IS 709/809: Computational Methods in IS Research

Algorithm Analysis (Sorting)

Nirmalya Roy
Department of Information Systems
University of Maryland Baltimore County
Sorting Problem

- Given an array $A[0...N - 1]$, modify $A$ such that $A[i] \leq A[i + 1]$ for $0 \leq i \leq N - 1$

- Internal vs. external sorting
  - Main memory and disk access

- Stable vs. unstable sorting
  - Equal elements retain original order
  - Keep elements with equal keys in the same relative order in the output as they were in the input
  - Input: $1, 5_x, 3, 5_y, 2, 4$
  - Output: $1, 2, 3, 4, 5_x, 5_y$
Sorting Problem

- **In-place sorting**
  - Transform input using a data structure with a constant amount of extra storage space; $O(1)$ extra memory
    - Constant additional storage for the auxiliary variables (i and temp)
  - Input is overwritten by the output as the algorithm executes
  - Example: Bubble sort, Selection sort, Insertion sort, Heap sort, Shell sort etc

- **Comparison sorting vs. non-comparison sorting**
  - Besides assignment operator; “<” and “>” operators are allowed on the input data
  - Function template `sort with comparator cmp`
    - `void sort(Iterator begin, Iterator end, Comparator cmp)`
Sorting Algorithms

- Insertion sort
- Selection sort
- Shell sort
- Heap sort
- Merge sort
- Quick sort
- ...
- Simple data structure; focus on analysis
Insertion Sort

- Algorithm:
  - Start with empty list S and unsorted list A of N items
  - For each item x in A
    - Insert x into S, in sorted order

- Example:
Insertion Sort (cont’d)

- In-place
- Stable
- Best-case?
  - $O(N)$
- Worst-case?
  - $O(N^2)$
- Average-case?
  - Consists of $N-1$ passes
  - For pass $p = 1$ to $N-1$
    - Position 0 thru $p$ are in sorted order
    - Move the element in position $p$ left until its correct place; among first $p+1$ elements

```
InsertionSort(A) {
    for p = 1 to N - 1 {
        tmp = A[p]
        j = p
        while (j > 0) and (tmp < A[j - 1]) {
            j = j - 1
        }
        A[j] = tmp
    }
}
```
## Insertion Sort Example

<table>
<thead>
<tr>
<th>Original</th>
<th>34</th>
<th>8</th>
<th>64</th>
<th>51</th>
<th>32</th>
<th>21</th>
<th>Positions Moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>After $p = 1$</td>
<td>8</td>
<td>34</td>
<td>64</td>
<td>51</td>
<td>32</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>After $p = 2$</td>
<td>8</td>
<td>34</td>
<td>64</td>
<td>51</td>
<td>32</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>After $p = 3$</td>
<td>8</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>32</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>After $p = 4$</td>
<td>8</td>
<td>32</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>After $p = 5$</td>
<td>8</td>
<td>21</td>
<td>32</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>4</td>
</tr>
</tbody>
</table>

- Consists of N-1 **passes**
- For pass $p = 1$ to N-1
  - Position 0 thru $p$ are in sorted order
  - Move the element in position $p$ left until its correct place is found; among first $p+1$ elements
Selection Sort

Algorithm:
- Start with empty list $S$ and unsorted list $A$ of $N$ items
- for $(i = 0; i < N; i++)$
  - $x \leftarrow$ item in $A$ with smallest key
  - Remove $x$ from $A$
  - Append $x$ to end of $S$

```
7  3  9  5
S  A
```
```
3  7  9  5
S  A
```
```
3  5  9  7
S  A
```
```
3  5  7  9
S  A
```
Selection Sort (cont’d)

- In-place
- Unstable
- Best-case: $O(N^2)$
- Worst-case: $O(N^2)$
- Average-case: $O(N^2)$
Shell Sort

- Shell sort is a multi-pass algorithm
  - Each pass is an **insertion sort** of the sequences consisting of every **$h$-th** element for a fixed gap $h$, known as the **increment**
  - This is referred to as **$h$-sorting**
- Consider shell sort with gaps 5, 3 and 1
  - Input array: $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}$
  - First pass, **5-sorting**, performs insertion sort on separate sub-arrays $(a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10})$
  - Next pass, **3-sorting**, performs insertion sort on the sub-arrays $(a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12})$
  - Last pass, **1-sorting**, is an ordinary insertion sort of the entire array $(a_1,..., a_{12})$
Shell Sort (cont’d)

- Works by comparing the elements that are distant.
- The distance between comparisons decreases as the algorithm runs until its last phase.

