Algorithm Analysis

Purpose

- Why bother analyzing code; isn't getting it to work enough?
  - Estimate time and memory in the average case and worst case
  - Identify bottlenecks, i.e., where to reduce time
  - Compare different approaches
  - Speed up critical algorithms

Problem – Algorithm – Data Structures & Techniques

When designing algorithms, we may end up breaking the problem into smaller "sub-problems"

Many algorithms can exist to solve one problem

Problem
(e.g., searching)

solves

Algorithm
(e.g., binary search)

contains

Specification
(Input => Output)

uses

Data structures & Techniques

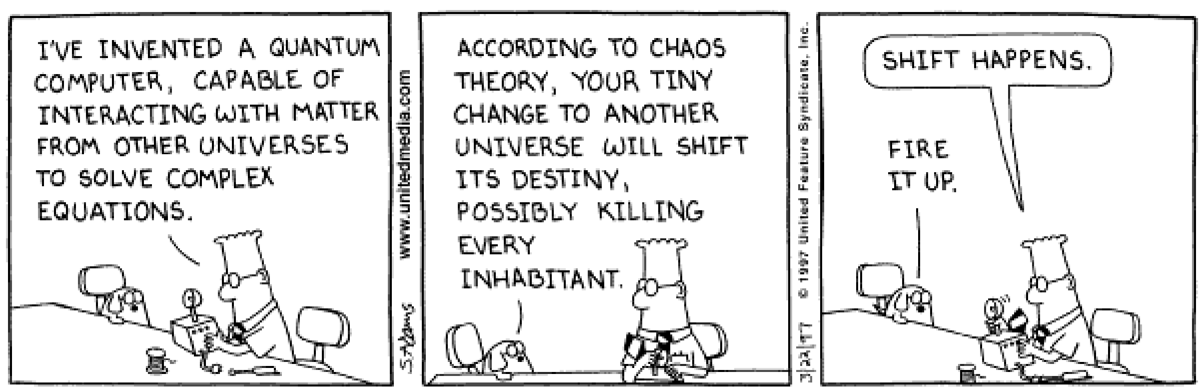
(e.g., sorting is a technique
array is a data struct.)

Algorithm

- Algorithm
  - Well-defined computational procedure for transforming inputs to outputs
- Problem
  - Specifies the desired input-output relationship
- Correct algorithm
  - Produces the correct output for every possible input in finite time
  - Solves the problem

Algorithm Analysis

- Predict resource utilization of an algorithm
  - Running time
  - Memory
- Dependent on architecture & model
  - Serial
  - Parallel
  - Quantum
  - DNA
  - …

