Eric Roberts
CSCI 121

Handout #39
October 26, 2018

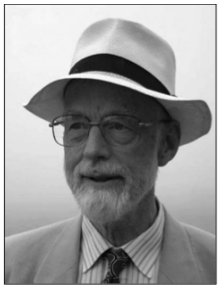
Algorithmic Efficiency

Algorithmic Analysis

Eric Roberts
CSCI 121
October 26, 2018

Tony Hoare

- Charles Antony Richard Hoare (always called Tony) is professor emeritus in computer science at Oxford.
- He is probably best known as the inventor of Quicksort, which is one of the most efficient sorting algorithms.
- In his Turing Award lecture, "The Emperor's Old Clothes," Hoare made an impassioned defense for simplicity in software design that included an explicit criticism of the complexity in the design of Ada by the Defense Department.



C. A. R. Hoare (1934–)

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 selection sort is

  $$O(N^2)$$

- If you read these formulas aloud, you would pronounce them as "big-O of *N*" and "big-O of N^2" 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 loop that searches for the smallest value in the rest of the list. This statement is part of two `for` loops. The total number of executions is

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

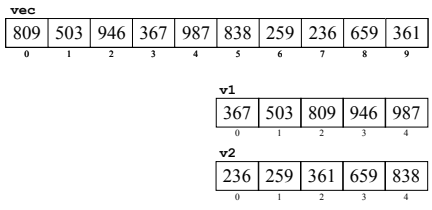
  which is $O(N^2)$.

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. Merge elements into the original vector by choosing the smallest element from `v1` or `v2` on each cycle.



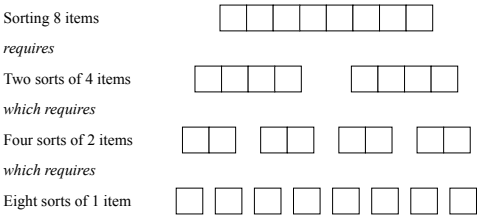
The Merge Sort Implementation

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

def sort(array):
    if len(array) > 1:
        mid = len(array) // 2
        a1 = array[:mid]
        a2 = array[mid:]
        sort(a1)
        sort(a2)
        merge(array, a1, a2)
```

The Merge Sort Implementation

```
# Merges the arrays a1 and a2, which must already be
# sorted, into array, whose length must be equal to
# the combined lengths of a1 and a2.

def merge(array, a1, a2):
    n1 = len(a1)
    n2 = len(a2)
    p1 = 0
    p2 = 0
    for i in range(len(array)):
        if p2 == n2 or (p1 < n1 and a1[p1] < a2[p2]):
            array[i] = a1[p1]
            p1 += 1
        else:
            array[i] = a2[p2]
            p2 += 1
```

The Complexity of Merge Sort



The work done at each level (*i.e.*, the sum of the work done by all the calls at that level) is proportional to the size of the vector. The running time is therefore proportional to N times the number of levels.

How Many Levels Are There?

- The number of levels in the merge sort decomposition is equal to the number of times you can divide the original vector in half until there is only one element remaining. In other words, what you need to find is the value of k that satisfies the following equation:

$$1 = N / \underbrace{2 / 2 / 2 / 2 \cdots / 2}_{k \text{ times}}$$

- As you saw in the discussion of binary search, solving for k gives the formula:

$$k = \log_2 N$$

- The complexity of merge sort is therefore $O(N \log N)$.

The Quicksort Idea

- The idea behind Tony Hoare's Quicksort algorithm is similar to that of merge sort in the sense that both depend on the recursive strategy of divide-and-conquer.
- Quicksort begins with a ***partition phase***, in which the elements of the array are reordered so that "small" elements move to the beginning and "large" elements move to the end. The terms "small" and "large" refer to the relationship between each element and a value selected to be the ***pivot***.
- After the partition phase, all that remains is to sort the two halves of the array recursively.
- On average, Quicksort has $O(N \log N)$ complexity, but the performance degrades to $O(N^2)$ if the pivot is poorly chosen.

Coding Quicksort

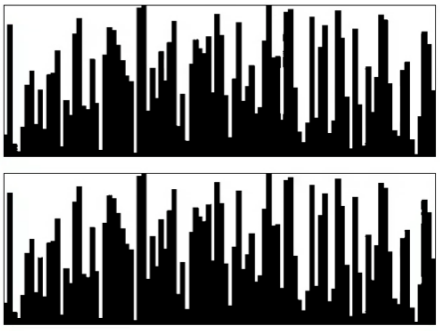
```
def sort(array):
    quicksort(array, 0, len(array))
```

```
def quicksort(array, p1, p2):
    if p2 - p1 > 1:
        boundary = partition(array, p1, p2)
        quicksort(array, p1, boundary)
        quicksort(array, boundary + 1, p2)
```

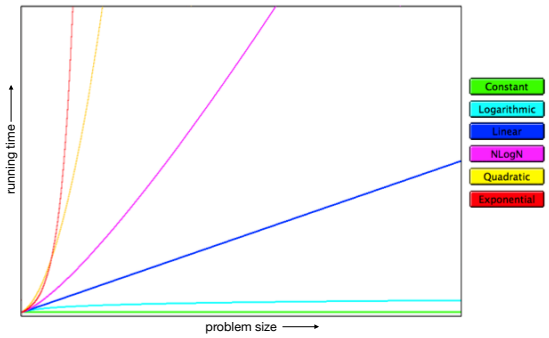
Coding Quicksort

