Chapter 2
Algorithm Analysis

Introduction

- algorithm
  - set of simple instructions to solve a problem
  - analyzed in terms, such as time and memory, required
    - too long (minutes, hours, years) – no good
    - too much memory (terabytes) – no good

- we can estimate resource usage using a formal mathematical framework to count basic operations
  - big-O $O(f(N))$
  - big-omega $\Omega(f(N))$
  - big-theta $\Theta(f(N))$

Asymptotic Notation

- formal definitions

$$f(n) = O\big(g(n)\big)$$
$$f(n) = \Omega\big(g(n)\big)$$
$$f(n) = \Theta(g(n))$$

$f(n)$ and $g(n)$ are positive for $n > 1$

when we say that X is true when n is sufficiently large, we mean there exists N such that X is true for all $n > N$

we are comparing relative rates of growth

Asymptotic Notation: Big-O

$f(n) = O(g(n))$ if there exist constants c and N such that

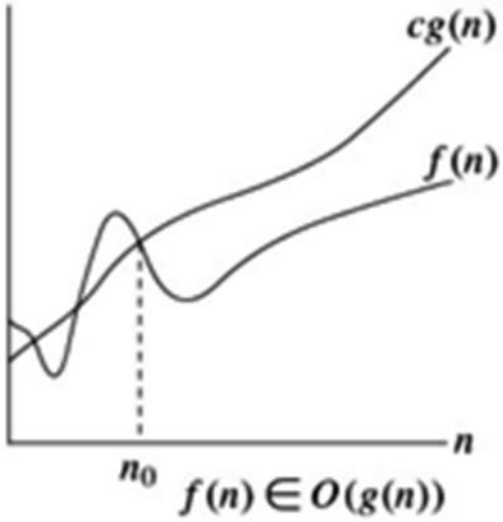
$$f(n) \leq c \cdot g(n) \text{ for all } n > N$$

loosely speaking, $f(n) = O\big(g(n)\big)$ is an analog of $f \leq g$

example: if $f(n) = 54n^2 + 42n$ and
$g(n) = n^2$ then $f = O(g)$ since

$$f(n) \leq 55g(n)$$

for all $n > 42$

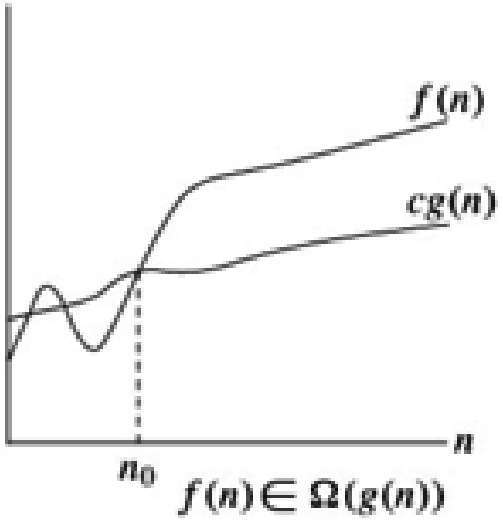


Asymptotic Notation: Ω

$f(n) = \Omega(g(n))$ if there exist constants c and N such that

$$f(n) \geq c \cdot g(n) \text{ for all } n > N$$

loosely speaking, $f(n) = \Omega\big(g(n)\big)$ is an analog of $f \geq g$

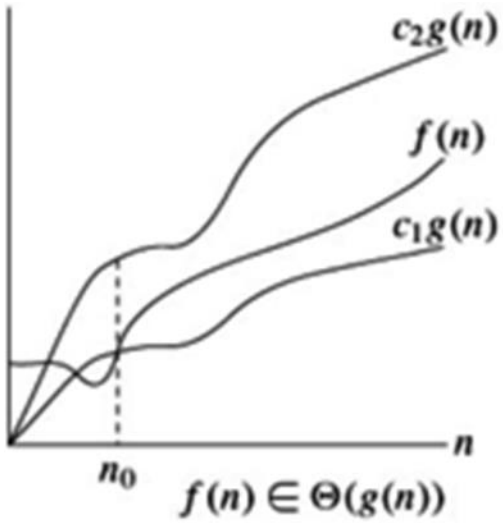


Asymptotic Notation: Θ

$f(n) = \Theta(g(n))$ if there exist constants c_1, c_2, and N such that

$$c_1 \cdot g(n) < f(n) < c_2 \cdot g(n) \text{ for all } n > N$$

loosely speaking, $f(n) = \Theta\big(g(n)\big)$ is an analog of $\underline{f \approx g}$



Relationship of O, Ω, and Θ

- note that

$$f(n) = O\big(g(n)\big) \Leftrightarrow g(n) = \underline{\Omega(f(n))}$$

since

$$f(n) < cg(n) \Leftrightarrow g(n) > \frac{1}{c} f(n)$$

- also

$$f(n) = \Theta\big(g(n)\big) \Leftrightarrow f(n) = \underline{O\big(g(n)\big)} \text{ and } f(n) = \underline{\Omega\big(g(n)\big)}$$

since $f(n) = \Theta(g(n))$ means that there exist c_1, c_2, and N such that

$$c_1 \cdot g(n) < f(n) < c_2 \cdot g(n) \text{ for all } n > N$$

Asymptotic Notation: First Observation

- for sufficiently large n,

$f(n) = O(g(n))$ gives an upper bound on f in terms of g

$f(n) = \Omega(g(n))$ gives a lower bound on f in terms of g

$f(n) = \Theta(g(n))$ gives both an upper and lower bound on f in terms of g

Asymptotic Notation: Second Observation

- it is the existence of constants that matters for the purposes of asymptotic complexity analysis – not the exact values of the constants
  - example: all of the following imply $n^2 + n = O(n^2)$:

