Analysis of Algorithms
An Introduction

CS 3358
Spring 2015

Jill Seaman

Note: in this lecture "function" almost always refers to a mathematical function, as in f(x) = x+101

Sections 6.1, 6.2, 6.4 (optional), 6.6 (not 6.6.3)

Algorithms

- An algorithm is a clearly specified set of instructions a computer follows to solve a problem.
- An algorithm should be
  - correct
  - efficient: not use too much time or space
- Algorithm analysis: determining how much time and space a given algorithm will consume.

Algorithms

- Note that two very different algorithms can solve the same problem
  - bubble sort vs. quicksort
  - List insert in an array-based implementation vs. a linked-list-based implementation.
- How do we know which is faster (more efficient in time)?
- Why not just run both on same data and compare?

Algorithms

- Could measure the time each one takes to execute, but that is subject to various external factors
  - multitasking operating system
  - speed of computer
  - language solution is written in (compiler)
- Need a way to quantify the efficiency of an algorithm independently of execution platform, language, or compiler

Estimating execution time

- The amount of time it takes an algorithm to execute is a function of the input size.
- We use the number of statements executed (given a certain input size) as an approximation of the execution time.
- Count up statements executed for a program or algorithm as a function of the amount of data
  - For a list of length N, it may require executing $3N^2+2N+125$ statements to sort it using a given algorithm.

Counting statements

- Each single statement (assignment, output) counts as 1 statement
- A boolean expression (in an if stmt or loop) is 1 statement
- A function call is equal to the number of statements executed by the function.
- A loop is basically the number of times the loop executes times the number of statements executed in the loop.
  - usually counted in terms of N, the input size.

Counting statements example

```
int total(int[] values, int numValues)
{  int result = 0;
   for(int i = 0; i < numValues; i++)
      result += values[i];
   return result;
}
```

- What does N (input size) represent in this case?
  - the number of values in the array (==numValues)
- Tally up the statement count:
  - int result = 0; (1)
  - int i=0; (1)
  - i < numValues (N+1)
  - i++ (N)
  - result += values[i]; (N)
  - return result; (1)

Total = 3N + 4

Comparing functions

- Is 3N+4 good? Is it better (less) than
  - 5N+5 ?
  - N+1,000 ?
  - $N^2 + N + 2$?

  for all values of N?
- Hard to say without graphing them.
- Even then, are the differences significant?

Comparing functions

- When comparing these functions in algorithm analysis
  - We are concerned with very large values of N.
  - We tend to ignore all but the “dominant” term.

    At large values of N, 3N dominates the 4 in 3N+4

  - We also tend to ignore the constant factor (3).
- We want to know which function is growing faster (getting bigger for bigger values of N).

Function classifications

- Constant $f(x)=b$ $O(1)$
- Logarithmic $f(x)=\log_b(x)$ $O(\log n)$
- Linear $f(x)=ax+b$ $O(n)$
- Linearithmic $f(x)=x \log_b(x)$ $O(n \log n)$
- Quadratic $f(x)=ax^2+bx+c$ $O(n^2)$
- Exponential $f(x)=b^x$ $O(2^n)$

Last column is “big Oh” notation

Classifications of (math) functions

Constant	$f(x)=b$	$O(1)$
Logarithmic	$f(x)=\log_b(x)$	$O(\log n)$
Linear	$f(x)=ax+b$	$O(n)$
Linearithmic	$f(x)=x \log_b(x)$	$O(n \log n)$
Quadratic	$f(x)=ax^2+bx+c$	$O(n^2)$
Exponential	$f(x)=b^x$	$O(2^n)$

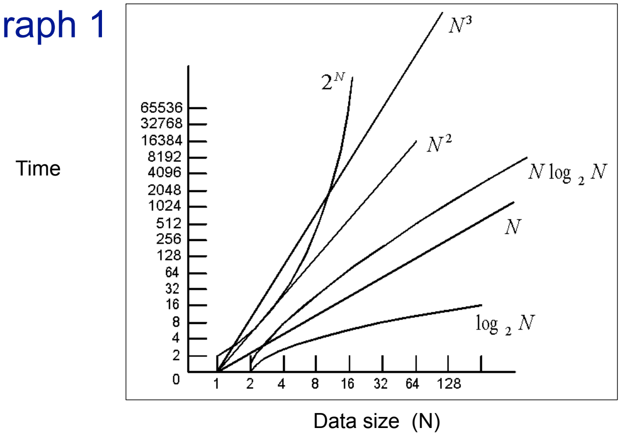
- Last column is “big Oh notation”, used in CS.

Comparing functions

- For a given function expressing the time it takes to execute a given algorithm in terms of N,
  - we ignore all but the dominant term and put it in one of the function classifications.
- Which classifications are more efficient?.
  - The ones that grow more slowly.

Comparing functions

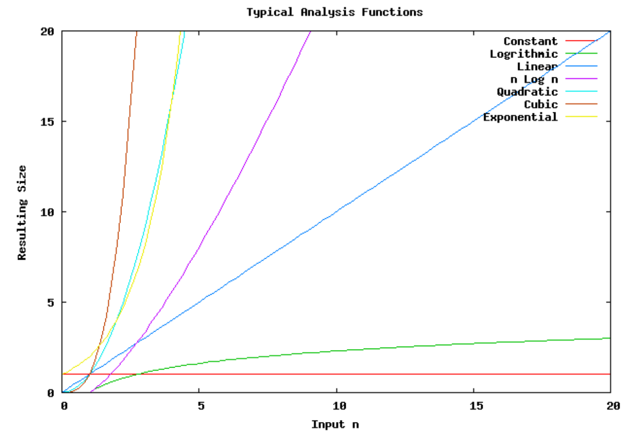
- Graph 1



We want small Time value for large N values

Comparing functions

- Graph 2



Comparing functions

- Assume N is 100,000, processing speed is 1,000,000,000 operations per second

Function	Running Time
2^N	3.2×10^{30086} years
N^4	3171 years
N^3	11.6 days
N^2	10 seconds
N log N	0.0017 seconds
N	0.0001 seconds
square root of N	3.2×10^{-7} seconds
log N	1.2×10^{-8} seconds