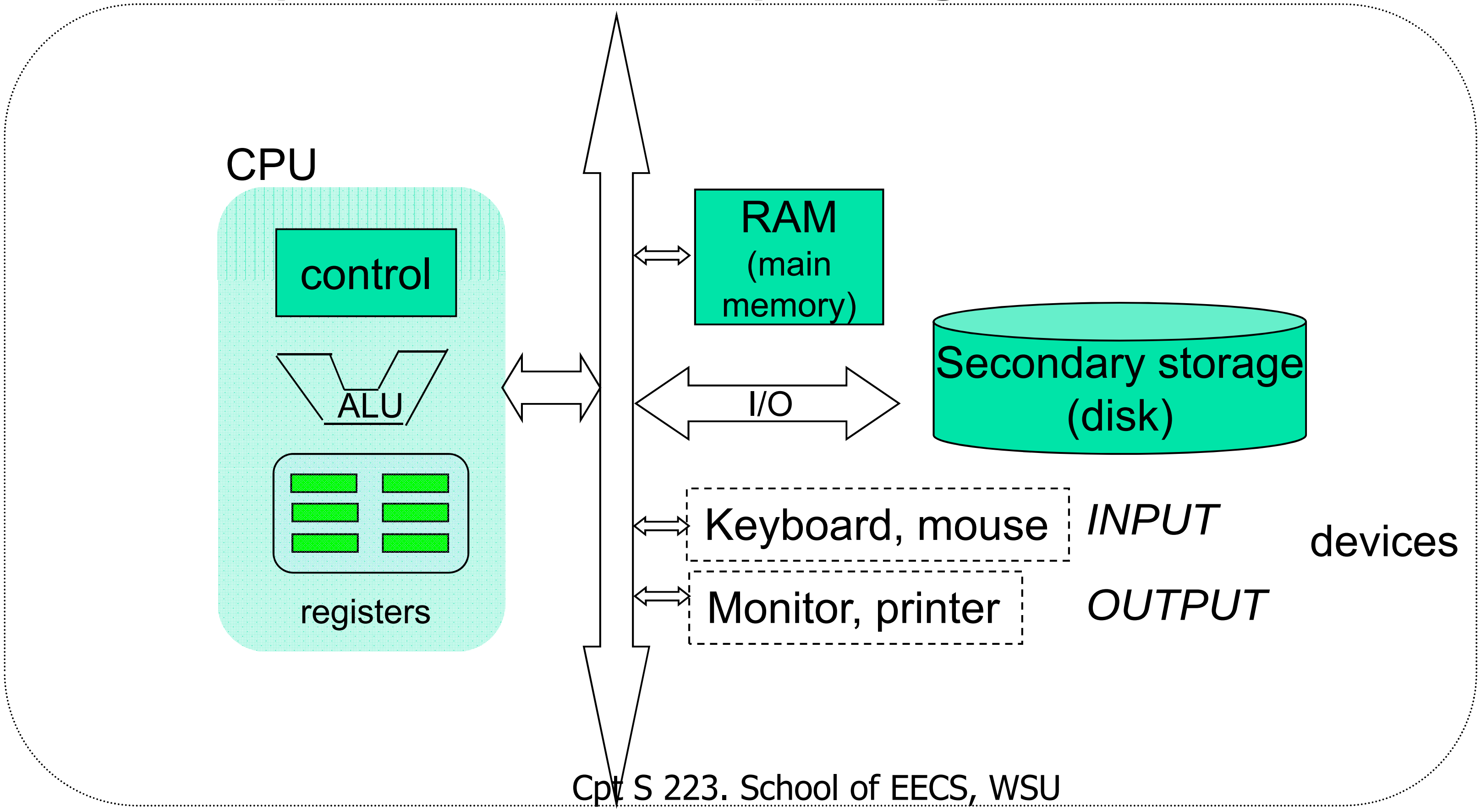


Factors for Algorithmic Design Consideration

- Run-time
- Space/memory
- Suitability to the problem’s application domain (contextual relevance)
- Scalability
- Guaranteeing correctness
- Deterministic vs. randomized
- Computational model
- System considerations: cache, disk, network speeds, etc.

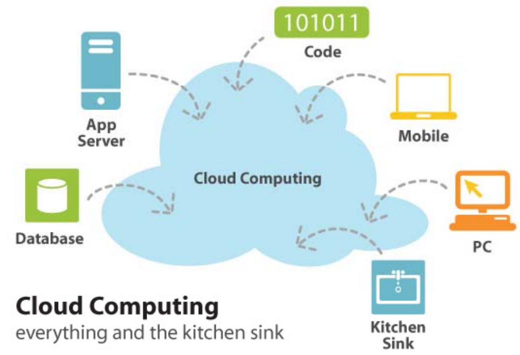
Model that we will assume

- Simple serial computing model



Other Models

- Multi-core models
- Co-processor model
- Multi-processor model
  - Shared memory machine
    - Multiple processors sharing common RAM
  - Memory is cheap ($20 per GB)
    - Memory bandwidth is NOT

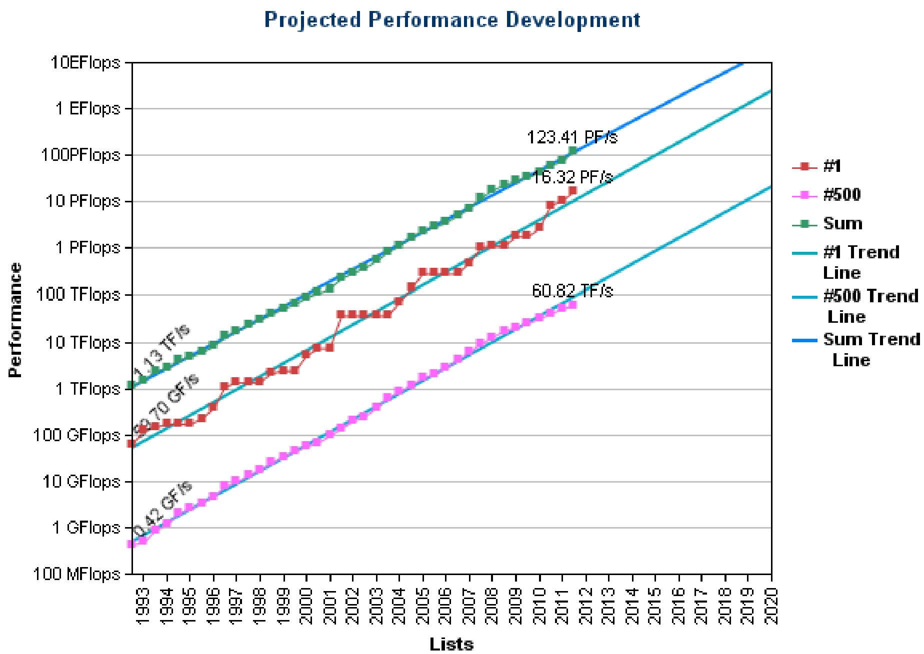


Cray XMT Supercomputer
- Up to 64 TB (65,536 GB) shared memory
- Up to 8000 processors
- 128 independent threads per processor
- $150M

Other Models

Supercomputer speed is measured in number of floating point operations per sec (or *FLOPS*)

- Distributed memory model
- www.top500.org



Fastest supercomputer (as of June 2012):

• Sequoia @ Lawrence Livermore National Lab

•16.32 **PetaFlop/s**
(16.32 x 10^{15} floating point ops per sec)

• IBM BlueGene/Q architecture
•#processing cores: 1.5M cores
•#aggregate memory: 1,572 TB