Sorting each column (Insertion Sort):

10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45

5-sorting

13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10
Shell Sort (cont’d)

Sorting each column

10 14 73
25 23 13
27 94 33
39 25 59
94 65 82
45

10 14 13
25 23 33
27 25 59
39 65 73
45 94 82
94

10 14 13 25 23 33 27 25 59 39 65 73 45 94 82 94

3-sorting
Shell Sort (cont’d)

- Insertion Sort:
  - Start with empty list S and unsorted list A of N items
  - For each item x in A
    - Insert x into S, in sorted order

Unsorted A

10 14 13 25 23 33 27 25 59 39 65 73 45 94 82 94

1-sorting/ Insertion Sort

Sorted S

10 13 14 23 25 25 27 33 39 45 59 65 73 82 94 94
Shell Sort (cont’d)

- In-place
- Unstable
- Best-case
  - Sorted: $\Theta(N \log_2 N)$
- Worst-case
  - Shell’s increments (by $2^k$): $\Theta(N^2)$
  - Hibbard’s increments (by $2^k - 1$): $\Theta(N^{3/2})$
- Average-case: $\Theta(N^{7/6})$
- Later sorts do not undo the work done in previous sorts
  - If an array is 5-sorted and then 3-sorted, the array is now not only 3-sorted, but both 5- and 3-sorted

ShellSort(A) {
    gap = N
    while (gap > 0) {
        gap = gap / 2
        B = <A[0], A[gap], A[2*gap], ...>
        InsertionSort(B)
    }
}
Merge Sort

- Idea: We can merge 2 sorted lists into 1 sorted list in linear time
- Let Q1 and Q2 be 2 sorted queues
- Let Q be empty queue
- Algorithm for merging Q1 and Q2 into Q:
  - While (neither Q1 nor Q2 is empty)
    - item1 = Q1.front()
    - item2 = Q2.front()
    - Move smaller of item1, item2 from present queue to end of Q
  - Concatenate remaining non-empty queue (Q1 or Q2) to end of Q
Merge Sort (cont’d)

- Recursive divide-and-conquer algorithm

- Algorithm:
  - Start with unsorted list A of N items
  - Break A into halves A₁ and A₂, having \( \left\lfloor N/2 \right\rfloor \) and \( \left\lceil N/2 \right\rceil \) items
  - Sort A₁ recursively, yielding S₁
  - Sort A₂ recursively, yielding S₂
  - Merge S₁ and S₂ into one sorted list S
Merge Sort (cont’d)

Divide with \(O(\log n)\) steps

Conquer with \(O(\log n)\) steps

Sorted S

\(1 + \lceil \log_2 N \rceil \) levels

Dividing is trivial
Merging is non-trivial

Sorted S1

Sorted S2
Merging Two Sorted Arrays

1. \( C[k++] = \text{Populate min}\{ A[i], B[j] \} \)
2. Advance the minimum contributing pointer

\( \Theta(N) \) time
Merge Sort (cont’d)

- Not in-place
- Stable
- Analysis: All cases
  - $T(1) = \Theta(1)$
  - $T(N) = 2T(N/2) + \Theta(N)$
  - $T(N) = \Theta(N \log_2 N)$
  - See whiteboard

```
MergeSort(A)
   MergeSort2(A, 0, N-1)

MergeSort2(A, i, j)
   if (i < j)
      k = (i + j) / 2
      MergeSort2(A, i, k)
      MergeSort2(A, k + 1, j)
      Merge(A, i, k, j)

Merge(A, i, k, j)
   Create auxiliary array B
   Copy elements of sorted A[i...k] and sorted A[k+1...j] into B (in order)
   A = B
```
Quick Sort

- Like merge sort, quick sort is a divide-and-conquer algorithm, except
  - Don’t divide the array in half
  - Partition the array based on elements being less than or greater than some element of the array (the pivot)

- In-place, unstable

- Worst-case running time: $O(N^2)$

- Average-case running time: $O(N \log_2 N)$

- Fastest generic sorting algorithm in practice
Quick Sort (cont’d)

