Quick Sort
Previously: Elementary Sorts

- Selection Sort
- Insertion Sort
- Bubble Sort

Try to state an advantage for each one...
Last Lecture: Shell Sort

- Gradually bring order to array by:
  - Sort sub-arrays including every $k^{th}$ element
  - For a sequence where the last $k = 1$

- Produces a series of nearly sorted arrays where insertion sort is very efficient
  - Theoretically, $N \cdot (\log N)^2$ complexity
Pictorial Representation

- Array gradually gains order
- Eventually, we approach the ideal case where insertion sort is $O(N)$
Shell Sort Properties

- $h$-sorting a $k$-sorted array, produces an array that is $h$- and $k$- sorted

- If $h$ and $k$ are relatively prime, $a[i]$ must be smaller than any $a[i+j]$ where $j > (h - 1)(k - 1)$
  - No more than $(h-1)(k-1)/g$ comparisons to $g$-sort array that is $h$ and $k$ sorted
Property I

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

- Consider 4 elements:
  - $a[i] \leq a[i+k]$
  - $a[i+h] \leq a[i+k+h]$

- After sorting, $a[i]$ contains minimum and $a[i+k+h]$ contains maximum of 4 elements
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).
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

- How many increments must be used for such a sequence?
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
```
Today: Quick Sort

- Most widely used sorting algorithm
  - Possibly, except for 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
C. A. R. Hoare Quote

“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 most cases
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

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 (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;
}
```
Partitioning Notes

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

- N + 1 comparisons
  - N - 1 for each element vs. partitioning element
  - Extra for 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:

\[ C_N = N + 1 + \frac{1}{N} \sum_{k=1}^{N} C_{k-1} + C_{N-k} \]

\[ \approx 2N \log_e N \approx 1.4N \log_2 N \]
Complexity of Quick Sort

The graph illustrates the relationship between the number of elements and the number of comparisons required for Quick Sort. The x-axis represents the number of elements in thousands, while the y-axis represents the number of comparisons in thousands. The graph shows three lines:

- **Cost** (blue line): This line represents the actual cost of Quick Sort.
- **Cost2** (pink line): This line represents an alternative cost model.
- **Optimal** (yellow line): This line represents the optimal number of comparisons.

As the number of elements increases, the number of comparisons also increases, and the lines diverge, indicating the different costs and efficiencies of the sorting methods.
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 often better for small problems
  - In this case, insertion sort is a good option
Sorting Small Sub-arrays

- **Possibility 1:**
  - if \((\text{stop} - \text{start} \leq M)\)
    
    \[
    \text{insertion(a, start, stop);}
    \]

- **Possibility 2:**
  - \(\textbf{if} \ (\text{stop} - \text{start} \leq M) \ \textbf{return;}\)
  - Make a single call to \text{insertion()} at the end

- \(5 < M < 25\) gives \(\approx 10\%\) speed increase
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);
}
Summary

- Together, median-of-three partitioning and using insertion sort for small sub-files improve performance about 20%

- Problem for next lecture:
  - Avoiding very deep recursions…
Selection

- A problem related to sorting …

- Find the $k$ smallest element in an array
  - Minimum
  - Maximum
  - Median
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 element k in position
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, k);
    if (i < k) select(a, i + 1, stop, k);
}
C Code: Without Recursion

// Places element k in position within array
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