$n^2 + n < 4n^2$ for all $n > 1$

$n^2 + n < 2n^2$ for all $n > 1$

$n^2 + n < 1.01n^2$ for all $n > 100$

$n^2 + n < 1.000001n^2$ for all $n > 1000$

Asymptotic Notation: Third Observation

- we typically ignore constant factors and lower-order terms, since all we want to capture is growth trends
  - example:

$$f(n) = 1{,}000n^2 + 10{,}000n + 42$$

$$\Rightarrow f(n) = O(n^2)$$

but the following are discouraged:

$f(n) = O(1{,}000n^2)$	true, but bad form
$f(n) = O(n^2 + n)$	true, but bad form
$f(n) < O(n^2)$	< built into O

Asymptotic Notation: Fourth Observation

$O(g(n))$ assures that you can't do any worse than $g(n)$, but it can be unduly pessimistic

$\Omega(g(n))$ assures that you can't do any better than $g(n)$, but it can be unduly optimistic

$\Theta(g(n))$ is the strongest statement: you can't do any better than $g(n)$, but you can't do any worse

A Shortcoming of O

an $f(n) = O(g(n))$ bound can be misleading

example: $n = O(n^2)$, since $n < n^2$ for all $n > 1$

however, for large n, $n \ll n^2$ – the functions $f(n) = n$ and $g(n) = n^2$ are nothing alike for large n

we don't write something like $f(n) = O(n^2)$ if we know, say, that $f(n) = O(n)$

upper bounds should be as small as possible!

A Shortcoming of Ω

an $f(n) = \Omega(g(n))$ bound can be misleading

example: $n^2 = \Omega(n)$, since $n^2 > n$ for all $n > 1$

however, for large n, $n^2 \gg n$ – the functions $f(n) = n$ and $g(n) = n^2$ are nothing alike for large n

we don't write something like $f(n) = \Omega(n)$ if we know, say, that $f(n) = \Omega(n^2)$

lower bounds should be as large as possible!

Θ is the Most Informative

a $f(n) = \Theta(g(n))$ relationship is the most informative:

$$c_1 \cdot g(n) < f(n) < c_2 \cdot g(n) \text{ for all } n > N$$

example: $2n^2 + n = \Theta(n^2)$

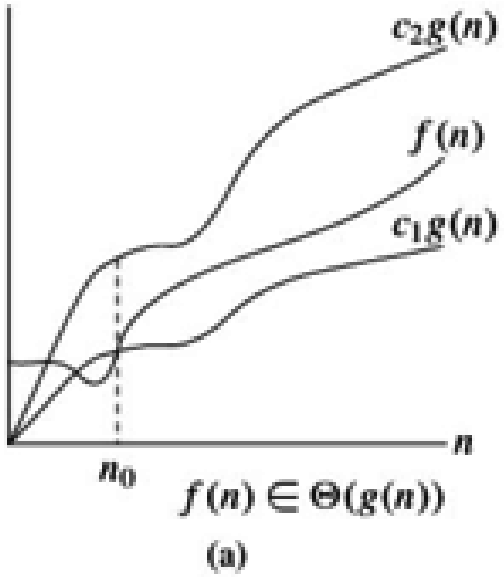
$$2n^2 < 2n^2 + n < 3n^2$$

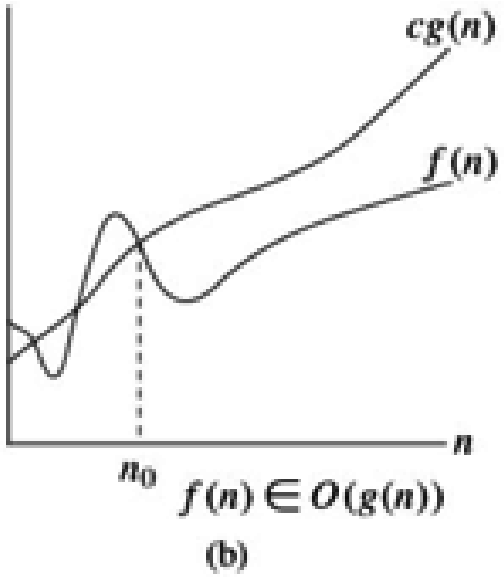
for all $n > 1$

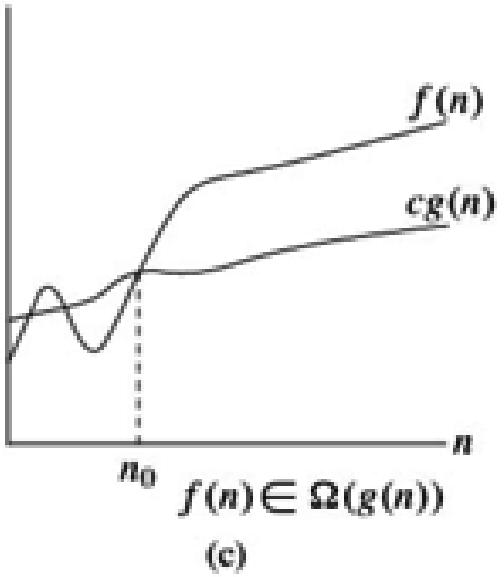
for large values of n, the ratio $(2n^2 + n)/n^2$ tends to 2;
in this sense,

$$2n^2 + n \approx 2n^2$$

Typical Growth Rates







https://expcode.wordpress.com/2015/07/19/big-o-big-omega-and-big-theta-notation/

Additional Rules

Rule 1

if $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ then

(a) $T_1(n) + T_2(n) = O(f(n) + g(n))$ (or just $\underline{O(\max(f(n), g(n)))}$)

