The Efficiency of Algorithms

Chapter 3
Topics:
Attributes of Algorithms
A Choice of Algorithms
Measuring Efficiency
Analysis of Algorithms
When Things Get Out of Hand

Attributes of Algorithms
• Correctness
  – Give a correct solution to the problem!
• Efficiency
  – Time: How long does it take to solve the problem?
  – Space: How much memory is needed?
  – Benchmarking vs. Analysis
• Ease of understanding
  – Program maintenance
• Elegance

A Choice of Algorithms
• Possible to come up with several different algorithms to solve the same problem.
• Which one is the "best"?
  – Most efficient
    – Time vs. Space?
    – Easiest to maintain?
• How do we measure time efficiency?
  – Running time? Machine dependent!
  – Number of steps?

The Data Cleanup Problem
• We look at three algorithms for the same problem, and compare their time- and space-efficiency.
• Problem: Remove 0 entries from a list of numbers.

1. The Shuffle-Left Algorithm
• We scan the list from left to right, and whenever we encounter a 0 element we copy ("shuffle") the rest of the list one position left.
Chapter 3: The Efficiency of Algorithms

2. The Copy-Over Algorithm

- We scan the list from left to right, and whenever we encounter a nonzero element we copy it over to a new list.

3. The Converging-Pointers Algorithm

- We scan the list from both left (L) and right (R). Whenever L encounters a 0 element, the element at location R is copied to location L, then R reduced.

Data-Cleanup Algorithm Comparison

- Which one is the most space efficient?
  - Shuffle-left: no additional space
  - Copy-over: needs a new list
  - Converging-pointers: no additional space

- Which one is the most time efficient?
  - Shuffle-left: many comparisons
  - Copy-over: goes through list only once
  - Converging-pointers: goes through list only once

- How do we measure time efficiency?
**Exercise**

- Can you come up with a more efficient algorithm for the data-cleanup problem, that does:
  - not require any additional space
  - less copying than shuffle-left
  - maintain the order of the none-zero elements
- Hint:
  - Can the copy-over algorithm be modified to copy the element into the same list?

**Measuring Efficiency**

- Need a metric to measure time efficiency of algorithms:
  - How long does it take to solve the problem?
    - Depends on machine speed
  - How many steps does the algorithm execute?
    - Better metric, but a lot of work to count all steps
  - How many “fundamental steps” does the algorithm execute?
    - Depends on size and type of input, interested in knowing:
      - Best-case, Worst-case, Average-case behavior
    - Need to analyze the algorithm!

**Sequential Search**

1. Get values for \( N, N_1,..., N_n, T_1,..., T_n \)
2. Set the value \( i \) to 1 and set the value of \( \text{Found} \) to \( \text{NO} \)
3. Repeat steps 4 through 7 until \( \text{Found} = \text{YES} \) or \( i > n \)
4. If \( \text{Name} = N_i \) then
5.    Print the value of \( T_i \)
6. Else
7.    Set the value of \( \text{Found} \) to \( \text{YES} \)
8. Add 1 to the value of \( i \)
9. If \( \text{Found} = \text{NO} \) then print “Sorry, name not in directory”
10. Stop

**Sequential Search - Analysis**

- How many steps does the algorithm execute?
  - Steps 2, 5, 6, 8 and 9 are executed at most once.
  - Steps 3, 4, and 7 depend on input size.
- Worst case:
  - Step 3, 4, and 7 are executed at most \( n \)-times.
- Best case:
  - Step 3 and 4 are executed only once.
- Average case:
  - Step 3, 4 are executed approximately \( (n/2) \)-times.
- Can use name comparisons as a fundamental unit of work!

**Order of Magnitude**

- We are:
  - Not interested in knowing the exact number of steps the algorithm performs.
  - Mainly interested in knowing how the number of steps grows with increased input size!
- Why?
  - Given large enough input, the algorithm with faster growth will execute more steps.
- Order of magnitude, \( O(...) \), measures how the number of steps grows with input size \( n \).
Linear Algorithms - $O(n)$

- If the number of steps grows in proportion, or linearly, with input size, it's a linear algorithm, $O(n)$.
  - Sequential search is linear, denoted $O(n)$.
- On a graph, will show as a straight line.

Non-linear Algorithm

- Think of an algorithm for filling out the $n$-times multiplication table.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>...</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>$n$</td>
</tr>
</tbody>
</table>

- As $n$ increases the work the algorithm does will increase by $n^2$, the algorithm is $O(n^2)$.

Data Cleanup - Analysis

<table>
<thead>
<tr>
<th></th>
<th>Shuffle-Left</th>
<th>Copy-Over</th>
<th>Converging pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>O($n$)</td>
<td>O($n$)</td>
<td>O($n$)</td>
</tr>
<tr>
<td>Space</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Best Case</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst Case</td>
<td>O($n^2$)</td>
<td>O($n$)</td>
<td>2n</td>
</tr>
<tr>
<td>Average Case</td>
<td>O($n^2$)</td>
<td>O($n$)</td>
<td>$n \leq x \leq 2n$</td>
</tr>
<tr>
<td>Space</td>
<td>$n$</td>
<td>$n$</td>
<td>O($n$)</td>
</tr>
</tbody>
</table>

Sorting

- Sorting is a very common task, for example:
  - Sorting a list of names into alphabetical order
  - Numbers into numerical order
- Important to find efficient algorithms for sorting:
  - Selection sort
  - Bubble sort
  - Quick sort
  - Heap sort
- We will analyze the complexity of selection sort.

