COMP 8547
Advanced Computing Concepts

Dr. Olena Syrotkina

Chapter 4 – Sorting

Contents

- Sorting
- Merge sort
- Quicksort
- Heap sort
- Lower bounds for sorting
- Selection
- Bucket sort
- Radix sort
- Counting sort

Sorting

- Sorting is one of the most important problems in computer science
- Applications are innumerable
  - Databases, data compression, coding, networking, data security, bioinformatics, etc, etc
  - In most cases, sorting is part of other algorithms and methods
- Main idea
  - Given a sequence of objects $S = s_1, s_2, \ldots, s_n$
  - Possibly stored in an array or a linked list
  - Output a list of sorted objects $S = s_{i1}, s_{i2}, \ldots, s_{in}$, such that

$$s_{i1} \leq s_{i2} \leq \ldots \leq s_{in}$$

- Objects can be arbitrary or composite objects, as long as they are comparable
  - Last name + first name
  - City + province
  - Integers, points on a plane, dates, etc, etc.

Types of sorting approaches

Comparison-based

- Objects compared based on a Comparator

Smith Jack < Smith John
2014-09-08 < 2014-09-09
2.0 < 2.05
2 < 14 or "14" < "2 "
(3,4) < (3,5)

Algorithms

- Mergesort
- Quicksort
- Heap sort
- Shell sort
- Insertion/selection

Index-based

- Objects (or attributes) placed in placeholders based on an Index

25, 30, 83, …

0	30
1	
2	
3	83
4	
5	25
…	

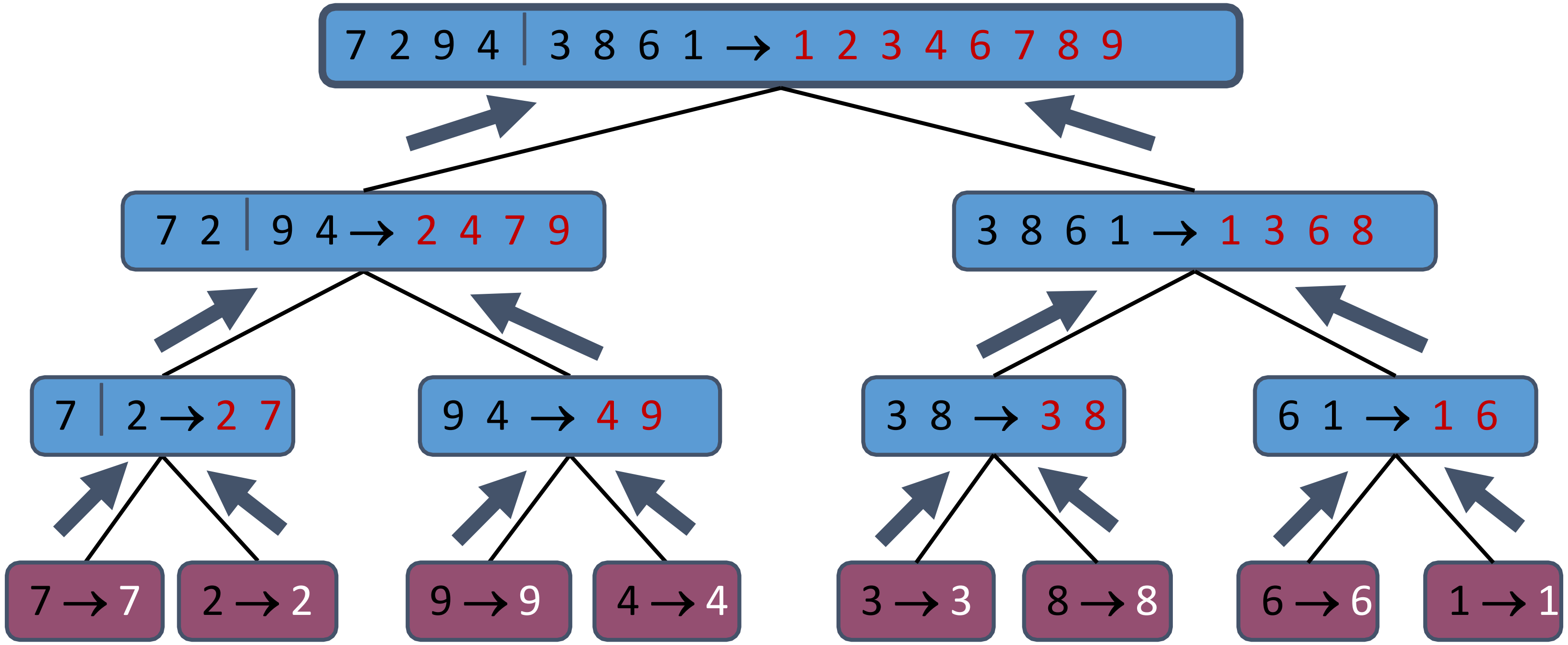
Algorithms

- Bucket sort
- Radix sort
- Lexicographic sort
- Counting sort

Mergesort

- Mergesort uses the principles of the divide-and-conquer paradigm
- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the sub-problems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are sub-problems of size 0 or 1

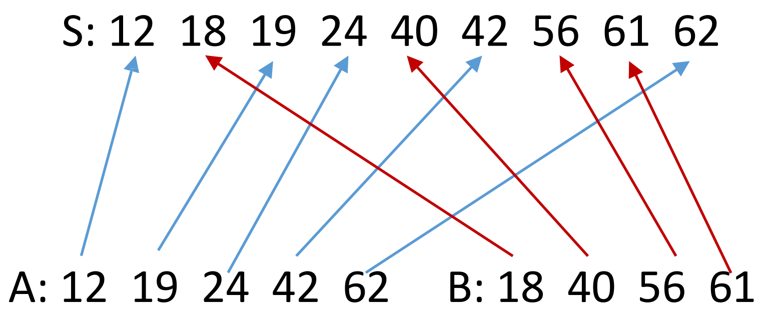
Algorithm mergeSort(S, C)
Input: sequence S with n objects and comparator C
Output: sequence S sorted according to C
if S.size() > 1 **then**
    (S_1, S_2) $\leftarrow$ partition(S, n/2)
    mergeSort(S_1, C)
    mergeSort(S_2, C)
    S $\leftarrow$ merge(S_1, S_2)