(b) $T_1(n) * T_2(n) = O(f(n) * g(n))$

Rule 2

if $T(n)$ is a polynomial of degree k, then $T(n) = \underline{\Theta(n^k)}$

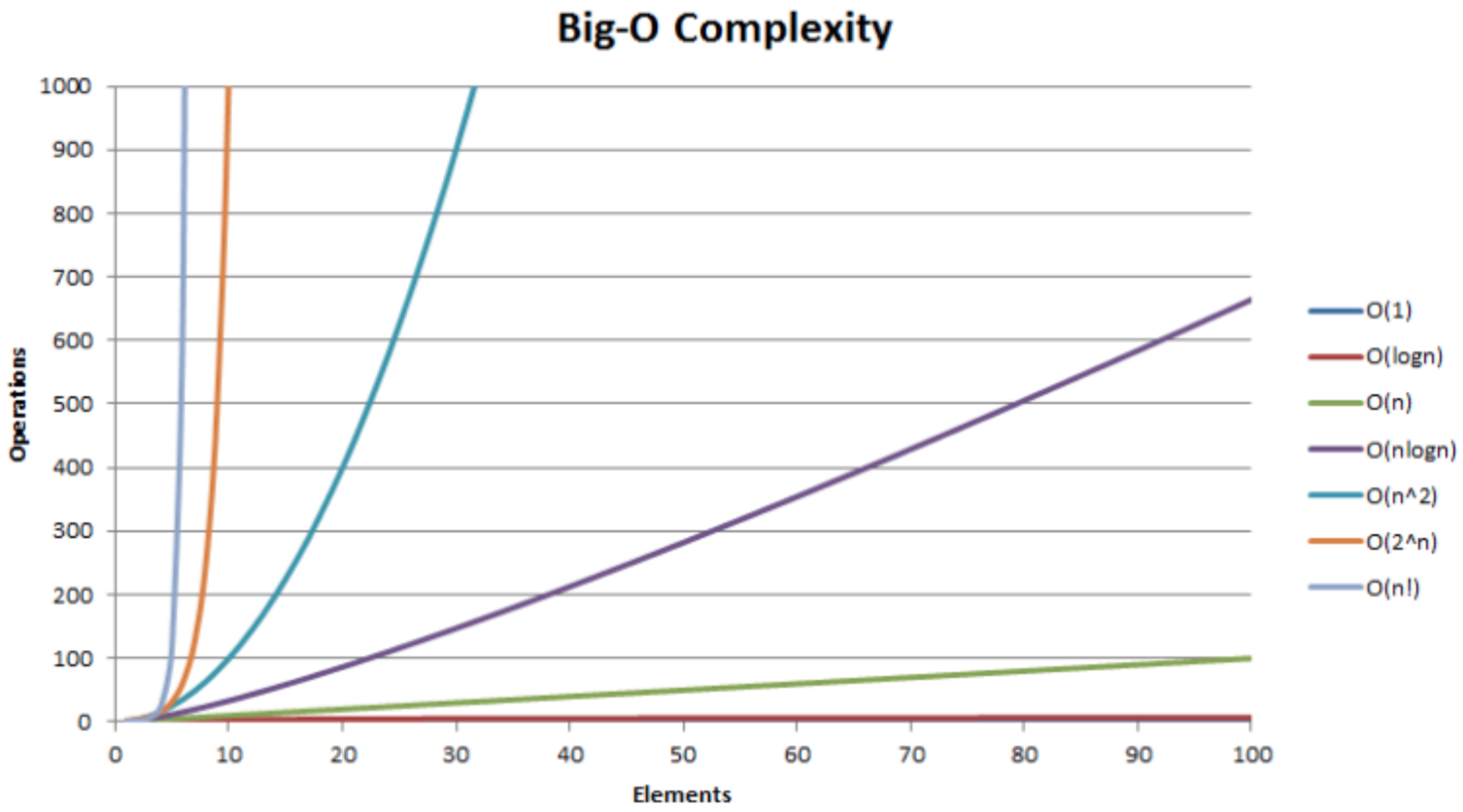
Rule 3

$\log^k n = \underline{O(n)}$ for any constant k (logarithms grow very slowly)

Typical Growth Rates

Function	Name
c	constant
$\log n$	logarithmic
n	linear
$n \log n$	
n^2	quadratic
n^3	cubic
2^n	exponential

Typical Growth Rates



http://www.cosc.canterbury.ac.nz/cs4hs/2015/files/complexity-tractability-julie-mcmahon.pdf

Complexity and Tractability

	$T(n)$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01ms	.03ms	.1ms	1ms	10ms	10s	1ms
20	.02ms	.09ms	.4ms	8ms	160ms	2.84h	1ms
30	.03ms	.15ms	.9ms	27ms	810ms	6.83d	1s
40	.04ms	.21ms	1.6ms	64ms	2.56ms	121d	18m
50	.05ms	.28ms	2.5ms	125ms	6.25ms	3.1y	13d
100	.1ms	.66ms	10ms	1ms	100ms	3171y	4×10^{13}y
10^3	1ms	9.96ms	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10ms	130ms	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100ms	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^{7}y	3.17×10^{43}y	

assume computer speed of 1 billion ops/sec

Order Comparison

the order of an algorithm is useful for comparison purposes

example

- an algorithm takes 10ms for input size 60
  - how long will it take for input size 600 if the order of the algorithm is linear?

$$\frac{10\text{ ms}}{x} = \frac{60}{600} \qquad 60x = 6000\text{ ms} \qquad x = 100\text{ ms}$$

  - quadratic?

$$\frac{10\text{ ms}}{x} = \frac{60^2}{600^2} \qquad \frac{10\text{ ms}}{x} = \left(\frac{60}{600}\right)^2 \qquad \frac{10\text{ ms}}{x} = \frac{1}{10^2} \qquad x = 1000\text{ ms}$$

Order Comparison

example (cont.)

- an algorithm takes 10ms for input size 60
  - how large a problem can be solved in 1s if the order of the algorithm is linear?
  - quadratic?

