Eric Roberts
CS 106B

Handout #21
April 20, 2009

Sorting and Efficiency

Sorting

- Of all the algorithmic problems that computer scientists have studied, the one with the broadest practical impact is certainly the *sorting problem*, which is the problem of arranging the elements of an array or a vector in order.
- The sorting problem comes up, for example, in alphabetizing a telephone directory, arranging library records by catalogue number, and organizing a bulk mailing by ZIP code.
- There are many algorithms that one can use to sort an array. Because these algorithms vary enormously in their efficiency, it is critical to choose a good algorithm, particularly if the application needs to work with large arrays.

The Selection Sort Algorithm

- Of the many sorting algorithms, the easiest one to describe is **selection sort**, which appears in the text like this:

```
void Sort(Vector<int> vec) {
   int n = vec.size();
   for (int lh = 0; lh < n; lh++) {
      int rh = lh;
      for (int i = lh + 1; i < n; i++) {
         if (vec[i] < vec[rh]) rh = i;
      }
      int temp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = temp;
   }
}
```

- Coding this algorithm as a single function makes sense for efficiency but complicates the analysis. The next two slides decompose selection sort into a set of functions that make the operation easier to follow.

Decomposition of the `Sort` Function

```
/*
 * Function: Sort
 * --------------
 * Sorts a Vector<int> into increasing order.  This implementation
 * uses an algorithm called selection sort, which can be described
 * in English as follows:  With your left hand (lh), point at each
 * element in the vector in turn, starting at index 0.  At each
 * step in the cycle:
 *
 * 1. Find the smallest element in the range between your left
 *    hand and the end of the vector, and point at that element
 *    with your right hand (rh).
 *
 * 2. Move that element into the correct position by swapping
 *    the elements indicated by your left and right hands.
 */

void Sort(Vector<int> & vec) {
   for (int lh = 0; lh < vec.size(); lh++) {
      int rh = FindSmallest(vec, lh, vec.size() - 1);
      Swap(vec[lh], vec[rh]);
   }
}
```

Decomposition of the `Sort` Function

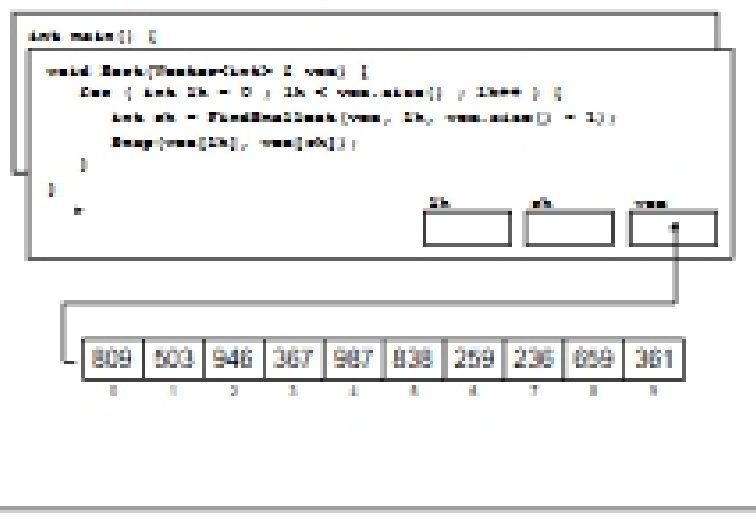
```
/*
 * Function: FindSmallest
 * ----------------------
 * Returns the index of the smallest value in the range between
 * index positions p1 and p2, inclusive.
 */

int FindSmallest(Vector<int> & vec, int p1, int p2) {
   int smallestSoFar = p1;
   for (int i = p1 + 1; i <= p2; i++) {
      if (vec[i] < vec[smallestSoFar]) smallestSoFar = i;
   }
   return smallestSoFar;
}

/*
 * Function: Swap
 * --------------
 * Exchanges the integer values passed by reference.
 */

void Swap(int & x, int & y) {
   int temp = x;
   x = y;
   y = temp;
}
```

Simulating Selection Sort



Efficiency of Selection Sort

- The primary question for today is how one might evaluate the efficiency of an algorithm such as selection sort.
- One strategy is to measure the actual time it takes to run for arrays of different sizes. In C++, you can measure elapsed time by calling the `time` function, which returns the current time in milliseconds. Using this strategy, however, requires some care:
  - The `time` function is often too rough for accurate measurement. It therefore makes sense to measure several runs together and then divide the total time by the number of repetitions.
  - Most algorithms show some variability depending on the data. To avoid distortion, you should run several independent trials with different data and average the results.
  - Some measurements are likely to be wildly off because the computer needs to run some background task. Such data points must be discarded as you work through the analysis.