• Price tag: millions of $$$

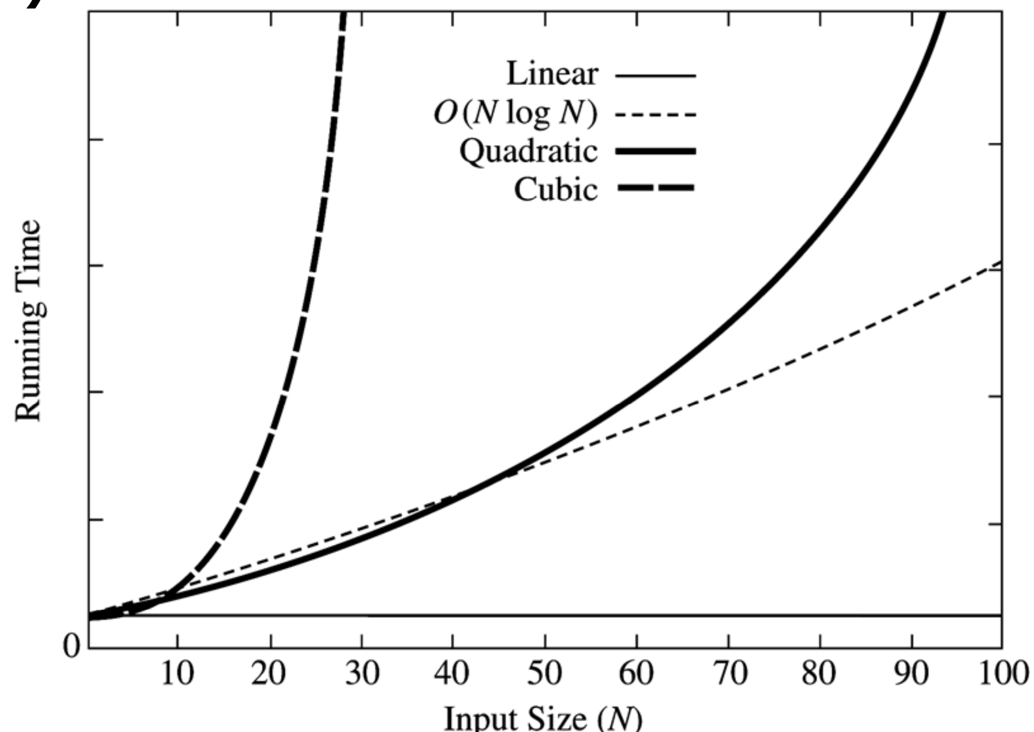
What to Analyze: T(n)

- Running time T(n)
  - N or n is typically used to denote the size of the input
    - Sorting?
    - Multiplying two integers?
    - Multiplying two matrices?
    - Traversing a graph?

- T(n) measures number of primitive operations performed
  - E.g., addition, multiplication, comparison, assignment

How to Analyze T(n)?

- As the input (n) grows what happens to the time T(n)?



Example for calculating T(n)

```
                                              #operations
int sum (int n)                               0
{
     int partialSum;                          0

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

$T(n) = 6n+4$

Example: Another less-precise but equally informative analysis

```
                                                   #operations
int sum (int n)                                    0
{
     int partialSum;                               0

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

$T(n) \propto n$

Do constants matter?

- What happens if:
  - N is small (< 100)?
  - N is medium (<1,000,000)?
  - N is large (> 1,000,000)?
- Asymptotically, curves matter more than absolute values!
  - Let:
    - $T_1(N) = 3N^2 + 100N + 1$
    - $T_2(N) = 50N^2 + N + 500$
  - Compare $T_1(N)$ vs $T_2(N)$?

Both codes will show a quadratic behavior

Algorithmic Notation & Analysis

- Big-O $O()$
- Omega $\Omega()$
- small-O $o()$
- small-omega $\omega()$
- Theta $\Theta()$

Asymptotic notations that help us quantify & compare costs of different solutions

- Worst-case
- Average-case
- Best-case

Describes the input

- "Algorithms"
- Lower-bound
- Upper-bound
- Tight-bound
- "Optimality"

Describes the problem & its solution

Asymptotic Notation

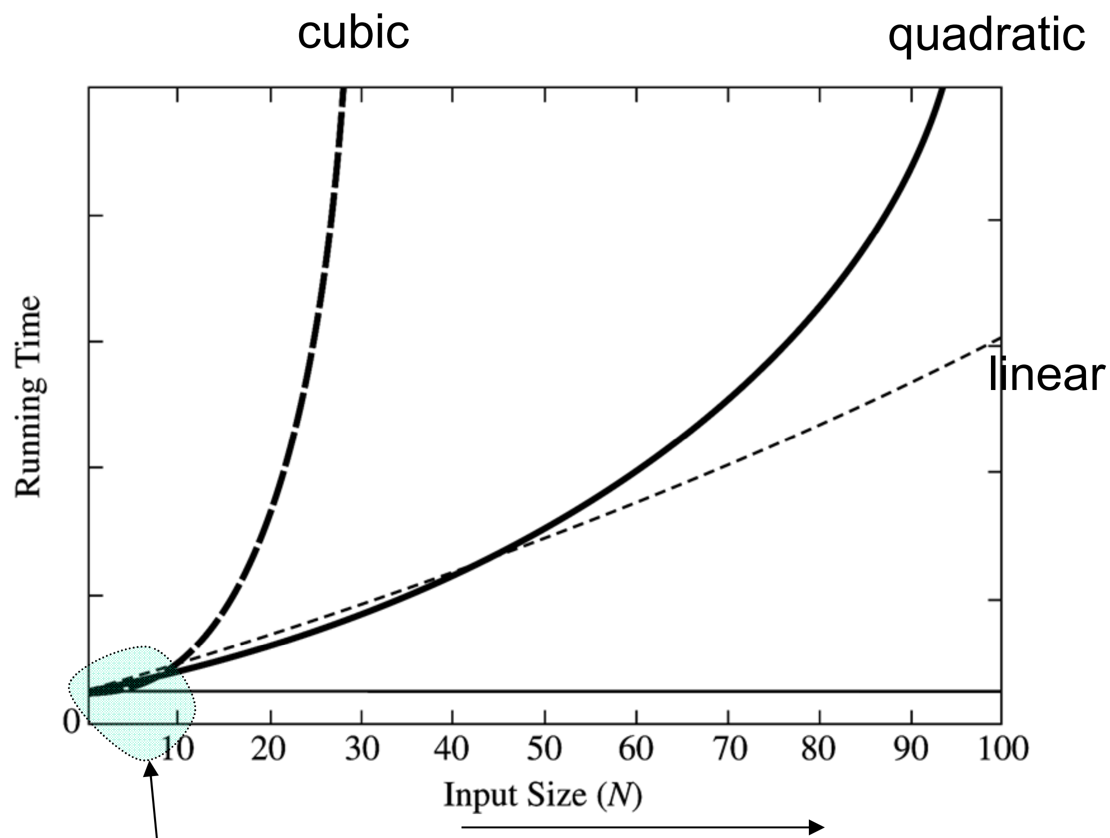
- Theta
  - Tight bound
- Big-Oh
  - Upper bound
- Omega
  - Lower bound

