Overview

- Topics
  - Measuring time and space used by algorithms
  - Machine-independent measurements
  - Costs of operations
  - Comparing algorithms
  - Asymptotic complexity – \( O() \) notation and complexity classes

Comparing Algorithms

- Example: We've seen two different list implementations
  - Dynamic expanding array
  - Linked list
- Which is "better"?
- How do we measure?
  - Stopwatch? Why or why not?

Program Efficiency & Resources

- Goal: Find way to measure "resource" usage in a way that is independent of particular machines/implementations
- Resources
  - Execution time
  - Execution space
  - Network bandwidth
  - others
- We will focus on execution time
  - Basic techniques/vocabulary apply to other resource measures

Example

- What is the running time of the following method?
  ```java
  double sum(double[] rainMeas) {
    double ans = 0.0;
    for (int k = 0; k < rainMeas.length; k++) {
      ans = ans + rainMeas[k];
    }
    return ans;
  }
  ```
- How do we analyze this?
Analysis of Execution Time

1. First: describe the size of the problem in terms of one or more parameters
   - For sum, size of array makes sense
   - Often size of data structure, but can be magnitude of some numeric parameter, etc.
2. Then, count the number of steps needed as a function of the problem size
   - Need to define what a "step" is.
   - First approximation: one simple statement
   - More complex statements will be multiple steps

Cost of operations: Constant Time Ops

- Constant time operations: each take one abstract time "step"
  - Simple variable declaration/initialization (double sum = 0.0;)
  - Assignment of numeric or reference values (var = value;)
  - Arithmetic operation (+, -, *, /, %)
  - Array subscripting (a[index])
  - Simple conditional tests (x < y, p != null)
  - Operator new itself (not including constructor cost)
    Note: new takes significantly longer than simple arithmetic or assignment, but its cost is independent of the problem we're trying to analyze
  - Note: watch out for things like method calls or constructor invocations that look simple, but are expensive

Cost of operations: Zero-time Ops

- Compiler can sometimes pay the whole cost of setting up operations
  - Nothing left to do at runtime
- Variable declarations without initialization
  ```java
double[] overdrafts;
```
- Variable declarations with compile-time constant initializers
  ```java
  static final int maxButtons = 3;
  ```
- Casts (of reference types, at least)
  ```java
  ...  (Double) checkBalance
  ```

Sequences of Statements

- Cost of
  ```java
  S1; S2; ... Sn
  ```
  is sum of the costs of S1 + S2 + ... + Sn

Conditional Statements

- The two branches of an if-statement might take different times. What to do??
  ```java
  if (condition) {
     S1;
  } else {
     S2;
  }
  ```
- Hint: Depends on analysis goals
  - "Worst case": the longest it could possibly take, under any circumstances
  - "Average case": the expected or average number of steps
  - "Best case": the shortest possible number of steps, under some special circumstance
  - Generally, worst case is most important to analyze

Analyzing Loops

- Basic analysis
  1. Calculate cost of each iteration
  2. Calculate number of iterations
  3. Total cost is the product of these
    Caution: sometimes need to add up the costs differently if cost of each iteration is not roughly the same
- Nested loops
  - Total cost is number of iterations or the outer loop times the cost of the inner loop
  - same caution as above
Method Calls

- Cost for calling a function is cost of...
  - Cost of evaluating the arguments (constant or non-constant)
  - Cost of actually calling the function (constant overhead)
  - Cost of executing the function body (constant or non-constant?)

```java
System.out.print(this.lineNumber);
System.out.println("Answer is " + Math.sqrt(3.14159));
```

Terminology note: "evaluating" and "passing" an argument are two different things!

Exact Complexity Function

- Careful analysis of an algorithm leads to an algebraic formula
- The "exact complexity function" gives the number of steps as a function of the problem size
- What can we do with it?
  - Predict running time in a particular case (given n, given type of computer)?
  - Predict comparative running times for two different n (on same type of computer)?
  - Get a general feel for the potential performance of an algorithm
  - Compare predicted running time of two different algorithms for the same problem (given same n)

A Graph is Worth A Bunch of Words

- Graphs are a good tool to illustrate, study, and compare complexity functions

- Fun math review for you
  - How do you graph a function?
  - What are the shapes of some common functions? For example, ones mentioned in these slides or the textbook.