Selection Sort

- Divide the list into an unsorted and a sorted section, initially the sorted section is empty.
- Locate the largest element in the unsorted section and replace that with the last element of the unsorted section.
- Move the marker between the unsorted and sorted section one position to the left.
- Repeat until unsorted section of the list is empty.
Selection Sort - Animation
• Exchange the largest element of the unsorted section with the last element of the unsorted section
• Move marker separating the unsorted and sorted section one position to the left (forward in the list)
• Continue until unsorted section is empty.

Selection Sort - Analysis
• What order of magnitude is this algorithm?
  – Use number of comparisons as a fundamental unit of work.
• Total number of comparisons:
  \[(n-1) + (n-2) + \ldots + 2 + 1\]
  \[= \frac{(n-1)}{2} \cdot n\]
  \[= \frac{1}{2} n^2 - \frac{1}{2} n\]
• This is an \(O(n^2)\) algorithm.
• Worst, best, average case behavior the same (why?)

Binary Search
• How do we look up words in a list that is already sorted?
  – Dictionary
  – Phone book
• Method:
  – Open up the book roughly in the middle.
  – Check in which half the word is.
  – Split that half again in two.
  – Continue until we find the word.

Binary Search - Example
To find Nancy, we go through:
Garry (midpoint at 4)
Pat (midpoint of 5-7)
Nancy (midpoint of a single item)

Binary Search - Odd number of elements
Whom that can be found
in one step: Garry
in two steps: Bob, Pat
in three steps: all remaining persons

Binary Search - Even number of elements
Let’s choose the end of first half as midpoint.
Whom that can be found
in one step: Dave
in two steps: Ann, Nancy
in three steps: all remaining persons
Binary Search - Analysis

- Looking for a name is like walking branches in a tree

<table>
<thead>
<tr>
<th>Ann</th>
<th>Bob</th>
<th>Dave</th>
<th>Garry</th>
<th>Nancy</th>
<th>Pat</th>
<th>Sue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Binary Search - Analysis (cont.)

- We cut the number of remaining names in half.
- The number of times a number $n$ can be cut in half and not get below 1 is called
  - Logarithm of $n$ to the base 2
  - Notation: $\log_2 n$ or $\lg n$
- Max. number of name comparisons = depth of tree.
  - 3 in the previous example.
  - $n$ names then approx. $\lg n$ comparisons needed
- Binary search is $O(\lg n)$

Logarithm vs. Linear

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\lg n$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>128</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>256</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>512</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>1024</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>32768</td>
<td>15</td>
<td>...</td>
</tr>
<tr>
<td>1048576</td>
<td>20</td>
<td>...</td>
</tr>
</tbody>
</table>

When Things Get Out of Hand

- Polynomial algorithms (exponent is a constant)
  - For example: $\lg n$, $n$, $n^2$, ..., $n^{3000}$, ...
  - More generally: $n^a$
- Exponential algorithms (exponent function of $n$)
  - For example: $2^n$
  - More generally: $a^n$
- An exponential algorithm:
  - Given large enough $n$ will always perform more work than a polynomially bounded one.
- Problem for which there exist only exponential algorithms are called intractable
  - Solvable, but not within practical time limits
  - Most often it is infeasible to solve but the smallest problems!

Growth Rate

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\lg(n)$</th>
<th>$n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>.0003 sec</td>
<td>.001 sec</td>
<td>.01 sec</td>
<td>.1024 sec</td>
</tr>
<tr>
<td>50</td>
<td>.0006 sec</td>
<td>.005 sec</td>
<td>.25 sec</td>
<td>3570 years</td>
</tr>
<tr>
<td>100</td>
<td>.0007 sec</td>
<td>.01 sec</td>
<td>1 sec</td>
<td>4*10^{16} centuries</td>
</tr>
<tr>
<td>1000</td>
<td>.001 sec</td>
<td>.1 sec</td>
<td>1.67 min</td>
<td>Too big</td>
</tr>
</tbody>
</table>
Summary

• We are concerned with the efficiency of algorithms
  – Time- and Space-efficiency
  – Need to analyze the algorithms
• Order of magnitude measures the efficiency
  – E.g. O(lg n), O(n), O(n^2), O(n^3), O(2^n), ...
  – Measures how fast the work grows as we increase the
    input size n.
  – Desirable to have slow growth rate.

Summary

• We looked at different algorithms
  – Data-Cleanup: Shuffle-left O(n^2), Copy-over O(n),
    Converging-pointers O(n)
  – Search: Sequential-search O(n), Binary-search O(lg n)
  – Sorting: Selection-sort O(n^2)
• Some algorithms are exponential
  – Not polynomially bounded
  – Problems for which there exists only exponential
    algorithms are called intractable
  – Only feasible to solve small instances of such problems