Data Structures – LECTURE 4

Comparison-based sorting

- Why sorting?
- Formal analysis of Quick-Sort
- Comparison sorting: lower bound
- Summary of comparison-sorting algorithms

Sorting

Definition

Input: A sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$

Output: A permutation (reordering)

$(a_1, \ldots, a'_n)$ such that $a'_1 \leq \ldots \leq a'_n$

Why sorting?

- – Fundamental problem in Computer Science
- – Many algorithms use it as a key subroutine
- – Wide variety with a rich set of techniques
- – Known lower bounds, asymptotically optimal
- – Many programming and implementation issues come up!

Sorting algorithms

Two types of sorting algorithms:

1. Comparison sorting: the basic operation is the comparison between two elements: $a_i \leq a_j$
   - – *Merge-Sort, Insertion-Sort, Bubble-Sort*
   - – *Quick-Sort*: analysis with recurrence equations
   - – Lower bounds for comparison sorting: $T(n) = \Omega(n \lg n)$ and $S(n) = \Omega(n)$
   - – *Heap Sort* with priority queues (later)
2. Non comparison-based: does not use comparisons!
   - – Requires additional assumptions
   - – Sorting in linear time: $T(\mathrm{n}) = \Omega(n)$ and $S(n) = \Omega(n)$

Quick-Sort

Uses a “Divide-and-Conquer” strategy:

- Split $A[Left..Right]$ into $A[Left..Middle - 1]$ and $A[Middle+1..Right]$
- Elements of $A[Left..Middle - 1]$ are smaller or equal than those in
- $A[Middle+1..Right]$
- Sort each part recursively

Quick-Sort(A, $Left$, $Right$)

```
1. if Left < Right then do
2.        Middle ← Partition(A, Left, Right)
3.        Quick-Sort(A, Left, Middle –1)
4.        Quick-Sort(A, Middle +1, Right)
```

Partition

Rearranges the array and returns the partitioning index

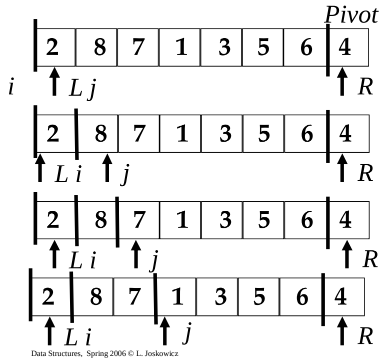
The partition is the *leftmost element larger than the last*

Partition(A, $Left$, $Right$)

```
1. Pivot ← A[Right]
2. i ← Left – 1
3. for j ← Left to Right–1
4.     do if (A[j] ≤ Pivot)
5.         then i ← i + 1
6.              Exchange(A[i], A[j])
7. Exchange (A[i+1], A[Right])
8. return i +1
```

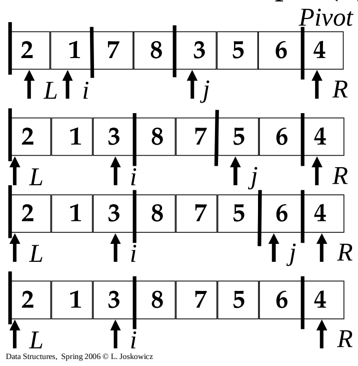
Example (1)

1st iteration



2 swapped with itself

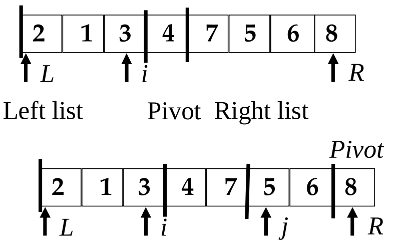
Example (2)



1 and 8 swapped

3 and 7 swapped

Example (3)



2nd iteration

Left list Pivot Right list

general pattern

Left list $A[i] \leq$ Pivot

Right list $A[i] >$ Pivot

Unrestricted list

Quick-Sort complexity

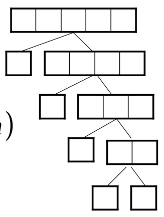
The complexity of Quick-Sort depends on whether the partitioning is balanced or unbalanced, which depends on which elements are used for partitioning

1. Unbalanced partition: there is no partition, so the sub-problems are of size $n - 1$ and 0.
2. Perfect partition: the partition is always in the middle, so the sub-problems are both of size $\leq n/2$.
3. Balanced partition: the partition is somewhere in the middle, so the sub-problems are of size $n - k$ and k.

Let us study each case separately!

