CS 171: Introduction to Computer Science II

Algorithm Analysis

Li Xiong
Summary

• Fundamental data structures
  – Arrays
  – Linked lists

• Abstract data types
  – Stacks
  – Queues

• Algorithms
  – Linear search, Binary search
  – Backtracking algorithm (Hw2)
  – Path finding algorithm (Hw3)

• Java and OO
  – Inheritance
  – Interface
Road Map

- Algorithm Analysis
- Sorting algorithms
- Binary search trees
- Hash tables
- Graphs
Algorithm Analysis

• An algorithm is a method for solving a problem expressed as a sequence of steps that is suitable for execution by a computer (machine)
  – E.g. Search in an ordered array
  – E.g. N-Queens problem
• We are interested in designing good algorithms
  – Linear search vs. binary search
  – Brute-force search vs. backtracking
• Good algorithms
  – Running time
  – Space usage (amount of memory required)
Running time of an algorithm

• Running time typically increases with the input size (problem size)
• Also affected by hardware and software environment
• We would like to focus on the relationship between the running time and the input size
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time
- Plot the results
- E.g. Stopwatch.java
Limitations of Experiments

• It is necessary to implement the algorithm, which may be difficult

• In order to compare two algorithms, the same hardware and software environments must be used

• Results may not be indicative of the running time on other inputs not included in the experiment.
MY HOBBY: EXTRAPOLATING

As you can see, by late next month you'll have over four dozen husbands.

Better get a bulk rate on wedding cake.
Algorithm Analysis - insight

• Total running time of a program is determined by two primary factors:
  – Cost of executing each statement (property of computer, Java compiler, OS)
  – Frequency of execution of each statement (property of program and input)
Algorithm Analysis

• Algorithm analysis:
  – Determine frequency of primitive operations
  – Characterizes it as a function of the input size

• A primitive operation:
  – Corresponds to a low-level (basic) computation with a constant execution time
  – E.g. Evaluating an expression; assigning a value to a variable; indexing into an array

• Benefit:
  – Takes into account all possible inputs
  – Independent of the hardware/software environment
Average-case vs. worst-case

• An algorithm may run faster on some inputs than it does on others (with the same input size)

• Average case: taking the average over all possible inputs of the same size
  – Depends on input distribution

• Best case

• Worst case
  – Easier analysis
  – Typically leads to better algorithms
Misleading Average

A statistician who put her head in the oven and her feet in the refrigerator.

She said, “On average, I feel just fine.”
Misleading Average

Statistician drowning in a pond with an average depth of 3ft
Loop Analysis

• Programs typically use loops to enumerate through input data items
• Count number of operations or steps in loops
• Each statement within the loop is counted as a step
Example 1

double sum = 0.0;
for (int i = 0; i < n; i ++) {
    sum += array[i];
}

How many steps?
Only count the loop statements (update to the loop variable $i$ is ignored)
double sum = 0.0;
for (int i = 0; i < n; i++) {
    sum += array[i];
}

How many steps?
Only count the loop statements (update to the loop variable i is ignored)

n
Example 2

double sum = 0.0;
for (int i = 0; i < n; i += 2) {
    sum += array[i];
}

How many steps?
Example 2

double sum = 0.0;
for (int i = 0; i < n; i += 2) {
    sum += array[i];
}

How many steps?

Loops will be executed n/2 times:

n/2
Example 3 – Multiple Loops

```c
definition
for (int i = 0; i < n; i ++) {
    for (int j = 0; j < n; j ++) {
        int x = i*j;
        sum += x;
    }
}
```

How many steps?
Example 3 – Multiple Loops

```java
for (int i = 0; i < n; i ++) {
    for (int j = 0; j < n; j ++) {
        int x = i*j;
        sum += x;
    }
}
```

How many steps?

Nested loops, each loop will be executed n times:

\[ n^2 \]
Increase of Cost w.r.t. n

• Example 1: $n$
• Example 2: $n/2$
• Example 3: $2n^2$

• What if $n$ is 100? What if $n$ is 3 times larger?

• Example 1 and 2 are linear to the input size
• Example 3 is quadratic to the input size
Mathematical notations
for algorithm analysis

• The cost function can be complicated and
lengthy mathematical expressions
  – E.g. $3n^3 + 20n^2 + 5$
• We care about how the cost increases w.r.t.
the problem size, rather than the absolute
cost
• Use simplified mathematical notions
  – Tilde notation
  – Big O notation
Tilde Notation

• Tilde notation: ignore insignificant terms
• Definition: we write $f(n) \sim g(n)$ if $f(n)/g(n)$ approaches 1 as $n$ grows

• $2n + 10 \sim 2n$
• $3n^3 + 20n^2 + 5 \sim 3n^3$
Big-Oh Notation

• Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

• Example: $2n + 10$ is $O(n)$
  − pick $c = 3$ and $n_0 = 10$
Big-Oh Example

• Example: the function $n^2$ is not $O(n)$
  
  $-n^2 \leq cn$

  $-n \leq c$

  The above inequality cannot be satisfied since $c$ must be a constant
Big-Oh and Growth Rate