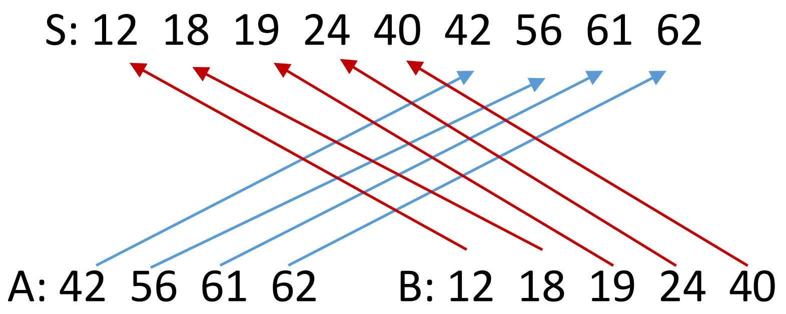


Mergesort - Merge

- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements takes $O(n)$ time

```
Algorithm merge(A, B, S)
    Input seq. A and B with n/2 objects each
    Output S = sorted sequence of A ∪ B
    S ← empty sequence
    i ← j ← 0
    while i < A.size()  and  j < B.size()
        if A.get(i) < B.get(j)
            S.insertLast(A.get(i))
            i ← i + 1
        else    // B.get(j) ≤ A.get(i)
            S.insertLast(B.get(j))
            j ← j + 1
    while  i < A.size()
        S.insertLast(A.get(i)); i ← i + 1
    while  j < B.size()
        S.insertLast(B.get(j)); j ← j + 1
    return S
```

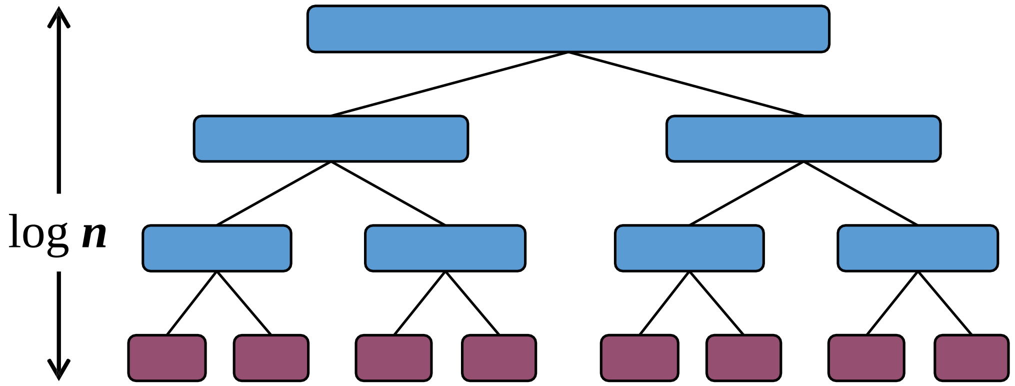




Mergesort - performance

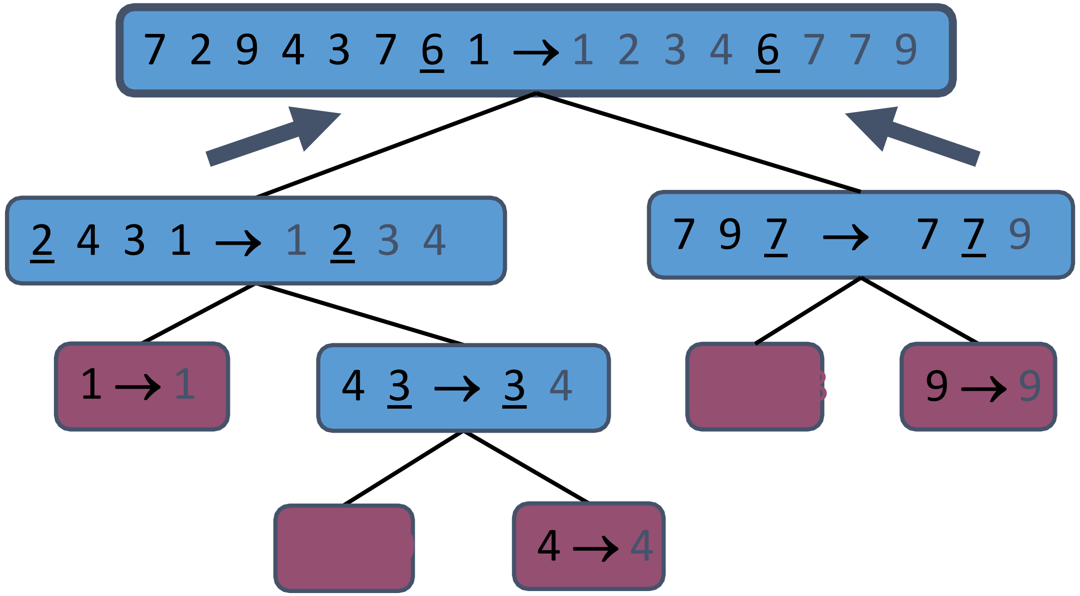
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we split the sequence into two halves
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total worst-case running time of merge-sort is $O(n \log n)$
- Complexity of mergesort can be analyzed using recurrences (studied later)
- In-place mergesort is rather complex and not suitable for practical applications