The main idea:

Express cost relative to standard functions (e.g., lg n, n, n^2, etc.)

Some Standard Function Curves

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential



(curves not to scale)

Initial aberrations do NOT matter!

What you want to measure

A standard function (e.g., n, n^2)

Big-oh : O()

$\leq$

- $f(N) = O(g(N))$ if there exist positive constants c and n_0 such that:
  - $f(N) \leq c*g(N)$, for all $N>n_0$

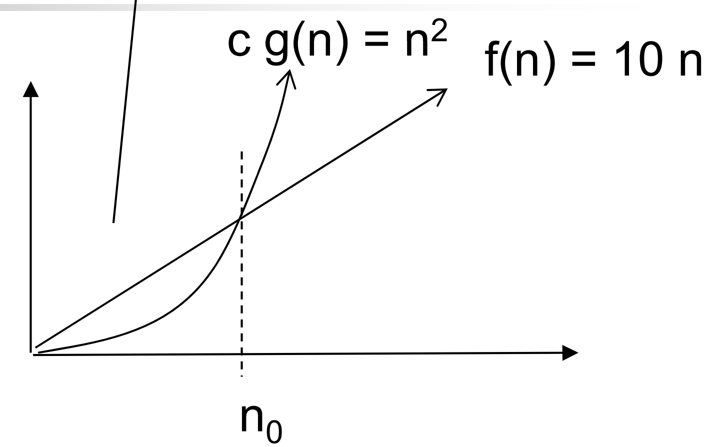
Upper bound for f(N)

- Asymptotic upper bound, possibly tight

Example for big-Oh

Up & down fluctuation allowed in this zone

- E.g., let f(n)=10 n
  - ==> $f(n) = O(n^2)$
- <u>Proof:</u>
  - If f(n) = O(g(n))
    - ==> $f(n) \le c\, g(n)$, for all $n>n_0$
    - (show such a $<c,n_0>$ combination exists)
    - ==> $c=1$, $n_0 = 9$
  - (Remember: always try to find the lowest possible n_0)



c=1

Breakeven point (n_0)

n	10 n	c n²
1	10	1
2	20	4
3	30	9
4	40	16
5	50	25
..	…	..
9	90	81
10	100	100
11	110	121
…	..	…

Ponder this

- If $f(n) = 10\ n$, then:
  - Is $f(n) = O(n)$?
  - Is $f(n) = O(n^2)$?
  - Is $f(n) = O(n^3)$?
  - Is $f(n) = O(n^4)$?
  - Is $f(n) = O(2^n)$?
  - …

Yes: e.g., for <$c=10$, $n_0=1$>

Yes: e.g., for <$c=1$, $n_0=9$>

- If all of the above, then what is the best answer?
- $f(n) = O(n)$

Little-oh : o()

$<$

- f(N) = o(g(N)) if there exist positive constants c and n_0 such that:
  - $f(N) < c*g(N)$ when $N>n_0$

Strict upper bound for f(N)

- E.g., f(n)=10 n; g(n) = n^2
  - ==> f(n) = o(g(n))

Big-omega : Ω ()

$\geq$

- f(N) = Ω(g(N)) if there exist positive constants c and n_0 such that:
  - $f(N) \geq c*g(N)$, for all $N>n_0$

Lower bound for f(N), possibly tight

- E.g., $f(n)= 2n^2$; g(n) =n log n

  ==> f(n) = Ω (g(n))

Little-omega : $\omega()$

$>$

- $f(N) = \omega(g(N))$ if there exist positive constants c and n_0 such that:
  - $f(N) > c*g(N)$ when $N>n_0$

Strict lower bound for f(N)

- E.g., $f(n)= 100\ n^2$; $g(n) =n$

  $\Longrightarrow f(n) = \omega\ (g(n))$

Theta : $\Theta()$

$\equiv$

- $f(N) = \Theta(h(N))$ if there exist positive constants c_1, c_2 and n_0 such that:
  - $c_1 h(N) \leq f(N) \leq c_2 h(N)$ for all $N > n_0$