What to Analyze

in order to analyze algorithms, we need a model of computation

- standard computer with sequential instructions
- each operation (+, -, *, /, =, etc.) takes one time unit, even memory accesses
- infinite memory
- obviously unrealistic

Complex Arithmetic Operations

- examples: x^n $\log_2 x$ $\sqrt{x}$
  - count the number of basic operations required to execute the complex arithmetic operations
  - for instance, we could compute x^n as

$$x^n \leftarrow x * x * \cdots * x * x$$

there are $n - 1$ multiplications, so there are $n - 1$ basic operations to compute x^n plus one more basic operation to assign or return the result

What to Analyze

different types of performance are analyzed

- best case
  - may not represent typical behavior
- average case
  - often reflects typical behavior
  - difficult to determine what is average
  - difficult to compute
- worst case
  - guarantee on performance for any input
  - typically considered most important
- should be implementation-independent

Maximum Subsequence Sum Problem

example: Maximum Subsequence Sum Problem

- given (possibly negative) integers, A_1, A_2, …, A_N, find the maximum value of

$$\sum_{k=i}^{j} A_k$$

- e.g, for input -2, 11, -4, 13, -5, -2, the answer is 20 (A_2 through A_4)

Maximum Subsequence Sum Problem

many algorithms to solve this problem

- we will focus on four with run time in the table below

Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 10$	0.000009	0.000004	0.000006	0.000003
$N = 100$	0.002580	0.000109	0.000045	0.000006
$N = 1{,}000$	2.281013	0.010203	0.000485	0.000031
$N = 10{,}000$	NA	1.2329	0.005712	0.000317
$N = 100{,}000$	NA	135	0.064618	0.003206

38 min

26 days

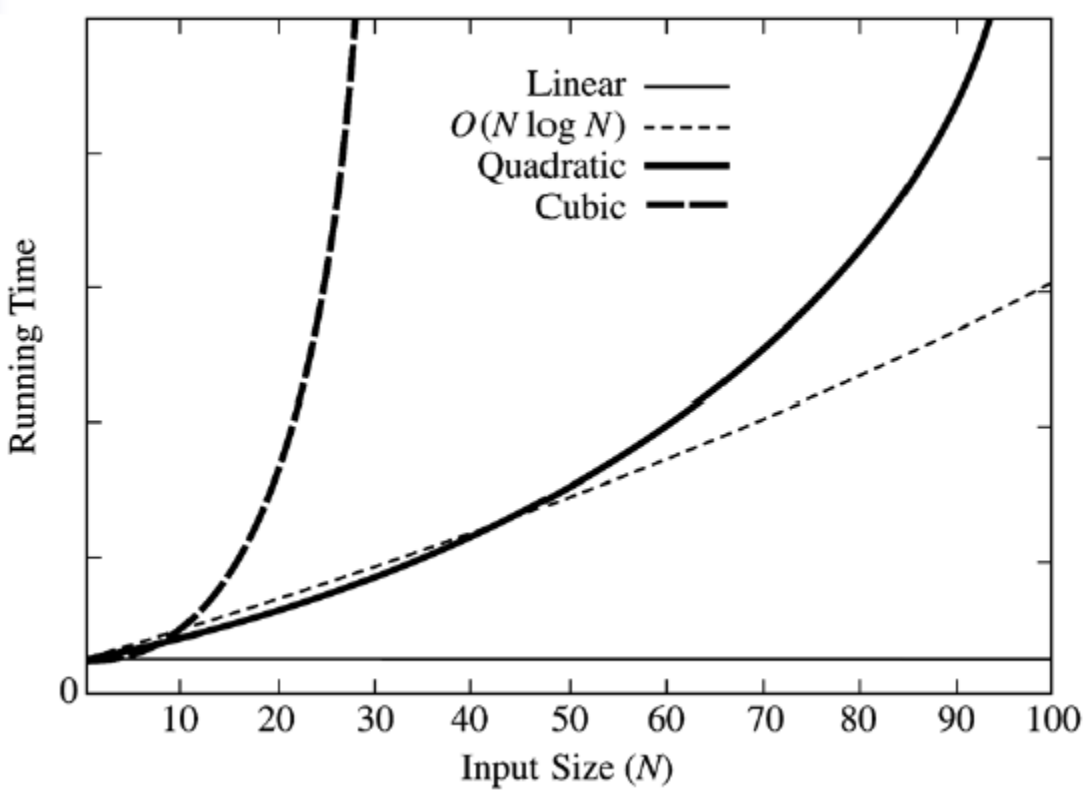
Do not include array read times.