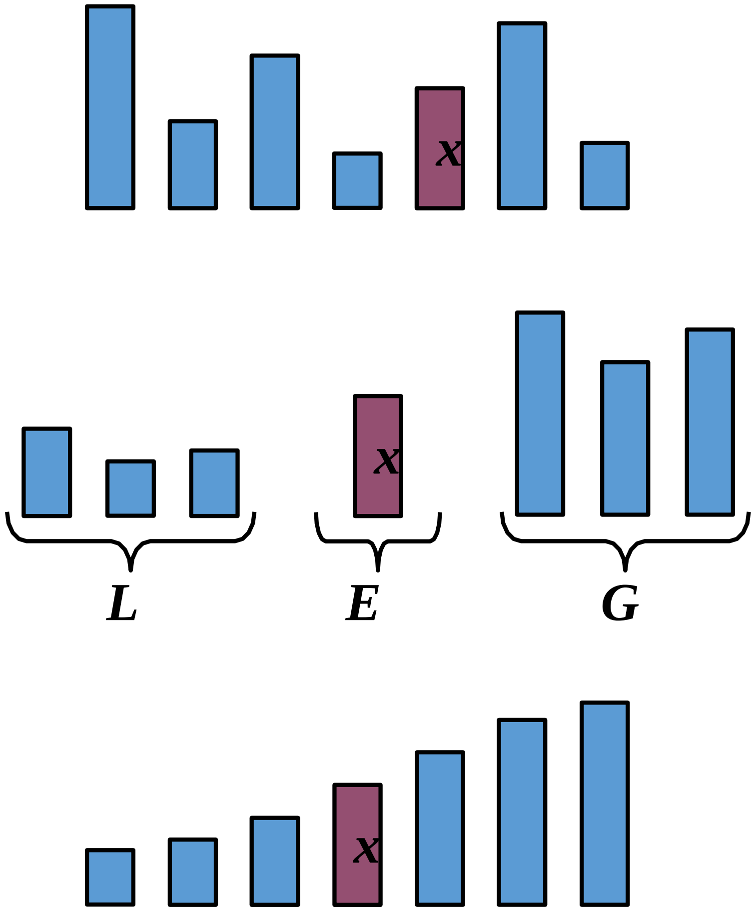
depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Quicksort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random pivot x and partition S into
    - L elements less than x
    - E elements equal to x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G





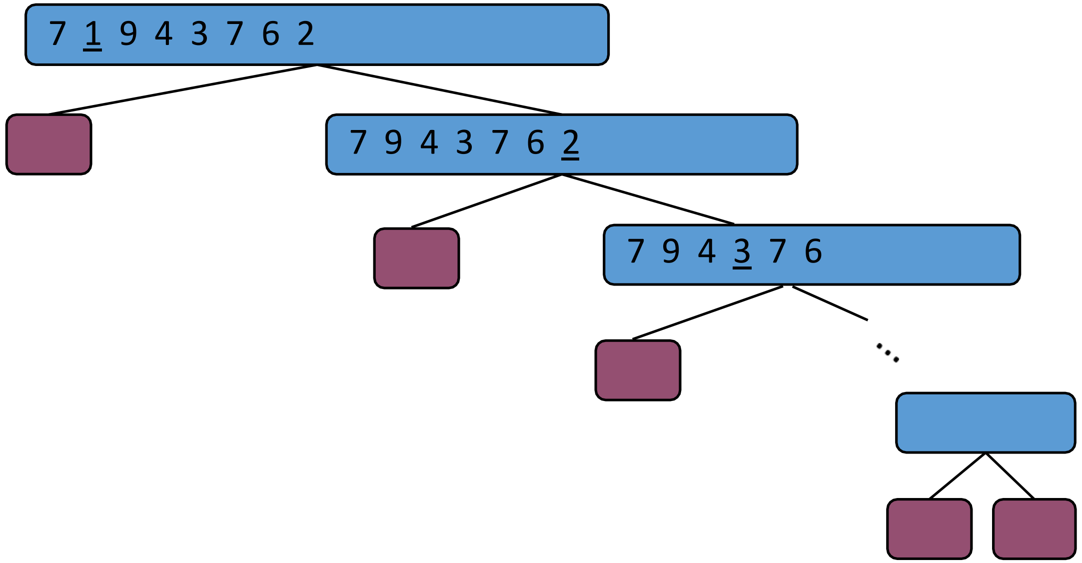
Worst-case analysis of Quicksort

- The worst case for Quicksort occurs when the pivot is the unique minimum or maximum element
- One of $\boldsymbol{L}$ and $\boldsymbol{G}$ has size $\boldsymbol{n} - 1$ and the other has size 0
- The running time is proportional to the sum (comparisons)

$$(\mathrm{n}-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$$

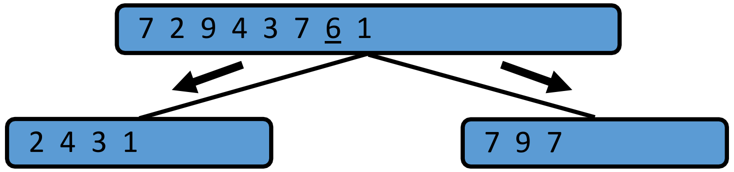
- Thus, the worst-case running time of Quicksort is $\boldsymbol{O}(\boldsymbol{n}^2)$

depth	time
0	$\boldsymbol{n} - 1$
1	$\boldsymbol{n} - 2$
…	…
$\boldsymbol{n} - 2$	1

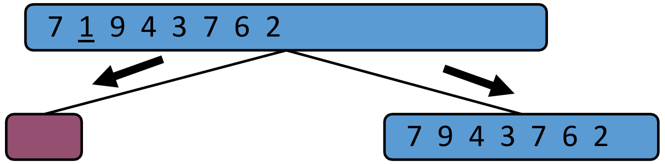


Average-case analysis of Quicksort

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size $\geq 3s/4$

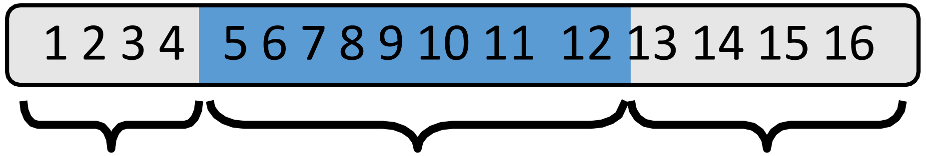


Good call



Bad call

- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Bad pivots **Good pivots** **Bad pivots**