Exercise

- Analyze the running time of `printMultTable`
  - Pick the problem size
  - Count the number of steps

```java
// print triangular multiplication table with n rows
void printMultTable( int n) {
  for ( int k=0; k <=n; k++) {
    printRow(k);
  }
}

// print row r with length r of a multiplication table
void printRow( int r) {
  for ( int k = 0; k <= r; k++) {
    System.out.print( r * k + " ");
  }
  System.out.println( );
}
```

Comparing Algorithms

- Suppose we analyze two algorithms and get these times (numbers of steps):
  - Algorithm 1: $37n + 2n^2 + 120$
  - Algorithm 2: $50n + 42$

How do we compare these? What really matters?

- Answer: In the long run, the thing that is most interesting is the cost as the problem size n gets large
- What are the costs for n=10, n=100; n=1,000; n=1,000,000?
- Computers are so fast that how long it takes to solve small problems is rarely of interest

Orders of Growth

- Examples:

<table>
<thead>
<tr>
<th></th>
<th>log2N</th>
<th>5N</th>
<th>N log2N</th>
<th>N^2</th>
<th>2^N</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>40</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>80</td>
<td>64</td>
<td>256</td>
<td>65536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>160</td>
<td>1024</td>
<td>~10^9</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>320</td>
<td>384</td>
<td>4096</td>
<td>~10^13</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>640</td>
<td>896</td>
<td>16384</td>
<td>~10^18</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>1280</td>
<td>2048</td>
<td>65536</td>
<td>~10^16</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
<td>50000</td>
<td>10^5</td>
<td>10^8</td>
<td>~10^310</td>
</tr>
</tbody>
</table>
Asymptotic Complexity

- Asymptotic: Behavior of complexity function as problem size gets large
  - Only thing that really matters is higher-order term
  - Can drop low order terms and constants

- The asymptotic complexity gives us a (partial) way to answer “which algorithm is more efficient”
  - Algorithm 1: $37n + 2n^2 + 120$ is proportional to $n^2$
  - Algorithm 2: $50n + 42$ is proportional to $n$
- Graphs of functions are handy tool for comparing asymptotic behavior

Big-O Notation

- Definition: If $f(n)$ and $g(n)$ are two complexity functions, we say that $f(n) = O(g(n))$ (pronounced $f(n)$ is $O(g(n))$ or is order $g(n)$) if there is a constant $c$ such that $f(n) \leq cg(n)$ for all sufficiently large $n$

Exercises

- Prove that $5n+3$ is $O(n)$
- Prove that $5n^2 + 42n + 17$ is $O(n^2)$

Implications

- The notation $f(n) = O(g(n))$ is not an equality
- Think of it as shorthand for
  - “$f(n)$ grows at most like $g(n)$”
  - “$f$ grows no faster than $g$”
  - “$f$ is bounded by $g$”

- $O(\cdot)$ notation is a worst-case analysis
  - Generally useful in practice
  - Sometimes want average-case or expected-time analysis if worst-case behavior is not typical (but often harder to analyze)

Complexity Classes

- Several common complexity classes (problem size $n$)
  - Constant time: $O(k)$ or $O(1)$
  - Logarithmic time: $O(\log n)$ [Base doesn’t matter. Why?]  
  - Linear time: $O(n)$
  - “$n \log n$” time: $O(n \log n)$
  - Quadratic time: $O(n^2)$
  - Cubic time: $O(n^3)$
  - Exponential time: $O(k^n)$
- $O(n^k)$ is often called polynomial time

Rule of Thumb

- If the algorithm has polynomial time or better: practical
  - Typical pattern: examining all data, a fixed number of times
- If the algorithm has exponential time: impractical
  - Typical pattern: examine all combinations of data
- What to do if the algorithm is exponential?
  - Try to find a different algorithm
  - Some problems can be proved not to have a polynomial solution
  - Other problems don’t have known polynomial solutions, despite years of study and effort.
  - Sometimes you settle for an approximation:
    - The correct answer most of the time, or
    - An almost-correct answer all of the time
Big-O Arithmetic

- For most commonly occurring functions, comparison can be enormously simplified with a few simple rules of thumb.
- Memorize complexity classes in order from smallest to largest: \( O(1) \), \( O(\log n) \), \( O(n) \), \( O(n \log n) \), \( O(n^2) \), etc.
- Ignore constant factors
  \[ 300n + 6n + 2 = O(n + n^4 + 2n) \]
- Ignore all but highest order term
  \[ O(n + n^4 + 2n) = O(2^n) \]

Analyzing List Operations (1)

- We can use \( O() \) notation to compare the costs of different list implementations
- Operation | Dynamic Array | Linked List
- Construct empty list
- Size of the list
- isEmpty
- clear

Analyzing List Operations (2)

- Operation | Dynamic Array | Linked List
- Add item to end of list
- Locate item (contains, indexOf)
- Add or remove item once it has been located

How long is a Computer-Day?

