More Quicksort
Mergesort

Biostatistics 615
Lecture 10
Scheduling ...

- I will hand out sample midterm next week

- Revision Q & A
  - February 17

- Mid-term Exam
  - February 19
  - Take Home
Dynamic Programming

- **Top Down**
  Recursive implementation, with additional code to store results of each evaluation (at the end) and to use previously stored results (at the beginning)

- **Bottom Up**
  Evaluate small values of the function and proceed to successively larger values.
Problem 1

- Using top-down dynamic programming, evaluate:

\[
\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1}
\]

- Must initialize matrix or results could be wrong
Initializing Matrix in R

- `matrix(nrow = 10, ncol = 10)`
  - Creates a matrix with 10 rows and columns

- `matrix(data = 0, nrow = 10, ncol = 10)`
  - The optional parameter allows the matrix to be pre-initialized with an element of choice, which could even be a vector!
Dynamic Matrix in C

```c
int ** matrix;

// Allocate an array of pointers
matrix = malloc(sizeof(int *) * nrow);

// Allocate an array of integers for each row
for (i = 0; i < nrow; i++)
  matrix[i] = malloc(sizeof(int) * ncol);
```
Problem 2

- Using bottom-up dynamic programming, evaluate:

\[ C(N) = \begin{cases} 
N + \frac{1}{N} \sum_{k=1}^{N} C(k-1) + C(N-k) & N \geq 2 \\
0 & N \leq 1 
\end{cases} \]

- Calculation is still slow, due to nested sum… but this can be simplified
int comparisons[Nmax];
double inner_sum = 0.0;

comparisons[1] = comparisons[0] = 0;

for (i = 2; i < Nmax; i++)
{
    inner_sum += 2 * comparisons[i - 1];
    comparisons[i] = i + inner_sum / i;
}
Last Lecture ... 

- Quick Sort
  - Choice of Median
  - Sorting Small Sub-arrays

- Quick Sort-based Selection
  - Finding quantiles of a distribution
Today

- Further improvements to Quick Sort
  - Maintaining explicit stacks

- Merge Sort
  - Another $N \log N$ sort
  - Fastest stable sort
Quick Sort: The Idea

- Divide array into smaller sub-arrays
  - Sort right and left sub-arrays independently

- Find a partitioning element such that:
  - All elements to the right are greater
  - All elements to the left are smaller
void quicksort(Item * a, int start, int stop)
{
    int i;

    if (stop <= start) return;

    i = partition(a, start, stop);
    quicksort(a, start, i - 1);
    quicksort(a, i + 1, stop);
}
C Code: Partitioning

```c
int partition(Item * a, int start, int stop)
{
    int up = start, down = stop - 1, part = a[stop];

    while (1)
    {
        while (isLess(a[up], part))
            up++;
        while (isLess(part, a[down]) && (up < down))
            down--;
        if (up >= down) break;
        Exchange(a[up], a[down]);
        up++, down--;
    }
    Exchange(a[up], a[stop]);
    return up;
}
```
Improvements We Considered

- Delaying sort for small sub-arrays
- Using median-of-three partitioning
  - Random partitioning element can also help!
- Avoiding deep recursion
## Sedgewick’s Timings

<table>
<thead>
<tr>
<th>N</th>
<th>Basic</th>
<th>Insertion</th>
<th>Insertion After</th>
<th>Ignore Duplicates</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,500</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>25,000</td>
<td>16</td>
<td>14</td>
<td>13</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>50,000</td>
<td>37</td>
<td>31</td>
<td>31</td>
<td>41</td>
<td>45</td>
</tr>
<tr>
<td>100,000</td>
<td>91</td>
<td>78</td>
<td>76</td>
<td>113</td>
<td>103</td>
</tr>
</tbody>
</table>

Arrays including first $N$ words in text of “Moby Dick”. 
The Problem

- The computer stack has a limited size
- Quick Sort can call itself up to N-1 times
  - Unlikely, but very deep recursion is possible!
- Can we provide a guarantee on depth of recursion?
The Solution

- After partitioning, handle smaller half first
  - At most, $\log_2 N$ smaller halves!

