Review: Improvements to Insertion Sort

- Stop comparisons early
- Carry out half-exchanges
- Place smallest element in first position
  - Approach is called using a “Sentinel” element
Sentinels in Insertion Sort

- Notice that inner loop continues until:
  - First element reached, or
  - Smaller element reached

- In each iteration, we must check both conditions

- Unless, smallest element is known to be the first element …
void sort(Item *a, int start, int stop) 
{
    int i, j;

    for (i = start + 1; i <= stop; i++)
    {
        int j = i;
        while (isLess(a[j], a[j-1]) && (j > start))
        {
            Exchange(a[j], a[j-1]);
            j--;
        }
    }
}
Insertion Sort – With Sentinel

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;
        while (isLess(a[j], a[j-1]))
        {
            Exchange(a[j], a[j-1]);
            j--;
        }
    }
}
Last Lecture: Shell Sort

- Gradually bring order to array by:
  - Using a series of step sizes $k$, ending with $k = 1$
  - Sort sub-arrays including every $k^{th}$ element

- Each pass handles nearly sorted arrays where insertion sort is efficient

- Theoretically, $N (\log N)^2$ complexity
Array gradually gains order

Eventually, we approach the ideal case where insertion sort is $O(N)$
Today: Quick Sort

- Most widely used sorting algorithm
  - Possibly excluding those bubble sorts that should be banished!

- Extremely efficient
  - $O(N \log N)$

- Divide-and-conquer algorithm
The Inventor of Quicksort

- Sir Charles A. R. Hoare
  - 1980 ACM Turing Award

- British computer scientist
  - Studied statistics as a graduate student

- Made major contributions to developing computer languages
"I conclude that there are two ways of constructing a software design:

One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."
Caution!

- Quicksort is fragile!
  - Small mistakes can be hard to spot
  - Shellsort is more robust (but slower)

- The worst case running time is $O(N^2)$
  - Can be avoided in nearly all practical settings
Divide-And-Conquer

- Divide a problem into smaller sub-problems

- Find a partitioning element such that:
  - All elements to the right are greater
  - All elements to the left are smaller
  - Sort right and left sub-arrays independently
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);
}
```
Quicksort Notes

- Each round places one element into position
  - The partitioning element

- Recursive calls handle each half of array

- What would be:
  - a good partitioning element?
  - a bad partitioning element?
Partitioning

- Choose an arbitrary element
  - Any suggestions?

- Place all smaller values to the left

- Place all larger values to the right
Partitioning

- Smaller elements
- Larger elements
- Partitioning element

Direction:
- Up
- Down
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;
}
```
Partitioning Notes

- The check (up < down) required when partitioning element is also smallest element.

- N comparisons
  - N - 1 for each element vs. partitioning element
  - One extra is possible when pointers cross
Quick Sort

Array is successively subdivided, around partitioning element.

Within each section, items are arranged randomly
Complexity of Quick Sort

- **Best case:**
  \[ C_N = 2C_{N/2} + N = N \log_2 N \]

- **Random input:**
  \[ C_N = N + \frac{1}{N} \sum_{k=1}^{N} C_{k-1} + C_{N-k} \]
  \[ \approx 2N \log_e N \approx 1.4N \log_2 N \]
Complexity of Quick Sort

![Graph showing the complexity of Quick Sort with lines for average case and best case. The x-axis represents the number of elements in thousands, ranging from 0 to 5. The y-axis represents the number of comparisons in thousands, ranging from 0 to 90. The graph demonstrates the linear relationship between the number of elements and the number of comparisons for both average and best cases.]

- **Average Case**
- **Best Case**

The graph illustrates how the number of comparisons increases as the number of elements grows, with a steeper increase in the average case compared to the best case.
Improvements to Quicksort

- Sorting small sub-arrays
  - Quicksort is great for large arrays
  - Inefficient for very small ones

- Choosing a better partitioning element
  - A poor choice could make sort quadratic!
Small Sub-arrays

- Most recursive calls are for small sub-arrays
  - Commonplace for many recursive programs

- “Brute-force” algorithms are often better for small problems
  - In this case, insertion sort is a good option
Sorting Small Sub-arrays

- Possibility 1:
  - `if (stop - start <= M)
    insertion(a, start, stop);`

- Possibility 2:
  - `if (stop - start <= M) return;`
  - Make a single call to `insertion()` at the end

- $5 < M < 25$ gives $\sim 10\%$ speed increase
Arrays including first $N$ words in text of “Moby Dick”.

<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>
Improved Partitioning

How do we avoid picking smallest or largest element for partitioning?

- Could take a random element…
- Could take a sample …
Median-of-Three Partitioning

- Take sample of three elements
- Usually, first, last and middle element
  - Sort these three elements
- Partition around median
  - Much more unlikely for worst case to occur
void quicksort(Item * a, int start, int stop) {
    int i;

    // Leave small subsets for insertion sort
    if (stop - start <= M) return;

    // Place median of 3 in position stop - 1
    Exchange(a[(start + stop)/2], a[stop - 1]);
    CompExch(a[start], a[stop - 1]);
    CompExch(a[start], a[stop]);
    CompExch(a[stop - 1], a[stop]);

    // The first and the last elements are “prepartitioned”
    i = partition(a, start + 1, stop - 1);
    quicksort(a, start, i - 1);
    quicksort(a, i + 1, stop);
}
So far...

- Basic Quick Sort
- Median of Three Partitioning
- Brute Force Sorts for Small Problems
- Combined, median-of-three partitioning and insertion sorts for smaller sub-arrays improve performance about 20%
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

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

- After partitioning, handle smaller half first
  - At most, $\log_2 N$ smaller halves!
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 - 1;
        }
    }
}
```
Explicit Stacks

- A common feature in computer programs
  - Similar to a “TO DO” list or an “INBOX”

- 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
A Related 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

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

    if (stop <= start) 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

// 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;

    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
    - ...

- Common application: finding $k$ smallest values in a simulation to save for further analyses
Recommended Reading

- Sedgewick, Chapter 7