Sorting Algorithms

CptS 223 – Advanced Data Structures

Larry Holder
School of Electrical Engineering and Computer Science
Washington State University
Sorting Problem

- Given array $A[0...N-1]$, modify $A$ such that $A[i] \leq A[i+1]$ for $0 \leq i < N-1$
- Internal vs. external sorting
- Stable vs. unstable sorting
  - Equal elements retain original order
- In-place sorting ($O(1)$ extra memory)
- Comparison sorting vs. ???
Sorting Algorithms

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

- In-place
- Stable
- Best case?
- Worst case?
- Average case?

```
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
}
```
ShellSort

- 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: $\Theta(N^{7/6})$?
HeapSort

- In-place
- Unstable
- All cases
  - $\Theta(N \log_2 N)$
MergeSort

- Not in-place
- Stable

Analysis: All cases
\[ T(1) = \Theta(1) \]
\[ T(N) = 2T(N/2) + \Theta(N) \]
\[ T(N) = \Theta(?) \]

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)

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

- In-place, unstable
- Like MergeSort, 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)
- Worst case running time $O(N^2)$
- Average case running time $O(N \log N)$
- Fastest generic sorting algorithm in practice
- Even faster if use simple sort (e.g., InsertionSort) when array is small
QuickSort Algorithm

- Given array S
- Modify S so elements in increasing order
  1. If size of S is 0 or 1, return
  2. Pick any element v in S as the pivot
  3. Partition S \(-\{v\}\) into two disjoint groups
     - \(S_1 = \{x \in (S - \{v\}) \mid x \leq v\}\)
     - \(S_2 = \{x \in (S - \{v\}) \mid x \geq v\}\)
  4. Return QuickSort(S1), followed by v, followed by QuickSort(S2)
QuickSort Example
Why so fast?

- MergeSort always divides array in half
  - QuickSort might divide array into subproblems of size 1 and N-1
    - When?
      - Leading to $O(N^2)$ performance
  - Need to choose pivot wisely (but efficiently)

- MergeSort requires temporary array for merge step
  - QuickSort can partition the array in place
  - This more than makes up for bad pivot choices
Picking 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
Picking the Pivot

- Best choice of pivot?
  - Median of array
- Median is expensive to calculate
- Estimate median as the median of three elements
  - 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%
Partitioning Strategy

- Partitioning is conceptually straightforward, but easy to do inefficiently

- Good strategy
  - Swap pivot with last element \( S[right] \)
  - Set \( i = left \)
  - Set \( j = (right - 1) \)
  - While \( i < j \)
    - Increment \( i \) until \( S[i] > pivot \)
    - Decrement \( j \) until \( S[j] < pivot \)
    - If \( i < j \), then swap \( S[i] \) and \( S[j] \)
  - Swap pivot and \( S[i] \)
Partitioning Example

Initial array

Swap pivot; initialize i and j

Position i and j

After first swap
Partitioning Example (cont.)

Before second swap

\[
\begin{array}{cccccccccc}
2 & 1 & 4 & 9 & 0 & 3 & 5 & 8 & 7 & 6 \\
i & j
\end{array}
\]

After second swap

\[
\begin{array}{cccccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 9 & 8 & 7 & 6 \\
i & j
\end{array}
\]

Before third swap

\[
\begin{array}{cccccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 9 & 8 & 7 & 6 \\
j & i
\end{array}
\]

After swap with pivot

\[
\begin{array}{cccccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 6 & 8 & 7 & 9 \\
i & p
\end{array}
\]
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 $O(N^2)$ performance
Partitioning Strategy

- How to handle duplicates?
- Alternative approach
  - Don’t skip elements equal to pivot
    - Increment i while $S[i] < \text{pivot}$
    - Decrement j while $S[j] > \text{pivot}$
  - Adds some unnecessary swaps
  - But results in perfect partitioning for array of identical elements
    - Unlikely for input array, but more likely for recursive calls to QuickSort
Small Arrays

