SORTING

by

David Toman

University of Waterloo

List of Slides

Sorting by Key Comparisons

- Motivation

  $\Rightarrow$ To make repeated searching an efficient operation

- Basic Operations

  $\Rightarrow$ Comparison of keys
  $\Rightarrow$ Moving entries within an array/list

- Key requirements

  $\Rightarrow$ Total ordering on sorted of elements

- Complexity

  $\Rightarrow$ optimal: $\Theta(n \log n)$
  $\Rightarrow$ for small n we can use $O(n^2)$ algorithms

Considerations

Many algorithms (for seemingly similar problem)

$\Rightarrow$ what's the difference?

- comparisons of *whole* keys (vs. digital sort)
- internal (fits in memory) vs. external sort
- stable (for equal keys)
- input in an array or a list?
- resources

  $\Rightarrow$ time (what if the input is mostly sorted?)
  $\Rightarrow$ space (in place?)
- ...

Selection Sort

IDEA: repeatedly pick the largest (smallest) element one-by-one from the input and place it in the output.

Example (in an array):

input					output				
4	$\boxed{9}$	3	7	6					
4	6	3	$\boxed{7}$						9
4	$\boxed{6}$	3						7	9
$\boxed{4}$	3						6	7	9
$\boxed{3}$						4	6	7	9
					3	4	6	7	9

Features:

$\Rightarrow$ comparison-based, internal, stable, in-place (for arrays)

Complexity Analysis

Algorithm (pseudo-code):

$i := 0$
repeat
  find largest element in $n - i$ elements
  remove from the input (swap with the i-th one)
while "something left" ($n - i > 0$)

Worst case (the largest element is always the last of the $n - i$ remaining elements):

$$(n-1) + (n-2) + \ldots + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{n}{2}(n-1) \in O(n^2)$$

Best case (presorted input): $O(n)$

Insertion Sort

IDEA: take input elements one-by one and insert them into a sorted array/list.

Example (in an array):

input	output
$4 \quad 9 \quad 3 \quad 7 \quad 6$	
$9 \quad 3 \quad 7 \quad 6$	4
$3 \quad 7 \quad 6$	$4 \quad 9$
$7 \quad 6$	$3 \quad 4 \quad 9$
6	$3 \quad 4 \quad 7 \quad 9$
	$3 \quad 4 \quad 6 \quad 7 \quad 9$

Features:

$\Rightarrow$ comparison-based, internal, stable, in-place (for arrays)

Complexity Analysis

Algorithm (pseudo-code):

for all elements in the input
insert the i th element into the $i-1$ *sorted* ones

Worst case (input sorted “backwards”):

$$1+2+\ldots+(n-2)+(n-1)=\sum_{i=1}^{n-1} i=\frac{n}{2}(n-1)\in O(n^2)$$

Best case (presorted input): $O(n)$

Exchange (bubble) Sort

IDEA: go repeatedly through array/list and swap adjacent elements if they're not in order.

Example (in an array):

Pass	Array/List				
1	$\boxed{4}$	$\boxed{9}$	3	7	6
1	4	$\boxed{9}$	$\boxed{3}$	7	6
1	4	3	$\boxed{9}$	$\boxed{7}$	6
1	4	3	7	$\boxed{9}$	$\boxed{6}$
$1-2$	$\boxed{4}$	$\boxed{3}$	7	6	9
2	3	$\boxed{4}$	$\boxed{7}$	6	9
2	3	4	$\boxed{7}$	$\boxed{6}$	9
2	3	4	6	$\boxed{7}$	$\boxed{9}$
2	3	4	6	7	9

Features:

$\Rightarrow$ comparison-based, internal, stable, in-place

Complexity Analysis

- every pass: $n - 1$ comparisons (and possibly swaps)
- how many passes can we have?

  $\Rightarrow$ up to n times (why?)

Total $O(n^2)$

Shell Sort

Developed by D.L.Shell in 1959; Tries to solve problems of insertion sort (which moves adjacent entries one at a time) and selection sort(which moves entries efficiently, but does many redundant operations.

IDEA: instead of comparing *adjacent* elements, compare (and order) elements “far apart” in the input first

$\Rightarrow$ an h-sort: $A[i] \leq A[i+h]$
$\Rightarrow$ decrease h until it reaches 1
an **increment sequence** $1 = h_1 < \ldots < h_t \Rightarrow$
choice of increments *crucial*
we use $1, 3, 7, \ldots, 2^k - 1$ (Hibbard)

Complexity Analysis

A single "phase" (for h_k): $O(N^2/h_k)$

$\Rightarrow$ essentially h_k insertion sorts of N/h_k elements.

