Shell Sort

Biostatistics 615
Lecture 8
Homework 2

- Results still pretty impressive!

- Topics stressed:
  - Sequential Search
  - Floating Point Precision
Sequential Search

- Note that efficiency of search improves slightly when there are more matches.

- Tip:
  - Use arrays to cycle through different parameter settings.
## Sampling With Replacement (I)

### Numbers Between 0 and 9,999

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Matches</th>
<th>StDev</th>
<th>Comparisons</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>0.1</td>
<td>0.3</td>
<td>999</td>
<td>0.10</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1.0</td>
<td>1.0</td>
<td>9995</td>
<td>1.00</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>1.0</td>
<td>1.0</td>
<td>9951</td>
<td>1.00</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>9.9</td>
<td>3.1</td>
<td>99504</td>
<td>9.95</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>9.5</td>
<td>2.9</td>
<td>95244</td>
<td>9.52</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>95.2</td>
<td>9.3</td>
<td>952385</td>
<td>95.17</td>
</tr>
</tbody>
</table>
### Sampling With Replacement (II)

#### Numbers Between 1,000 and 9,999

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Matches</th>
<th>StDev</th>
<th>Comparisons</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>0.1</td>
<td>0.3</td>
<td>999</td>
<td>0.11</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1.1</td>
<td>1.1</td>
<td>9994</td>
<td>1.11</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>1.1</td>
<td>1.0</td>
<td>9945</td>
<td>1.11</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>11.1</td>
<td>3.3</td>
<td>99446</td>
<td>11.05</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>10.5</td>
<td>3.1</td>
<td>94737</td>
<td>10.52</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>105.0</td>
<td>9.7</td>
<td>947497</td>
<td>105.17</td>
</tr>
</tbody>
</table>
Floating Point Precision

- Smallest value
  - That can be added to 1
    - $2^{-52}$ or $2.2 \times 10^{-16}$
  - That can be subtracted from 1
    - $2^{-53}$ or $1.1 \times 10^{-16}$
  - That is distinct from zero
    - $2^{-1074}$ or $4.9 \times 10^{-324}$

- Note: Need two variables to calculate this quantity
Last Lecture ...

- Properties of Sorting Algorithms
  - Adaptive
  - Stable

- Elementary Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort

- Several Inches of Snow!...
Recipe: Selection Sort

- Find the smallest element
  - Place it at beginning of array

- Find the next smallest element
  - Place it in the second slot

- ...
Selection Sort

**Recipe:**

- Find the smallest element
  - Place it at beginning of array

- Find the next smallest element
  - Place it in the second slot

- ...
C Code: Selection Sort

```c
void sort(Item *a, int start, int stop)
{
    int i, j;

    for (i = start; i < stop - 1; i++)
    {
        int min = i;
        for (j = i + 1; j < stop; j++)
            if (isLess(a[j], a[min]))
                min = j;
        Exchange(a[i], a[min]);
    }
}
```
Recipe: Insertion Sort

- The “Simple Sort” we first considered

- Consider one element at a time
  - Place it among previously considered elements
  - Must move several elements to “make room”

- Can be improved, by “adapting to data”
Insertion Sort

Recipe:

• Consider one element at a time
  • Place it among previously ordered elements

  • Optimizations:
    • Stop comparisons early
    • Avoid extra exchanges
    • Identify smallest element first
Adaptive Insertion Sort

```c
void sort(Item *a,
          int start, int stop)
{
    int i, j;

    for (i = stop; i > start; i--)
        CompExch(a[i-1], a[i]);

    for (i = start + 2; i <= stop; i++)
    {
        int j = i; Item val = a[j];
        while (isLess(val, a[j-1]))
        {
            a[j] = a[j-1]; j--; }
        a[j] = val;
    }
}
```
Recipe: Bubble Sort

- Pass through the array
  - Exchange elements that are out of order

- Repeat until done…

- Very “popular”
  - Very inefficient too!
C Code: Bubble Sort

```c
void sort(Item *a,
          int start, int stop)
{
    int i, j;

    for (i = start; i <= stop; i++)
        for (j = stop; j > i; j--)
            CompExch(a[j-1], a[j]);
}
```
Bubble Sort

Notice:

Each pass moves one element into position.

Right portion of array is partially sorted
Shaker Sort

**Notice:**

Things improve slightly if bubble sort alternates directions…
Notes on Bubble Sort

- Similar to non-adaptive Insertion Sort
  - Moves through unsorted portion of array

- Similar to Selection Sort
  - Does more exchanges per element

- Stop when no exchanges performed
  - Adaptive, but not as effective as Insertion Sort
<table>
<thead>
<tr>
<th>Selection</th>
<th>Insertion</th>
<th>Bubble</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Selection Diagram" /></td>
<td><img src="image2" alt="Insertion Diagram" /></td>
<td><img src="image3" alt="Bubble Diagram" /></td>
</tr>
</tbody>
</table>
Performance Characteristics

- Selection, Insertion, Bubble Sorts

- All quadratic
  - Running time differs by a constant

- Which sorts do you think are stable?
Selection Sort

- Exchanges
  - $N - 1$

- Comparisons
  - $N \cdot (N - 1) / 2$

- Requires about $N^2 / 2$ operations
- Ignoring updates to min variable
Adaptive Insertion Sort

- **Half - Exchanges**
  - About $N^2 / 4$ on average (random data)
  - $N * (N – 1) / 2$ (worst case)

- **Comparisons**
  - About $N^2 / 4$ on average (random data)
  - Slightly more than $N * (N – 1) / 2$ (worst case)

- Requires about $N^2 / 4$ operations
- Requires nearly linear time on nearly sorted data
Bubble Sort

- Exchanges
  - $N \times (N - 1) / 2$

- Comparisons
  - $N \times (N - 1) / 2$

- Average case and worst case very similar, even for adaptive method
## Empirical Comparison

<table>
<thead>
<tr>
<th></th>
<th>Selection</th>
<th>Insertion</th>
<th>Insertion (adaptive)</th>
<th>Bubble</th>
<th>Shaker</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N</strong></td>
<td><strong>1000</strong></td>
<td><strong>2000</strong></td>
<td><strong>4000</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>29</td>
<td>15</td>
<td>45</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>119</td>
<td>62</td>
<td>182</td>
<td>138</td>
</tr>
</tbody>
</table>

(Running times in seconds)
New Today ...

