Introduction to Algorithms
Analysis

CSE 680

Prof. Roger Crawfis

Run-time Analysis

- Depends on
  - input size
  - input quality (partially ordered)
- Kinds of analysis
  - Worst case (standard)
  - Average case (sometimes)
  - Best case (never)

What do we mean by Analysis?

- Analysis is performed with respect to a computational model
- We will usually use a generic uniprocessor random-access machine (RAM)
  - All memory is equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls
  - Constant word size
    - Unless we are explicitly manipulating bits

Example: Searching

- Assume we have a sorted array of integers, X[1..N] and we are searching for “key”

```
                                    Cost    Times
found = 0;                          C0      1
i  = 0;                             C1      1
while (!found && i < N)  {          C2      0 <= L < N
      if (key == X[i])              C3      1 <= L <= N
         found = 1;                 ?       ?
      i++;                          C5      1 <= L <= N
}
```

T(n) = C0 + C1 + L*(C2 + C3 + C4), where 1 <= L <= N is the number of times that the loop is iterated.

Example: Searching

- What's the best case? Loop iterates just once =>
  - T(n) = C0 + C1 + C2 + C3 + C4
- What's the average (expected) case? Loop iterates N/2 times =>
  - T(n) = C0 + C1 + N/2 * (C2 + C3 + C4)
  - Notice that this can be written as T(n) = a + b*n where a, b are constants
- What's the worst case? Loop iterates N times =>
  - T(n) = C0 + C1 + N * (C2 + C3 + C4)
  - Notice that this can be written as T(n) = a + b*n where a, b are constants

Worst Case Analysis

- We will only look at WORST CASE running time of an algorithm. Why?
  - Worst case is an upper bound on the running time. It gives us a guarantee that the algorithm will never take any longer
  - For some algorithms, the worst case happens fairly often. As in this search example, the searched item is typically not in the array, so the loop will iterate N times
  - The "average case" is often roughly as bad as the "worst case". In our search algorithm, both the average case and the worst case are linear functions of the input size "n"

Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
      }
      A[j+1] = key
  }
}
```

How many times will this loop execute?

Insertion Sort

Statement	Cost	times
`InsertionSort(A, n) {`		
`  for i = 2 to n {`	c_1	n
`      key = A[i]`	c_2	n-1
`      j = i - 1;`	c_4	n-1
`      while (j > 0) and (A[j] > key) {`	c_5	$\sum_{j=2}^{n} t_j$
`          A[j+1] = A[j]`	c_6	$\sum_{j=2}^{n} (t_j - 1)$
`          j = j - 1`	c_7	$\sum_{j=2}^{n} (t_j - 1)$
`      }`	0	
`      A[j+1] = key`	c_8	n-1
`  }`	0	
`}`		

Analyzing Insertion Sort

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- What can T(n) be?
  - Best case -- inner loop body never executed
    - t_i = 1 ➔ T(n) is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - t_i = i ➔ T(n) is a quadratic function
  - Average case
    - ???

So, Is Insertion Sort Good?

- Criteria for selecting algorithms
  - Correctness
  - Amount of work done
  - Amount of space used
  - Simplicity, clarity, maintainability
  - Optimality

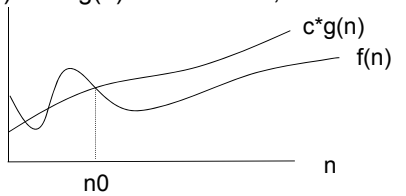
Asymptotic Notation

- We will study the asymptotic efficiency of algorithms
  - To do so, we look at input sizes large enough to make only the order of growth of the running time relevant
  - That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit as the size of the input increases without bound.
  - Usually an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.
    - Real-time systems, games, interactive applications need to limit the input size to sustain their performance.
- 3 asymptotic notations
  - Big O, Θ, Ω Notations

Big-Oh Notation: Asymptotic Upper Bound

Want g(n) to be simple.

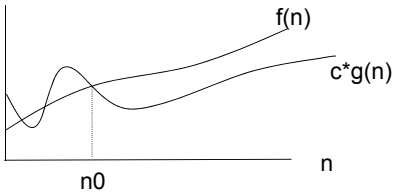
- **T(n) = f(n) = O(g(n))**
  - if f(n) <= c*g(n) for all n > n0, where c & n0 are constants > 0



- Example: T(n) = 2n + 5 is O(n). Why?
  - 2n+5 <= 3n, for all n >= 5
- T(n) = 5*n^2 + 3*n + 15 is O(n^2). Why?
  - 5*n^2 + 3*n + 15 <= 6*n^2, for all n >= 6

Ω Notation: Asymptotic Lower Bound

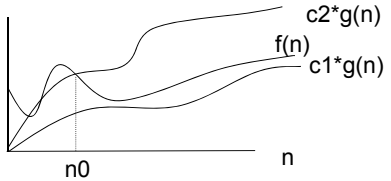
- **T(n) = f(n) = Ω(g(n))**
  - if f(n) >= c*g(n) for all n > n0, where c and n0 are constants > 0



- Example: T(n) = 2n + 5 is Ω(n). Why?
  - 2n+5 >= 2n, for all n > 0
- T(n) = 5*n^2 - 3*n is Ω(n^2). Why?
  - 5*n^2 - 3*n >= 4*n^2, for all n >= 4

Θ Notation: Asymptotic Tight Bound

- T(n) = f(n) = Θ(g(n))
  - if c1*g(n) <= f(n) <= c2*g(n) for all n > n0, where c1, c2 and n0 are constants > 0



- Example: T(n) = 2n + 5 is Θ(n). Why?
  - 2n <= 2n+5 <= 3n, for all n >= 5
- T(n) = 5*n^2 - 3*n is Θ(n^2). Why?
  - 4*n^2 <= 5*n^2 - 3*n <= 5*n^2, for all n >= 4

Big-Oh, Theta, Omega

Tips to guide your intuition:

- Think of O(f(N)) as "greater than or equal to" f(N)
  - Upper bound: "grows slower than or same rate as" f(N)
- Think of Ω(f(N)) as "less than or equal to" f(N)
  - Lower bound: "grows faster than or same rate as" f(N)
- Think of Θ(f(N)) as "equal to" f(N)
  - "Tight" bound: same growth rate

(*True for large N and ignoring constant factors*)

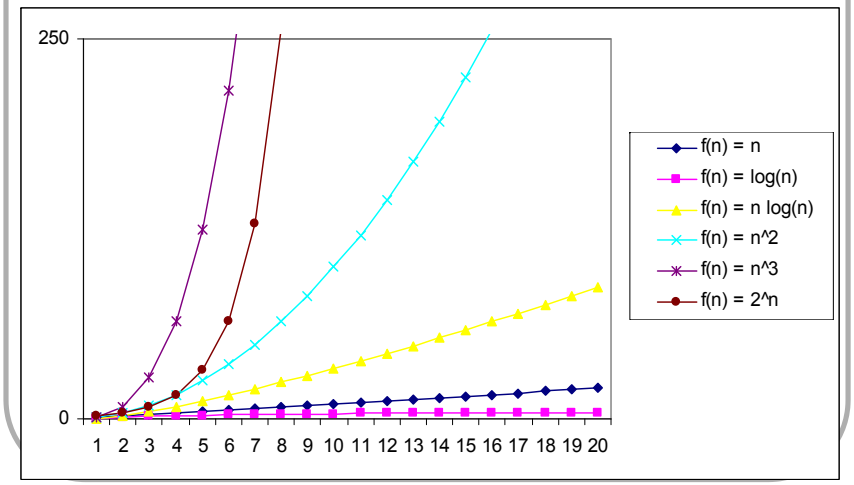
Common Functions

Increasing cost

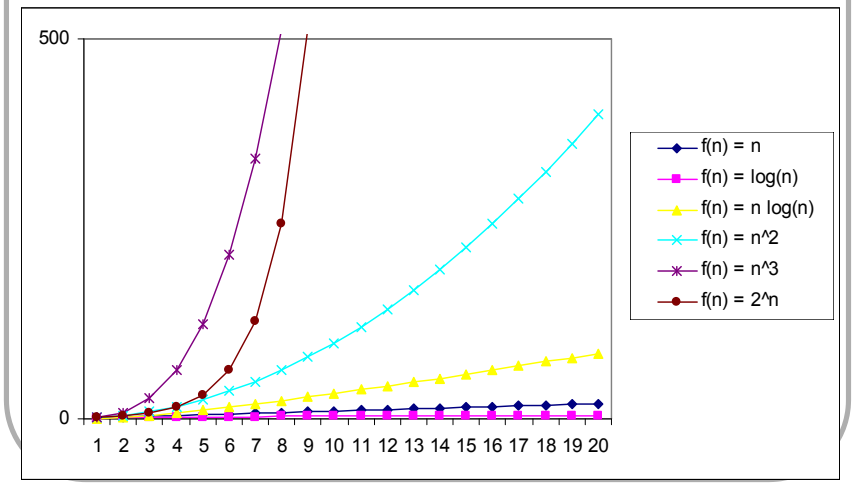
Name	Big-Oh	Comment
Constant	O(1)	Can't beat it!