(same as)

Tight bound for f(N)

- $f(N) = \Theta(h(N))$ if and only if $f(N) = O(h(N))$ and $f(N) = \Omega(h(N))$

Example (for theta)

- $f(n) = n^2 - 2n \rightarrow f(n) = \Theta(?)$
- Guess: $f(n) = \Theta(n^2)$
- Verification:
  - Can we find valid c_1, c_2, and n_0?
- If true:

$\rightarrow c_1 n^2 \leq f(n) \leq c_2 n^2$

$\rightarrow c_1 n^2 \leq n^2 - 2n \leq c_2 n^2$

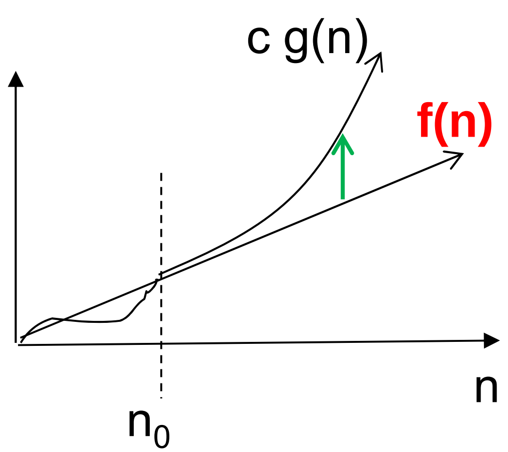
$\rightarrow$ …

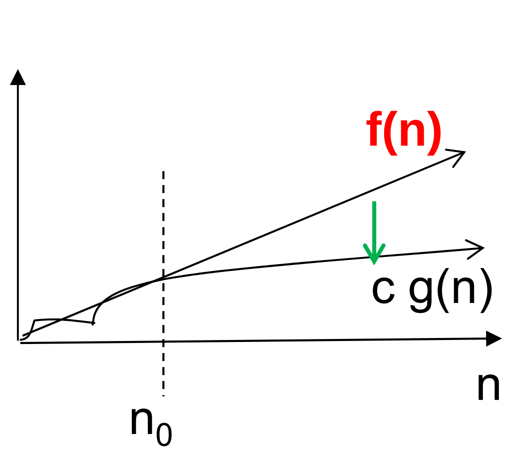
The Asymptotic Notations

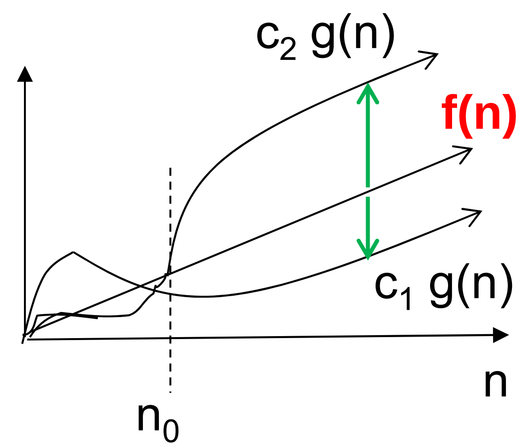
Notation	Purpose	Definition
O(n)	Upper bound, possibly tight	$f(n) \le c\ g(n)$
Ω(n)	Lower bound, possibly tight	$f(n) \ge c\ g(n)$
Θ(n)	Tight bound	$c_1\ g(n) \le f(n) \le c_2\ g(n)$
o(n)	Upper bound, strict	$f(n) < c\ g(n)$
ω(n)	Lower bound, strict	$f(n) > c\ g(n)$


DILBERT
By Scott Adams
BLAH, BLAH, BLAH, BLAH. . .
I HAVE NO IDEA WHAT HE'S TALKING ABOUT.

Asymptotic Growths







f(n) = O(g(n))
f(n) = o(g(n))

f(n) = Ω(g(n))
f(n) = ω (g(n))

f(n) = Θ(g(n))

Rules of Thumb while using Asymptotic Notations

<u>Algorithm's complexity:</u>

- When asked to analyze an algorithm's complexities:

Tight bound

1st Preference:
Whenever possible, use $\Theta()$

Upper bound

2nd Preference:
If not, use $O()$ - or $o()$

Lower bound

3rd Preference:
If not, use $\Omega()$ - or $\omega()$

Rules of Thumb while using Asymptotic Notations…

Algorithm’s complexity:

- Unless otherwise stated, express an algorithm’s complexity in terms of its worst-case

Rules of Thumb while using Asymptotic Notations…

Problem's complexity

- Ways to answer a problem's complexity:

  Q1) This problem is at least as hard as … ?

  Use lower bound here

  Q2) This problem cannot be harder than … ?

  Use upper bound here

  Q3) This problem is as hard as … ?

