Today’s Outline

• Announcements
  – HW #6-7
    • Assignment due Thurs March 12th.
  • Sorting

Why Sort?

Sorting: The Big Picture

Problem: 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>Bucket sort</td>
<td>External sorting</td>
<td></td>
</tr>
<tr>
<td>Selection sort</td>
<td>Merge sort</td>
<td>Radix sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bubble sort</td>
<td>Quick sort</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 …
Mystery(int array a[]) {
    for (int p = 1; p < length; p++) {
        int tmp = a[p];
        for (int j = p; j > 0 && tmp < a[j-1]; j--)
            a[j] = a[j-1];
        a[j] = tmp;
    }
}

What sort is this?

What is its running time?
Best?
Avg?
Worst?

Selection Sort: Code

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 : 

Sorts using other data structures:

How?
Runtime?

AVL Sort?

Heap Sort?

Splay Sort?

HeapSort:
Using Priority Queue ADT (heap)

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

Runtime:

AVL Sort

Runtime:

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

Merge Sort?
Merge Sort

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

```
MergeSort[Array [1..n]]
```

1. Split Array in half
2. Recursively sort each half
3. Merge two halves together

**Merge** (a1[1..n], a2[1..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++
    }
}
```

Now throw in the dregs...

"The 2-pointer method"

---

Quick Sort

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

**QuickSort** Example

```
0 1 2 3 4 5 6 7 8 9
```

- 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**

```
0 1 2 3 4 5 6 7 8 9
```

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

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.

Recurrence Relations

Write the recurrence relation for Quicksort:

• Best Case:

• Worst Case:

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.
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

Are the following: stable? in-place?

<table>
<thead>
<tr>
<th>Sort</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>Heap 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?
  – 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), (c a b), (c b a)
  – 6 orderings = \(3!\) = 3! (ie, “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
  – \((N-1)(N-2)\cdots(2)(1) = N!\) possible orderings

Decision Tree

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?
- So the decision tree has height: $h_{\text{root}}$

$log(N!)$ is $\Omega(N \log N)$

$log(N!) = \log(N - 1) + \log(N - 2) + \ldots + \log 1$

$\geq \log N + \log(N - 1) + \log(N - 2) + \ldots + \log \frac{N}{2}$

$= \frac{N}{2} \log \frac{N}{2}$

$\geq \frac{N}{2} (\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2}$

$= \Omega(N \log N)$

$\Omega(N \log N)$

- 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, CountingSort)

If all values to be sorted are known to be between 1 and $K$, create an array $count$ 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)

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>721</td>
<td>3</td>
</tr>
<tr>
<td>537</td>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td>9</td>
<td>67</td>
<td>123</td>
</tr>
<tr>
<td>721</td>
<td>3</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>38</td>
<td>478</td>
</tr>
<tr>
<td>123</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>67</td>
<td>9</td>
<td>9</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.

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

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>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>537</td>
<td>67</td>
<td>537</td>
</tr>
<tr>
<td>67</td>
<td>478</td>
<td>478</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

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>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>721</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>537</td>
<td>537</td>
<td>537</td>
</tr>
<tr>
<td>38</td>
<td>478</td>
<td>478</td>
</tr>
<tr>
<td>67</td>
<td>721</td>
<td>721</td>
</tr>
</tbody>
</table>

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. Why?
  - 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 in section 7.10