- If a program needs \( f(n) \) microseconds to solve some problem, what is the largest single problem it can solve in one full day?
- One day = \( 1,000,000 \times 24 \times 60 \times 60 = 10^{16} \times 24 \times 36 \times 102 = 8.64 \times 10^{16} \) microseconds.
- To calculate, set \( f(n) = 8.64 \times 10^{16} \) and solve for \( n \) in each case

```
\( f(n) \quad n \text{ such that } f(n) \)
\-----------------------------
\( n \quad 8.64 \times 10^{11} \)
\( 5n \quad 1.73 \times 10^{12} \)
\( n \log n \quad 2.75 \times 10^9 \)
\( n^2 \quad 2.94 \times 10^5 \)
\( n^3 \quad 4.42 \times 10^3 \)
\( 2^n \quad 36 \)
```

Wait! Isn’t this totally bogus??

- Write better code!!
- More clever hacking in the inner loops (assembly language, special-purpose hardware in extreme cases)
- Moore’s law: Speeds double every 18 months
- Wait and buy a faster computer in a year or two!

- But …

Speed Up The Computer by 1,000,000

- Suppose technology advances so that a future computer is 1,000,000 times faster than today’s.
- In one day there are now = \( 8.64 \times 10^{16} \times 10^6 \) ticks available
- To calculate, set \( f(n) = 8.64 \times 10^{16} \) and solve for \( n \) in each case

```
f(n) \quad \text{original } n \text{ for one day} \quad \text{new } n \text{ for one day}
-----------------------------------------------
n \quad 8.64 \times 10^{11} \quad ??????????
5n \quad 1.73 \times 10^{12} \quad ??????????
n \log n \quad 2.75 \times 10^9 \quad \text{etc.}
n^2 \quad 2.94 \times 10^5 \quad \text{etc.}
n^3 \quad 4.42 \times 10^3 \quad \text{etc.}
2^n \quad 36
```
How Much Does 1,000,000-faster Buy?

- Divide the new max \( n \) by the old max \( n \), to see how much more we can do in a day

<table>
<thead>
<tr>
<th>( f(n) )</th>
<th>( n ) for 1 day</th>
<th>new ( n ) for 1 day</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( 8.64 \times 10^{10} )</td>
<td>( 8.64 \times 10^{16} )</td>
</tr>
<tr>
<td>( 5n )</td>
<td>( 1.73 \times 10^{10} )</td>
<td>( 1.73 \times 10^{16} )</td>
</tr>
<tr>
<td>( n \log_2 n )</td>
<td>( 2.75 \times 10^9 )</td>
<td>( 1.71 \times 10^{15} )</td>
</tr>
</tbody>
</table>

Practical Advice For Speed Lovers

- First pick the right algorithm and data structure
- Implement it carefully, ensuring correctness
- Then optimize for speed — but only where it matters
- Constants do matter in the real world
- Clever coding can speed things up, but result can be harder to read, modify
- Current state of the art approach: Use measurement tools to find hotspots, then tweak those spots.

"Premature optimization is the root of all evil" — Donald Knuth

Summary

- Analyze algorithm sufficiently to determine complexity
- Compare algorithms by comparing asymptotic complexity
- For large problems an asymptotically faster algorithm will always trump clever coding tricks

"Premature optimization is the root of all evil" — Donald Knuth

Computer Science Note

- Algorithmic complexity theory is one of the key intellectual contributions of Computer Science
- Typical problems
  - What is the worst/average/best-case performance of an algorithm?
  - What is the best complexity bound for all algorithms that solve a particular problem?
- Interesting and (in many cases) complex, sophisticated math
  - Probabilistic and statistical as well as discrete
  - Still some key open problems
  - Most notorious: \( P \neq NP \)