Log log	O(loglogN)	Extrapolation search
Logarithmic	O(logN)	Typical time for good searching algorithms
Linear	O(N)	This is about the fastest that an algorithm can run given that we need O(n) just to read the input
N logN	O(NlogN)	Most sorting algorithms
Quadratic	O(N^2)	Acceptable when the data size is small (N<1000)
Cubic	O(N^3)	Acceptable when the data size is small (N<1000)
Exponential	O(2^N)	Only good for really small input sizes (n<=20)

Polynomial time

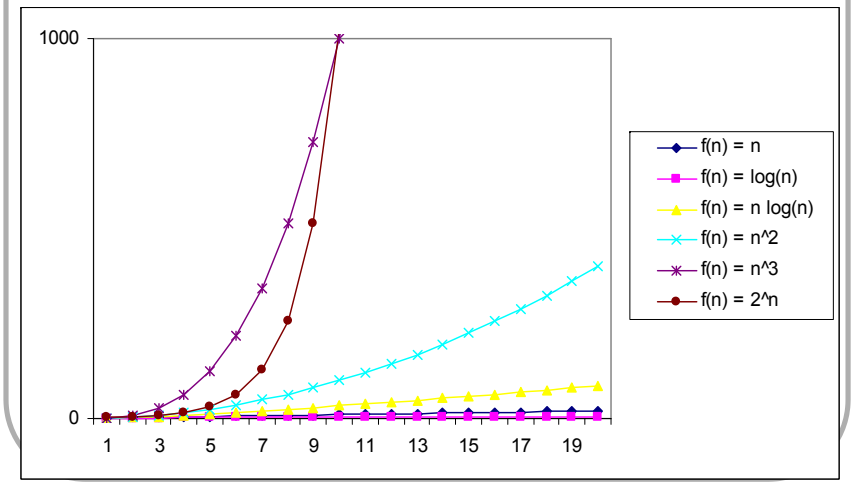
Asymptotic Complexity



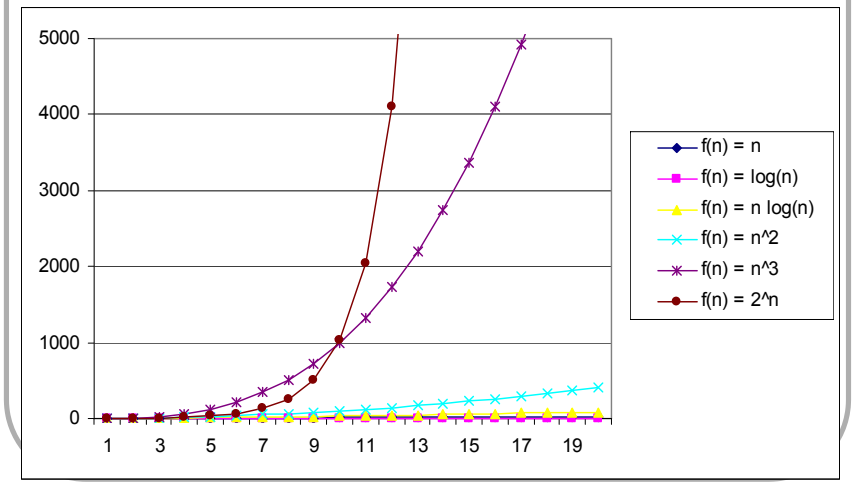
Asymptotic Complexity



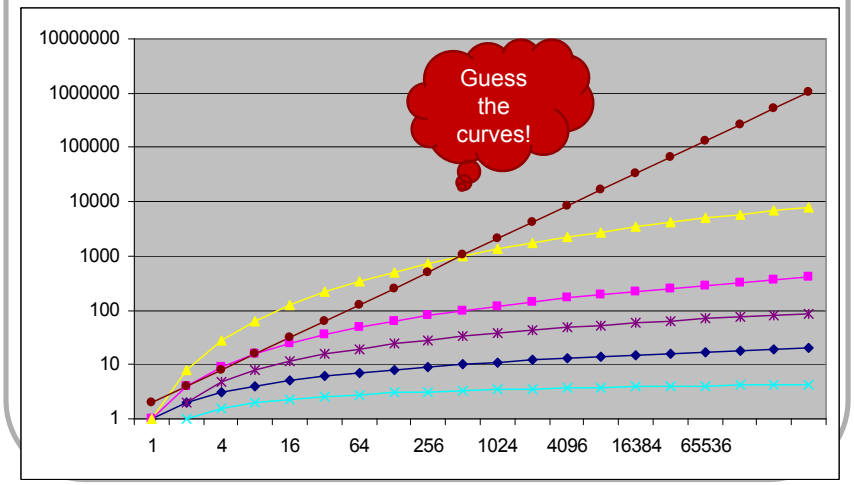
Asymptotic Complexity



Asymptotic Complexity



Asymptotic Complexity



Math Review

- S(N) = 1 + 2 + 3 + 4 + … N = $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$
- Sum of Squares:

$$\sum_{i=1}^{N} i^2 = \frac{N*(N+1)*(2n+1)}{6} \approx \frac{N^3}{3}$$

- Geometric Series:

$$\sum_{i=0}^{N} A^i = \frac{1-A^{N+1}}{1-A} = \Theta(1) \qquad A > 1$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1} \qquad A < 1$$

Math Review

- Linear Geometric Series:

$$\sum_{i=0}^{n} ix^i = x + 2x^2 + 3x^3 + \ldots + nx^n = \frac{(n-1)x^{(n+1)} - nx^n + x}{(x-1)^2}$$

- Harmonic Series:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = (\ln n) + O(1)$$

Math Review

- Logarithms:

$$\log A^B = B * \log A$$

$$\log(A * B) = \log A + \log B$$

$$\log\left(\frac{A}{B}\right) = \log A - \log B$$

Math Review

- Summations with general bounds:

$$\sum_{i=a}^{b} f(i) = \sum_{i=0}^{b} f(i) - \sum_{i=0}^{a-1} f(i)$$

- Linearity of Summations:

$$\sum_{i=1}^{n} (4i^2 - 6i) = 4\sum_{i=1}^{n} i^2 - 6\sum_{i=1}^{n} i$$

Review: Induction

- Suppose
  - S(k) is true for fixed constant k
    - Often k = 0
  - S(n) → S(n+1) for all n >= k
- Then S(n) is true for all n >= k

Proof By Induction

- Claim:S(n) is true for all n >= k
- Basis:
  - Show formula is true when n = k
- Inductive hypothesis:
  - Assume formula is true for an arbitrary n
- Step:
  - Show that formula is then true for n+1

Induction Example: Gaussian Closed Form

- Prove 1 + 2 + 3 + … + n = n(n+1) / 2
  - Basis:
    - If n = 0, then 0 = 0(0+1) / 2
  - Inductive hypothesis:
    - Assume 1 + 2 + 3 + … + n = n(n+1) / 2
  - Step (show true for n+1):

    1 + 2 + … + n + n+1 = (1 + 2 + …+ n) + (n+1)

    = n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2

    = (n+1)(n+2)/2 = (n+1)(n+1 + 1) / 2

Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$

    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - Inductive hypothesis:
    - Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$
  - Step (show true for n+1):

    $a^0 + a^1 + \ldots + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$

    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$

Induction

- We've been using *weak induction*
- *Strong induction* also holds
  - Basis: show S(0)
  - Hypothesis: assume S(k) holds for arbitrary k <= n
  - Step: Show S(n+1) follows
- Another variation:
  - Basis: show S(0), S(1)
  - Hypothesis: assume S(n) and S(n+1) are true
  - Step: show S(n+2) follows

Sample "for" loops

