Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University
2010
Chapter 8. Algorithmic Analysis
Outline

1. Introduction
2. Selection Sort Algorithm
3. Merge Sort Algorithm
4. Big-O Notation
5. Quick Sort Algorithm
Outline

1. Introduction
2. Selection Sort Algorithm
3. Merge Sort Algorithm
4. Big-O Notation
5. Quick Sort Algorithm
Introduction

Analyze the efficiency of algorithms.

- What does the term **efficiency** mean in an algorithmic context?
- What is the measurement for efficiency?

Study the efficiency of some sorting algorithms.

**Sorting**: Rearrange the elements (integers) of an array so that they fall in ascending order.
Outline

1. Introduction
2. Selection Sort Algorithm
3. Merge Sort Algorithm
4. Big-O Notation
5. Quick Sort Algorithm
Selection sort: Idea

vec[0] to vec[lh-1] already sorted
vec[rh] is the smallest among vec[lh] to vec[i]
when i reaches the end (n-1), vec[rh] is the smallest among vec[lh] to vec[n-1], swap vec[rh] and vec[lh]
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;
        }
        if (rh > lh) {
            int temp = vec[lh];
            vec[lh] = vec[rh];
            vec[rh] = temp;
        }
    }
}
Running time as a measurement for efficiency. 
*N*: vector size or number of elements to be sorted.

Experimental results:

<table>
<thead>
<tr>
<th>(N)</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1.46 msec</td>
</tr>
<tr>
<td>400</td>
<td>135.42 msec</td>
</tr>
<tr>
<td>4,000</td>
<td>13.42 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>83.90 sec</td>
</tr>
</tbody>
</table>
Running time as a measurement for efficiency. 

$N$: vector size or number of elements to be sorted.

Experimental results:

<table>
<thead>
<tr>
<th>$N$</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1.46 msec</td>
</tr>
<tr>
<td>400</td>
<td>135.42 msec</td>
</tr>
<tr>
<td>4,000</td>
<td>13.42 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>83.90 sec</td>
</tr>
</tbody>
</table>

Can you see the growth pattern?
Running time

Running time as a measurement for efficiency. 
$N$: vector size or number of elements to be sorted.

Experimental results:

<table>
<thead>
<tr>
<th>$N$</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1.46 msec</td>
</tr>
<tr>
<td>400</td>
<td>135.42 msec</td>
</tr>
<tr>
<td>4,000</td>
<td>13.42 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>83.90 sec</td>
</tr>
</tbody>
</table>

Can you see the growth pattern?

Problem: Implementation and machine dependent.
Analyzing the performance

The operation in the inner most loop is

\[ \text{vec}[i] < \text{vec}[rh] \quad rh = i; \]

the comparison.

Use the number of comparisons as an efficiency measurement.
**Analyzing the performance**

The operation in the inner most loop is

\[ \text{vec}[i] < \text{vec}[rh] \quad \text{rh} = i; \]

the comparison.

Use the number of comparisons as an efficiency measurement.

Why?

The operations in the inner most loop are executed most frequently, meaning that they are the major contribution to the total computational cost.
Counting the number of comparisons

<table>
<thead>
<tr>
<th>Value of $lh$</th>
<th>Number of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>1</td>
<td>$n - 2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n - 2$</td>
<td>1</td>
</tr>
</tbody>
</table>

The total number of comparisons:

$$(n - 1) + (n - 2) + ... + 1 = \frac{n(n - 1)}{2} = \frac{n^2 - n}{2}$$
Computational complexity

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\frac{N^2 - N}{2}$</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>780</td>
<td>1.46 msec</td>
</tr>
<tr>
<td>400</td>
<td>79,800</td>
<td>135.42 msec</td>
</tr>
<tr>
<td>4,000</td>
<td>7,998,000</td>
<td>13.42 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>49,995,000</td>
<td>83.90 sec</td>
</tr>
</tbody>
</table>

About the same growth rate.
### Computational complexity

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\frac{N^2 - N}{2}$</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>780</td>
<td>1.46 msec</td>
</tr>
<tr>
<td>400</td>
<td>79,800</td>
<td>135.42 msec</td>
</tr>
<tr>
<td>4,000</td>
<td>7,998,000</td>
<td>13.42 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>49,995,000</td>
<td>83.90 sec</td>
</tr>
</tbody>
</table>

About the same growth rate.

Problem size $N$: vector size or the number of elements to be sorted.

