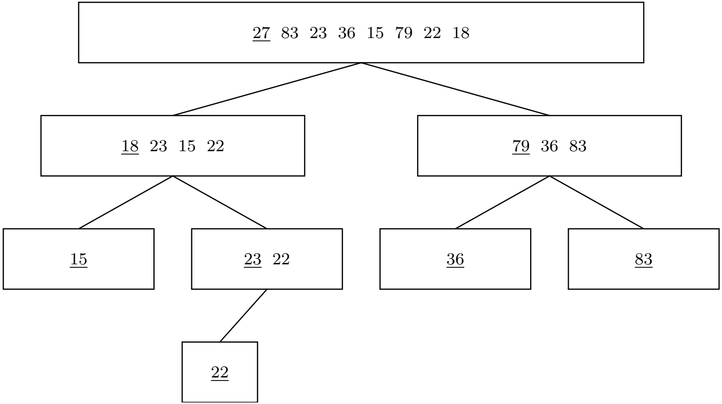
Example of split step



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.



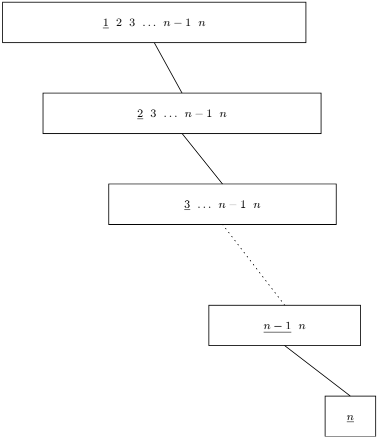
Worst-case Analysis of Quicksort

- 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: $1, 2, 3, \ldots, n-1, n$



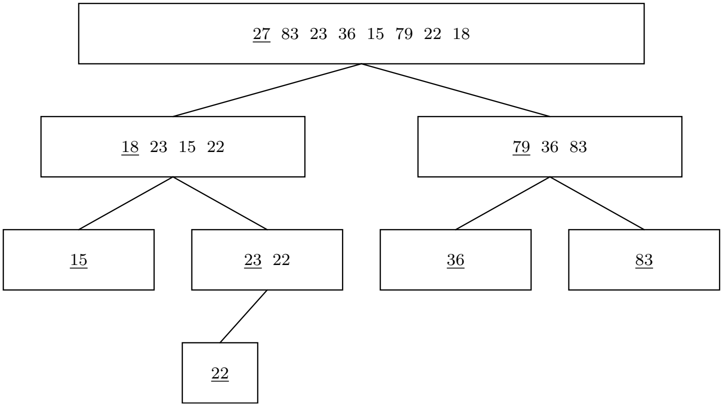
$\binom{n}{2}$ comparisons required. So the worst-case running time for Quicksort is $\Theta(n^2)$. But what about the average case ...?

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: 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:

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

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

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

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

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