Algorithm Analysis and Big Oh Notation

Courtesy of Prof. Ajay Gupta
(with updates by Dr. Leszek T. Lilien)

CS 1120 – Fall 2007
Department of Computer Science
Western Michigan University
Measuring the Efficiency of Algorithms

- Analysis of algorithms
  
  - Area of computer science

  - Provides tools for determining efficiency of different methods of solving a problem
    - E.g., the sorting problem - which sorting method is more efficient

- Comparing the efficiency of different methods of solution.
  
  - Concerned with significant differences
  
  - E.g.:
    
    - $n$ - the number of items to be sorted
    - Is the running time proportional to $n$ or proportional to $n^2$?
    - Big difference: e.g., for $n = 100$ it results in 100-fold difference; for $n = 1000$ it results in 1000-fold difference; for $n = ...$
How To Do Algorithm Comparison?

- **Approach 1:**
  Implement the algorithms in C#, run the programs, measure their performance, compare

  Many issues that affect the results:
  - How are the algorithms coded?
  - What computer should you use?
  - What data should the programs use?

- **Approach 2:**
  Analyze algorithms independently of their implementations

  How?

  For measuring/comparing execution time of algorithms
  - Count the number of basic operations of an algorithm
  - Summarize the count
The Execution Time of Algorithms

- Count the number of basic operations of an algorithm
  - Read, write, compare, assign, jump, arithmetic operations (increment, decrement, add, subtract, multiply, divide), open, close, logical operations (not/complement, AND, OR, XOR), ...
The Execution Time of Algorithms

- **Counting** an algorithm’s operations
  - Example: calculating a sum of array elements

```c
int sum = item[0]; <- 1 assignment
int j = 1; <- 1 assignment
while (j < n) <- n comparisons
{
    sum += item[j]; <- n-1 plus/assignments
    ++j; <- n-1 plus/assignments
}
```

**Total:** $3n$ operations

- **Notice:**
  - **Problem size** = $n$ = number of elements in an array
  - This problem of size $n$ requires solution with $3n$ operations
Algorithm Growth Rates

- Measure an algorithm’s time requirement as a function of the problem size
  - E.g., problem size = number of elements in an array
    - Algorithm A requires $n^2/5$ time units
    - Algorithm B requires $5n$ time units

- Algorithm efficiency is a concern for large problems only
  - For smaller values of $n$, $n^2/5$ and $5n$ not “that much” different
    - Imagine how big is the difference for $n > 1,000,000$
Common Growth-Rate Functions - I

(b)

Value of growth-rate function

\[ n \]
\[ n^2 \]
\[ n^3 \]
\[ 2^n \]
\[ n \log_2 n \]
\[ \log_2 n \]
### Differences among the growth-rate functions grow with \( n \)

- See the differences growing on the diagram on the previous page
- The bigger \( n \), the bigger differences -
  - that’s why algorithm efficiency is “concern for large problems only”
Similar table from the textbook:

<table>
<thead>
<tr>
<th>$n = \vdots$</th>
<th>$O(\log n)$</th>
<th>$O(n)$</th>
<th>$O(n \log n)$</th>
<th>$O(n^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>100</td>
<td>200</td>
<td>10000</td>
</tr>
<tr>
<td>1,000</td>
<td>3</td>
<td>1000</td>
<td>3000</td>
<td>$10^6$</td>
</tr>
<tr>
<td>1,000,000</td>
<td>6</td>
<td>1000000</td>
<td>6000000</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>9</td>
<td>1000000000</td>
<td>90000000000</td>
<td>$10^{18}$</td>
</tr>
</tbody>
</table>

**Fig. 24.13**  | Number of comparisons for common Big O notations.
Big-Oh Notation

- Algorithm A is order \( f(n) \) —denoted \( O(f(n)) \)— if there exist constants \( k \) and \( n_0 \) such that A requires \( \leq k \times f(n) \) time units to solve a problem of size \( n \geq n_0 \)

- Examples:
  - \( n^2/5 \)
    - \( O(n^2) \): \( k=1/5, \ n_0=0 \)
  - \( 5*n \)
    - \( O(n) \): \( k=5, \ n_0=0 \)
More Examples

How about $n^2 - 3n + 10$?

- It is $O(n^2)$ if there exist $k$ and $n_0$ such that $kn^2 \geq n^2 - 3n + 10$ for all $n \geq n_0$
- We see (fig.) that: $3n^2 \geq n^2 - 3n + 10$ for all $n \geq 2$
- So $k=3$, $n_0=2$
- More $k$-$n_0$ pairs could be found, but finding just one is enough to prove that $n^2 - 3n + 10$ is $O(n^2)$
Properties of Big-Oh

- Ignore low-order terms
  - E.g., $O(n^3+4n^2+3n)=O(n^3)$

- Ignore multiplicative constant
  - E.g., $O(5n^3)=O(n^3)$

- Combine growth-rate functions
  - $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
  - E.g., $O(n^2) + O(n\cdot \log_2 n) = O(n^2 + n\cdot \log_2 n)$
    - Then, $O(n^2 + n\cdot \log_2 n) = O(n^2)$
Worst-case vs. Average-case Analyses

- An algorithm can require different times to solve different problems of the same size.

- **Worst-case analysis** = find the maximum number of operations an algorithm can execute in all situations
  - Worst-case analysis is easier to calculate
  - More common

- **Average-case analysis** = enumerate all possible situations, find the time of each of the m possible cases, total and divide by m
  - Average-case analysis is harder to compute
  - Yields a more realistic expected behavior
Bigger Example: Analysis of Selection Sort

<table>
<thead>
<tr>
<th>values</th>
<th>0</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>

Divides the array into two parts: already sorted, and not yet sorted.

