Algorithm Analysis & Program Testing

An introduction
Time Analysis: reasoning about speed of algorithm

• Need to decide how time will be measured
  – actual performance time: not always accurate
    • different system
    • same system, different time
  – number of operations performed -- more accurate measure
What’s an operation?

• Declaration

• Assignment statement:
  
  \[ x = 2; \quad // \text{one operation} \]
  
  \[ x = y + 2; \quad // \text{two operations} \]

• Input/Output statement

• Method call

• Most conditional statements
public class MessyFixed {
    public static void main (String [] args) {
        int first = 4;
        int second = first * 2;
        int third = first + second;
        System.out.println ("Values are: " + first + ", " + second + ", " + third);
        System.out.println ("first + second = " + (first + second));
        System.out.println ("third = " + third);
    }
}
Big-O notation

• Can use number of operations in an algorithm to determine rough measure of amount of time needed to perform it
• Size of data set (amount of input) largely determines relative speed, so Big-O takes this into account
Big-O examples

• An algorithm executes in constant time if it contains statements that don’t depend on size of data set -- said to be $O(1)$

• An algorithm executes in linear time if the number of operations is equal to the size of the data set -- $O(N)$
Big-O is an approximation

- Algorithm is still $O(1)$ if it takes 20 constant statements or 20,000
- A loop is linear if it depends on the size of the data set, but constant if it terminates at a pre-set value
- Constant values are ignored -- $O(N + 1000)$ is still $O(N)$
Analysis example -- loops

• For any loop, analysis has two parts:
  – number of iterations
  – amount of work done during each iteration

• Example:

```c
int sum=0, j;
for (j=0; j<N; j++)
  sum=sum+j;
```

• Loop executes N times

• 4 operations in loop:
  – j<N
  – sum+j
  – sum=
  – j++ (could count as 2)

• O(4) == O(1)

• Total is N * O(1) or O(N)
Another loop example

```c
int sum = 0, j;
for (j=0; j<100; j++)
    sum = sum + j;
```

- Loop executes 100 times
- Work inside loop is still $O(4)$, which is $O(1)$
- Total is $100 \cdot O(1)$, or $O(100)$, which is still $O(1)$
Nested Loops

- Treat like single loop, evaluating each level of nesting as needed:
  ```
  int j,k, sum=0;
  for (j=0; j<N; j++)
     for (k=N; k>0; k--)
       sum+=k+j;
  ```

- Start with outer loop -- N iterations -- amount of work depends on inner loop
  - Inner loop is \( O(N) \)
  - Total is \( N \times O(N) \), or \( O(N^2) \)
Conditional Statements

- To evaluate an if-statement, compute costs of “then” & “else” parts, then take larger:
  ```
  if (InputDone)
  {
      for (j=0; j<N; j++)
          cout << Data[j];
  }
  else
      cout << “Not done yet” << endl;
  ```

- **Would be O(N) because O(N) > O(1)**
What is the order of magnitude (Big O)?

```java
if (digit >= 5)
    output = output + "D";
digit = digit % 5;
while (digit > 0)
{
    output = output + "C";
    digit--;
}
```
Method Calls

• Treat method like any other block of code, use its complexity function (Big-O) wherever it is called

• If method call occurs in a loop, determine cost of function call & use to determine cost of loop
Commonly Used Terminology for Complexity of Algorithms

- O(1)  
  Constant complexity
- O(log N)  
  Logarithmic complexity
- O(N)  
  Linear complexity
- O(N \log N)  
  N log N complexity
- O(N^b)  
  Polynomial complexity
- O(b^N)  
  Exponential complexity (where b > 1)
- O(N!)  
  Factorial complexity
Tractability

• A problem that is solvable using an algorithm of polynomial complexity (or less) is said to be tractable

• This means there is an expectation that a solution can be found in a reasonable amount of time

• Generally true if degree and coefficient of polynomial are relatively small
Tractability

• An intractable problem is one that cannot be solved with worst-case polynomial time
• In practice, an intractable problem is not necessarily unsolvable; remember, we’re only looking at the worst case
• An unsolvable problem is one for which no algorithm exists
Limitations of Big-O

• Estimate of time complexity of algorithm expresses how the amount of time required to solve a problem changes with larger and larger input sets
• Can’t be directly translated into amount of time used by algorithm
Comparisons of Actual Computer Time for Algorithms

• The next slide presents a table of actual time required to solve problems of specific sizes given these assumptions:
  – All operations reduced to bit level, so algorithm complexity translates directly to number of bit operations
  – Each operation requires exactly $10^{-9}$ seconds to perform
## Computer Time Used by Algorithms

<table>
<thead>
<tr>
<th>Problem Size(N)</th>
<th>Log N</th>
<th>Bit Operations Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>10</td>
<td>3x10^{-9}</td>
<td>10^{-8}</td>
</tr>
<tr>
<td>10^2</td>
<td>7x10^{-9}</td>
<td>10^{-7}</td>
</tr>
<tr>
<td>10^3</td>
<td>10^{-8}</td>
<td>10^{-6}</td>
</tr>
<tr>
<td>10^4</td>
<td>1.3x10^{-8}</td>
<td>10^{-5}</td>
</tr>
<tr>
<td>10^5</td>
<td>1.7x10^{-8}</td>
<td>10^{-4}</td>
</tr>
<tr>
<td>10^6</td>
<td>2.0x10^{-8}</td>
<td>10^{-3}</td>
</tr>
</tbody>
</table>
Notes on Computer Times

• Most algorithms, even $O(N^2)$, execute in a reasonable amount of time for small inputs
• But note difference as $N$ grows for quadratic algorithm:
  – for $N=10,000$, takes .1 second
  – for $N=100,000$, takes 10 seconds
  – for $N=1,000,000$, takes 17 minutes
• The difference grows even faster for $O(2^N)$ and $O(N!)$. 
Big-O Only One Measure to Consider

• Number of inputs can differ wildly from one run of the program to another

• Useful to think in terms of 3 possible cases to analyze:
  – worst-case: maximum number of required operations
  – best-case: minimum number
  – average case: average of best & worst
Program Testing & Debugging

• Testing: running a program to observe its behavior

• Need to select & use good test data to determine whether or not run-time errors will occur

• Purpose of testing is to find bugs so they can be eliminated; not to prove bugs don’t exist
Properties of Good Test Data

• Known expected output for given input
• Include inputs most likely to cause error
• Test boundary values
Fully Exercising Code

• Make sure each line of code is executed at least once by some of your test data
• If some portion of code should be skipped under certain conditions, make sure you include test data that creates those conditions, and that the code gets skipped
Debugging Do’s

• Limit code changes to corrections of known errors
• Rerun test cases each time a change is made
• Keep track of changes as you make them
• Isolate potential problem areas
  – commenting out
  – using breakpoints
Algorithm Analysis & Program Testing

.ends.