Algorithm:
- Start with list A of N items
- Choose pivot item v from A
- Partition A into 2 unsorted lists A₁ and A₂
  - A₁: All keys smaller than v’s key
  - A₂: All keys larger than v’s key
  - Items with same key as v can go into either list
  - The pivot v does not go into either list
- Sort A₁ recursively, yielding sorted list S₁
- Sort A₂ recursively, yielding sorted list S₂
- Concatenate S₁, v, and S₂, yielding sorted list S

How to choose pivot?
Quick Sort (cont’d)

For now, let the pivot v be the first item in the list.

Dividing (“Partitioning”) is non-trivial
Merging is trivial

O(N log₂ N)
Quick Sort Algorithm

- quicksort (array: S)
  1. If size of S is 0 or 1, return
  2. Pivot = Pick an element v in S
  3. Partition S − {v} into two disjoint groups
     S1 = {x ∈ (S − {v}) | x < v}
     S2 = {x ∈ (S − {v}) | x > v}
  4. Return {quicksort(S1), followed by v, followed by quicksort(S2)}
Quick Sort (cont’d)

What if the list is already sorted?

For now, let the pivot $v$ be the first item in the list.

When input already sorted, choosing first item as pivot is disastrous.

$O(N^2)$
Quick Sort (cont’d)

We need a better pivot-choosing strategy.
Quick Sort (cont’d)

- Merge sort always divides array in half
  - Quick sort might divide array into sub problems of size 1 and N – 1
    - When?
    - Leading to $O(N^2)$ performance
  - Need to choose pivot wisely (but efficiently)

- Merge sort requires temporary array for merge step
  - Quick sort can partition the array in place
  - This more than makes up for bad pivot choices
Quick Sort (cont’d)

- Choosing the pivot
  - Choosing the first element
    - What if array already or nearly sorted?
    - Good for random array
  - Choose random pivot
    - Good in practice if truly random
    - Still possible to get some bad choices
    - Requires execution of random number generator
    - On average, generates $\frac{1}{4}$, $\frac{3}{4}$ split
Quick Sort (cont’d)

Choosing the pivot

- Best choice of pivot?
  - Median of array
  - Median is expensive to calculate
  - Estimate median as the median of three elements (called the median-of-three strategy)
    - Choose first, middle, and last elements
    - E.g., <8, 1, 4, 9, 6, 3, 5, 2, 7, 0>
  - Has been shown to reduce running time (comparisons) by 14%
Quick Sort (cont’d)

- Partitioning strategy
  - Partitioning is conceptually straightforward, but easy to do inefficiently
  - Good strategy
    - Swap pivot with last element A[right]
    - Set i = left
    - Set j = (right – 1)
    - While (i < j)
      - Increment i until A[i] > pivot
      - Decrement j until A[j] < pivot
      - If (i < j) then swap A[i] and A[j]
    - Swap pivot and A[i]
Partitioning Example

Initial array

Swap pivot; initialize i and j

Position i and j

After first swap

Before second swap

After second swap

Before third swap

After swap with pivot
Quick Sort (cont’d)

- Partitioning strategy
  - How to handle duplicates?
  - Consider the case where all elements are equal.
    - Current approach: Skip over elements equal to pivot
      - No swaps (good)
      - But then \( i = (right - 1) \) and array partitioned into \( N - 1 \) and 1 elements
    - Worst-case performance: \( O(N^2) \)
Quick Sort (cont’d)

- Partitioning strategy
  - How to handle duplicates?
  - Alternate approach
    - Don’t skip elements equal to pivot
      - Increment i while A[i] < pivot
      - Decrement j while A[j] > pivot
    - Adds some unnecessary swaps
    - But results in perfect partitioning for the array of identical elements
      - Unlikely for input array, but more likely for recursive calls to quick sort
Which Sort to Use?

- When array A is small, generating lots of recursive calls on small sub-arrays is expensive.

