Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University
2010

Chapter 8. Algorithmic Analysis

Outline

Outline

Introduction

Analyze the efficiency of algorithms.

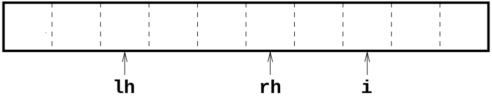
- What does the term efficiency mean in an algorithmic context?
- What is the measurement for efficiency?

Study the efficiency of some sorting algorithms.

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

Outline

Selection sort: Idea



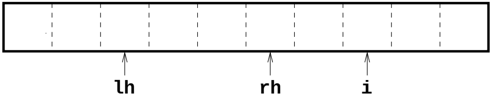
vec[0] to vec[lh-1] already sorted

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]

Selection sort algorithm

```
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

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

Experimental results:

N	Running Time
40	1.46 msec
400	135.42 msec
4,000	13.42 sec
10,000	83.90 sec

Running time

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

Experimental results:

N	Running Time
40	1.46 msec
400	135.42 msec
4,000	13.42 sec
10,000	83.90 sec

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:

N	Running Time
40	1.46 msec
400	135.42 msec
4,000	13.42 sec
10,000	83.90 sec

Can you see the growth pattern?

Problem: Implementation and machine dependent.

Analyzing the performance

The operation in the inner most loop is

```
vec[i] < vec[rh] rh = i;
```

the comparison.

Use the number of comparisons as an efficiency measurement.

Analyzing the performance

The operation in the inner most loop is

```
vec[i] < vec[rh] 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

Value of `lh`	Number of comparisons
0	$n-1$
1	$n-2$
$\vdots$	$\vdots$
$n-2$	1

The total number of comparisons:

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}=\frac{n^2-n}{2}$$

Computational complexity

N	$\frac{N^2-N}{2}$	Running Time
40	780	1.46 msec
400	79,800	135.42 msec
4,000	7,998,000	13.42 sec
10,000	49,995,000	83.90 sec

About the same growth rate.

Computational complexity

N	$\frac{N^2-N}{2}$	Running Time
40	780	1.46 msec
400	79,800	135.42 msec
4,000	7,998,000	13.42 sec
10,000	49,995,000	83.90 sec

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

Merge sort algorithm (pseudo codes)

```
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

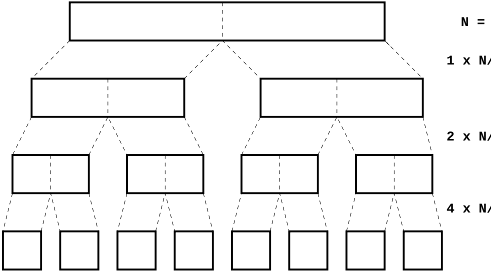
add smaller of v1[p1] and v2[p2] to the end of vec, then increment p1 or p2

when v1 (or v2) has been merged to vec, add the remaining v2[p2:end] (or v1[p1:end]) to the end of vec

Merge short algorithm (pseudo codes)

```
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



$\log_2 N$ recursive levels. At each level, N elements are sorted in places.

Complexity: $N \log_2 N$

Outline

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 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.

Big-O notation (cont.)

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 $\frac{N^2}{2}$

Big-O notation (cont.)

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

Big-O notation (cont.)

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

	selection sort	merge sort
big-O notation	$O(N^2)$	$O(N \log_2 N)$

Big-O notation (cont.)

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

	selection sort	merge sort
big-O notation	$O(N^2)$	$O(N \log_2 N)$

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

N	N^2	$N \log_2 N$
100	10,000	664
1,000	1,000,000	9965
10,000	100,000,000	132,877

Standard complexity classes

constant	$O(1)$	Find the first element in an array
logarithmic	$O(\log N)$	Binary search in an sorted
linear	$O(N)$	Compute the average of an array
$N \log N$	$O(N \log N)$	Merge sort
quadratic	$O(N^2)$	Selection sort
cubic	$O(N^3)$	Conventional matrix multiplication
exponential	$O(2^N)$	Tower of Hanoi

Standard complexity classes

constant	$O(1)$	Find the first element in an array
logarithmic	$O(\log N)$	Binary search in an sorted
linear	$O(N)$	Compute the average of an array
$N \log N$	$O(N \log N)$	Merge sort
quadratic	$O(N^2)$	Selection sort
cubic	$O(N^3)$	Conventional matrix multiplication
exponential	$O(2^N)$	Tower of Hanoi

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

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

Outline

Quick sort algorithm (recursive)

C.A.R. Hoare

The most used algorithm in sorting programs.

Idea:

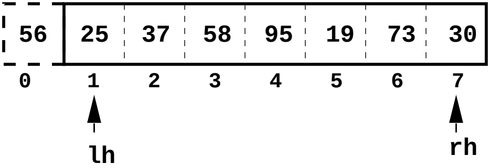
1. Choose an element, usually the first, as the pivot, the boundary between small and large.
2. Rearrange the elements so that large elements are moved toward the end and small elements toward the beginning.
3. Sort the elements in each part of the vector

small: those are smaller than the pivot

large: those are larger than or equal to the pivot

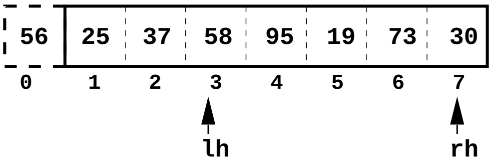
Partitioning the vector

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 util `rh == lh`
6. Swap `vec[lh]` (= `vec[rh]`) and `pivot` (= `vec[start]`)



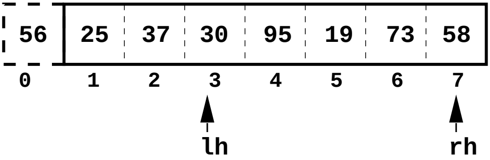
Partitioning the vector (cont.)

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 util `rh == lh`
6. Swap `vec[lh]` (= `vec[rh]`) and `pivot` (= `vec[start]`)



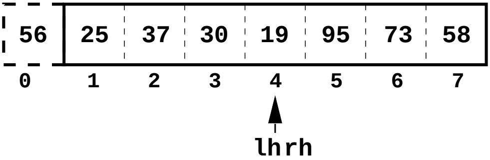
Partitioning the vector (cont.)

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 util `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 util `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 util `rh == lh`
6. Swap `vec[lh]` (= `vec[rh]`) and `pivot` (= `vec[start]`)

19	25	37	30	56	95	73	58
0	1	2	3	4	5	6	7

boundary

Performance

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

Experimental results

N	Merge sort	Quick sort
40	2.54 msec	0.52 msec
400	31.25 msec	8.85 msec
4,000	383.33 msec	129.17 msec
10,000	997.67 msec	341.67 msec

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.