Merge Sort

Biostatistics 615/815
Lecture 11
Scheduling ...

- I will hand out sample midterm next week

- Revision Q & A
  - October 26 or November 1?

- Mid-term Exam
  - November 1 or November 3?
  - Take Home
Problem Set 3 Notes

- 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
Notes

- **Use** `double` **to store function results**
  - Allows solution to larger problems
  - `long` **and** `long long` are alternatives in Unix

- **Additional rules**
  - Choose\((N, k) = \text{Choose}(N, N - k)\)

- **Use global variable for intermediate results**
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
Speedy Solution ...

```c
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

- Choose a partitioning element …

- Organize array such that:
  - All elements to the right are greater
  - All elements to the left are smaller

- Sort right and left sub-arrays independently
Improvements We Considered

- Delay sort for small sub-arrays
  - Use insertion sort instead

- Use median-of-three partitioning
  - Random partitioning element can also help!

- Avoid recursion
C Code: QuickSort

```c
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];

    if (stop <= start) return start;

    while (true)
    {
        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;
}
```
The Selection Problem

- Consider the problem of finding the $k^{th}$ smallest element in an array.
- Useful when searching for the median, quartiles, deciles or percentiles.
Selection – small $k$

- We can solve the problem in $O(Nk) = O(N)$

- One approach:
  - Perform $k$ passes
  - For pass $j$, find $j$ smallest element

- Another approach:
  - Maintain a small array with $k$ smallest elements
Selection – for large k

- One option is to sort array…

- But we only need to bring $k$ into position

- Focus on one side of current partition
C Code: Selection

// Places k\textsuperscript{th} smallest element in the k\textsuperscript{th} position
// within array. Could move other elements.
void select(Item * a, int start, int stop, int k)
{
    int i;

    if (start <= stop) return;

    i = partition(a, start, stop);

    if (i > k) select(a, start, i - 1);
    if (i < k) select(a, i + 1, stop);
}
C Code: Without Recursion

void select(Item * a, int start, int stop, int k)
{
    int i;

    while (start < stop)
    {
        i = partition(a, start, stop);

        if (i >= k) stop = i - 1;
        if (i <= k) start = i + 1;
    }
}
Selection

- Quicksort based method is $O(N)$
  - Rough argument:
    - First pass through $N$ elements
    - Second pass through $N/2$ elements
    - Third pass through $N/4$ elements
    - ...
    - All passes will take time small constant * $N$

- Common application: finding $k$ smallest values in a simulation to save for further analyses
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
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 - 1; } 
    }
}
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 (stop <= start) 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 * merge,
    Item * a, int N, Item* b, int M)
{
    int i, j, k;
    for (k = 0; k < M + N; k++)
    {
        if (i == N) { merge[k] = b[j++]; continue; }
        if (j == M) { merge[k] = a[i++]; continue; }
        if (isLess(b[j], a[i]))
            { merge[k] = b[j++]; }
        else
            { merge[k] = a[i++]; }
    }
}
“In-Place” Merge

- For sorting, we would like to:
  - Starting with sorted halves
    - \(a[\text{start} \ldots \ m]\)
    - \(a[m + 1 \ldots \ \text{end}]\)
  - 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 two sub-arrays
- Replaces contents with merge results
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 <= stop; i++)
        aux[i] = a[i];

    for (i = k = start, j = m + 1; k <= stop; k++)
        if (j <= stop && isLess(aux[j], aux[i]) || i > m)
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
}
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

```c
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[m + 1 + stop - j] = a[j];

  for (i = k = start, j = stop; 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; * using insertion for small arrays
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 mergesort(Item* a, int start, int stop)
{
    int i, m;

    for (m = 1; m < stop - start; m += m)
        for (i = start; i < stop; i += m + m)
        {
            int from = i, mid = i + m - 1;
            int to = min(i + m + m - 1, stop);
            merge(a, from, mid, to);
        }
}
```
Merging Pattern for N = 21
## Sedgewick’s Timings (secs)

<table>
<thead>
<tr>
<th>N</th>
<th>QuickSort</th>
<th>Top-Down MergeSort</th>
<th>Bottom-Up MergeSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>24</td>
<td>53</td>
<td>59</td>
</tr>
<tr>
<td>200,000</td>
<td>52</td>
<td>111</td>
<td>127</td>
</tr>
<tr>
<td>400,000</td>
<td>109</td>
<td>237</td>
<td>267</td>
</tr>
<tr>
<td>800,000</td>
<td>241</td>
<td>524</td>
<td>568</td>
</tr>
</tbody>
</table>

Array of floating point numbers
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