Computational complexity $\frac{N^2 - N}{2}$

A function of the problem size, independent of implementation and machine.
Outline

1. Introduction
2. Selection Sort Algorithm
3. Merge Sort Algorithm
4. Big-O Notation
5. Quick Sort Algorithm
void Sort(Vector<int> & vec) {
    int n = vec.size();
    if (n <= 1) return;
    split vec into v1 and v2;
    Sort(v1);
    Sort(v2);
    vec.clear();
    Merge(vec, v1, v2);
}
Merging v1 and v2 into vec

Assume v1 and v2 are sorted.

v1[0:p1-1] and v2[0:p2-1] already merged to vec

add smaller of v1[p1] and v2[p2] to the end of vec, then increment p1 or p2

when v1 (or v2) has been merged to vec, add the remaining v2[p2:end] (or v1[p1:end]) to the end of vec
void Merge(Vector<int> vec, Vector<int> v1, Vector<int> v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    int p1 = 0;
    int p2 = 0;
    while (p1 < n1 && p2 < n2) {
        if (v1[p1] < v2[p2]) {
            vec.add(v1[p1++]);
        } else {
            vec.add(v2[p2++]);
        }
    }
    add remaining v1 or v2 to vec;
}
**Complexity**

\[ \log_2 N \text{ recursive levels. At each level, } N \text{ elements are sorted in places.} \]

**Complexity:** \( N \log_2 N \)
Outline

1. Introduction
2. Selection Sort Algorithm
3. Merge Sort Algorithm
4. Big-O Notation
5. Quick Sort Algorithm
Big-O notation

A simple **qualitative** approximation of the computational complexity of an algorithm.
Big-O notation

A simple qualitative approximation of the computational complexity of an algorithm.

We are more interested in the performance of large size problems than small size problems.

For example, in the selection sort experiments, the difference in running time between $N = 40$ and $N = 400$ is only a fraction of a second. Whereas the difference between 400 and 4,000 is more than a dozen seconds and the difference between 4,000 and 10,000 is more than a minute.
To simplify the notation, we eliminate any term whose contribution to the total ceases to be significant as $N$ becomes large.

Example. The complexity of the selection sort algorithm is $\frac{N^2 - N}{2}$. We know $\lim_{N \to \infty} \frac{N^2 - N}{2} = \frac{N^2}{2}$. That means the contribution of the term $\frac{N}{2}$ to the total ceases to be significant as $N$ grows large. So, we first eliminate the term $\frac{N}{2}$.

The complexity is first simplified to $\frac{N^2}{2}$.
To further simplify the notation, we eliminate and constant factors. Thus $\frac{N^2}{2}$ is simplified to $N^2$. 
To further simplify the notation, we eliminate and constant factors. Thus \( \frac{N^2}{2} \) is simplified to \( N^2 \).

| big-O notation | selection sort \( O(N^2) \) | merge sort \( O(N \log_2 N) \) |
To further simplify the notation, we eliminate and constant factors. Thus $\frac{N^2}{2}$ is simplified to $N^2$.

<table>
<thead>
<tr>
<th>big-O notation</th>
<th>selection sort $O(N^2)$</th>
<th>merge sort $O(N \log_2 N)$</th>
</tr>
</thead>
</table>

Difference between $O(N^2)$ and $O(N \log_2 N)$

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N^2$</th>
<th>$N \log_2 N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10,000</td>
<td>664</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000,000</td>
<td>9965</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000,000</td>
<td>132,877</td>
</tr>
</tbody>
</table>
### Standard complexity classes

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Big-O Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
<td>Find the first element in an array</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$O(\log N)$</td>
<td>Binary search in an sorted array</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(N)$</td>
<td>Compute the average of an array</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>$O(N \log N)$</td>
<td>Merge sort</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$O(N^2)$</td>
<td>Selection sort</td>
</tr>
<tr>
<td>Cubic</td>
<td>$O(N^3)$</td>
<td>Conventional matrix multiplication</td>
</tr>
<tr>
<td>Exponential</td>
<td>$O(2^N)$</td>
<td>Tower of Hanoi</td>
</tr>
</tbody>
</table>
Standard complexity classes

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Example</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>Find the first element in an array</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log N)$</td>
<td>Binary search in an sorted</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>Compute the average of an array</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>$O(N \log N)$</td>
<td>Merge sort</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>Selection sort</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(N^3)$</td>
<td>Conventional matrix multiplication</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>Tower of Hanoi</td>
</tr>
</tbody>
</table>

$O(N^k)$: polynomial algorithms, tractable

$O(2^N)$: exponential algorithms, intractable
Outline

1. Introduction
2. Selection Sort Algorithm
3. Merge Sort Algorithm
4. Big-O Notation
5. Quick Sort Algorithm
Quick sort algorithm (recursive)

C.A.R. Hoare

The most used algorithm in sorting programs.