- Shellsort
  - An algorithm that beats the $O(N^2)$ barrier
  - Suitable performance for general use

- Very popular
  - It is the basis of the default R sort() function
Shellsort

- Donald L. Shell (1959)
  - A High-Speed Sorting Procedure
    Communications of the Association for Computing Machinery 2:30-32
  - Systems Analyst working at GE

- Also called:
  - Diminishing increment sort
  - “Comb” sort
**Intuition**

- Insertion sort is effective:
  - For small datasets
  - For data that is nearly sorted

- Insertion sort is inefficient when:
  - Elements must move far in array
The Idea ...

- Allow elements to move large steps
- Bring elements close to final location
  - Make array almost sorted
- How?
Shellsort Recipe

- For some decreasing step size $h$
  - Every sequence must end at 1
  - $\ldots, 8, 4, 2, 1 \ldots$

- Sort the array so elements separated by exactly $h$ elements apart are in order
  - if $h = 4$
    - sort elements 1, 5, 9, 13 $\ldots$
    - sort elements 2, 6, 10, 14 $\ldots$
Shellsort Notes

- Any decreasing sequence that ends at 1 will do…
  - The final pass ensures array is sorted

- Different sequences can dramatically increase (or decrease) performance

- Code is similar to insertion sort
C Code: Shellsort

```c
void sort(Item * a, int * sequence, int start, int stop)
{
    int step = 0, i;

    for (step = 0; sequence[step] >= 1; step++)
    {
        int inc = sequence[step];

        for (i = start + inc; i <= stop; i++)
        {
            int j = i; Item val = a[i];
            while ((j >= start + inc) && val < a[j - inc])
            {
                a[j] = a[j - inc]; j -= inc; }
            a[j] = val;
        }
    }
}
```
Array gradually gains order

Eventually, we approach the ideal case where insertion sort is $O(N)$
Increment Sequences

- Good:
  - Consecutive numbers are relatively prime
  - Increments decrease roughly exponentially

- An example of a bad sequence:
  - 1, 2, 4, 8, 16, 32 …
  - What happens if the largest values are all in odd positions?
Shellsort Properties

- Not very well understood

- For good increment sequences, requires time proportional to
  - $N (\log N)^2$
  - $N^{1.25}$

- We will discuss them briefly …
h-Sorted Array

- An array such that taking every $h$th element (starting anywhere) yields a sorted array
- A set of several array interleaved together
Property I

- If we $h$-sort an array that is $k$-ordered...
- Result is an $h$- and $k$- ordered array

- Tricky to prove, but consider:
  - When sorting elements $i$ and $i+h$ ...
  - Elements $i+k$ and $i+k+h$ are sorted in parallel series
  - After sorting, $a[i] \leq a[i+k]$ and $a[i+h] \leq a[i+k+h]$
5 – Sorting an Array

5-sorting an array

Elements in each subarray color coded
4-Sorting 5-Sorted Array

4 - sorting an array

ABCDEABCDEABCDEABCDE

A1 E1 D2 C3 B4

ABCDEABCDEABCDEABCDE

B1 A2 E3 D4 C5
Property II

- No more than \((h-1)(k-1)/g\) comparisons to \(g\) sort a file that is \(h\)- and \(k\)-sorted

- For \(h\) and \(k\) relatively prime

- Elements further than \((h-1)(k-1)\) can be reached by a series of steps of size \(h\) or \(k\) (i.e. through elements known to be in order)
Optimal Performance?

Consider a triangle of increments:

- Each element is:
  - double the number above to the right
  - three times the number above to the left

1

2 3

4 6 9

8 12 18 27

16 24 36 54 81

32 48 72 108 162 243
Optimal Performance?

- Start from bottom to top, right to left

- After first row, every sub-array is 3-sorted and 2-sorted
  - No more than 1 exchange!

- In total, there are $\sim \log_2 N \log_3 N / 2$ increments
  - About $N (\log N)^2$ performance possible
## Running Time (in secs)

<table>
<thead>
<tr>
<th>N</th>
<th>Pow2</th>
<th>Knuth</th>
<th>Merged</th>
<th>Seq1</th>
<th>Seq2</th>
</tr>
</thead>
<tbody>
<tr>
<td>125000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>250000</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>500000</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1000000</td>
<td>14</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2000000</td>
<td>42</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4000000</td>
<td>118</td>
<td>10</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Pow2 – 1, 2, 4, 8, 16 ... \((2^i)\)  
Knuth – 1, 4, 13, 40, ... \((3 \times \text{previous} + 1)\)  
Seq1 – 1, 5, 41, 209, ... \((4^i - 3 \times 2^i + 1)\)  
Seq2 – 1, 19, 109, 505 ... \((9 \times 4^i - 9 \times 2^i + 1)\)  
Merged – Alternate between Seq1 and Seq2
Not Sensitive to Input ...
Today: Shellsort

- Breaks the $N^2$ barrier
  - Does not compare all pairs of elements, ever!

- Average and worst-case performance similar

- Difficult to analyze precisely
Reading

- Sedgewick, Chapter 6