On each pass, finds the smallest of the unsorted elements, and swap it into its correct place, thereby increasing the number of sorted elements by one.
To find the smallest in UNSORTED:

indexMin = 0

comp. 1: check if values[1] = 24 < values[indexMin] = 36 - yes => indexMin = 1
comp. 2: check if values[2] = 10 < values[indexMin] = 24 - yes => indexMin = 2
comp. 3: check if values[3] = 6 < values[indexMin] = 10 - yes => indexMin = 3
comp. 4: check if values[4] = 12 < values[indexMin] = 6 - NO

Thus indexMin = 3; swap values[0] = 36 with values[indexMin] = 6 – see next slide
Selection Sort: End of Pass One

values

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SORTED

UNSORTED
To find the smallest in UNSORTED:

indexMin = 1
comp. 1: check if values[2] = 10 < values[indexMin] = 24 - yes => indexMin = 2
comp. 2: check if values[3] = 36 < values[indexMin] = 10 - NO
comp. 3: check if values[4] = 12 < values[indexMin] = 10 - NO
Thus indexMin = 2; swap values[1] = 24 with values[indexMin] = 10 – see next slide
Selection Sort: End of Pass Two

values

[ 0 ] 6
[ 1 ] 10
[ 2 ] 24
[ 3 ] 36
[ 4 ] 12

SORTED

UNSORTED
Selection Sort: Pass Three

To find the smallest in UNSORTED:
indexMin = 2
comp. 1: check if values[3] = 36 < values[indexMin] = 24 - NO
comp. 2: check if values[4] = 12 < values[indexMin] = 24 - yes => indexMin = 4
Thus indexMin = 4; swap values[2] = 24 with values[indexMin] = 12 – see next slide
Selection Sort: End of Pass Three

values
[ 0 ] 6
[ 1 ] 10
[ 2 ] 12
[ 3 ] 36
[ 4 ] 24

SORTED

UNSORTED
## Selection Sort: Pass Four

<table>
<thead>
<tr>
<th>values [0]</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>10</td>
</tr>
<tr>
<td>[2]</td>
<td>12</td>
</tr>
<tr>
<td>[3]</td>
<td>36</td>
</tr>
<tr>
<td>[4]</td>
<td>24</td>
</tr>
</tbody>
</table>

To find the smallest in UNSORTED:

- **indexMin = 3**
- comp. 1: check if `values[4] = 24 < values[indexMin] = 36` - yes => **indexMin = 4**

Thus **indexMin = 4**; swap `values[3] = 36` with `values[indexMin] = 24` – see next slide
Selection Sort: End of Pass Four

values

[ 0 ] 6
[ 1 ] 10
[ 2 ] 12
[ 3 ] 24
[ 4 ] 36

SORTED
### Selection Sort: How Many Comparisons?

<table>
<thead>
<tr>
<th>Values</th>
<th>Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] 6</td>
<td>4 comparisons starting with indexMin = 0</td>
</tr>
<tr>
<td>[1] 10</td>
<td>3 comparisons starting with indexMin = 1</td>
</tr>
<tr>
<td>[2] 12</td>
<td>2 comparisons starting with indexMin = 2</td>
</tr>
<tr>
<td>[3] 24</td>
<td>1 comparison starting with indexMin = 3</td>
</tr>
<tr>
<td>[4] 36</td>
<td>0 comparisons starting with indexMin = 4</td>
</tr>
</tbody>
</table>

In addition, we have \( \leq 4 \) swaps
For Selection Sort in General

- Above: Array contained 5 elements
  4 + 3 + 2 + 1 + 0 comparisons and <= 4 swaps were needed

- Generalization for Selection Sort:
  When the array contains N elements, the number of comparisons is:
  \[(N-1) + (N-2) + ... + 2 + 1 + 0\]
  and the number of swaps is  <= N-1

- Lets use:
  \[\text{Sum} = (N-1) + (N-2) + ... + 2 + 1 + 0\]
Calculating Number of Comparisons

\[
\text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1
\]

+ \[
\text{Sum} = 1 + 2 + \ldots + (N-2) + (N-1)
\]

\[
= 2 \times \text{Sum} = N + N + \ldots + N + N = N \times (N-1)
\]

- Since:

\[
2 \times \text{Sum} = N \times (N-1)
\]

then:

\[
\text{Sum} = 0.5 \, N^2 - 0.5 \, N
\]

- This means that we have \(0.5 \, N^2 - 0.5 \, N\) comparisons
And the Big-Oh for Selection Sort is...

- \( 0.5 N^2 - 0.5 N \) comparisons = \( O(N^2) \) comparisons
- \( N-1 \) swaps = \( O(N) \) swaps

- This means that complexity of Selection Sort is \( O(N^2) \)
  - Because \( O(N^2) + O(N) = O(N^2) \)
void SelectionSort (int values[], int numValues)
// Post: Sorts array values[0 . . numValues-1 ]
// into ascending order by key value
{
    int endIndex = numValues - 1;
    for (int current=0; current<endIndex; current++)
        Swap (values, current, MinIndex(values, current, endIndex));
}
Pseudocode for Selection Sort (contd)

int MinIndex(int values[], int start, int end)
// Post: Function value = index of the smallest value
// in values [start] . . . values [end].
{
    int indexOfMin = start;
    for(int index = start + 1; index <= end; index++)
        if (values[index] < values[indexOfMin])
            indexOfMin = index;
    return indexOfMin;
}