- When S 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., InsertionSort)
- 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%
QuickSort Implementation

```cpp
1  /**
2   * Quicksort algorithm (driver).
3  */
4  template <typename Comparable>
5  void quicksort( vector<Comparable> & a )
6  {
7    quicksort( a, 0, a.size() - 1 );
8  }
```
QuickSort Implementation

```cpp
/**
 * Return median of left, center, and right.
 * Order these and hide the pivot.
 */

template <typename Comparable>
const Comparable & median3( vector<Comparable> & a, int left, int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ] < a[ left ] )
        swap( a[ left ], a[ center ] );
    if( a[ right ] < a[ left ] )
        swap( a[ left ], a[ right ] );
    if( a[ right ] < a[ center ] )
        swap( a[ center ], a[ right ] );
    // Place pivot at position right - 1
    swap( a[ center ], a[ right - 1 ] );
    return a[ right - 1 ];
}
```
Swap should be compiled inline.
Analysis of QuickSort

- Let I be the number of elements sent to the left partition
- Compute running time $T(N)$ for array of size $N$
  \[ T(0) = T(1) = O(1) \]
  \[ T(N) = T(i) + T(N - i - 1) + O(N) \]
Analysis of QuickSort

- Worst-case analysis
  - Pivot is the smallest element \((i = 0)\)

\[
T(N) = T(0) + T(N - 1) + O(N)
\]
\[
T(N) = O(1) + T(N - 1) + O(N)
\]
\[
T(N) = T(N - 1) + O(N)
\]
\[
T(N) = T(N - 2) + O(N - 1) + O(N)
\]
\[
T(N) = T(N - 3) + O(N - 2) + O(N - 1) + O(N)
\]
\[
T(N) = \sum_{i=1}^{N} O(i) = O(N^2)
\]
Analysis of QuickSort

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

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

<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>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 Sorting

<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>0.000002</td>
<td>0.000003</td>
<td>0.000002</td>
<td>0.000002</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>

~3 hours

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 sorting, no, we can’t do better
  - Can show lower bound of $\Omega(N \log N)$
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 tree for sorting three elements
Decision Tree for Sorting

- 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 depths of all leaves.
- There are $N!$ different orderings of $N$ elements.
Lower Bound for Comparison 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 $\lceil \log L \rceil$
- Thm. 7.6: Any comparison sort requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case
Lower Bound for Comparison Sorting

- Thm. 7.7: Any comparison sort requires $\Omega(N \log N)$ comparisons

- Proof (recall Stirling’s approximation)

  \[ N! = \sqrt{2\pi N} \left(\frac{N}{e}\right)^N \left(1 + \Theta\left(\frac{1}{N}\right)\right) \]

  \[ N! > \left(\frac{N}{e}\right)^N \]

  \[ \log(N!) > N \log N - N \log e = \Theta(N \log N) \]

  \[ \therefore \log(N!) = \Omega(N \log N) \]
Linear Sorting

- Some constraints on input array allow faster than $\Theta(N \log N)$ sorting (no comparisons)
- CountingSort$^1$
  - 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 BucketSort.
Linear Sorting

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

```cpp
#include <algorithm>
void sort (iterator start, iterator end);
void sort (iterator start, iterator end, Comparator cmp);
void stable_sort (iterator start, iterator end);
void stable_sort (iterator start, iterator end, Comparator cmp);
```

**STL sort** uses IntrospectiveSort

- QuickSort until recursion depth of \((\log N)\)
  - Median-of-3 pivot selection
- Then HeapSort

**STL stable_sort** uses MergeSort
Sorting in the STL

- Lists

```cpp
#include <list>
void sort ();
void sort (Comparator cmp);
```

- Uses MergeSort
  - Stable
  - No auxiliary array needed
- Iterators left intact
External Sorting

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

- N = number of elements in array A to be sorted
- M = number of elements that fit in memory
- K = \( \left\lfloor \frac{N}{M} \right\rfloor \)

Approach
- Read in M amount of A, sort it using QuickSort, and write it back to disk: 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
External MergeSort

- \( T(N,M) = O(K \times M \log M) + N \)
- \( T(N,M) = O((N/M) \times M \log M) + N \)
- \( T(N,M) = O((N \log M) + N) \)
- \( T(N,M) = O(N \log M) \)
- Disk accesses (all sequential)
  - \( P = \) page size
  - \( \text{Accesses} = 4N/P \) (read-all/write-all twice)
Sorting: Summary

- Need for sorting is ubiquitous in software
- Optimizing the sort algorithm to the domain is essential
- Good general-purpose algorithms available
  - QuickSort
- Optimizations continue…
  - Sort benchmark