Unbalanced partition

The recurrence equation is:

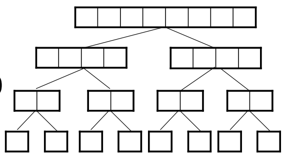


$T(n) = T(n - 1) + T(0) + \Theta(n)$

$$T(n) = T(n-1) + \Theta(n)$$
$$= \sum_{k=1}^{n} \Theta(k)$$
$$= \Theta\left(\sum_{k=1}^{n} k\right)$$
$$= \Theta(n^2)$$

Perfect partition

The recurrence equation is:



$T(n) \leq T(n/2) + T(n/2) + \Theta(n)$

$$T(n) = 2T(n/2) + \Theta(n)$$
$$T(n) = \Theta(n \lg n)$$

General case

The recurrence equation is:

$$T(n) = T(q) + T(n-q-1) + \Theta(n)$$
$$T(n) = \max_{0 \leq q < n} \{T(q) + T(n-q-1)\} + \Theta(n)$$

Average case is somewhere between unbalanced and perfect partition:

$$\Theta(n \lg n) \leq T(n) \leq \Theta(n^2)$$

which one dominates?

Example: 9-to-1 proportional split

- Suppose that the partitioning algorithm always produces a 9-to-1 proportional split.
- The complexity is:
  $T(n) = T(n/10) + T(9n/10) + \Theta(n)$
- At every level, the boundary condition is reached at depth $\log_{10} n$ with cost $\Theta(n)$. The recursion terminates at depth $\log_{10/9} n$
- Therefore, the complexity is $T(n) = O(n \lg n)$
- In fact, this is true for *any* proportional split!

Worst-case analysis: proof (1)

$$T(n) = \max_{1 \le q < n} \{T(q) + T(n-q)\} + \Theta(n)$$

Claim: $T(n) \le cn^2 = O(n^2)$

Proof:

Base of induction: True for n=1.

Induction step: Assume for $n < n'$, and prove for n'.

$$\begin{aligned} T(n') &= \max_{1 \le q < n'} \{T(q) + T(n'-q)\} + \Theta(n') \\ &\le cq^2 + c(n'-q)^2 + dn' \\ &= 2cq^2 + c(n')^2 - 2cqn' + dn' \end{aligned}$$

Worst-case analysis: proof (2)

To prove the claim, we need to show that this is smaller than $c(n')^2$, or equivalently that:

$$dn' \le 2cq\,(n'-q)$$

Since $q(n'-q)$ is always greater than $n/2$, as can be easily verified by checking the two cases:

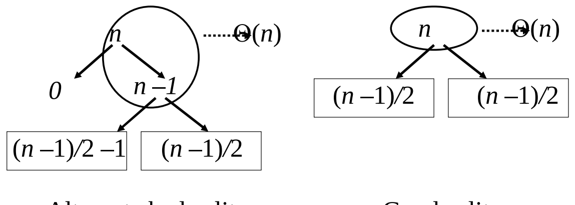
$$q < \tfrac{n}{2} \quad \text{or} \quad n \ge q \ge \tfrac{n}{2}$$

we can pick c such that the inequality holds.

Average case complexity

- We must first define what is an average case
- The behavior is determined by the *relative ordering* of the elements, not by the elements themselves.
- Thus, we are interested in the average of all permutations, where each permutation is equally likely to appear (uniformly random input).
- The average complexity is the number of steps averaged over a uniformly random input.
- The complexity is determined by the number of "bad splits" and the number of "good splits".

Bad splits and good splits -- intuition



Alternate bad split Good split

In both cases, the complexity is Θ(n). Thus the bad split was "absorbed" by a good one!

Randomization and average complexity

- One way of studying the average case analysis is to analyze the performance of a *randomized version* of the algorithm.
- In the randomized version, choices are made with a uniform probability, and this mimicks input generality – essentially, we reduce the chances of hitting the worst input!
- Randomization ensures that the performance is good without making assumptions on the input
- Randomness is one of the most important concepts and tools in modern Computer Science!

Randomized Quick-Sort

- Randomized Complexity: The number of steps, (for the WORST input !) averaged over the random choices of the algorithm.
- For Quick-Sort, the pivot determines the number of good and bad splits
- We chose the leftmost element to select a pivot. What if we choose instead *any element* randomly?
- In Partition, use *Pivot* ← *A*[Random(*Left,Right)*]
  instead of *Pivot* ← *A*[*Left*]
- Note that the algorithm remains correct!

Randomized complexity

- Randomized-case recurrence:

The pivot is equally likely to be in any place, and since there are *n* places, each case occurs in 1/*n* of the inputs.
We get:

$$T(n) = \frac{1}{n}\left(\sum_{q=1}^{n-1}\big(T(q)+T(n-q)\big)\right)+\Theta(n)$$

- This is "Recurrence with Full History", since it depends on all previous sizes of the problem.

It can be proven, using methods which we will not get into this time, that the solution for this recurrence satisfies:

Sorting with comparisons

- The basic operation of all the sorting algorithms we have seen so far is the comparison between two elements: $a_i \le a_j$
- The sorted order they determine is based only on comparisons between the input elements!
- We would like to prove that any comparison sorting algorithm must make Ω(*n* lg *n*) comparisons in the worst case to sort *n* elements (lower bound).
- Sorting without comparisons takes Ω(*n*) in the worst case, but we must make assumptions about the input.