```
function func(n)
1. x ← 0;
2. for i ← 1 to n do
3.     for j ← 1 to n do
4.         x ← x + (i - j);
5. return(x);
```

Sample "for" loops

```
function func(n)
1. x ← 0;
2. for i ← 1 to n do
3.     for j ← 1 to i do
4.         x ← x + (i - j);
5. return(x);
```

Sample “for” loops

```
function func(n)
1. x ← 0;
2. for i ← 1 to n do
3.     for j ← i to n do
4.         x ← x + (i - j);
5. return(x);
```

Sample “for” loops

```
function func(n)
1. x ← 0;
2. for i ← 1 to n do
3.     for j ← 1 to ⌊√n⌋ do
4.         x ← x + (i - j);
5. return(x);
```

Sample “for” loops

```
function func(n)
1. x ← 0;
2. for i ← 1 to n do
3.     for j ← 1 to ⌊√i⌋ do
4.         x ← x + (i - j);
5. return(x);
```

Sample “while” loops

```
function func(n)
1. x ← 0;
2. i ← 0;
3. while i < n do
4.     x ← x + 1;
5.     i ← i + 1;
6. return(x);
```

Sample “while” loops

```
function func(n)
1. x ← 0;
2. i ← 3;
3. while i < n do
4.     x ← x + 1;
5.     i ← i + 1;
6. return(x);
```

Sample “while” loops

```
function func(n)
1. x ← 0;
2. i ← 0;
3. while i < n do
4.     x ← x + 1;
5.     i ← i + 3;
6. return(x);
```

Sample “while” loops

```
function func(n)
1. x ← 0;
2. i ← 1;
3. while i < n do
4.     x ← x + 1;
5.     i ← i  * 2;
6. return(x);
```

Sample “while” loops

```
function func(n)
1. x ← 0;
2. i ← 1;
3. while i < n do
4.     x ← x + 1;
5.     i ← i  * 3;
6. return(x);
```