Observation: before the h_k phase of the shell sort the input is already h_{k+1} and h_{k+2}-sorted

$\Rightarrow$ if $i = c_1 \cdot h_{k+1} + c_2 \cdot h_{k+2}$ then $A[p] \leq A[p+i]$
$\Rightarrow$ $h_{k+2} = 2h_{k+1} - 1$ — can't share common factors
integers larger than $(h_{k+1} - 1)(h_{k+2} - 1) = 8h_k^2 + 4h_k$ are linear combinations of h_{k+1} and h_{k+2}.
$\Rightarrow$ we need to look only at $O(h_k)$ "neighbors"

This gives us a $O(Nh_k)$ bound on each pass of the algorithm. Putting it together:

$$\begin{aligned} T(n) &= O(\sum_{k=1}^{t/2} nh_k + \sum_{k=t/2+1}^{t} \frac{n^2}{h_k}) \\ &= O(nh_{t/2}) + O(\frac{n^2}{h_{t/2}}) = O(n^{3/2}) \end{aligned}$$

Merge Sort

IDEA: sort input by

1. splitting input in half
2. sorting (recursively) the two halves
3. merging the resulting sorted lists

Example

Depth	Array/List				
1	4	9	3	7	6
2	4	9	3	7	6
3	4	9	3	7	6
2	4	9	3	7	6
1	3	4	9	6	7
1	3	4	6	7	9

Complexity Analysis

Time to merge-sort n elements

$\Rightarrow$ time to do two recursive merge-sorts of size $n/2$
$\Rightarrow$ plus time to merge, which is linear in n.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(\lceil n/2 \rceil) + O(n) \end{aligned}$$

Solve recurrence relation: $T(n) \in O(n \log n)$

$\Rightarrow$ telescoping sums
$\Rightarrow$ characteristic equations
($\Rightarrow$ Maple)

Merge-sort more efficient than quadratic-cost sorting algorithms, but only for large n

$\Rightarrow$ as internal sorting method extra array needed
to merge sorted halves (really? [Knuth 648(ex.18)]
$\Rightarrow$ data copying is non-productive overhead

Quick Sort

IDEA: sort input by

1. splitting input into two parts
   of "small" and "large" elements

2. recursively sorting the two parts

3. *appending* the two sorted results, pivot in the middle

good choice of **pivot** essential

$\Rightarrow$ first element (poor)
$\Rightarrow$ best-of-three elements (better)
$\Rightarrow$ randomly (best)

Complexity Analysis

Time to quick-sort n elements

$\Rightarrow$ time to do two recursive quick-sorts
$\Rightarrow$ plus time to merge, which is linear in n.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(r) + T(n-r) + O(n) \end{aligned}$$

... looks like a merge-sort.

but it isn't: worst-case splits $r = 1$ and then

$$T(n) = O(n) + T(n-1)$$

and therefore $T(n) \in O(n^2)$.

Average Complexity Analysis

Assumption: all sizes of first partition equally likely
$\Rightarrow$ probability $1/n$
$\Rightarrow$ depends on selection of *pivot*

The average value of $T(i)$

$$T(i) = T(n-i-1) = \sum_{j=0}^{n-1} T(j)$$

and the recurrence therefore is

$$T(n) = O(n) + \frac{2}{n}\sum_{j=0}^{n-1} T(j)$$

which gives us $T(n) \in O(n \log n)$ expected time.

Tree and Heap Sorts

IDEA: Use balanced search tree or heap to sort.

$\Rightarrow$ operations on tree/heap $O(\log n)$

Algorithm(s):

for all elements in the input
    insert the i-th element into a tree/heap
for all elements
    delete the min/max element from the tree/heap

Running time $nO(\log n) + nO(\log n) \in O(n \log n)$.

Comparison of Sorting Methods

No one method is best
$\Rightarrow$ some are good for small data size, others for large n.

- Merge-sort has best worst-case behavior, but requires more storage than heap sort.
  $\Rightarrow$ external sorting
- Heap-sort when $O(n \log n)$ worst case is a must.
- Quick-sort has best average behavior, but worse than the average case for selection and insertion
  $\Rightarrow$ best method for large files
  $\Rightarrow$ too much overhead for small files
- Shell-sort also acceptable for large files
  $\Rightarrow$ not quite as good as quick-sort
  $\Rightarrow$ but no need to worry about data not being random
- Insertion sort is best for small data sets
  $\Rightarrow$ efficient for long lists if already nearly sorted
- Selection sort: easy to output intermediate results