General strategy

- When \( N < \) threshold, use a sort more efficient for small arrays (e.g. insertion sort).
- Good thresholds range from 5 to 20.
- Also avoids issue with finding median-of-three pivot for array of size 2 or less.
- Has been shown to reduce running time by 15%.
Analysis of Quick Sort

- Let \( m \) be the number of elements sent to the left partition
- Compute running time \( T(N) \) for array of size \( N \)
- \( T(0) = T(1) = O(1) \)
- \( T(N) = T(m) + T(N - m - 1) + O(N) \)

- Pivot selection takes constant time
Analysis of Quick Sort (cont’d)

- Recurrence formula:
  - $T(N) = T(m) + T(N - m - 1) + O(N)$

- Worst-case analysis
  - Pivot is the smallest element ($m = 0$)
    - $T(N) = T(0) + T(N - 1) + O(N)$
    - $T(N) = O(1) + T(N - 1) + O(N)$
    - $T(N) = T(N - 1) + O(N)$; since $T(N - 1) = T(N - 2) + O(N - 1)$;
    - $T(N) = T(N - 2) + O(N - 1) + O(N)$
    - $T(N) = T(N - 3) + O(N - 2) + O(N - 1) + O(N)$
    - $\cdots$

\[
T(N) = \sum_{i=1}^{N} O(i) = O(N^2)
\]
Analysis of Quick Sort (cont’d)

- Recurrence formula:
  - $T(N) = T(m) + T(N - m - 1) + O(N)$

- Best-case analysis
  - Pivot is in the middle $(m = N / 2)$
  - $T(N) = T(N / 2) + T(N / 2) + O(N)$
  - $T(N) = 2T(N / 2) + O(N)$
  - $T(N) = O(N \log N)$

- Average-case analysis
  - Assuming each partition equally likely
  - $T(N) = O(N \log N)$
# Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst Case</th>
<th>Average Case</th>
<th>Best Case</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N)$</td>
<td>Best Case is linear</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N^2)$</td>
<td>Best Case is quadratic</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N)$</td>
<td>Fast for small $N$</td>
</tr>
<tr>
<td>ShellSort</td>
<td>$\Theta(N^{3/2})$</td>
<td>$\Theta(N^{7/6})$</td>
<td>$\Theta(N \log N)$</td>
<td>Increment sequence?</td>
</tr>
<tr>
<td>HeapSort</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>Large constants</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>Requires memory</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>Small constants</td>
</tr>
</tbody>
</table>
Comparison of Sorting Algorithms (cont’d)

<table>
<thead>
<tr>
<th>N</th>
<th>Insertion Sort ( O(N^2) )</th>
<th>Shellsort ( O(N^{7/6}) (? )</th>
<th>Heapsort ( O(N \log N) )</th>
<th>Quicksort ( O(N \log N) )</th>
<th>Quicksort (opt.) ( O(N \log N) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.0000001</td>
<td>0.0000002</td>
<td>0.0000003</td>
<td>0.0000002</td>
<td>0.0000002</td>
</tr>
<tr>
<td>100</td>
<td>0.000106</td>
<td>0.000039</td>
<td>0.000052</td>
<td>0.000025</td>
<td>0.000023</td>
</tr>
<tr>
<td>1000</td>
<td>0.011240</td>
<td>0.000678</td>
<td>0.000750</td>
<td>0.000365</td>
<td>0.000316</td>
</tr>
<tr>
<td>10000</td>
<td>1.047</td>
<td>0.009782</td>
<td>0.010215</td>
<td>0.004612</td>
<td>0.004129</td>
</tr>
<tr>
<td>100000</td>
<td>110.492</td>
<td>0.13438</td>
<td>0.139542</td>
<td>0.058481</td>
<td>0.052790</td>
</tr>
<tr>
<td>1000000</td>
<td>NA</td>
<td>1.6777</td>
<td>1.7967</td>
<td>0.6842</td>
<td>0.6154</td>
</tr>
</tbody>
</table>

All times are in seconds

Good sorting applets:
- [http://www.sorting-algorithms.com](http://www.sorting-algorithms.com)
- [http://math.hws.edu/TMCM/java/xSortLab/](http://math.hws.edu/TMCM/java/xSortLab/)
Lower Bound on Sorting

- Best worst-case sorting algorithm (so far) is $O(N \log N)$
- Can we do better?
- Can we prove a lower bound on the sorting problem?