```
def partition(array, p1, p2):
    pivot = array[p1]
    lh = p1 + 1
    rh = p2 - 1
    while True:
        while lh < rh and array[rh] >= pivot:
            rh -= 1
        while lh < rh and array[lh] < pivot:
            lh += 1
        if lh == rh: break
        array[lh],array[rh] = array[rh],array[lh]
    if array[lh] >= pivot:
        return p1
    array[p1],array[lh] = array[lh],array[p1]
    return lh
```

Selection Sort *vs.* Quicksort



Graphs of the Complexity Classes



Standard Complexity Classes

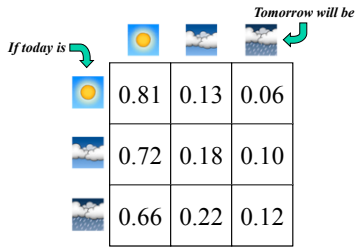
- The complexity of a particular algorithm tends to fall into one of a small number of standard complexity classes:

constant	$O(1)$	Finding first element in a vector
logarithmic	$O(\log N)$	Binary search in a sorted vector
linear	$O(N)$	Summing a vector; linear search
$N \log N$	$O(N \log N)$	Merge sort
quadratic	$O(N^2)$	Selection sort
cubic	$O(N^3)$	Obvious algorithms for matrix multiplication
exponential	$O(2^N)$	Branch and try all possibilities

- In general, theoretical computer scientists regard any problem whose complexity cannot be expressed as a polynomial as ***intractable***.

Matrix Multiplication

The following matrix shows the probability of tomorrow's weather based on the what the weather is today.



Matrix Multiplication

What, then, is the likely weather two days from now, given that you know what the weather looks like today?

The day after tomorrow will be

If today is

	0.81	0.13	0.06
	0.72	0.18	0.10
	0.66	0.22	0.12

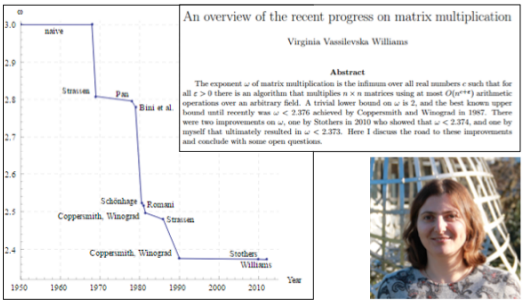
Implementing Matrix Multiplication

- The following is a straightforward implementation of matrix multiplication in Python:

```
def multiplyMatrices(a, b):
    n = len(a)
    c = createMatrix(n);
    for i in range(n):
        for j in range(n):
            for k in range(n):
                c[i][j] += a[i][k] + b[k][j]
    return c
```

- What is the big-O complexity of this algorithm?

Matrix Multiplication



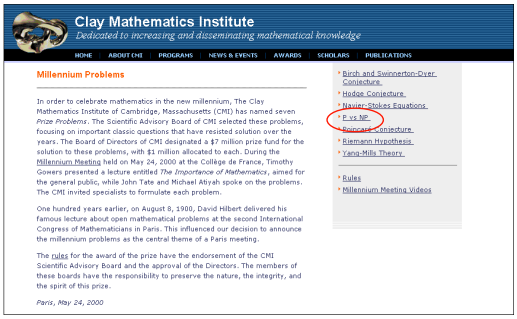
An overview of the recent progress on matrix multiplication

Virginia Vassilevska Williams

Abstract

The exponent ω of matrix multiplication is the infimum over all real numbers c such that for all $\varepsilon > 0$ there is an algorithm that multiplies $n \times n$ matrices using at most $O(n^{c+\varepsilon})$ arithmetic operations over an arbitrary field. A trivial lower bound on ω is 2, and the best known upper bound until recently was $\omega < 2.376$ achieved by Coppersmith and Winograd in 1987. There were two improvements on ω, one by Stothers in 2010 who showed that $\omega < 2.374$, and one by myself that ultimately resulted in $\omega < 2.373$. Here I discuss the road to these improvements and conclude with some open questions.

The **P** = **NP** Question



Clay Mathematics Institute

Dedicated to increasing and disseminating mathematical knowledge

HOME | ABOUT CMI | PROGRAMS | NEWS & EVENTS | AWARDS | SCHOLARS | PUBLICATIONS

Millennium Problems

In order to celebrate mathematics in the new millennium, The Clay Mathematics Institute of Cambridge, Massachusetts (CMI) has named seven *Prize Problems*. The Scientific Advisory Board of CMI selected these problems, focusing on important classic questions that have resisted solution over the years. The Board of Directors of CMI designated a $7 million prize fund for the solution to these problems, with $1 million allocated to each. During the Millennium Meeting held on May 24, 2000 at the Collège de France, Timothy Gowers presented a lecture entitled *The Importance of Mathematics*, aimed for the general public, while John Tate and Michael Atiyah spoke on the problems. The CMI invited specialists to formulate each problem.

One hundred years earlier, on August 8, 1900, David Hilbert delivered his famous lecture about open mathematical problems at the second International Congress of Mathematicians in Paris. This influenced our decision to announce the millennium problems as the central theme of a Paris meeting.

The rules for the award of the prize have the endorsement of the CMI Scientific Advisory Board and the approval of the Directors. The members of these boards have the responsibility to preserve the nature, the integrity, and the spirit of this prize.

Paris, May 24, 2000

- Birch and Swinnerton-Dyer Conjecture
- Hodge Conjecture
- Navier-Stokes Equations
- P vs NP
- Poincaré Conjecture
- Riemann Hypothesis
- Yang-Mills Theory

- Rules
- Millennium Meeting Videos