Measuring Sort Timings

The following table shows the average timing of the selection sort algorithm after removing outlying trials that differ by more than two standard deviations from the mean. The column labeled μ (the Greek letter mu, which is the standard statistical symbol for the mean) is a reasonably good estimate of running time.

N	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	μ	σ
10	.0021	.0025	.0022	.0026	.0020	.0020	.0022	.0023	.0022	.0025	.0024	.000239
20	.006	.007	.006	.007	.007		.007	.007	.007	.007	.007	.000436
40	.014	.014	.014	.015	.014	.014	.016	.016	.014	.014	.014	.000813
80	.026	.024	.025	.026	.023	.025	.025	.026	.025	.027	.025	.0014
160	.039	.037	.036	.041	.042	.039		.038	.034	.038	.039	.0025
320	.167	.152	.160	.170	.146	.146	.165	.148	.170	.154	.162	.0151
640	3.91	3.63	4.02	3.76	4.11	3.51	3.48	3.64	3.31	3.42	3.69	0.252
1000	13.40	12.90	13.80	17.60	12.90	14.10	12.70		16.00	15.50	14.32	1.89
2000	322.5	292.6	291.7	291.2	288.3	321.3	291.3	290.7	322.1	321.3	346.4	20.83
10000	1318.		1327.	1318.	1331.	1290.	1318.	1335.	1325.	1319	1326.	7.50

Selection Sort Running Times

- Many algorithms that operate on vectors have running times that are proportional to the size of the array. If you multiply the number of values by ten, you would expect those algorithms to take ten times as long.
- As the running times on the preceding slide make clear, the situation for selection sort is very different. The table on the right shows the average running time when selection sort is applied to 10, 100, 1000, and 10000 values.

N	time
10	.002m
100	0.162
1000	14.32
10000	1322.

- As a rough approximation—particularly as you work with larger values of N—it appears that every ten-fold increase in the size of the array means that selection sort takes about 100 times as long.

Counting Operations

- Another way to estimate the running time is to count how many operations are required to sort an array of size *N*.
- In the selection sort implementation, the section of code that is executed most frequently (and therefore contributes the most to the running time) is the body of the **`FindSmallest`** method. The number of operations involved in each call to **`FindSmallest`** changes as the algorithm proceeds:

  N values are considered on the first call to **`FindSmallest`**.
  N – 1 values are considered on the second call.
  N – 2 values are considered on the third call, and so on.

- In mathematical notation, the number of values considered in **`FindSmallest`** can be expressed as a summation, which can then be transformed into a simple formula:

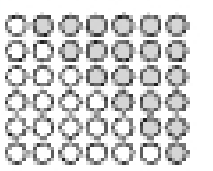
$$1 + 2 + 3 + \cdots + (N-1) + N = \sum_{i=1}^{N} i = \frac{N \cdot (N+1)}{2}$$

A Geometric Insight

- You can convince yourself that

$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \cdot (N+1)}{2}$$

  by thinking about the problem geometrically.
- The terms on the left side of the formula can be arranged into a triangle, as shown at the bottom of this slide for $N = 6$.
- If you duplicate the triangle and rotate it by 180°, you get a rectangle that in this case contains 6×7 dots, half of which belong to each triangle.



Quadratic Growth

- The reason behind the rapid growth in the running time of selection sort becomes clear if you make a table showing the value of $\frac{N \cdot (N+1)}{2}$ for various values of *N*:

N	$\frac{N \cdot (N+1)}{2}$
10	55
100	5050
1000	500,500
10000	50,005,000

- The growth pattern in the right column is similar to that of the measured running time of the selection sort algorithm. As the value of *N* increases by a factor of 10, the value of $\frac{N \cdot (N+1)}{2}$ increases by a factor of around 100, which is 10^2. Algorithms whose running times increase in proportion to the square of the problem size are said to be **quadratic**.