Maximum Subsequence Sum Problem

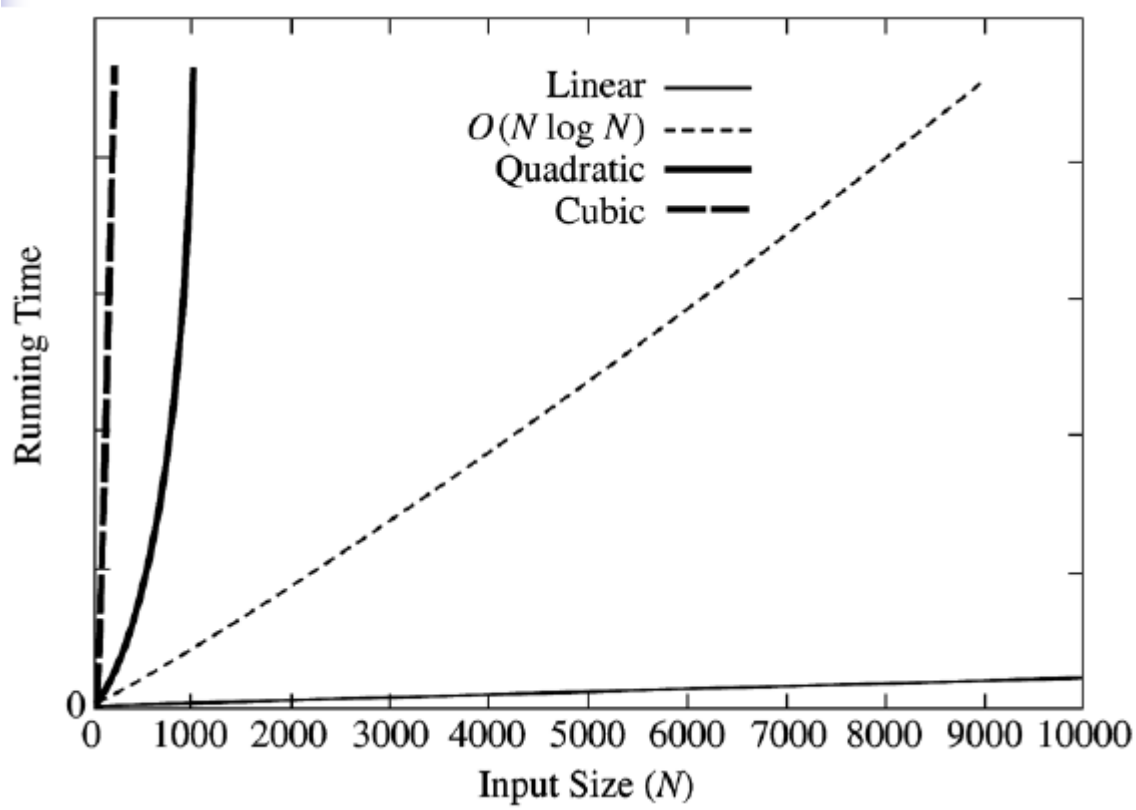
notes

- computer actually run on not important since we're looking to compare algorithms
- for small inputs, all algorithms perform well
- times do not include read time
  - algorithm 4: time to read would surpass time to compute
  - reading data is often the bottleneck
- for algorithm 4, the run time increases by a factor of 10 as the problem size increases by a factor of 10 (linear)
- algorithm 2 is quadratic; therefore an increase of input by a factor of 10 yields a run time factor of 100
- algorithm 1 (cubic): run time factor of 1000

Maximum Subsequence Sum Problem



Maximum Subsequence Sum Problem



Running Time Calculations

at least two ways to estimate the running time of a program

- empirically
  - as in previous results
  - realistic measurement
- analysis
  - helps to eliminate bad algorithms early when several algorithms are being considered
  - analysis itself provides insight into designing efficient algorithms
  - pinpoints bottlenecks, which need special coding

A Simple Example

calculate $\sum_{i=1}^{N} i^3$

```
    int sum( int n )
    {
        int partialSum;

1       partialSum = 0;
2       for( int i = 1; i <= n; ++i )
3           partialSum += i * i * i;
4       return partialSum;
    }
```

A Simple Example

analysis

- declarations count for no time; ignore call costs
- lines 1 and 4 count for 1 time unit each
- line 3 counts for 4 time units
  - 2 multiplications
  - 1 addition
  - 1 assignment
- line 2 has hidden costs
  - initializing i (1 time unit)
  - testing i $\leq$ n (N + 1)
  - incrementing i (N)
- total: $6N + 4$ or $O(N)$

A Simple Example

analysis

- too much work to analyze all programs like this
- since we're dealing with O, we can use shortcuts to help
  - line 3 counts for 4 time units, but is $O(1)$
  - line 1 is insignificant compared to the **`for`** loop

General Rules

concentrate on loops and recursive calls

- Rule 1 – FOR loops
  - running time of the statements inside (including tests) times the number of iterations
- Rule 2 – Nested loops
  - analyze from the inside out
  - running time of the statements times the product of the sizes of all the loops
- watch out for loops that
  - contribute only a constant amount of computation
  - contain break statements

Single Loops

```
for (int i = 0; i < n; i++)
    a[i]+= i;


for (int i = 5; i < n; i++)
    a[i]+= i;


for (int i = 0; i < n; i++)
{
    if (i >= 7) break;
    a[i]+= i;
}
```

Nested Loops

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        ++k;
```

- constant amount of computation inside nested loop
  - must multiply number of times both loops execute
  - $O(n^2)$

```
for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j)
        ++k;
```

- even though second loop is not executed as often, still $O(n^2)$

Nested Loops

```
for (int i = 0; i < 7; ++i)
    for (int j = 0; j < n; ++j)
        ++k;
```

- outer loop is executed a fixed number of times
  - $O(n)$

General Rules (cont.)

concentrate on loops and recursive calls

- Rule 3 – Consecutive statements
  - simply add the running times

```
for (i = 0; i < n; ++i)
    a[i] = 0
for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
        a[i] += a[j] + i + j
```

  - $O(n)$ work followed by $O(n^2)$ work, so $O(n^2)$

General Rules (cont.)

concentrate on loops and recursive calls

- Rule 4 – if/else

```
if (test)
    S1
else
    S2
```

  - total running time includes the test plus the larger of the running times of S1 and S2
  - could be overestimate in some cases, but never an underestimate

General Rules (cont.)

concentrate on loops and recursive calls

- function calls analyzed first
- recursion
  - if really just a for loop, not difficult (the following is $O(n)$)

```
long factorial (int n)
{
   if (n <= 1)
      return 1;
   else
      return n * factorial (n - 1);
}
```

General Rules (cont.)

previous example not a good use of recursion

- if recursion used properly, difficult to convert to a simple loop structure

what about this example?