• The big-Oh notation gives an upper bound on the growth rate of a function
• The statement “\( f(n) \) is \( O(g(n)) \)” means that the growth rate of \( f(n) \) is no more than the growth rate of \( g(n) \)
• We can use the big-Oh notation to rank functions according to their growth rate
Important Functions in Big-Oh Analysis

- Constant: \( 1 \)
- Logarithmic: \( \log n \)
- Linear: \( n \)
- N-Log-N: \( n \log n \)
- Quadratic: \( n^2 \)
- Cubic: \( n^3 \)
- Polynomial: \( n^d \)
- Exponential: \( 2^n \)
- Factorial: \( n! \)
In terms of the order: exponentials > polynomials > logarithms > constant.
<table>
<thead>
<tr>
<th>description</th>
<th>order of growth</th>
<th>typical code framework</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>1</td>
<td>( a = b + c; )</td>
<td>statement</td>
<td>add two numbers</td>
</tr>
<tr>
<td>logarithmic</td>
<td>( \log N )</td>
<td>[ see page 47 ]</td>
<td>divide in half</td>
<td>binary search</td>
</tr>
<tr>
<td>linear</td>
<td>( N )</td>
<td>double max = a[0]; for (int i = 1; i &lt; N; i++) if (a[i] &gt; max) max = a[i];</td>
<td>loop</td>
<td>find the maximum</td>
</tr>
<tr>
<td>linearithmic</td>
<td>( N \log N )</td>
<td>[ see ALGORITHM 2.4 ]</td>
<td>divide and conquer</td>
<td>mergesort</td>
</tr>
<tr>
<td>quadratic</td>
<td>( N^2 )</td>
<td>for (int i = 0; i &lt; N; i++) for (int j = i+1; j &lt; N; j++) if (a[i] + a[j] == 0) cnt++;</td>
<td>double loop</td>
<td>check all pairs</td>
</tr>
<tr>
<td>cubic</td>
<td>( N^3 )</td>
<td>for (int i = 0; i &lt; N; i++) for (int j = i+1; j &lt; N; j++) for (int k = j+1; k &lt; N; k++) if (a[i] + a[j] + a[k] == 0) cnt++;</td>
<td>triple loop</td>
<td>check all triples</td>
</tr>
<tr>
<td>exponential</td>
<td>( 2^N )</td>
<td>[ see CHAPTER 6 ]</td>
<td>exhausutive search</td>
<td>check all subsets</td>
</tr>
</tbody>
</table>
## Practical implications of Order-or-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1970s</td>
</tr>
<tr>
<td>1</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
</tr>
<tr>
<td>N²</td>
<td>hundreds</td>
</tr>
<tr>
<td>N³</td>
<td>hundred</td>
</tr>
<tr>
<td>2^N</td>
<td>20</td>
</tr>
</tbody>
</table>
Algorithm Analysis

• Derive cost function $f(n)$
• Throw away low-order terms (tilde notation)
• Drop constant factors (big Oh notation, order of growth)

<table>
<thead>
<tr>
<th>function</th>
<th>tilde approximation</th>
<th>order of growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N^3/6 - N^2/2 + N/3$</td>
<td>$\sim N^3/6$</td>
<td>$N^3$</td>
</tr>
<tr>
<td>$N^2/2 - N/2$</td>
<td>$\sim N^2/2$</td>
<td>$N^2$</td>
</tr>
<tr>
<td>$\lg N + 1$</td>
<td>$\sim \lg N$</td>
<td>$\lg N$</td>
</tr>
<tr>
<td>3</td>
<td>$\sim 3$</td>
<td>1</td>
</tr>
</tbody>
</table>
Useful Approximations

• Harmonic sum
  \[ 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N} \sim \ln N \]

• Triangular sum
  \[ 1 + 2 + 3 + \ldots + N = \frac{N(N+1)}{2} \sim \frac{N^2}{2} \]

• Geometric sum
  \[ 1 + 2 + 4 + \ldots + N = 2N - 1 \sim 2N \text{ when } N = 2^n \]

• Stirling’s approximation
  \[ \lg N! = \lg 1 + \lg 2 + \lg 3 + \ldots + \lg N \sim N \lg N \]
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        sum += i*j;
    }
}
Example 4

```c
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        sum += i*j;
    }
}
```

\[ n+(n-1)+(n-2)+...+1+0 = \frac{n(n+1)}{2} \text{ is } O(n^2) \]
double product = 1.0;
for (int i = 1; i <= n; i *= 2) {
    product *= i;
}

Example 5: Solution

double product = 1.0;
for (int i = 1; i <= n; i *= 2) {
    product *= i;
}

• This has a logarithmic cost:

\[ O(\log_2 n) \]

or \[ O(\log n) \] as the change of base is merely a matter of a constant factor.
Search in Ordered vs. Unordered Array

• What’s the big O function for linear search?
• Binary search?
Search in Ordered vs. Unordered Array

- What’s the big O function for linear search? $O(N)$
- Binary search? $O(\log N)$
- Binary search has much better running time, particularly for large-scale problems
Bonus Question

• What is the Order of growth (big-oh) of the following code?

```java
for (int i = 1; i <= N; ++i) {
    for (int j = 1; j < N; j *= 2) {
        count++;
    }
}
```