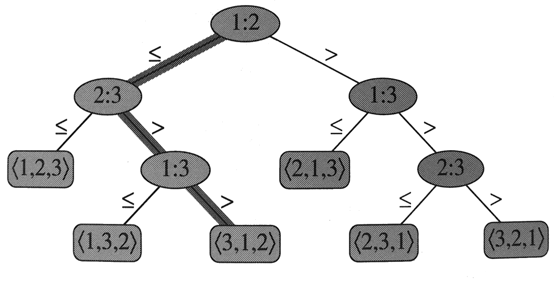
Comparison sorting – lower bound

- We want to prove a lower bound (Ω) on the worst-case complexity sorting for **ANY** sorting algorithm that uses comparisons.
- We will use the *decision tree model* to evaluate the number of comparisons that are needed in the worst case.
- Every algorithm A has its own decision tree T, depending on how it does the comparisons between elements.
- The length of the longest path from the root to the leaves in this tree T will determine the maximum number of comparisons that the algorithm must perform.

Decision trees

- A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular algorithm.
- The tree has *internal nodes, leaves, and branches*:
  - Internal node: two indices *i:j* for $1 \le i, j \le n$
  - Leaf*:* a permutation of the input $\pi(1), \ldots \pi(n)$
  - Branches: result of a comparison
    $a_i \le a_j$ (left) or $a_i > a_j$ (right)

Decision tree for 3 elements

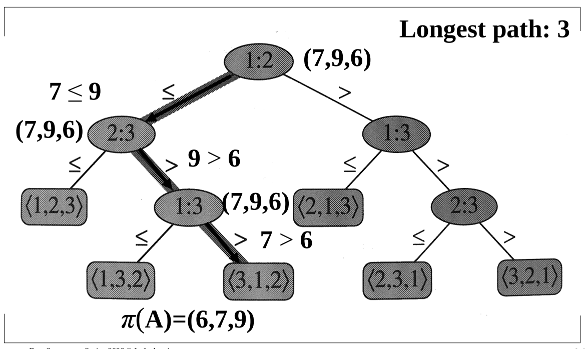


Paths in decision trees

The execution of sorting algorithm A *on input I corresponds to tracing a path in* T *from the root to a leaf*

- Each internal node is associated with a *yes*/*no* question, regarding the input, and the two edges that are coming out of it are associated with one of the two possible answers to the question.
- The leaves are associated with one possible outcome of the tree, and no edge is coming out of them.
- At the leaf, the permutation π is the one that sorts the elements!

Decision tree for 3 elements



Decision tree computation

- The computation for an input starts at the root, and progresses down the tree from one node to the next according to the answers to the questions at the nodes.
- The computation ends when we get to a leaf.
- ANY correct algorithm MUST be able to produce each permutation of the input.
- There are at most $n!$ permutations and they must all appear in the leafs of the tree.

Worst case complexity

- The worst-case number of comparisons is the length of the longest root-to-leaf path in the decision tree.
- The lower bound on the length of the longest path for a given algorithm gives a lower bound on the worst-case number of comparisons the algorithm requires.
- Thus, finding a lower bound on the length of the longest path for a decision tree based on comparisons provided a lower bound on the worst case complexity of comparison based sorting algorithms!

Comparison-based sorting algorithms

- Any comparison-based sorting algorithm can be described by a decision tree T.
- The number of leaves in the tree of any comparison based sorting algorithm must be at least $n!$, since the algorithm must give a correct answer to every possible input, and there are $n!$ possible answers.
- Why "at least"? Because there might be more than one leaf with the same answer, corresponding to different ways the algorithm treats different inputs.

Length of the longest path (1)

- $n!$ different possible answers.
- Consider all trees with $n!$ leaves.
- In each one, consider the longest path. Let d be the depth (height) of the tree.
- The minimum length of such longest path must be such that $n! \leq 2^d$
- Therefore, $\log (n!) \leq \log (2^d) = d$
- *Quick check:* $(n/2)^{(n/2)} \leq n! \leq n^n$

$$(n/2) \log (n/2) \leq \log (n!) \leq n \log n$$

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

Length of the longest path (2)

Claim: for $n \geq 2$

Proof:
$$\log(n!) = \log(\prod_{i=1}^{n} i) = \sum_{i=1}^{n} \log(i)$$
$$\geq \sum_{i=\frac{n}{2}}^{n} \log(\tfrac{n}{2}) = \tfrac{n}{2}\log(\tfrac{n}{2})$$
$$= \Omega(n\log(n)).$$

On the other hand:
$$\log(n!) = \sum_{i=1}^{n} \log(i) \leq n\log(n).$$

This is the lower bound on the number of comparisons in any comparison-based sorting algorithm.

Complexity of comparison-sorting algorithms

	Space	Worst case	Best case	Average case	Random. case
Bubble-Sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion-Sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$
Merge-Sort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	---
Quick-Sort	$O(n)$	$O(n^2)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$

Lower bounds for comparison sorting is $T(n) = \Omega(n \lg n)$ and $S(n) = \Omega(n)$ for worst and average case, deterministic and randomized algorithms.