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.

Algorithmic Analysis

Tony Hoare

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

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 array into two halves: \( a_1 \) and \( a_2 \).
2. Sort each of \( a_1 \) and \( a_2 \) recursively.
3. Merge elements into the original array by choosing the smallest element from \( a_1 \) or \( a_2 \) on each cycle.

- The merge sort algorithm consists of the following steps:
  1. Divide the array into two halves.
  2. Sort each of these smaller arrays recursively.
  3. Merge the two arrays back into the original one.

```python
def sort(array):
    """Rearranges the elements of array in ascending order."""
    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

```python
# 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

- Sorting 8 items requires
  - Two sorts of 4 items which requires
    - Four sorts of 2 items which requires
      - Eight sorts of 1 item

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 array. 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 the same as the number of steps required for the binary search algorithm and is therefore equal to the number of times you can divide the original array in half until there is only one element remaining.
- The number of steps \( k \) is therefore given by the formula \( k = \log_2 N \).
- The complexity of merge sort is therefore \( O(N \log N) \).

### Comparing \( N^2 \) and \( N \log N \)

- The difference between \( O(N^2) \) and \( O(N \log N) \) is enormous for large values of \( N \), as shown in this table:

<table>
<thead>
<tr>
<th>( N )</th>
<th>( N^2 )</th>
<th>( N \log_2 N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>664</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000,000</td>
<td>9,966</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000,000</td>
<td>132,877</td>
</tr>
<tr>
<td>100,000</td>
<td>10,000,000,000</td>
<td>1,660,964</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000,000,000</td>
<td>19,931,569</td>
</tr>
</tbody>
</table>

- Based on these numbers, the theoretical advantage of using merge sort over selection sort on an array of 1,000,000 values would be a factor of more than 50,000.
Merge sort

Selection sort

Standard Complexity Classes

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

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Example Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>Finding first element in a vector</td>
</tr>
<tr>
<td>logarithmic</td>
<td>Binary search in a sorted vector</td>
</tr>
<tr>
<td>linear</td>
<td>Summing a vector, linear search</td>
</tr>
<tr>
<td>(N \log N)</td>
<td>Merge sort</td>
</tr>
<tr>
<td>cubic</td>
<td>Selection sort</td>
</tr>
<tr>
<td>quadratic</td>
<td>Obvious algorithms for matrix multiplication</td>
</tr>
<tr>
<td>exponential</td>
<td>Branch and try all possibilities</td>
</tr>
</tbody>
</table>

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

Matrix Multiplication and the Weather

- Matrix multiplication can be used to simulate evolving processes, including weather prediction. For example, the following matrix predicts the weather for tomorrow given the weather for today:

\[
\begin{bmatrix}
0.85 & 0.10 & 0.05 \\
0.60 & 0.25 & 0.15 \\
0.40 & 0.40 & 0.20 \\
\end{bmatrix}
\]
Matrix Multiplication and the Weather

- Given that matrix, you can predict the weather for the day after tomorrow by multiplying the matrix by itself.

<table>
<thead>
<tr>
<th></th>
<th>Rain</th>
<th>Clouds</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Today</td>
<td>0.81</td>
<td>0.13</td>
<td>0.06</td>
</tr>
<tr>
<td>The day after tomorrow will be</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.72</td>
<td>0.18</td>
<td>0.10</td>
<td></td>
</tr>
<tr>
<td>0.66</td>
<td>0.22</td>
<td>0.12</td>
<td></td>
</tr>
</tbody>
</table>

- What if you then repeat the process for ten days?

Ten days from now will be

<table>
<thead>
<tr>
<th></th>
<th>Rain</th>
<th>Clouds</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>If today is</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.77</td>
<td>0.14</td>
<td>0.07</td>
<td></td>
</tr>
<tr>
<td>0.77</td>
<td>0.14</td>
<td>0.07</td>
<td></td>
</tr>
<tr>
<td>0.77</td>
<td>0.14</td>
<td>0.07</td>
<td></td>
</tr>
</tbody>
</table>

Implementing Matrix Multiplication

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

```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?