Divide and Conquer Sorting

CSE 326
Data Structures
Lecture 18

Insertion Sort

- What if first $k$ elements of array are already sorted?
  \[-4, 7, 12, 5, 19, 16\]
- We can shift the tail of the sorted elements list down and then insert next element into proper position and we get $k+1$ sorted elements
  \[-4, 5, 7, 12, 19, 16\]

“Divide and Conquer”

- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, recursively sort left and right halves, then merge two halves \(-\) known as Mergesort
- **Idea 2**: Partition array into small items and large items, then recursively sort the two sets \(-\) known as Quicksort

Mergesort

- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Mergesort Example

Auxiliary Array

- The merging requires an auxiliary array.
**Merging**

Merging requires an auxiliary array.

```
Merge(A[], T[] : integer array, left, right : integer) : {
    mid, i, j, k, l, target : integer;
    mid := (right + left)/2;
    i := left; j := mid + 1; target := left;
    while i < mid and j <= right do
        else T[target] := A[j]; j := j + 1;
        target := target + 1;
        if i > mid then //left completed//
            for k := left to target-1 do A[k] := T[k];
        if j > right then //right completed//
            k := mid; l := right;
            while k <= i do A[k] := A[l]; k := k+1; l := l-1;
            for k := left to target-1 do A[k] := T[k];
    }
}
```

**Recursive Mergesort**

```
Mergesort(A[], T[] : integer array, left, right : integer) : {
    if left < right then
        mid := (left + right)/2;
        Mergesort(A,T,left,mid);
        Mergesort(A,T,mid+1,right);
        Merge(A,T,left,right);
}
```

```
MainMergesort(A[1..n] : integer array, n : integer) : {
    T[1..n]: integer array;
    Mergesort(A,T,1,n);
}
```
Iterative Mergesort

- Merge by 1
- Merge by 2
- Merge by 4
- Merge by 8

Iterative pseudocode

• Sort(array A of length N)
  – Let \( m = 2 \), let \( B \) be temp array of length \( N \)
  – While \( m < N \)
    • For \( i = 1 \ldots N \) in increments of \( m \)
      – merge \( A[i..i+m/2] \) and \( A[i+m/2..i+m] \) into \( B[i..i+m] \)
    • Swap role of \( A \) and \( B \)
    • \( m = m \times 2 \)
  – If needed, copy \( B \) back to \( A \)

Mergesort Analysis

• Let \( T(N) \) be the running time for an array of \( N \) elements
• Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array
• Each recursive call takes \( T(N/2) \) and merging takes \( O(N) \)

Mergesort Recurrence Relation

• The recurrence relation for \( T(N) \) is:
  – \( T(1) \leq c \)
    • base case: 1 element array \( \Rightarrow \) constant time
  – \( T(N) \leq 2T(N/2) + dN \)
    • Sorting \( n \) elements takes
      – the time to sort the left half
      – plus the time to sort the right half
      – plus an \( O(N) \) time to merge the two halves
  • \( T(N) = O(N \log N) \)

Properties of Mergesort

• Not in-place
  – Requires an auxiliary array
• Very few comparisons
• Iterative Mergesort reduces copying.
Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does.
  - Partition array into left and right sub-arrays
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

“Four easy steps”

- To sort an array S
  - If the number of elements in S is 0 or 1, then return. The array is sorted.
  - Pick an element v in S. This is the pivot value.
  - Partition S-{v} into two disjoint subsets, S₁ = {all values x<v}, and S₂ = {all values x>v}.
  - Return QuickSort(S₁), v, QuickSort(S₂)

The steps of QuickSort

- The algorithm so far lacks quite a few of the details
- Picking the pivot
  - want a value that will cause |S₁| and |S₂| to be non-zero, and close to equal in size if possible
- Implementing the actual partitioning
- Dealing with the cases where the element equals the pivot

Details, details

- “The algorithm so far lacks quite a few of the details”
- Picking the pivot
  - want a value that will cause |S₁| and |S₂| to be non-zero, and close to equal in size if possible
  - Implementing the actual partitioning
  - Dealing with the cases where the element equals the pivot

Alternative Pivot Rules

- Chose A[left]
  - Fast, but too biased, enables worst-case
- Chose A[random], left ≤ random ≤ right
  - Completely unbiased
  - Will cause relatively even split, but slow
- Median of three, A[left], A[right], A[(left+right)/2]
  - The standard, tends to be unbiased, and does a little sorting on the side.