Big-O Notation

- The most common way to express computational complexity is to use **big-O notation**, which was introduced by the German mathematician Paul Bachmann in 1892.
- Big-O notation consists of the letter O followed by a formula that offers a qualitative assessment of running time as a function of the problem size, traditionally denoted as *N*. For example, the computational complexity of linear search is

$$O(N)$$

  and the computational complexity of radix sort is

$$O(N \log N)$$

- If you read these formulas aloud, you would pronounce them as "big-O of *N*" and "big-O of *N* log *N*" respectively.

Common Simplifications of Big-O

- Given that big-O notation is designed to provide a qualitative assessment, it is important to make the formula inside the parentheses as simple as possible.
- When you write a big-O expression, you should always make the following simplifications:
  1. Eliminate any term whose contribution to the running time ceases to be significant as N becomes large.
  2. Eliminate any constant factors.
- The computational complexity of selection sort is therefore

$$O(N^2)$$

and not

$$O\left(\frac{N \times (N+1)}{2}\right)$$

Deducing Complexity from the Code

- In many cases, you can deduce the computational complexity of a program directly from the structure of the code.
- The standard approach to doing this type of analysis begins with looking for any section of code that is executed more often than other parts of the program. As long as the individual operations involved in an algorithm take roughly the same amount of time, the operations that are executed most often will come to dominate the overall running time.
- In the selection sort implementation, for example, the most commonly executed statement is the `if` statement inside the `FindSmallest` method. This statement is part of two `for` loops, one in `FindSmallest` itself and one in `Sort`. The total number of executions is

$$1 + 2 + 3 + \cdots + (N-1) + N$$

which is $O(N^2)$.

Exercise: Computational Complexity

Assuming that none of the steps in the body of the following `for` loops depend on the problem size stored in the variable `n`, what is the computational complexity of each of the following examples:

a)
```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < i; j++) {
      ...loop body...
   }
}
```

b)
```
for (int i = 0; i <= n; i *= 2) {
   ...loop body...
}
```

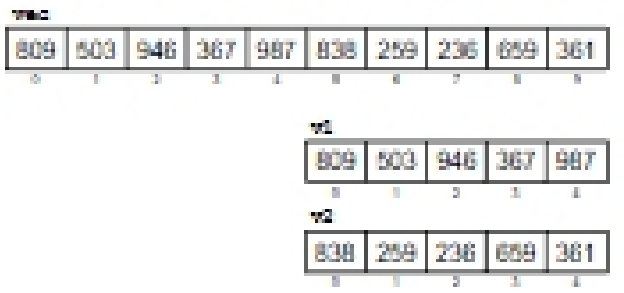
c)
```
for (int i = 0; i < 100; i++) {
   for (int j = 0; j < i; j++) {
      ...loop body...
   }
}
```

Finding a More Efficient Strategy

- As long as arrays are small, selection sort is a perfectly workable strategy. Even for 10,000 elements, the average running time of selection sort is just over a second.
- The quadratic behavior of selection sort, however, makes it less attractive for the very large arrays that one encounters in commercial applications. Assuming that the quadratic growth pattern continues beyond the timings reported in the table, sorting 100,000 values would require two minutes, and sorting 1,000,000 values would take more than three hours.
- The computational complexity of the selection sort algorithm, however, holds out some hope:
  - Sorting twice as many elements takes four times as long.
  - Sorting half as many elements takes only one fourth the time.
  - Is there any way to use sorting half an array as a subtask in a recursive solution to the sorting problem?

The Merge Sort Idea

1. Divide the vector into two halves: `v1` and `v2`.
2. Sort each of `v1` and `v2` recursively.
3. Clear the original vector.
4. Merge elements into the original vector by choosing the smallest element from `v1` or `v2` on each cycle.



The Merge Sort Implementation

```
/*
 * The merge sort algorithm consists of the following steps:
 *
 * 1. Divide the vector into two halves.
 * 2. Sort each of those smaller vectors recursively.
 * 3. Merge the two vectors back into the original one.
 */

void sort(vector<int> & vec) {
   int n = vec.size();
   if (n == 0) return;
   vector<int> v1;
   vector<int> v2;
   for (int i = 0; i < n; i++) {
      if (i < n / 2) {
         v1.add(vec[i]);
      } else {
         v2.add(vec[i]);
      }
   }
   sort(v1);
   sort(v2);
   vec.clear();
   merge(vec, v1, v2);
}
```