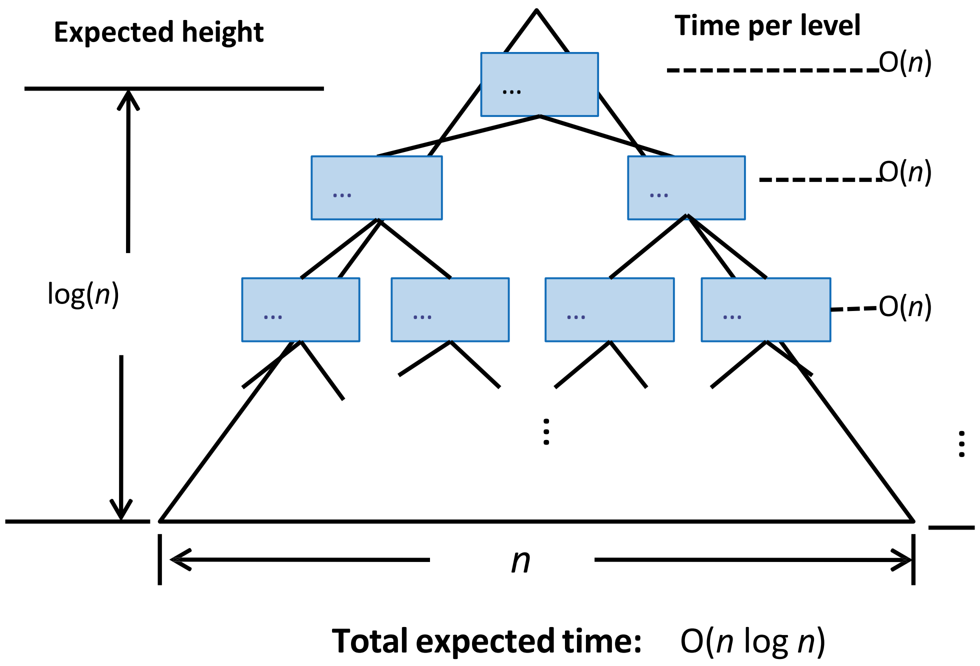
Average-case analysis of Quicksort

- Probabilistic Fact: The expected number of coin tosses required in order to get $\boldsymbol{k}$ heads is $2\boldsymbol{k}$
- For a node of depth $\boldsymbol{i}$, we expect
  - $\boldsymbol{i}/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{\boldsymbol{i}/2}\boldsymbol{n}$

◆ Therefore, we have
  - For a node of depth $2\log_{4/3}\boldsymbol{n}$, the expected input size is 1
  - The expected height of the Quicksort tree is $\boldsymbol{O}(\log \boldsymbol{n})$

◆ The amount of work done at the nodes of the same depth is $\boldsymbol{O}(\boldsymbol{n})$

◆ Thus, the average-case running time of Quicksort is $\boldsymbol{O}(\boldsymbol{n}\log \boldsymbol{n})$



$$\log_{4/3} n = \frac{\log_2 n}{\log_2 4/3} = \frac{1}{\log_2 4/3}\log_2 n$$

In-place Quicksort

- Quick-sort can be implemented to run in-place
- The pivot is the last element, but can be any other
- In the partition step, we use swap operations to rearrange the elements of the input sequence such that
  - the elements $\leq$ the pivot have rank less than $\boldsymbol{l}$
  - the elements $\geq$ the pivot have rank greater than $\boldsymbol{l}$
- The recursive calls consider
  - elements with rank less than $\boldsymbol{l}$
  - elements with rank greater than $\boldsymbol{l}$
- Note:
  - For in-place we "place" in L elements $\leq$ pivot (not just <)
- In-place Quicksort runs in:
  - O(n log n) average case
  - O(n^2) worst case
- Dual-pivot Quicksort runs faster than using a single pivot [4]

Algorithm *inPlaceQuickSort*(S, a, b)

Input sequence S, ranks a and b

Output sequence S with elements between a and b sorted

if $a \geq b$ **then return**

$p \leftarrow S[b]$ // the pivot (last element)

$l \leftarrow a$; $r \leftarrow b - 1$

while $l \leq r$ **do**

  while $l \leq r$ **and** $S[l] \leq p$ **do**

    $l \leftarrow l + 1$

  while $l \leq r$ **and** $S[r] \geq p$ **do**

    $r \leftarrow r - 1$

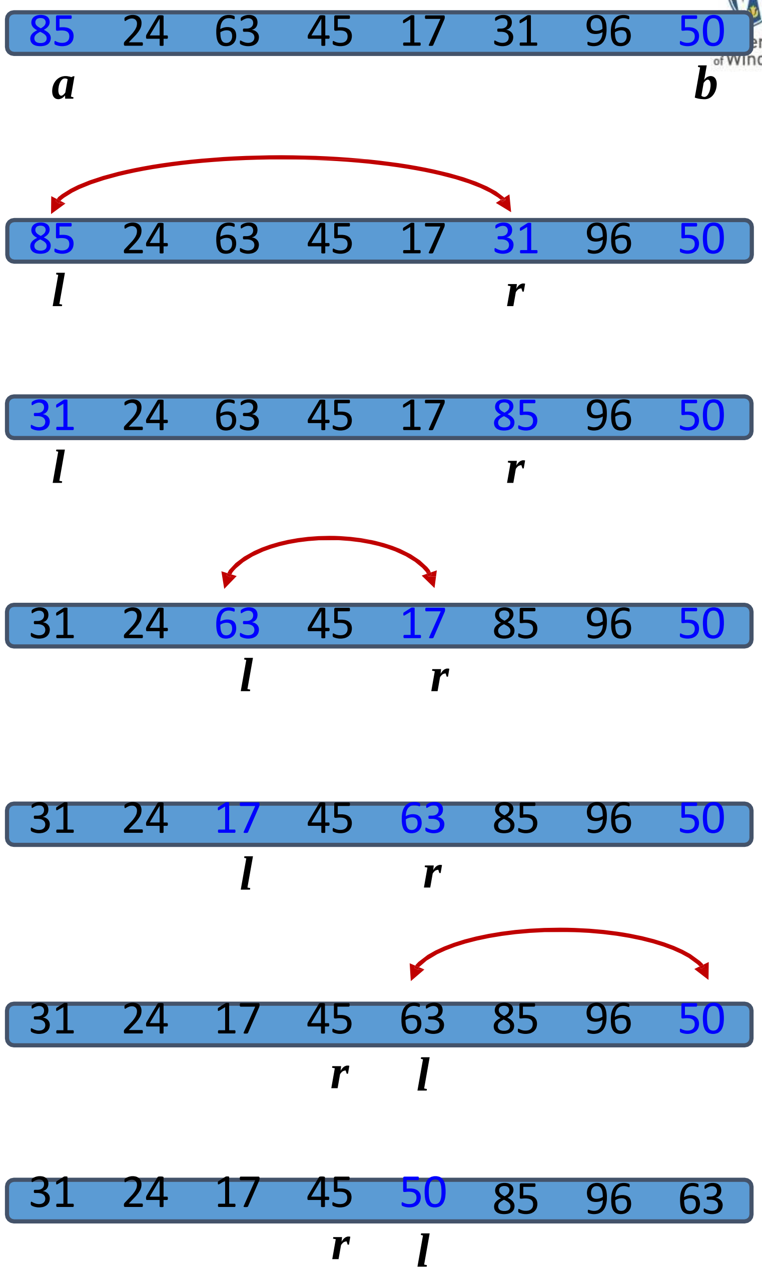
  if $l < r$ **then**

    swap $S[l]$ and $S[r]$

swap $S[l]$ and $S[b]$ // relocate pivot

inPlaceQuickSort(S, a, $l - 1$)

inPlaceQuickSort(S, $l + 1$, b)