- Keep track of sections to be solved in “explicit” stack
Non-Recursive QuickSort

```c
void quicksort(Item * a, int start, int stop)
{
    int i, s = 0, stack[64];

    stack[s++] = start; stack[s++] = stop;
    while (s > 0)
    {
        stop = stack[--s]; start = stack[--s];
        if (start >= stop) continue;

        i = partition(a, start, stop);
        if (i - start > stop - i)
        {
            stack[s++] = start; stack[s++] = i - 1;
            stack[s++] = i + 1; stack[s++] = stop; }
        else { stack[s++] = i + 1; stack[s++] = stop;
                stack[s++] = start; stack[s++] = i; }
    }
}
```
Explicit Stacks

- A common feature in computer programs
- A simple way to avoid recursion
  - More effort for the programmer
- Another application is in graph traversal
Quick Sort Summary

- Divide and Conquer Algorithm
  - Recursive calls can be “hidden”

- Optimizations
  - Choice of median
  - Threshold for brute-force methods
  - Limiting depth of recursion
Merge Sort

- Divide-And-Conquer Algorithm
  - Divides a file in two halves
  - Merges sorted halves

- The “opposite” of quick sort

- Requires additional storage
C Code: Merge Sort

```c
void mergesort(Item * a, int start, int stop)
{
    int m = (start + stop)/2;

    if (start <= stop) return;

    mergesort(a, start, m);
    mergesort(a, m + 1, stop);
    merge(a, start, m, stop);
}
```
Merge Pattern N = 21
Merging Sorted Arrays

- Consider two arrays
- Assume they are both in order
- Can you think of a merging strategy?
void merge(Item* c, Item* a, int N, Item* b, int M)
{
    int i, j, k;
    for (k = 0; k < M + N; k++)
    {
        if (i == N) { c[k] = b[j++]; continue; }
        if (j == M) { c[k] = a[i++]; continue; }
        if (isLess(b[j], a[i]))
            { c[k] = b[j++]; }
        else
            { c[k] = a[i++]; }
    }
}
“In-Place” Merge

- For sorting, we would like to:
  - Starting with sorted halves
    - \( a[\text{start} \ldots m] \)
    - \( a[m \ldots \text{end}] \)
  - Generate a sorted stretch
    - \( a[\text{start} \ldots \text{end}] \)

- We would like an in-place sort…
  - Or something that “looks” like one
Abstract In-Place Merge

- For caller, performs like in-place merge
- Creates copies of two sub-arrays
- Replaces contents with merge results
- Check for end of input can be avoided by inverting second array.
Avoiding End-of-Input Check

At each point, compare elements $i$ and $j$.

Then select the smallest element.

Move $i$ or $j$ towards the middle, as appropriate.
C Code: Abstract In-place Merge

Item aux[maxN];

void merge(Item* a, int start, int m, int stop)
{
    int i, j, k;
    for (i = start; i <= m; i++)
        aux[i] = a[i];
    for (j = m + 1; j <= stop; j++)
        aux[stop + m + 1 - j] = a[j];
    for (j = stop, i = k = start; k <= stop; k++)
        if (isLess(aux[j], aux[i])
            a[k] = aux[j--];
        else
            a[k] = aux[i++];
}
Merge Sort in Action
Merge Sort Notes

- Order $N \log N$
  - Number of comparisons independent of data
  - Exactly $\log N$ rounds
  - Each requires $N$ comparisons

- Merge sort is stable
- Insertion sort for small arrays is helpful
### Sedgewick’s Timings (secs)

<table>
<thead>
<tr>
<th>N</th>
<th>QuickSort</th>
<th>MergeSort</th>
<th>MergeSort*</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>24</td>
<td>53</td>
<td>43</td>
</tr>
<tr>
<td>200,000</td>
<td>52</td>
<td>111</td>
<td>92</td>
</tr>
<tr>
<td>400,000</td>
<td>109</td>
<td>237</td>
<td>198</td>
</tr>
<tr>
<td>800,000</td>
<td>241</td>
<td>524</td>
<td>426</td>
</tr>
</tbody>
</table>

Array of floating point numbers.
Non-Recursive Merge Sort

- First sort all sub-arrays of 1 element

- Perform successive merges
  - Merge results into sub-arrays of 2 elements
  - Merge results into sub-arrays of 4 elements
  - ...
Bottom-Up Merge Sort

```c
int min(int a, int b)
{ return a < b ? a : b; }

void merge(Item* a, int start, int stop)
{
    int i, m;

    for (m = 1; m < stop - start; m += m)
        for (i = start; i < stop; i += stop)
            merge(a, i, i+m-1, min(i+m+m-1, stop));
}
```
Merging Pattern for N = 21
Today ...

- Quick Sort
- Merge Sort
- Unraveled Recursive Sorts
- Contrasting approaches to divide and conquer
Recommended Reading

- For QuickSort
  - Sedgewick, Chapter 7

- For MergeSort
  - Sedgewick, Chapter 8