Idea:

1. Choose an element, usually the first, as the pivot, the boundary between small and large.
2. Rearrange the elements so that large elements are moved toward the end and small elements toward the beginning.
3. Sort the elements in each part of the vector

small: those are smaller than the pivot
large: those are larger than or equal to the pivot
### Partitioning the vector

1. **Initial lh and rh**
2. Move rh until rh == lh or vec[rh] is small
3. Move lh until lh == rh or vec[lh] is large
4. If rh != lh, swap vec[lh] and vec[rh]
5. Repeat 2-4 until rh == lh
6. Swap vec[lh] (= vec[rh]) and pivot (= vec[start])

```
<table>
<thead>
<tr>
<th></th>
<th>56</th>
<th>25</th>
<th>37</th>
<th>58</th>
<th>95</th>
<th>19</th>
<th>73</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

lh and rh are initialized and moved according to the rules.
Partitioning the vector (cont.)

1. Initial lh and rh
2. Move rh until rh == lh or vec[rh] is small
3. Move lh until lh == rh or vec[lh] is large
4. If rh != lh, swap vec[lh] and vec[rh]
5. Repeat 2-4 until rh == lh
6. Swap vec[lh] (== vec[rh]) and pivot (== vec[start])

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>25</td>
<td>37</td>
<td>58</td>
<td>95</td>
<td>19</td>
<td>73</td>
<td>30</td>
</tr>
</tbody>
</table>

lh

rh
Partitioning the vector (cont.)

1. Initial $lh$ and $rh$
2. Move $rh$ until $rh = lh$ or $vec[rh]$ is small
3. Move $lh$ until $lh = rh$ or $vec[lh]$ is large
4. If $rh \neq lh$, swap $vec[lh]$ and $vec[rh]$
5. Repeat 2-4 until $rh = lh$
6. Swap $vec[lh]$ ($= vec[rh]$) and pivot ($= vec[start]$)
Partitioning the vector (cont.)

1. Initial \(lh\) and \(rh\)
2. Move \(rh\) until \(rh == lh\) or \(vec[rh]\) is small
3. Move \(lh\) until \(lh == rh\) or \(vec[lh]\) is large
4. If \(rh != lh\), swap \(vec[lh]\) and \(vec[rh]\)
5. Repeat 2-4 until \(rh == lh\)
6. Swap \(vec[lh]\) (= \(vec[rh]\)) and pivot (= \(vec[start]\))
Partitioning the vector (cont.)

1. Initial \(lh\) and \(rh\)
2. Move \(rh\) until \(rh = lh\) or \(vec[rh]\) is small
3. Move \(lh\) until \(lh = rh\) or \(vec[lh]\) is large
4. If \(rh \neq lh\), swap \(vec[lh]\) and \(vec[rh]\)
5. Repeat 2-4 until \(rh = lh\)
6. Swap \(vec[lh]\) (\(= vec[rh]\)) and pivot (\(= vec[start]\))

<table>
<thead>
<tr>
<th></th>
<th>19</th>
<th>25</th>
<th>37</th>
<th>30</th>
<th>56</th>
<th>95</th>
<th>73</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

boundary
A simple implementation, Figure 8-6, pp. 299-300.

Experimental results

<table>
<thead>
<tr>
<th>$N$</th>
<th>Merge sort</th>
<th>Quick sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>2.54 msec</td>
<td>0.52 msec</td>
</tr>
<tr>
<td>400</td>
<td>31.25 msec</td>
<td>8.85 msec</td>
</tr>
<tr>
<td>4000</td>
<td>383.33 msec</td>
<td>129.17 msec</td>
</tr>
<tr>
<td>10000</td>
<td>997.67 msec</td>
<td>341.67 msec</td>
</tr>
</tbody>
</table>
Performance

Question

Why in the worst case when the array is already sorted the complexity of this quick sort algorithm is quadratic?
Performance

Question

Why in the worst case when the array is already sorted the complexity of this quick sort algorithm is quadratic?

Choosing the pivot.
Average-case performance vs worst-case performance.