Heapsort

- Priority queue sorting uses a priority queue to sort a set of comparable elements
  - Insert n objects with a series of insert operations
  - Remove n objects in sorted order with a series of removeMin operations
- Heapsort uses a heap to sort n objects
  - the space used is O(n)
  - methods insert and removeMin take O(log n) time
  - Heapsort runs in O(n log n)

Algorithm *PQ-Sort*(*S*, *C*)
    Input sequence *S*, comparator *C* for the objects of *S*
    Output sequence *S* sorted in increasing order according to *C*
    P ← priority queue with comparator *C*
    while ¬*S.isEmpty* ()
        e ← *S.remove* (*S. first* ())
        P.insertItem(*e*, *e*)
    while ¬*P.isEmpty*()
        e ← *P.removeMin*()
        S.insertLast(*e*)

In-place Heapsort

- Uses an array to store both the heap and the sequence
- Sorting in increasing order: Use a reverse comparator (largest key is at the top/root of the heap)
- Sorting in decreasing order: Keep minimum at top/root
- Phase 1: Start with an empty heap and:
  - Move the boundary from left to right, one step at a time
  - At step i, expand the heap by adding the element at index i
- Phase 2: Start with an empty sequence:
  - Move boundary between heap and sequence from right to left, one step at a time
  - At step i, remove max/min element from the heap and store it at index n – i + 1

In-place Heapsort – Example: sort S = 6, 2, 7, 9, 5

University of Windsor

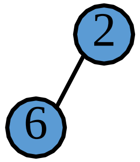
Phase 1:

Insert 6:

	6	2	7	9	5	
0	1	2	3	4	5	

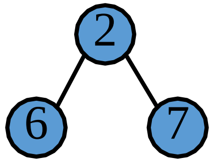
Insert 2:

	6	2	7	9	5	
0	1	2	3	4	5	



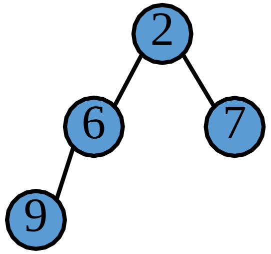
Insert 7:

	2	6	7	9	5	
0	1	2	3	4	5	



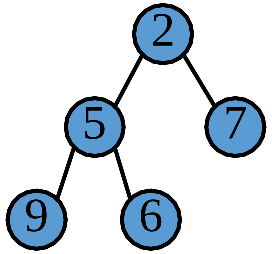
Insert 9:

	2	6	7	9	5	
0	1	2	3	4	5	



Insert 5 and UpHeap:

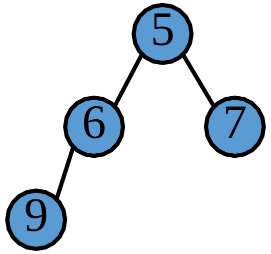
	2	5	7	9	6	
0	1	2	3	4	5	



Phase 2:

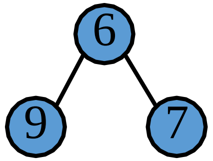
Remove 2 and DownHeap:

	5	6	7	9	2	
0	1	2	3	4	5	



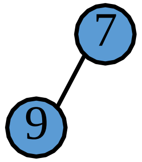
Remove 5 and DownHeap:

	6	9	7	5	2	
0	1	2	3	4	5	



Remove 6:

	7	9	6	5	2	
0	1	2	3	4	5	



Remove 7:

	9	7	6	5	2	
0	1	2	3	4	5	

Remove 9:

	9	7	6	5	2	
0	1	2	3	4	5	

Other comparison-based sorting algorithms

- Selection sort, Insertion Sort, Bubble sort
  - Simple but slow
  - Run in quadratic worst-case time
- Shellsort
  - Uses several sorting phases
  - Each sorting step sorts elements i_j positions apart
  - A sequence $i_1, i_2, \dots, i_k$ determines the positions to be sorted in each phase
  - Running time depends on the sequence
  - Hibbard's sequence (1,3,7, …,2^k-1) yields O($n^{3/2}$) worst case running time
  - Other analyses have been made to show some sequences yield O($n^{4/3}$)
- Shellsort is interesting only for academic purposes
- In practice, Quicksort and Heapsort are the preferred sorting algorithms

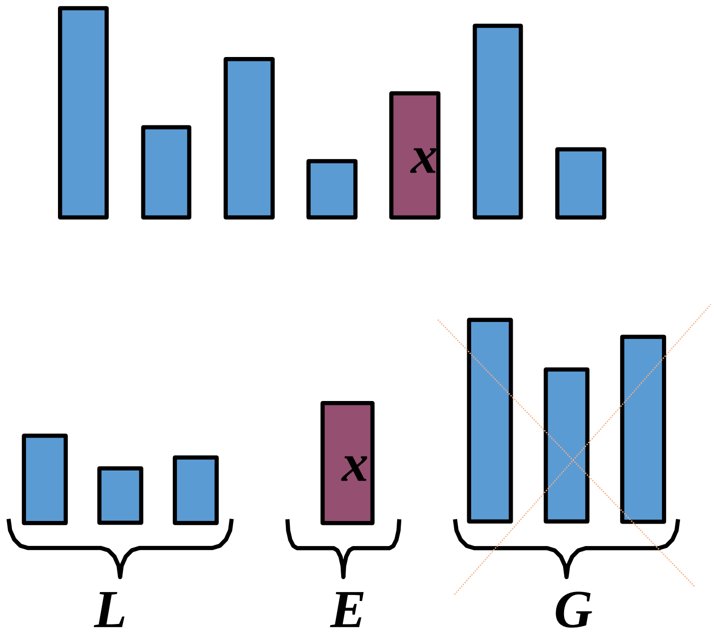
Summary of comparison-based sorting algorithms