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
  - the elements in left sub-array are ≤ pivot
  - elements in right sub-array are ≥ pivot
- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions
Partitioning is Done In-Place

- One implementation (there are others)
  - median3 finds pivot and sorts left, center, right
  - Swap pivot with next to last element
  - Set pointers i and j to start and end of array
  - Increment i until you hit element \(A[i] > \text{pivot}\)
  - Decrement j until you hit element \(A[j] < \text{pivot}\)
  - Swap \(A[i]\) and \(A[j]\)
  - Repeat until i and j cross
  - Swap pivot (= \(A[N-2]\)) with \(A[i]\)

Recursive Quicksort

```plaintext
Quicksort(A[]): integer array, left, right : integer:
    pivotindex := integer;
    if left + CUTOFF ≤ right then
        pivot := median3(A, left, right);
        pivotindex := 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 Performance

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - \(T(0) = T(1) = O(1)\)
    - constant time if 0 or 1 element
  - For \(N > 1\), 2 recursive calls plus linear time for partitioning
  - \(T(N) = 2T(N/2) + O(N)\)
    - Same recurrence relation as Mergesort
  - \(T(N) = O(N \log N)\)
QuickSort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - $T(N) \leq a$ for $N \leq C$
  - $T(N) \leq T(N-1) + bN$
  - $\leq T(N-2) + b(N-1) + bN$
  - $\leq T(C) + b(C+1) + bN$
  - $\leq a + b(C + C+1 + C+2 + \ldots + N)$
  - $T(N) = O(N^2)$
- Fortunately, *average case performance* is $O(N \log N)$ (see text for proof)

Properties of QuickSort

- No iterative version (without using a stack).
- Pure quicksort not good for small arrays.
- “In-place”, but uses auxiliary storage because of recursive calls.
- $O(n \log n)$ average case performance, but $O(n^2)$ worst case performance.

Folklore

- “QuickSort is the best in-memory sorting algorithm.”
- Mergesort and QuickSort make different tradeoffs regarding the cost of comparison and the cost of a swap.

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.

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?
  - Example: $a, b, c$ ($N = 3$)
Permutations

- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - 3 choices for the first position, 2 choices for the second position, 1 choice for the third position
  - 3 * 2 * 1 = 6 possible orderings

- For N elements
  - N(N-1)(N-2)...(2)(1) = N! possible orderings

Decision Trees

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for N distinct elements?
    - N!, a leaf for each possible ordering
  - Only 1 leaf has the ordering that is the desired correctly sorted arrangement

Decision Trees and Sorting

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - i.e., by making comparisons
  - Each decision reduces the possible solution space by one half
  - Run time is ≥ maximum no. of comparisons
    - maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree

Lower bound on Height

- A binary tree of height h has at most how many leaves?
  \[ L \leq 2^h \]
- The decision tree has how many leaves:
  \[ L = N! \]
- A binary tree with L leaves has height at least:
  \[ h \geq \log_2 L \]
- So the decision tree has height:
  \[ h \geq \log_2(N!) \]
\[ \log(N!) \text{ is } \Omega(N \log N) \]

\[
\begin{align*}
\log(N!) &= \log(N - 1) + (N - 2) + (N - 3) + \cdots + 1 \\
&= \log N + \log(N - 1) + \log(N - 2) + \cdots + 1 \\
&\geq \log N + \log(N - 1) + \log(N - 2) + \cdots + 1 \\
&\geq \frac{N}{2} \log N \\
&= \Omega(N \log N)
\end{align*}
\]

\[ \Omega(N \log N) \]

- 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 \( \text{count} \) of size \( K \), increment \( \text{counts} \) 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)

<table>
<thead>
<tr>
<th>Input data</th>
<th>Bucket sort by 1's digit</th>
<th>After 1st pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>478</td>
<td>9</td>
<td>721</td>
</tr>
<tr>
<td>537</td>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td>123</td>
<td>67</td>
<td>478</td>
</tr>
<tr>
<td>38</td>
<td>9</td>
<td>123</td>
</tr>
</tbody>
</table>

This example uses \( B=10 \) and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.
Radix Sort Example (2\textsuperscript{nd} pass)

<table>
<thead>
<tr>
<th>After 1\textsuperscript{st} pass</th>
<th>Bucket sort by 10’s digit</th>
<th>After 2\textsuperscript{nd} pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>721 3 123 537 67 478 38 9</td>
<td></td>
<td>3 9 721 537 38 478</td>
</tr>
</tbody>
</table>

Radix Sort Example (3\textsuperscript{rd} pass)

<table>
<thead>
<tr>
<th>After 2\textsuperscript{nd} pass</th>
<th>Bucket sort by 100’s digit</th>
<th>After 3\textsuperscript{rd} pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 9 721 123 537 38 478</td>
<td>3 9 123 537 38 478</td>
<td></td>
</tr>
</tbody>
</table>

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

RadixSort

BucketSort on lsd:

0 1 2 3 4 5 6 7 8 9

BucketSort on next-higher digit:

0 1 2 3 4 5 6 7 8 9

BucketSort on msd:

0 1 2 3 4 5 6 7 8 9

Your Turn

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

Summary of sorting

- Sorting choices:
  - \(O(N^2)\) – Bubblesort, Insertion Sort
  - \(O(N \log N)\) average case running time:
    - Heapsort: In-place, not stable.
    - Mergesort: \(O(N)\) extra space, stable.