```
long fib (int n)
{
   if (n <= 1)
      return 1;
   else
      return fib (n - 1) + fib (n - 2);
}
```

General Rules (cont.)

even worse!

- extremely inefficient, especially for values > 40
- analysis
  - for $N = 0$ or $N = 1$, $T(N)$ is constant: $T(0) = T(1) = 1$
  - since the line with recursive calls is not a simple operation, it must be handled differently

$$T(N) = T(N-1) + T(N-2) + 2 \quad \text{for } N \geq 2$$

  - we have seen that this algorithm $< \left(\frac{5}{3}\right)^N$
    - similarly, we could show it is $> \left(\frac{3}{2}\right)^N$
    - exponential!

General Rules (cont.)

with current recursion

- lots of redundant work
  - violating compound interest rule
- intermediate results thrown away

running time can be reduced substantially with simple for loop

Maximum Subsequence Sum Problem Revisited

example: Maximum Subsequence Sum Problem

- given (possibly negative) integers, A_1, A_2, …, A_N, find the maximum value of

$$\sum_{k=i}^{j} A_k$$

- e.g, for input -2, 11, -4, 13, -5, -2, the answer is 20 (A_2 through A_4)
- previously, we reviewed running time results for four algorithms

Maximum Subsequence Sum Problem Revisited

many algorithms to solve this problem

- we will focus on four with run time in the table below

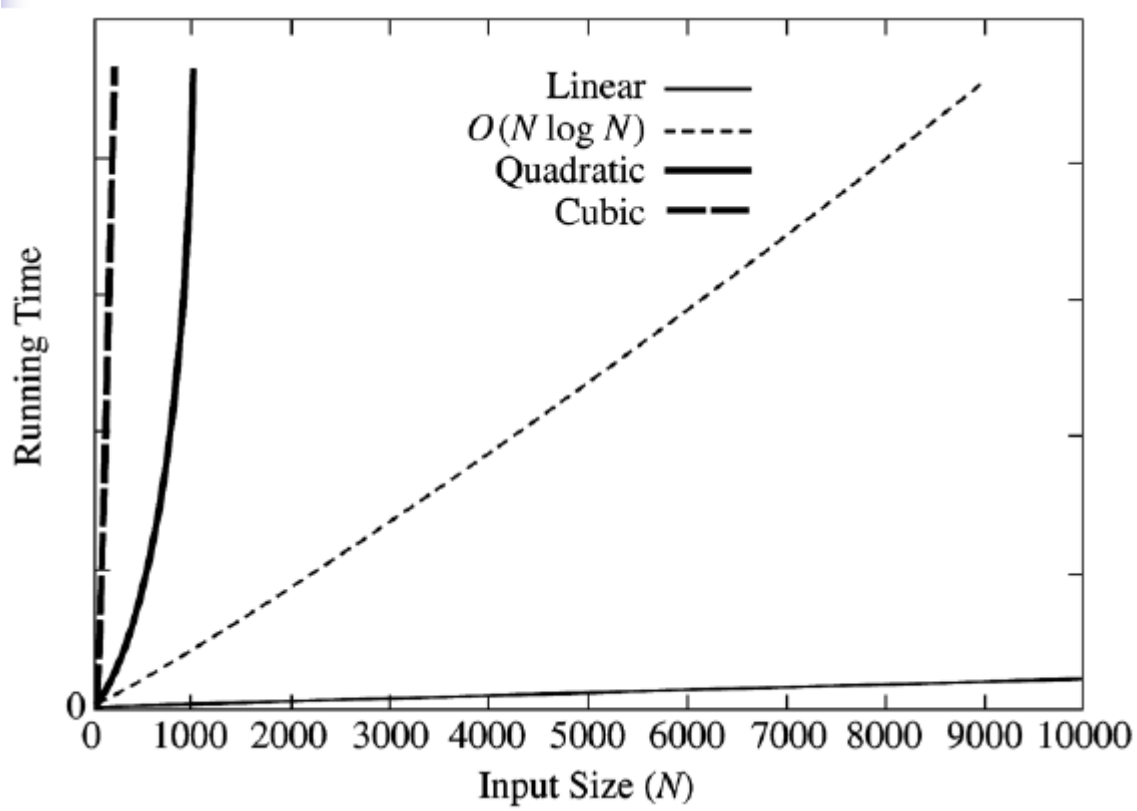
Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 10$	0.000009	0.000004	0.000006	0.000003
$N = 100$	0.002580	0.000109	0.000045	0.000006
$N = 1{,}000$	2.281013	0.010203	0.000485	0.000031
$N = 10{,}000$	NA	1.2329	0.005712	0.000317
$N = 100{,}000$	NA	135	0.064618	0.003206

38 min

26 days

Do not include array read times.

Maximum Subsequence Sum Problem Revisited



Algorithm 1

```
1   /**
2    * Cubic maximum contiguous subsequence sum algorithm.
3    */
4   int maxSubSum1( const vector<int> & a )
5   {
6       int maxSum = 0;
7
8       for( int i = 0; i < a.size( ); ++i )
9           for( int j = i; j < a.size( ); ++j )
10          {
11              int thisSum = 0;
12
13              for( int k = i; k <= j; ++k )
14                  thisSum += a[ k ];
15
16              if( thisSum > maxSum )
17                  maxSum = thisSum;
18          }
19
20      return maxSum;
21  }
```

Algorithm 1

exhaustively tries all possibilities

- first loop iterates N times
- second loop iterates $N - i$ times, which could be small, but we must assume the worst
- third loop iterates $j - i + 1$ times, which we must also assume to be size N
- total work on line 14 is $O(1)$, but it's inside these nested loops
- total running time is therefore $O(1 \cdot N \cdot N \cdot N) = O(N^3)$
- what about lines 16-17?

Algorithm 1

more precise calculation

- use better bounds on loops to compute how many times line 14 is calculated

$$\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} 1$$

- compute from the inside out

$$\sum_{k=i}^{j} 1 = j - i + 1$$

in general, top limit – bottom limit + 1

- then

$$\sum_{j=i}^{N-1} (j - i + 1) = \frac{(N - i + 1)(N - i)}{2}$$

Algorithm 1

more precise calculation (cont.)

- to complete the calculation

add 1 to limits → replace *i* with *i* - 1 in function

$$\sum_{i=0}^{N-1} \frac{(N-i+1)(N-i)}{2} = \sum_{i=1}^{N} \frac{(N-i+1)(N-i+2)}{2}$$

$$= \frac{1}{2}\sum_{i=1}^{N} i^2 - \left(N+\frac{3}{2}\right)\sum_{i=1}^{N} i + \frac{1}{2}(N^2+3N+2)\sum_{i=1}^{N} 1$$

$$= \frac{1}{2}\frac{N(N+1)(2N+1)}{6} - \left(N+\frac{3}{2}\right)\frac{N(N+1)}{2} + \frac{N^2+3N+2}{2}N$$

$$= \frac{N^3+3N^2+2N}{6}$$

Algorithm 1

another example

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        k = i * j;
```

