Today’s Outline

• Announcements
  – Homework #6/7 due Thurs 12/9 at 11:45pm.

• Sorting

Why Sort?

Sorting: The Big Picture

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

<table>
<thead>
<tr>
<th>Simple algorithms: $O(n^2)$</th>
<th>Fancier algorithms: $O(n \log n)$</th>
<th>Comparison lower bound: $\Omega(n \log n)$</th>
<th>Specialized algorithms: $O(n)$</th>
<th>Handling huge data sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>Heap sort</td>
<td>Merge sort</td>
<td>Bucket sort</td>
<td>External sorting</td>
</tr>
<tr>
<td>Selection sort</td>
<td>Radix sort</td>
<td>Quick sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bubble sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shell sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Insertion Sort: Idea

• At the $k^{th}$ step, put the $k^{th}$ input element in the correct place among the first $k$ elements
• Result: After the $k^{th}$ step, the first $k$ elements are sorted.

Runtime:

worst case : 
best case : 
average case :

Selection Sort: Idea

• Find the smallest element, put it 1st
• Find the next smallest element, put it 2nd
• Find the next smallest, put it 3rd
• And so on …
Selection Sort: Code

```c
void SelectionSort (Array a[0..n-1]) {
    for (i=0; i<n; ++i) {
        j = Find index of smallest entry in a[i..n-1]
        Swap(a[i],a[j])
    }
}
```

Runtime:
- worst case : 
- best case : 
- average case :

12/06/2010 8

Sorts using other data structures:

AVL Sort?

Heap Sort?

HeapSort:
Using Priority Queue ADT (heap)

23 44 76 13 18 801 27

Shove all elements into a priority queue, take them out smallest to largest.

Runtime:
12/06/2010 9

AVL Sort

Runtime:
Would the simpler “Splay sort” take any longer than this?

12/06/2010 10

Divide and conquer

- A common and important technique in algorithms
  - Divide problem into parts
  - Solve parts
  - Merge solutions

12/06/2010 12

Divide and Conquer Sorting

- MergeSort:
  - Divide array into two halves
  - Recursively sort left and right halves
  - Merge halves
- QuickSort:
  - Partition array into small items and large items
  - Recursively sort the two smaller portions

12/06/2010 13
Merge Sort?

Merge Sort

- MergeSort (Array [1..n])
  1. Split Array in half
  2. Recursively sort each half
  3. Merge two halves together

```
MergeSort [a1..n, a2..n]

i1=1, i2=1

While (i1<n, i2<n) {
    if (a1[i1] < a2[i2]) {
        Next is a1[i1]
        i1++
    } else {
        Next is a2[i2]
        i2++
    }
}
```

"The 2-pointer method"

Now throw in the dregs...

Perform mergeSort

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>9</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
</table>

Divide:
Divide:
Divide:
Divide:
Merge:
Merge:

Merge Sort: Complexity

Auxiliary array

- The merging requires an auxiliary array

```
2 4 8 9 1 3 5 6
```

Properties of MergeSort

- Definition: In-place
  - Can be done without extra memory
- MergeSort: Not in-place
  - Requires Auxiliary array
Quicksort

- Uses divide and conquer
- Doesn’t require $O(N)$ extra space like MergeSort
- Partition into left and right
  - Left less than pivot
  - Right greater than pivot
- Recursively sort left and right
- Concatenate left and right

Quick Sort

1. Pick a “pivot”
2. Divide into less-than & greater-than pivot
3. Sort each side recursively

The steps of QuickSort

Selecting the pivot

- Ideas?

Perform quicksort

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>9</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
</table>

Divide:

Divide:

Divide:

Merge:

QuickSort Example

Choose the pivot as the median of three.

Place the pivot and the largest at the right and the smallest at the left.
QuickSort Example

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>7</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
</table>

Move i to the right to be larger than pivot.
Move j to the left to be smaller than pivot.
Swap

Recursive Quicksort

```plaintext
QuickSort(A[]): integer array, left, right : integer:
  pivotindex : integer;
  if left < 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.

Cutoff for quicksort

• Quicksort performs poorly on small sets
  – In fact insertion sort does better
• Small sets occur often due to the recursion
• So below a certain set size, or cutoff, switch to insertion sort

Recurrence Relations

Write the recurrence relation for QuickSort:

• Best Case:
• Worst Case:
QuickSort:
Worst case complexity

QuickSort:
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.

Quicksort Complexity

- Worst case: $O(n^2)$
- Best case: $O(n \log n)$
- Average Case: $O(n \log n)$

Mergesort and massive data

- MergeSort is the basis of massive sorting
- Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
- Mergesort scans linearly through arrays, leading to (relatively) efficient sequential disk access
- In-memory sorting of reasonable blocks can be combined with larger mergesorts
- Mergesort can leverage multiple disks

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)
  - (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  - 6 orderings = 3·2·1 = 3! (i.e., “3 factorial”)
  - 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)...(2)(1) = N! possible orderings

Decision Tree

- a < b < c,  b < c < a,  c < a < b,  a < c < b,  b < a < c,  c < b < a

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:

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

\[ \log(N!) = \log(N - (N-1) - (N-2) - (2) - (1)) \]
\[ = \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \]
\[ \geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2} \]
\[ \geq \frac{N}{2} \log \frac{N}{2} \]
\[ \geq \frac{N}{2} (\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2} \]
\[ = \Omega(N \log N) \]

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

- 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 \( \text{count} \) 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>

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)

Input data

<table>
<thead>
<tr>
<th>Bucket sort by 1’s digit</th>
<th>After 1st pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>678</td>
<td>9</td>
</tr>
<tr>
<td>537</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

This example uses 8-10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

Radix Sort Example (2nd pass)

<table>
<thead>
<tr>
<th>After 1st pass</th>
<th>Bucket sort by 10’s digit</th>
<th>After 2nd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>537</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>478</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

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

Radix Sort Example (3rd pass)

<table>
<thead>
<tr>
<th>After 2nd pass</th>
<th>Bucket sort by 100’s digit</th>
<th>After 3rd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>721</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>537</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>478</td>
<td>67</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After 3rd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>123</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>67</td>
</tr>
<tr>
<td>478</td>
</tr>
</tbody>
</table>
RadixSort

- Input: 126, 328, 636, 341, 416, 131, 328

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 |

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

Summary of sorting

- $O(n^2)$ average, worst case:
  - Selection Sort, Bubblesort, Insertion sort
- $O(n^{4/3})$ worst case:
  - Shell sort
- $O(n \log n)$ average case:
  - Heapsort: in-place, not stable
  - Mergesort: $O(n)$ extra space, stable, massive data
  - Quicksort: Claimed fastest in practice, but $O(n^2)$ worst case.
- $\Omega(n \log n)$ worst and average case:
  - Any comparison-based sorting algorithm
- $O(n)$
    - Poor memory use can undercut performance.