Algorithm	Worst	Average	Best	Notes
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	◈ slow ◈ in-place ◈ for small data sets (< 1K)
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$	◈ slow ◈ in-place ◈ for small data sets (< 1K)
Shellsort	$O(n^{4/3})$	$O(n \log n)$	Depends on sequence	◈ Interesting for academic purposes only
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	◈ fast ◈ in-place ◈ for large data sets (1K — 1M)
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	◈ fast ◈ sequential data access ◈ for huge data sets (> 1M)
Quick-sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	◈ fast ◈ in-place ◈ for large data sets (1K – 1M)

Selection

- Problem: given an unsorted list S of n keys, find the k^{th} smallest key
- Already seen:
  - Naïve approach
  - Heap-based approach
- Prune-and-search
  - Based on the principles of divide-and-conquer + pruning
  - Algorithm: Randomized quick select
  - It selects the k^{th} smallest key in O(n) average-case running time
  - Worst-case running time could be dropped to O(n) too.
    - ➢ But hidden constants make the algorithm not efficient in practice
  - Randomized quick select can be implemented in-place

Algorithm quickSelect(S, k)
Input: A sequence S of n keys and k
Output: The k^{th} smallest key of S
if $n = 1$ **then**
    return $S[0]$
Pick a random pivot x of S
Divide S into three susbseq:
    L with elements $< x$
    E with elements $= x$
    G with elements $> x$
if $k \le |L|$ **then**
    return quickSelect(L, k)
else if $k \le |L| + |E|$ **then**
    return x
else
    return quickSelect(G, $k-|L|-|E|$)



Prune G if $k \le |L| + |E|$
This is done recursively on smaller subsequences

Selection – lower bounds

Given a sequence S of n keys, or n comparable objects

To find:

- The smallest key:
  - Any comparison-based algorithm needs at least n – 1 comparisons
- The two smallest keys:
  - At least $n + \lceil \log n \rceil - 2$ comparisons are needed
- The median:
  - At least $\lceil 3n/2 \rceil - O(\log n)$ comparisons are needed
- The k^{th} smallest key:

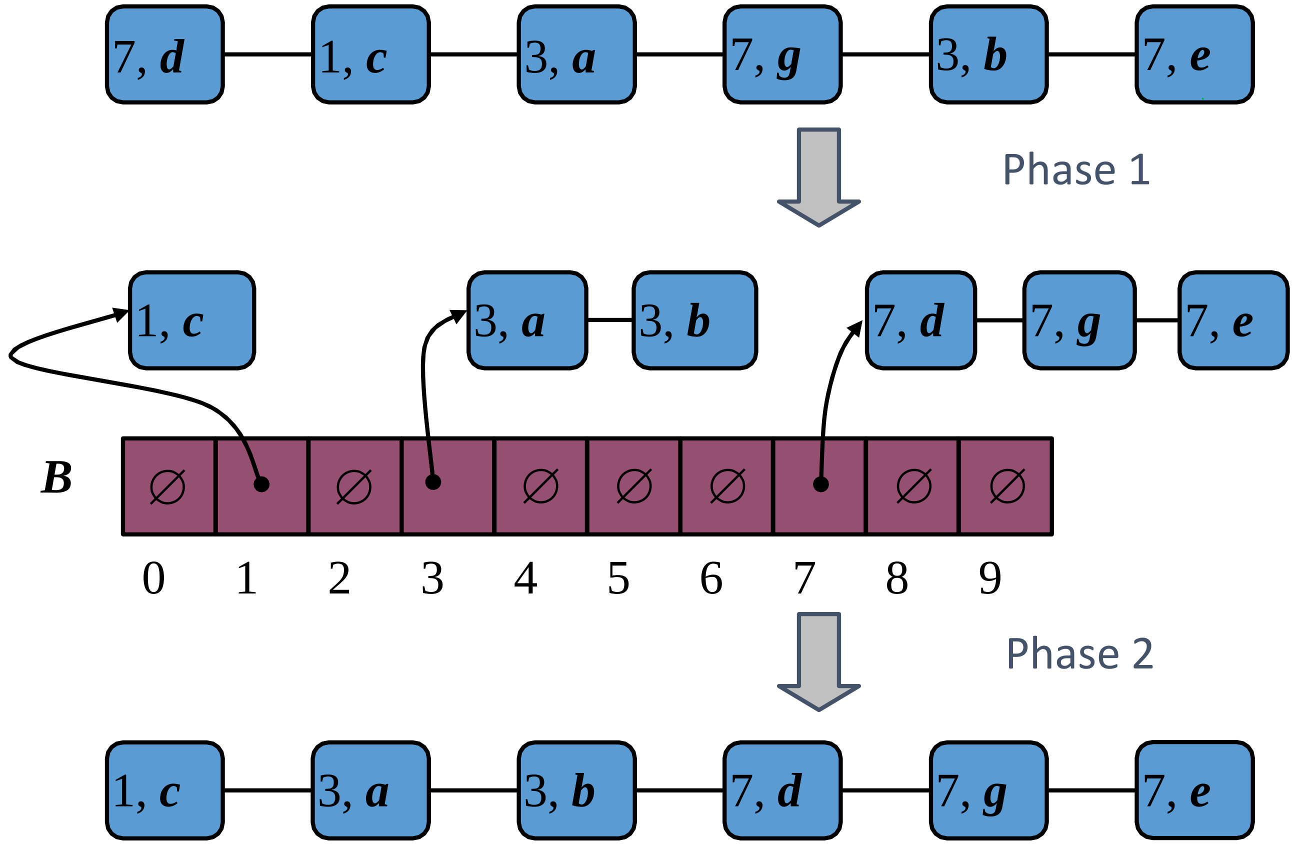
  At least $n - k + \left\lceil \log \binom{n}{k-1} \right\rceil$ comparisons are needed
- The minimum and maximum:
  - At least $\lceil 3n/2 \rceil - 2$ comparisons are needed

Bucket sort

- Let be S be a sequence of n (key, element) entries with keys in the range $[0, N - 1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N - 1$, move the entries of bucket $B[i]$ to the end of sequence S
- Performance:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

  Bucket-sort takes $O(n + N)$ time
- **Stable** sort property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm
  - Bucket sort is stable

Example: Key range: $[0, 9]$



Radix sort

- Radix-sort is a specialization of lexicographic-sort that uses Bucket sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, N-1]$
- Radix-sort runs in worst case time: $O(d(n+N))$

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm *radixSort*(S, N)