use summations to find # of operations = $4n^2 + 4n + 2$

Algorithm 2

to speed up the algorithm, we can remove a for loop

- unnecessary calculation removed
- note that

$$\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$$

- new algorithm is $O(n^2)$

Algorithm 2

```
1   /**
2    * Quadratic maximum contiguous subsequence sum algorithm.
3    */
4   int maxSubSum2( const vector<int> & a )
5   {
6       int maxSum = 0;
7
8       for( int i = 0; i < a.size( ); ++i )
9       {
10          int thisSum = 0;
11          for( int j = i; j < a.size( ); ++j )
12          {
13              thisSum += a[ j ];
14
15              if( thisSum > maxSum )
16                  maxSum = thisSum;
17          }
18      }
19
20      return maxSum;
21  }
```

Algorithm 3

recursive algorithm runs even faster

- divide and conquer strategy
  - divide: split problem into two roughly equal subproblems
  - conquer: merge solutions to subproblems
- maximum subsequence can be in one of three places
  - entirely in left half of input
  - entirely in right half of input
  - in both halves, crossing the middle
- first two cases solved recursively
- last case solved by finding the largest sum in the first half that includes the last element of the first half, plus the largest sum in the second half that includes the first element in the second half

Algorithm 3

consider the following example

First Half				Second Half			
4	−3	5	−2	−1	2	6	−2

- maximum subsequence for the first half: 6
- maximum subsequence for the second half: 8
- maximum subsequence that crosses the middle
  - maximum subsequence in the first half that includes the last element of the first half: 4
  - maximum subsequence in the second half that includes the first element in the second half: 7
  - total: 11

Algorithm 3

```
1   /**
2    * Recursive maximum contiguous subsequence sum algorithm.
3    * Finds maximum sum in subarray spanning a[left..right].
4    * Does not attempt to maintain actual best sequence.
5    */
6   int maxSumRec( const vector<int> & a, int left, int right )
7   {
8       if( left == right )  // Base case
9           if( a[ left ] > 0 )
10              return a[ left ];
11          else
12              return 0;
13
14      int center = ( left + right ) / 2;
15      int maxLeftSum  = maxSumRec( a, left, center );
16      int maxRightSum = maxSumRec( a, center + 1, right );
17
18      int maxLeftBorderSum = 0, leftBorderSum = 0;
19      for( int i = center; i >= left; --i )
20      {
21          leftBorderSum += a[ i ];
22          if( leftBorderSum > maxLeftBorderSum )
23              maxLeftBorderSum = leftBorderSum;
24      }
```

Algorithm 3 (cont.)

```
25
26      int maxRightBorderSum = 0, rightBorderSum = 0;
27      for( int j = center + 1; j <= right; ++j )
28      {
29          rightBorderSum += a[ j ];
30          if( rightBorderSum > maxRightBorderSum )
31              maxRightBorderSum = rightBorderSum;
32      }
33
34      return max3( maxLeftSum, maxRightSum,
35                   maxLeftBorderSum + maxRightBorderSum );
36  }
37
38  /**
39   * Driver for divide-and-conquer maximum contiguous
40   * subsequence sum algorithm.
41   */
42  int maxSubSum3( const vector<int> & a )
43  {
44      return maxSumRec( a, 0, a.size( ) - 1 );
45  }
```

Algorithm 3

notes

- driver function needed
  - gets solution started by calling function with initial parameters that delimit entire array
- if `left == right`
  - base case
- recursive calls to divide the list
  - working toward base case
- lines 18-24 and 26-32 calculate max sums that touch the center
- **`max3`** returns the largest of three values

Algorithm 3

analysis

- let $T(N)$ be the time it takes to solve a maximum subsequence problem of size N
  - if $N = 1$, program takes some constant time to execute lines 8-12; thus $T(1) = 1$
  - otherwise, the program must perform two recursive calls
- the two **`for`** loops access each element in the subarray, with constant work; thus $O(N)$
  - all other non-recursive work in the function is constant and can be ignored
- recursive calls divide the list into two $N/2$ subproblems if N is even (and more strongly, a power of 2)
  - time: $2T(N/2)$

Algorithm 3

total time for the algorithm is therefore

$$T(N) = 2T\left(\frac{N}{2}\right) + N \quad \text{where } T(1) = 1$$

- note that

$$T(n) = T(n-1) + c \text{ is } T(n) = cn \text{ or just } O(n)$$

and

$$T(n) = T(n-1) + cn \text{ is } T(n) = cn(n)$$

or just $O(n^2)$

Algorithm 3

we can solve the recurrence relation directly (later)

$$T(N) = 2T\left(\frac{N}{2}\right) + N \quad \text{where } T(1) = 1$$

- for now, note that

$$T(1) = 1$$
$$T(2) = 4 = 2 * 2$$
$$T(4) = 12 = 4 * 3$$
$$T(8) = 32 = 8 * 4$$
$$T(16) = 80 = 16 * 5$$

thus, if $N = 2^k$,

$$T(N) = N * (k + 1) = N \log N + N = O(N \log N)$$

Algorithm 4

linear algorithm to solve the maximum subsequence sum

- observation: we never want a negative sum
- as before, remember the best sum we've encountered so far, but add a running sum
  - if the running sum becomes negative, reset the starting point to the first positive element

Algorithm 4

```
1   /**
2    * Linear-time maximum contiguous subsequence sum algorithm.
3    */
4   int maxSubSum4( const vector<int> & a )
5   {
6       int maxSum = 0, thisSum = 0;
7
8       for( int j = 0; j < a.size( ); ++j )
9       {
10          thisSum += a[ j ];
11
12          if( thisSum > maxSum )
13              maxSum = thisSum;
14          else if( thisSum < 0 )
15              thisSum = 0;
16      }
17
18      return maxSum;
19  }
```