  Use tight bound here

A few examples

- $N^2 = O(N^2) = O(N^3) = O(2^N)$
- $N^2 = \Omega(1) = \Omega(N) = \Omega(N^2)$
- $N^2 = \Theta(N^2)$
- $N^2 = o(N^3)$
- $2N^2 + 1 = \Theta(?)$
- $N^2 + N = \Theta(?)$
- $N^3 - N^2 = \Theta(?)$
- $3N^3 - N^3 = \Theta(?)$

Reflexivity

- Is $f(n) = \Theta(f(n))$? yes
- Is $f(n) = O(f(n))$? yes
- Is $f(n) = \Omega(f(n))$? yes

reflexive

- Is $f(n) = o(f(n))$? no
- Is $f(n) = \omega(f(n))$? no

Not reflexive

Symmetry

- $f(n) = \Theta(g(n))$ "iff" $g(n) = \Theta(f(n))$

"If and only if"

Transpose symmetry

- $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

Transitivity

- $f(n) = \Theta(g(n))$ <u>and</u> $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$ <u>and</u> $g(n) = O(h(n)) \Rightarrow$?
- $f(n) = \Omega(g(n))$ <u>and</u> $g(n) = \Omega(h(n)) \Rightarrow$?
- …

More rules…

- Rule 1: If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then
  - additive: $T_1(n) + T_2(n) = O(f(n) + g(n))$
  - multiplicative: $T_1(n) * T_2(n) = O(f(n) * g(n))$
- Rule 2: If $T(n)$ is a polynomial of degree k, then $T(n) = \Theta(n^k)$
- Rule 3: $\log^k n = O(n)$ for any constant k
- Rule 4: $\log_a n = \Theta(\log_b n)$ for any constants a & b

Some More Proofs

- Prove that: $n \lg n = O(n^2)$
  - We know $\lg n \le n$, for $n \ge 1$ ($\Rightarrow n_0 = 1$)
  - Multiplying n on both sides:

    $n \lg n \le n^2$

    $\Rightarrow \quad n \lg n \le 1.\ n^2$

Some More Proofs…

- Prove that: $6n^3 \neq O(n^2)$
  - By contradiction:
    - If $6n^3 = O(n^2)$
    - → $6n^3 \leq c\, n^2$
    - → $6n \leq c$
    - → It is not possible to bound a variable with a constant
    - → Contradiction ϟ

Maximum subsequence sum problem

- Given N integers $A_1, A_2, \ldots, A_N$, find the maximum value (≥ 0) of:

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

  - Don't need the actual sequence (i,j), just the sum
  - If final sum is negative, output 0
- E.g., [1, -4, $\underline{4, 2, -3, 5, 8}$, -2]

16 is the answer

j

i

MaxSubSum: Solution 1

- Compute for all possible subsequence ranges (i,j) and pick the maximum range

```
MaxSubSum1 (A)
   maxSum = 0
   for i = 1 to N
      for j = i to N
         sum = 0
         for k = i to j
            sum = sum + A[k]
         if (sum > maxSum)
         then maxSum = sum
   return maxSum
```

All possible start points

All possible end points

Calculate sum for range [i .. j]

$\Theta(N)$

$\Theta(N)$

$\Theta(N)$

Total run-time $= \Theta(N^3)$

```
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); i++ )
        for( int j = i; j < a.size( ); j++ )
        {
            int thisSum = 0;

            for( int k = i; k <= j; k++ )
                thisSum += a[ k ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }

    return maxSum;
}
```

Solution 1: Analysis

- More precisely

$$T(N) = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} \Theta(1)$$

$$= \Theta(N^3)$$

New sum

A[j]

Old sum

MaxSubSum: Solution 2

- Observation: $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$
  - ==> So, re-use sum from previous range

Old code:

```
MaxSubSum1 (A)
   maxSum = 0
   for i = 1 to N
      for j = i to N
         sum = 0
         for k = i to j
            sum = sum + A[k]
         if (sum > maxSum)
         then maxSum = sum
   return maxSum
```

Sum (new k) = sum (old k) + A[k]
=> So *NO* need to recompute sum for range A[i..k-1]

New code:

```
MaxSubSum2 (A)
   maxSum = 0
   for i = 1 to N
      sum = 0
      for j = i to N
         sum = sum + A[j]
         if (sum > maxSum)
         then maxSum = sum
   return maxSum
```

$\Theta(N)$

$\Theta(N)$

Total run-time = $\Theta(N^2)$

```
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); i++ )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); j++ )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

Solution 2: Analysis

$$T(N) = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \Theta(1)$$

$$T(N) = \Theta(N^2)$$

Can we do better than this?

Use a Divide & Conquer technique?

MaxSubSum: Solution 3

- Recursive, *"divide and conquer"*
  - Divide array in half
    - $A_{1..center}$ and $A_{(center+1)..N}$
  - Recursively compute MaxSubSum of left half
  - Recursively compute MaxSubSum of right half
  - Compute MaxSubSum of sequence constrained to use A_{center} and $A_{(center+1)}$
  - Return max { left_max, right_max, center_max }

- E.g., <1, -4, 4, 2, -3, 5, 8, -2>

max_{left} max_{center} max_{right}

MaxSubSum: Solution 3

```
MaxSubSum3 (A, i, j)
    maxSum = 0
    if (i = j)
    then if A[i] > 0
         then maxSum = A[i]
    else k = floor((i+j)/2)
         maxSumLeft = MaxSubSum3(A,i,k)
         maxSumRight = MaxSubSum3(A,k+1,j)
         compute maxSumThruCenter
         maxSum = Maximum (maxSumLeft,
                           maxSumRight,
                           maxSumThruCenter)
    return maxSum
```