Preview
- For comparison-based sorting, we can’t do better
- We can show lower bound of $\Omega(N \log N)$
Decision Trees

- A decision tree is a binary tree
  - Each node represents a set of possible orderings of the array elements
  - Each branch represents an outcome of a particular comparison
- Each leaf of the decision tree represents a particular ordering of the original array elements
Decision Trees (cont’d)

Decision tree for sorting three elements

```
1. a < b, b < a
   2. a < c
      4. a < b < c, a < c < b
         8. a < b < c
         9. a < c < b
      3. c < a
         5. c < a < b
   6. b < a < c
      10. b < a < c
      11. b < c < a
```
Decision Trees (cont’d)

- The logic of every sorting algorithm that uses comparisons can be represented by a decision tree.
- In the worst case, the number of comparisons used by the algorithm equals the depth of the deepest leaf.
- In the average case, the number of comparisons is the average of the depth of all leaves.
- There are $N!$ different orderings of $N$ elements.
Lower Bound for Comparison-based Sorting

- Lemma 7.1: A binary tree of depth $d$ has at most $2^d$ leaves
- Lemma 7.2: A binary tree with $L$ leaves must have depth at least $\lceil \log L \rceil$
- Theorem 7.6: Any comparison-based sorting requires at least $\lceil \log (N!) \rceil$ comparison in the worst case
- Theorem 7.7: Any comparison-based sorting requires $\Omega(N \log N)$ comparisons
Linear Sorting

- Some constraints on input array allow faster than $\Theta(N \log N)$ sorting (no comparisons)

- Counting Sort$^1$
  - Given array $A$ of $N$ integer elements, each less than $M$
  - Create array $C$ of size $M$, where $C[i]$ is the number of $i$’s in $A$
  - Use $C$ to place elements into new sorted array $B$
  - Running time $\Theta(N + M) = \Theta(N)$ if $M = \Theta(N)$

$^1$Weiss incorrectly calls this Bucket Sort.
Linear Sorting (cont’d)

- **Bucket Sort**
  - Assume N elements of A uniformly distributed over the range \([0, 1)\)
  - Create N equal-size buckets over \([0, 1)\)
  - Add each element of A into appropriate bucket
  - Sort each bucket (e.g. with insertion sort)
  - Return concatenation of buckets
  - Average-case running time \(\Theta(N)\)
    - Assumes each bucket will contain \(\Theta(1)\) elements
External Sorting

- What if the number of elements $N$ we wish to sort do not fit in main memory?
- Obviously, our existing sorting algorithms are inefficient
  - Each comparison potentially requires a disk access
- Once again, we want to minimize disk accesses
External Merge Sort

- $N =$ number of elements in array $A$ to be sorted
- $M =$ number of elements that fit in main memory
- $K = \left\lceil \frac{N}{M} \right\rceil$

**Approach**

- Read in $M$ amount of $A$, sort it using quick sort, and write it back to disks: $O(M \log M)$
- Repeat above $K$ times until all of $A$ processed
- Create $K$ input buffers and 1 output buffer, each of size $M / (K + 1)$
- Perform a $K$-way merge: $O(N)$
  - Update input buffers one disk-page at a time
  - Write output buffer one disk-page at a time
Multiway Merge (3-way) Example

<table>
<thead>
<tr>
<th></th>
<th>Ta1</th>
<th>Ta2</th>
<th>Ta3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tb1</td>
<td>11</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>Tb2</td>
<td>81</td>
<td>35</td>
<td>96</td>
</tr>
<tr>
<td>Tb3</td>
<td>94</td>
<td>58</td>
<td>75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ta1</th>
<th>Ta2</th>
<th>Ta3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tb1</td>
<td>11</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>Tb2</td>
<td>81</td>
<td>35</td>
<td>96</td>
</tr>
<tr>
<td>Tb3</td>
<td>94</td>
<td>58</td>
<td>75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ta1</th>
<th>Ta2</th>
<th>Ta3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tb1</td>
<td>11</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>Tb2</td>
<td>81</td>
<td>35</td>
<td>96</td>
</tr>
<tr>
<td>Tb3</td>
<td>94</td>
<td>58</td>
<td>75</td>
</tr>
</tbody>
</table>
Sorting: Summary

- Need for sorting is ubiquitous in software
- Optimizing the sorting algorithm to the domain is essential
- Good general-purpose algorithms available
  - Quick sort
- Optimizations continue...
  - Sort benchmark
    - http://sortbenchmark.org/