Sorting: The Big Picture

Given \(n\) comparable elements in an array, sort them in an increasing (or decreasing) order.

Simple algorithms: \(O(n^2)\)

Fancier algorithms: \(O(n \log n)\)

Comparison lower bound: \(\Omega(n \log n)\)

Specialized algorithms: \(O(n)\)

Handling huge data sets

Insertion sort
Selection sort
Bubble sort
Shell sort

Heap sort
Merge sort
Quick sort

Bucket sort
Radix sort
External sorting

The steps of QuickSort

1. Choose the pivot as the median of three.
2. Place the pivot and the largest at the right and the smallest at the left.
3. Move \(i\) to the right to be larger than pivot.
4. Move \(j\) to the left to be smaller than pivot.
5. Swap

Recursive Quicksort

QuickSort(A[]): integer array, left,right : integer:
if left + CUTOFF < right then
    pivotindex := median3(A,left,right);
    pivot := Partition(A,left,right-1,pivot);
    QuickSort(A, left, pivotindex – 1);
    QuickSort(A, pivotindex + 1, right);
else
    Insertionsort(A,left,right);

Don't use quicksort for small arrays.
CUTOFF = 10 is reasonable.
QuickSort:

Best case complexity

Worst case complexity

Average case complexity

Turns out to be $O(n \log n)$

See Section 7.7.5 for an idea of the proof.

Don’t need to know proof details for this course.

Features of Sorting Algorithms

• In-place
  – Sorted items occupy the same space as the original items. (No copying required, only $O(1)$ extra space if any.)

• Stable
  – Items in input with the same value end up in the same order as when they began.

Sort Properties

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>stable?</th>
<th>in-place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MergeSort?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>QuickSort?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

How fast can we sort?

• Heapsort, Mergesort, and Quicksort all run in $O(N \log N)$ best case running time
• Can we do any better?
• No, if the basic action is a comparison.
### Sorting Model
- Recall our basic assumption: we can only compare two elements at a time
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given N elements
  - Assume no duplicates
- How many possible orderings can you get?
  - This is the number of potential inputs the algorithm must separate

### Permutations
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
    - (a b c), (a c b), (b a c), (c a b), (c b a)
    - 6 orderings = 3-2-1 = 3!
  - All the possible permutations of a set of 3 elements
- For N elements
  - N choices for the first position, (N-1) choices for the second position, …, (2) choices, 1 choice
  - \( N(N-1)(N-2) \cdot \frac{2}{2} \cdot \frac{1}{1} = N! \) possible orderings

### Decision Tree
- **a < b < c, b < c < a, c < a < b, a < c < b, b < a < c, c < b < a**
- The leaves contain all the possible orderings of a, b, c

### Lower bound on Height
- A binary tree of height \( h \) has at most how many leaves?
- \( L \)
- A binary tree with \( L \) leaves has height at least:
- \( h \)
- The decision tree has how many leaves:
- \( h \)
- So the decision tree has height:
- \( h \)

### \( \log(N!) \) is \( \Omega(N \log N) \)
- \( \log(N!) = \log(N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 2 \cdot 1) \)
- \( = N \log N + \log(N-1) + \log(N-2) + \ldots + \log 2 + \log 1 \)
- \( \geq N \log N + (\log N - 1) + (\log N - 2) + \ldots + \log N + \log N - 2 \)
- \( \geq \frac{N}{2} \log N \)
- \( \geq \frac{N}{2} (\log N - \log 2) \)
- \( = \Omega(N \log N) \)

### \( \Omega(N \log N) \)
- Run time of any comparison-based sorting algorithm is \( \Omega(N \log N) \)
- Can we do better if we don’t use comparisons?
BucketSort (aka BinSort)
If all values to be sorted are known to be between 1 and $K$, create an array array of size $K$, increment counts while traversing the input, and finally output the result.

**Example** $K=5$. Input = $(5,1,3,4,3,2,1,1,5,4,5)$

<table>
<thead>
<tr>
<th>count array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Running time to sort n items?

BucketSort Complexity: $O(n+K)$

- Case 1: $K$ is a constant
  - BinSort is linear time
- Case 2: $K$ is variable
  - Not simply linear time
- Case 3: $K$ is constant but large (e.g. $2^{32}$)
  - ???

Fixing impracticality: RadixSort

- **Radix** = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything

- **Idea**: BucketSort on each **digit**
  - least significant to most significant (lsd to msd)

Radix Sort Example (1st pass)

Radix Sort Example (2nd pass)

Radix Sort Example (3rd pass)

Invariant: after k passes the low order k digits are sorted.
Radixsort: Complexity

- How many passes?
- How much work per pass?
- Total time?
- Conclusion?
- In practice
  - Radixsort only good for large number of elements with relatively small values
  - Hard on the cache compared to MergeSort/QuickSort

Internal versus External Sorting

- Need sorting algorithms that minimize disk/tape access time
- **External sorting** – Basic Idea:
  - Load chunk of data into RAM, sort, store this “run” on disk/tape
  - Use the Merge routine from Mergesort to merge runs
  - Repeat until you have only one run (one sorted chunk)
  - Text gives some examples