```
/**
 * Recursive maximum contiguous subsequence sum algorithm.
 * Finds maximum sum in subarray spanning a[left..right].
 * Does not attempt to maintain actual best sequence.
 */
int maxSumRec( const vector<int> & a, int left, int right )
{
    if( left == right )  // Base case
        if( a[ left ] > 0 )
            return a[ left ];
        else
            return 0;

    int center = ( left + right ) / 2;
    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );
```

// how to find the max that passes through the center

```
    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    int maxRightBorderSum = 0, rightBorderSum = 0;
    for( int j = center + 1; j <= right; j++ )
    {
        rightBorderSum += a[ j ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                     maxLeftBorderSum + maxRightBorderSum );
}
```

Keep right end fixed at *center* and vary left end

Keep left end fixed at *center+1* and vary right end

Add the two to determine max through center

```
/**
 * Driver for divide-and-conquer maximum contiguous
 * subsequence sum algorithm.
 */
int maxSubSum3( const vector<int> & a )
{
    return maxSumRec( a, 0, a.size( ) - 1 );
}
```

Solution 3: Analysis

- $T(1) = \Theta(1)$
- $T(N) = 2T(N/2) + \Theta(N)$
- $T(N) = \Theta(?)$

Can we do even better?

$T(N) = \Theta(N \log N)$

A	1	-4	4	2	-3	5	8	-2
Sum	1	0	4	6	3	8	$\boxed{16}$	14

MaxSubSum: Solution 4

E.g., <1, -4, 4, 2, -3, 5, 8, -2>

- Observation
  - Any negative subsequence cannot be a prefix to the maximum sequence
  - Or, only a positive, contiguous subsequence is worth adding

```
MaxSubSum4 (A)
    maxSum = 0
    sum = 0
    for j = 1 to N
        sum = sum + A[j]
        if (sum > maxSum)
        then maxSum = sum
        else if (sum < 0)
             then sum = 0
    return maxSum
```

$T(N) = \Theta(N)$

Can we do even better?

MaxSubSum: Solution 5 (another $\Theta(N)$ algorithm)

- Let's define: Max(i) <= maxsubsum for range A[1..i]
  - ==> Max(N) is the final answer
- Let; Max'(i) <= maxsubsum <u>ending</u> at A[i]
- <u>Base case:</u>
  - Max(1) = = Max'(1) = max { 0, A[1]}
- <u>Recurrence</u> (i.e., FOR i=2 to N):
  - Max(i) = ?
  - = max { Max'(i), Max(i-1) }
  - Max'(i) = ?

    = max { Max'(i-1) + A[i], A[i], 0}

<u>Case:</u>
Max ends at i

<u>Case:</u>
Max *does not* end at i

This technique is called "*dynamic programming*"

Algorithm 5 pseudocode

<u>Dynamic programming:</u> (re)use the optimal solution for a sub-problem to solve a larger problem

```
MaxSubSum5 (A)
   if N==0 return 0

   max = MAX(0,A[1])
   max' = MAX(0,A[1])

   for j = 2 to N
       max' = MAX(max'+A[j], A[j], 0)
       max = MAX(max, max')

   return max
```

$T(N) = \Theta(N)$

```
/**
 * Linear-time maximum contiguous subsequence sum algorithm.
 */
int maxSubSum4( const vector<int> & a )
{
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 )
            thisSum = 0;
    }