    Input sequence S of d-tuples such that $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \le (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S

    Output sequence S sorted in lexicographic order

    for $i \leftarrow d$ **downto** 1

        bucketSort(S, N)

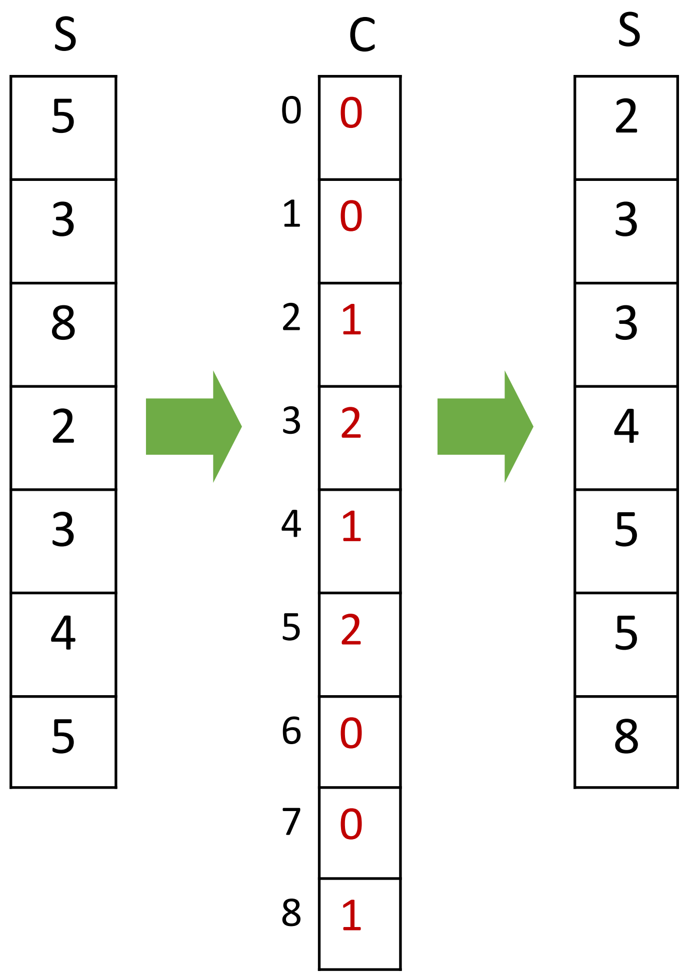
- If d = log n and N = O(n), Radix sort runs in O(n log n)
- If d is constant and N = O(n), Radix sort runs in O(n)

Note: All these depend on the assumptions we make about the keys. Comparison-based sorting algorithms make no assumptions about the keys

Counting sort

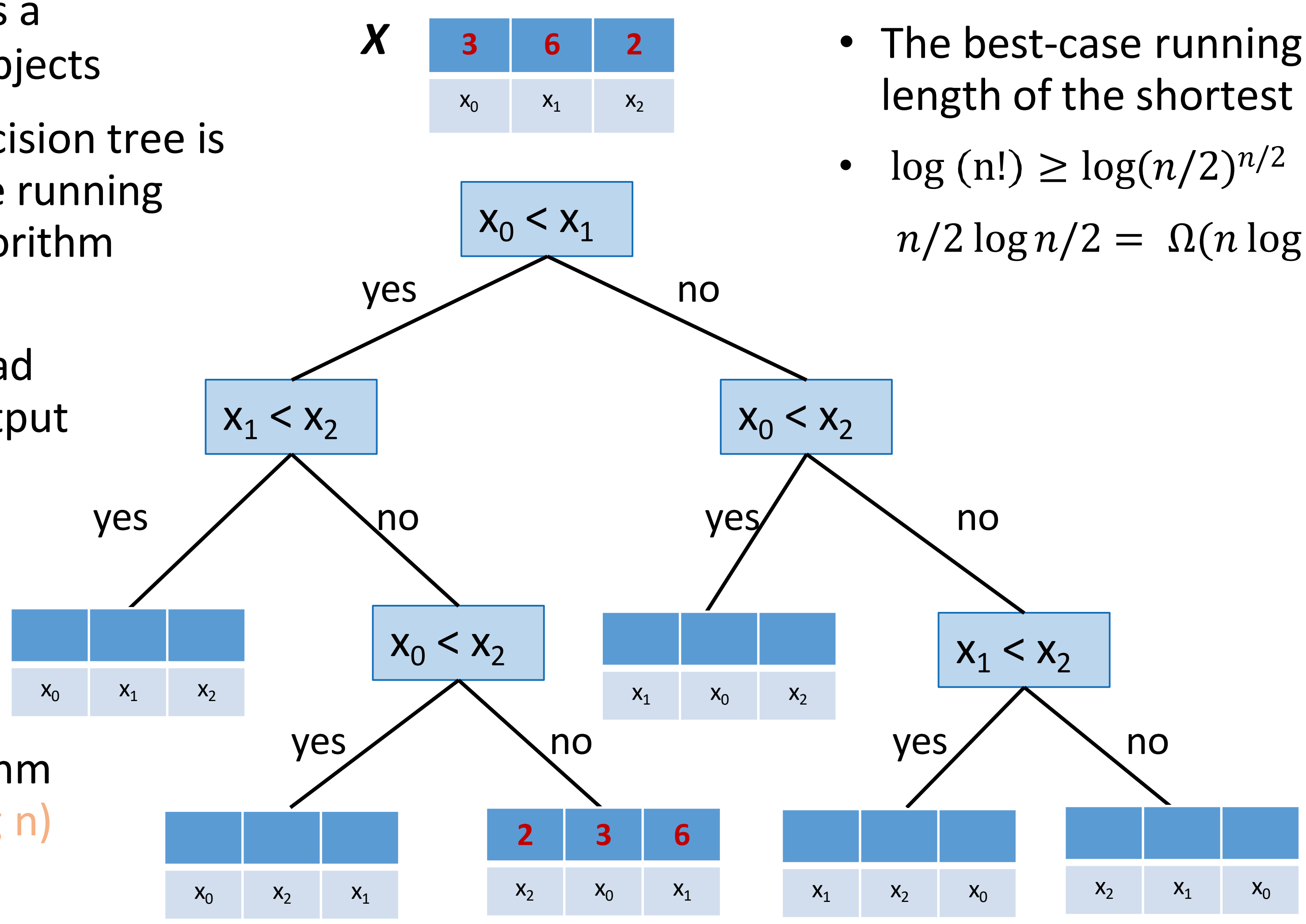
- Counting sort is a simplification (particular case) of Radix sort
- Instead of using buckets, it uses N “counters”
- The sorted list is obtained directly from the counters
- The worst-case running time is O(n + N)
- If N = O(n), the running time is O(n)
- Counting sort can be applied to arrays of positive integers
- Strings and other types have to be converted to integers