Algorithm 4

notes

- if a[i] is negative, it cannot possibly be the first element of a maximum sum, since any such subsequence would be improved by beginning with a[i+1]
- similarly, any negative subsequence cannot possibly be a prefix for an optimal subsequence
  - this allows us to advance through the original sequence rapidly
- correctness may be difficult to determine
  - can be tested on small input by running against more inefficient brute-force programs
- additional advantage: data need only be read once (online algorithm) – extremely advantageous

Logarithms in the Running Time

logarithms will show up regularly in analyzing algorithms

- we have already seen them in divide and conquer strategies
- general rule: an algorithm is $O(N \log N)$ if it takes constant $O(1)$ time to cut the problem size by a fraction (typically $\frac{1}{2}$)
- if constant time is required to merely reduce the problem by a constant amount (say, smaller by 1), then the algorithm is $O(N)$
  - only certain problems fall into the $O(\log N)$ category since it would take $\Omega(N)$ just to read in the input
  - for such problems, we assume the input is pre-read

Binary Search

binary search

- given an integer X and integers $A_0, A_1, \ldots, A_{N\_1}$, which are presorted and already in memory, find I such that $A_i = X$, or return $i = -1$ if X is not in the input
- obvious solution: scan through the list from left to right to find X
  - runs in linear time
  - does not take into account that elements are presorted
- better solution: check if X is in the middle element
  - if yes, done
  - if smaller, apply same strategy to sorted left subarray
  - if greater, use right subarray

Binary Search

```
1   /**
2    * Performs the standard binary search using two comparisons per level.
3    * Returns index where item is found or -1 if not found.
4    */
5   template <typename Comparable>
6   int binarySearch( const vector<Comparable> & a, const Comparable & x )
7   {
8       int low = 0, high = a.size( ) - 1;
9
10      while( low <= high )
11      {
12          int mid = ( low + high ) / 2;
13
14          if( a[ mid ] < x )
15              low = mid + 1;
16          else if( a[ mid ] > x )
17              high = mid - 1;
18          else
19              return mid;    // Found
20      }
21      return NOT_FOUND;      // NOT_FOUND is defined as -1
22  }
```

Binary Search

analysis

- all the work done inside the loop takes $O(1)$ time per iteration
- first iteration of the loop is for $N - 1$
  - subsequent iterations of the loop halve this amount
  - total running time is therefore $O(\log N)$
- another example using sorted data for fast lookup
  - periodic table of elements
  - 118 elements
    - at most 8 accesses would be required

Euclid's Algorithm

Euclid's algorithm

- computes gcd (greatest common divisor) of two integers
  - largest integer that divides both

Euclid's Algorithm

```
1   long long gcd( long long m, long long n )
2   {
3       while( n != 0 )
4       {
5           long long rem = m % n;
6           m = n;
7           n = rem;
8       }
9       return m;
10  }
```

Euclid's Algorithm

notes

- algorithm computes $\gcd(M, N)$ assuming $M \geq N$
  - if $N > M$, the values are swapped
- algorithm works by continually computing remainders until 0 is reached
- the last non-0 remainder is the answer
- for example, if $M = 1{,}989$ and $N = 1{,}590$, the sequence of remainders is 399, 393, 6, 3, 0
  - therefore gcd = 3
- good, fast algorithm

Euclid's Algorithm

analysis

- need to determine how many remainders will be computed
  - $\log N$ is a good guess, but as can be seen from the example, the values do not decrease in a uniform way
- we can prove that the remainder is at most half of its original value after two iterations
  - this would show that the number of iterations is at most $2\log N = O(\log N)$

Euclid's Algorithm

analysis (cont.)

- Show: if $M > N$, then $N < M/2$
- Proof: 2 cases
  - if $N \leq M/2$, then since the remainder is smaller than N, the theorem is true
  - if $N > M/2$, then N goes into M once with a remainder of $M - N < M/2$, and the theorem is true
- in our example, $2\log N$ is about 20, but we needed only 7 operations
  - the constant can be refined to $1.44\log N$
  - the average case (complicated!) is $(12 \ln 2 \ln N)/\pi^2 + 1.47$

Exponentiation

exponentiation

- raising an integer to an integer power
  - results are often large, so machine must be able to support such values
  - number of multiplications is the measurement of work
- obvious algorithm: for X^N uses $N - 1$ multiplications
- recursive algorithm: better
  - $N \leq 1$ base case
  - otherwise, if N is even, $X^N = X^{N/2} \cdot X^{N/2}$
  - if N is odd, $X^N = X^{(N-1)/2} \cdot X^{(N-1)/2} \cdot X$
  - example: $X^{62} = (X^{31})^2$

$X^{31} = (X^{15})^2 X$, $X^{15} = (X^7)^2 X$, $X^7 = (X^3)^2 X$, $X^3 = (X^2)X$

Exponentiation

```
1   long long pow( long-long x, int n )
2   {
3       if( n == 0 )
4           return 1;
5       if( n == 1 )
6           return x;
7       if( isEven( n ) )
8           return pow( x * x, n / 2 );
9       else
10          return pow( x * x, n / 2 ) * x;
11  }
```

Exponentiation

analysis

- number of multiplications: $2\log N$
  - at most, 2 multiplications are required to halve the problem (if N is odd)
- some additional modifications can be made, but care must be taken to avoid errors and inefficiencies

Limitations of Worst-Case Analysis

analysis can be shown empirically to be an overestimate

- analysis can be tightened
- average running time may be significantly less than worst-case running time
  - often, the worst-case is achievable by very bad input, but still a large overestimate in the average case
  - unfortunately, average-case analysis is extremely complex (and in many cases currently unsolved), so worst-case is the best that we've got