    return maxSum;
}
```

What are the space complexities of all 5 versions of the code?

$\Theta(n)$

MaxSubSum Running Times

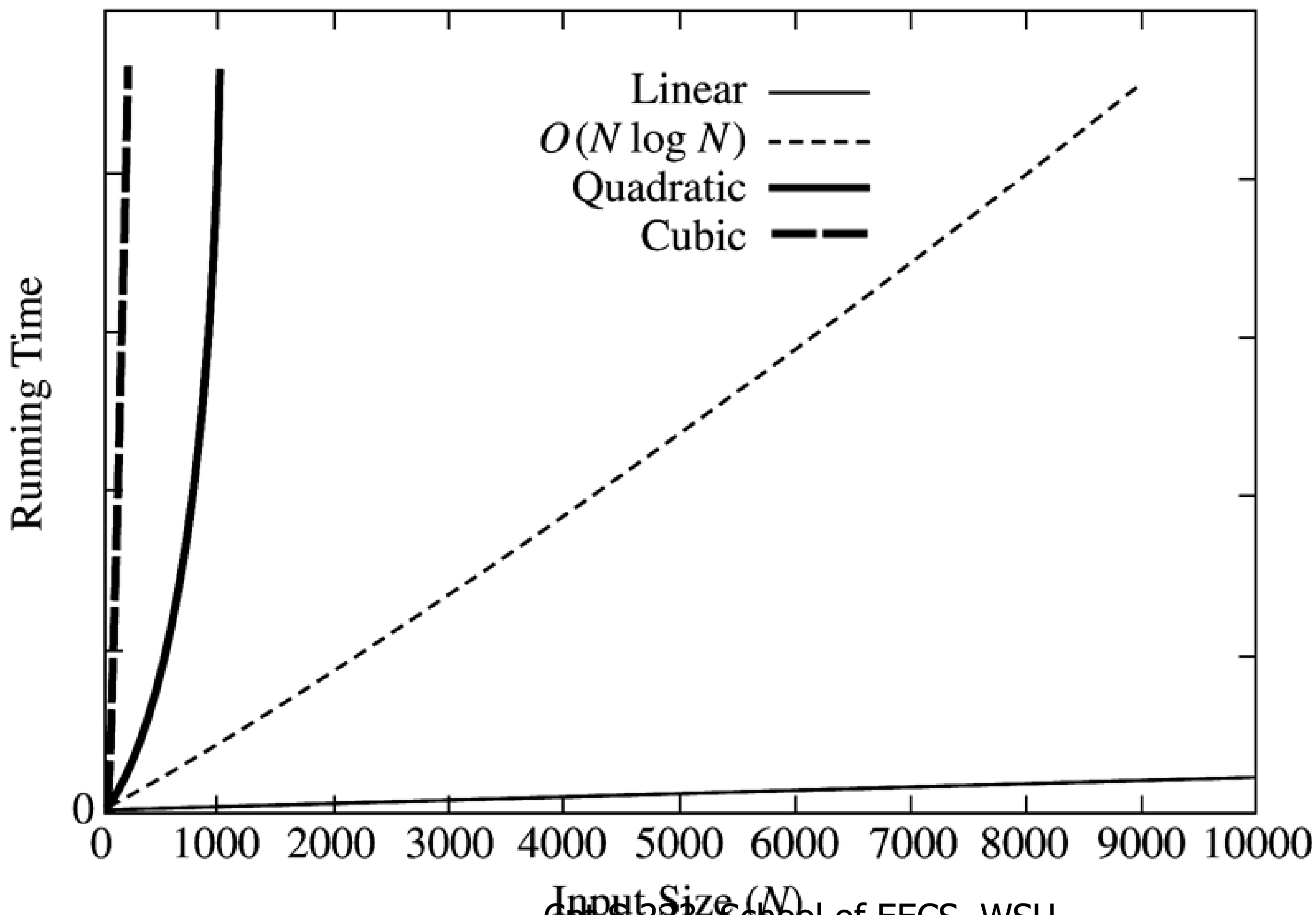
Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 and 5 $O(N)$
$N = 10$	0.000009	0.000004	0.000006	0.000003
$N = 100$	0.002580	0.000109	0.000045	0.000006
$N = 1{,}000$	2.281013	0.010203	0.000485	0.000031
$N = 10{,}000$	NA	1.2329	0.005712	0.000317
$N = 100{,}000$	NA	135	0.064618	0.003206

38 min

26 days

Do not include array read times.

MaxSubSum Running Times



Logarithmic Behavior

- $T(N) = O(\log_2 N)$
- Usually occurs when
  - Problem can be halved in constant time
  - Solutions to sub-problems combined in constant time
- Examples
  - Binary search
  - Euclid's algorithm
  - Exponentiation

For off-class reading:
- read book
- slides for lecture notes from course website

Binary Search

- Given an integer X and integers $A_0, A_1, \ldots, A_{N-1}$, which are <u>presorted</u> and already in memory, find i such that $A_i = X$, or return i = -1 if X is not in the input.
- $T(N) = O(\log_2 N)$

```
/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found.
 */
template <typename Comparable>
int binarySearch( const vector<Comparable> & a, const Comparable & x )
{
    int low = 0, high = a.size( ) - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid;   // Found
    }
    return NOT_FOUND;     // NOT_FOUND is defined as -1
}
```

Euclid's Algorithm

- Compute the greatest common divisor gcd(M,N) between the integers M and N
  - I.e., largest integer that divides both
  - Used in encryption

Euclid’s Algorithm

```
long gcd( long m, long n )
{
    while( n != 0 )
    {
        long rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

Example: gcd(3360,225)

- m = 3360, n = 225
- m = 225, n = 210
- m = 210, n = 15
- m = 15, n = 0

Euclid's Algorithm: Analysis

- Note: After two iterations, remainder is at most half its original value
  - Thm. 2.1: If M > N, then M mod N < M/2
- $T(N) = 2 \log_2 N = O(\log_2 N)$
  - $\log_2 225 = 7.8$, $T(225) = 16$ (?)
- Better worst case: $T(N) = 1.44 \log_2 N$
  - $T(225) = 11$
- Average case: $T(N) = (12 \ln 2 \ln N) / \pi^2 + 1.47$
  - $T(225) = 6$

Exponentiation

- Compute X^N
- Obvious algorithm:

```
pow(x,n)
  result = 1
  for i = 1 to n
    result = result * x
  return result
```

- Observation
  - $X^N = X^{N/2} * X^{N/2}$ (for even N)
  - $X^N = X^{(N-1)/2} * X^{(N-1)/2} * X$ (for odd N)
- Minimize multiplications T(N)
- $T(N) = 2 \log_2 N = O(\log_2 N)$

Exponentiation

```
long pow( long x, int n )
{
    if( n == 0 )
        return 1;
    if( n == 1 )
        return x;
    if( isEven( n ) )
        return pow( x * x, n / 2 );
    else
        return pow( x * x, n / 2 ) * x;
}
```

$T(N) = \Theta(1)$, $N \leq 1$
$T(N) = T(N/2) + \Theta(1)$, $N > 1$

$T(N) = O(\log_2 N)$
$T(N) = \Theta(\log_2 N)$?

Summary

- Algorithm analysis
- Bound running time as input gets big
- Rate of growth
- Compare algorithms
- Recursion and logarithmic behavior