Formal Definition of Big O

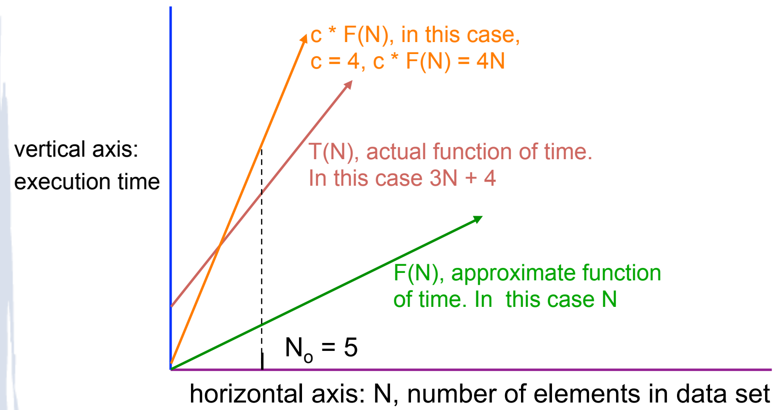
"Order F of N"

- T(N) is O(F(N)) if there are positive constants c and N_0 such that T(N) <= cF(N) when N >= N_0
  - N is the size of the data set the algorithm works on
  - T(N) is the function that characterizes the actual running time of the algorithm (like 3N+4)
  - F(N) is a function that characterizes an upper bounds on T(N). It is a limit on the running time of the algorithm. (The typical Big O functions)
  - c and N_0 are constants. We pick them to make the definition work.

Example using definition

- Given $T(N) = 3N + 4$, prove it is $O(N)$.
  - $F(N)$ in the definition is N
  - We need to choose constants c and N_0 to make $T(N) <= cF(N)$ when $N >= N_0$ true.
  - Lets try $c = 4$ and $N_0 = 5$.
  - Graph on next slide shows: 3N+4 is less than 4N whenever N is bigger than 5

Demonstrating 3N+4 is O(N)



Big O is an upper bound
(any upper bound)

- If $T(N)$ is $O(F(N))$, then for big values of N, $cF(N)$ is greater than $T(N)$.
  - If $T(N)$ is $O(n)$, it is also $O(n^2)$.
  - If $T(N)$ is $O(n)$, it is also $O(n^3)$.
  - If $T(N)$ is $O(n)$, it is also $O(n \log n)$.
- But these statements are not very informative.
  - If $T(N)$ is $O(n)$, it may or may not be $O(1)$.
- We prefer to state the tightest bound possible when analyzing algorithms.

Best, Average, Worst case analyses

Because certain data values may affect execution time (still functions of N):

- Best case: fewest possible statements executed considering all possible cases of input of size N
- Average case: average number of statements executed for all possible cases of input of size N
- Worst case: maximum number of statements that could be executed for all possible cases of input of size N

Consider examples 1 and 2 that follow:

Example 1:

```
bool findNum(double[] values, int numValues, double num)
{
   int found = false;
   for(int i = 0; i < numValues; i++)
      if(values[i] == num)
         found = true;
   return found;
}
```

- T(N) is O(F(N)) for what function F?
  - best case?
  - average case?
  - worst case?

Note: all have the same efficiency function for this code

Example 2:

```
bool findNum(double[] values, int numValues, double num)
{
   for(int i = 0; i < numValues; i++)
      if(values[i] == num)
         return true;
   return false;
}
```

- T(N) is O(F(N)) for what function F?
  - best case?
  - average case of all inputs that return true?
  - worst case?

Example 3:

```
Matrix Matrix::add(Matrix rhs)
{  Matrix sum = new Matrix(numRows(), numCols(), 0);
   for(int row = 0; row < numRows(); row++)
      for(int col = 0; col < numCols(); col++)
         sum.myMatrix[row][col] = myMatrix[row][col]
             + rhs.myMatrix[row][col];
   return sum;
}
```

- T(N) is O(F(N)) for what function F?

Is there a best case or worst case?

Example 4:

```
public void selectionSort(double[] data, int numValues)
{  int n = numValues;
   int min;
   double temp;
   for(int i = 0; i < n; i++)
   {  min = i;
      for(int j = i+1; j < n; j++)
          if(data[j] < data[min])
             min = j;
      temp = data[i];
      data[i] = data[min];
      data[min] = temp;
   }// end of outer loop, i
}
```

Note: 1+2+3+...+N = N*(N+1)/2

- T(N) is O(F(N)) for what function F?

Example 5:

```
public int func(int[] list, int length){
  int total = 0;
  for(int i = 0; i < length; i++){
     total += countDups(list[i], list);
  }
  return total;
}
// method countDups is O(N) where N is the
// length of the array it is passed
```

- T(N) is O(F(N)) for what function F?

Example 6:

- Insert (and remove) for List_3358
  - implemented using arrays (in class: see below)
  - implemented using linked lists (see PA#2 solution)
- These operations are O(__) ?

```
void List_3358::remove() {
   assert(!atEOL() && !isEmpty());
   for (int i=cursor; i < currentSize-1; i++)
      values[i] = values[i+1];
   currentSize--;
   if (isEmpty())
      cursor = EOL;
}
```