Algorithm Counting-sort(S, N)
Input: Sequence S of integers in range [0, N-1]
Output: Sorted list S
Create an array C of N counters
for $i \leftarrow 0$ **to** $n - 1$ **do**
  $C[S[i]] \leftarrow C[S[i]] + 1$
$i \leftarrow 0$; $j \leftarrow 0$
while $i < N$ **do**
  if $C[i] > 0$ **then**
    $S[j] \leftarrow i$
    $j \leftarrow j + 1$
    $C[i] \leftarrow C[i] - 1$
  else
    $i \leftarrow i + 1$



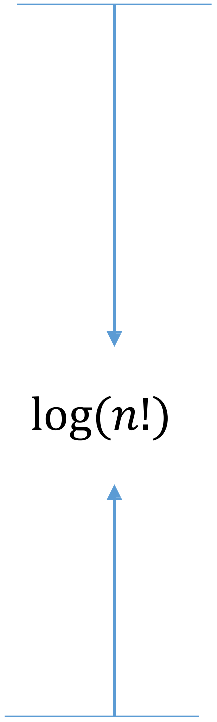
Comparison-based sorting – lower bound (Optional; not in exam)

- A decision tree can be used to derive the lower bound
- Each node represents a comparison of two objects
- The height of this decision tree is a lower bound on the running time of a specific algorithm
- Every possible input permutation must lead to a separate leaf output
- Since there are n! leaves, the height of the tree is at least log (n!)
- Any comparison-based sorting algorithm takes at least Ω(n log n) time



- The worst-case running time for an algorithm is the length of the longest path
- The best-case running time is the length of the shortest path
- $\log(n!) \geq \log(n/2)^{n/2}$

  $n/2 \log n/2 = \Omega(n \log n)$



Review and Further Reading

- Class Collections in Java 8 provides an implementation of Mergesort in its sort method. Implementers can change the sorting algorithm, provided they use a stable sorting method
  - http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html
- Class Arrays in Java 8 provides an implementation of Quicksort (dual pivot version) in its sort method
  - http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html
- Heapsort, Quicksort, Mergesort:
  - Sec. 7.5, 7.6, 7.7 of [3], Sec. 9.4, 12.1, 12.2 of [2], Secs. 5.4, 8.1 and 8.2 of [1]
- Radix sort:
  - Sec. 12.3 of [2]
- Bounds on sorting:
  - Sec. 7.8 of [3], Sec. 12.3 of [2], Sec. 8.3 of [1]
- Selection:
  - Sec. 7.9 of [3], Sec. 12.5 of [2], Sec. 9.2 of [1]

References

1. Algorithm Design and Applications by M. Goodrich and R. Tamassia, Wiley, 2015.
2. Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014.
3. Data Structures and Algorithm Analysis in Java, 3rd Edition, by M. Weiss, Addison-Wesley, 2012.
4. Dual-pivot (triple, quad, penta-pivot) Quicksort
   - https://iopscience.iop.org/article/10.1088/1757-899X/180/1/012051/pdf
5. Java documentation:
   - http://docs.oracle.com/javase/8/docs/api/overview-summary.html
   - http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html
   - http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

Lab – Practice

1. Use class Sort.java provided in the code, the dual-pivot Quicksort of Java 7 (Arrays.sort), and RadixSort.java.
2. Repeat the following for Mergesort, Quicksort, Heapsort and dual-pivot Quicksort:
   a) Create one million random keys (of type long) and sort them
   b) Repeat (a) 10 times
   c) Compute the average CPU time taken to sort the keys for the four methods
   d) Comment on the results and compare them to the average-case complexities
3. Do the following for the four sorting methods of #2, and for Radix sort
   a) Create one million random strings of length 4 and sort them using the five sorting methods
   b) Repeat (a) 100 times Compute the average CPU time taken to sort the keys for the five methods
   c) Repeat (a) and (b) with strings of length 6, 8, ..., 10

Exercises

1. Sort sequence 9,8,7,6,5,4,3,2,1 using all sorting methods seen in class. Comment. Do the same with 1,2,3,4,5,6,7,8,9
2. Sort sequence S = 1, 2, 4, 5, 3, 7, 8, 10, 11, 9, 6 using Mergesort, Quicksort and Heapsort. Comment on the comparisons used and the complexities of the algorithms.
3. Repeat #2 for quicksort by choosing the pivot as the (a) first element of list, and (b) the element in the middle. Comment on the running time.
4. What is the running time of Mergesort, Quicksort and Heapsort if all elements are equal?
5. What is the running time of Radix sort and Counting sort if all n numbers are in range [1,n] and are all equal?
6. For the in-place quicksort, derive the running time for: (a) all numbers are sorted in increasing order, (b) all numbers are in decreasing order, (c) the numbers have a random order. Do the same for Mergesort and Heapsort.
7. Write an algorithm that implements Bucket sort.
8. Using the implementation of Quickselect, write an algorithm that finds the k smallest keys. Implement the algorithm in Java. What is the worst-case running time of your algorithm? And the average-case.
9. Give an example of 8 integers that exemplifies the worst-case input of: Quicksort, Mergesort, and Heapsort.
10. *Suppose that quicksort receives "depth" as a parameter, which represents the depth of the tree. When the depth of the tree is 2*log n, the algorithm is changed to run heapsort to sort that sublist. Implement the algorithm and show that its worst-case running time is O(n log n).
11. * Modify quicksort to do heapsort on the largest sub-list after partitioning. The smallest sub-list is then sorted using quicksort recursively. Does the complexity of this new quicksort change? Why/why not?
12. *Design an algorithm for finding the minimum and the second smallest of a list, and which uses the smallest number of comparisons.
13. Given 2^5 integer keys in the range $[0..\ 2^{32}-1]$. What is the running time of Radixsort? And counting sort? Give asymptotic values, disregarding the constants.
14. For #13, do the same for Quicksort, Mergesort and Heapsort.