Shell Sort

Biostatistics 615/815
Lecture 6
Housekeeping Note: Homework Grading

- Weihua Guan is the GSI
- He requests that you e-mail him source code for your assignments to:
  
  wguan@umich.edu

- Thanks!
Last Lecture ...

- Properties of Sorting Algorithms
  - Adaptive
  - Stable

- Elementary Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
### “Stable” and “Unstable” Sorts

**Stable Sort by State**

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Season</th>
</tr>
</thead>
<tbody>
<tr>
<td>Las Vegas</td>
<td>NV</td>
<td>All Year</td>
</tr>
<tr>
<td>Denver</td>
<td>CO</td>
<td>All Year</td>
</tr>
<tr>
<td>Traverse City</td>
<td>MI</td>
<td>Summer</td>
</tr>
<tr>
<td>Holland</td>
<td>MI</td>
<td>Summer</td>
</tr>
<tr>
<td>Boulder</td>
<td>CO</td>
<td>Winter</td>
</tr>
</tbody>
</table>

**Unstable Sort by State**

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Season</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boulder</td>
<td>CO</td>
<td>Winter</td>
</tr>
<tr>
<td>Denver</td>
<td>CO</td>
<td>All Year</td>
</tr>
<tr>
<td>Holland</td>
<td>MI</td>
<td>Summer</td>
</tr>
<tr>
<td>Traverse City</td>
<td>MI</td>
<td>Summer</td>
</tr>
<tr>
<td>Las Vegas</td>
<td>NV</td>
<td>All Year</td>
</tr>
</tbody>
</table>

The table shows the cities, states, and seasons for both stable and unstable sorts.
Recap

- Selection, Insertion, Bubble Sorts

- Can you think of:
  - One property that all of these share?
  - One useful advantage for Selection sort?
  - One useful advantage for Insertion sort?

- Situations where these sorts can be used?
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

- **Tunable algorithm**
  - Can use different orderings for comparisons
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
  - Back then, most computers read punch-cards

- Also called:
  - Diminishing increment sort
  - “Comb” sort
  - “Gap” 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 in large steps
- Bring elements close to final location
  - First, ensure array is nearly sorted ...
  - ... then, run insertion sort
- How?
  - Sort interleaved arrays first
Shellsort Recipe

- Decreasing sequence of step sizes $h$
  - Every sequence must end at 1
  - ... , 8, 4, 2, 1

- For each $h$, sort sub-arrays that start at arbitrary element and include every $h^{th}$ element
  - if $h = 4$
    - Sub-array with elements 1, 5, 9, 13 ...
    - Sub-array with elements 2, 6, 10, 14 ...
    - Sub-array with elements 3, 7, 11, 15 ...
    - Sub-array with elements 4, 8, 12, 16 ...
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
Sub-arrays when Increment is 5

5-sorting an array

Elements in each subarray color coded
```c
void sort(Item a[], int sequence[], int start, int stop)
{
    int step, i;

    for (int 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 $O(N)$
C Code: Using a Shell Sort

```c
#include "stdlib.h"
#include "stdio.h"

#define Item int

void sort(Item a[], int sequence[], int start, int stop);

int main(int argc, char * argv[])
{
    printf("This program uses shell sort to sort a random array\n\n");
    printf("Parameters: [array-size]\n\n");

    int size = 100;
    if (argc > 1) size = atoi(argv[1]);

    int sequence[] = {364, 121, 40, 13, 4, 1, 0};
    int * array = (int *) malloc(sizeof(int) * size);

    srand(123456);
    printf("Generating %d random elements ...\n", size);
    for (int i = 0; i < size; i++)
        array[i] = rand();

    printf("Sorting elements ...\n", size);
    sort(array, sequence, 0, size - 1);

    printf("The sorted array is ...\n");
    for (int i = 0; i < size; i++)
        printf("%d  ", array[i]);
    printf("\n");
    free(array);
}
```
Note on Example Code: Declaring Variables “Late”

- Instead of declaring variables immediately after opening a `{}` block, wait until first use
  - Possibility introduced with C++

- Supported by most modern C compilers
  - In UNIX, use g++ instead of gcc to compile
## Running Time (in seconds)

<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...
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 …
Definition: $h$-Sorted Array

- An array where taking every $h^{th}$ element (starting anywhere) yields a sorted array.
- Corresponds to a set of several* sorted arrays interleaved together.
  - * There could be $h$ such arrays.
Property 1

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

- $h$-sort preserves $k$-order!

- Seems tricky to prove, but considering a set of 4 elements as they are sorted in parallel makes things clear...
Property I

- Result of $h$-sorting an array that is $k$-ordered is an $h$- and $k$- ordered array

- Consider 4 elements, in $k$-ordered array:
  - $a[i] \leq a[i+k]
  - $a[i+h] \leq a[i+k+h]

- After $h$-sorting, $a[i]$ contains minimum and $a[i+k+h]$ contains maximum of all 4
Property II

- If $h$ and $k$ are relatively prime …

- Items that are more than $(h-1)(k-1)$ steps apart must be in order
  - Possible to step from one to the other using steps size $h$ or $k$
  - That is, by stepping through elements known to be in order.

- Insertion sort requires no more $(h-1)(k-1)$ comparisons per item to sort array that is $h$- and $k$-sorted
  - Or $(h-1)(k-1)/g$ comparisons to carry a $g$-sort
Property II

- Consider $h$ and $k$ sorted arrays
  - Say $h = 4$ and $k = 5$

- Elements that must be in order
Property II

- Consider $h$ and $k$ sorted arrays
  - Say $h = 4$ and $k = 5$

- More elements that must be in order …
Property II

- Combining the previous series gives the desired property that elements \((h-1)(k-1)\) elements away must be in order.
An optimal series?

- Considering the two previous properties...

- A series where every sub-array is known to be 2- and 3- ordered could be sorted with a single round of comparisons

- Is it possible to construct series of increments that ensures this?
  - Before $h$-sorting, ensure $2h$ and $3h$ sort have been done …
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
  - \(< \log_2 N \log_3 N\) increments

```
  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 ~ $\log_2 N \log_3 N / 2$ increments
  - About $N (\log N)^2$ performance possible
Today’s Summary: 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