Algorithm Analysis

Ph.D. Truong Dinh Huy

Background

Suppose we have two algorithms, how can we tell which is better?

We could implement both algorithms, run them both.

– Expensive and error prone

Preferably, we should analyze them mathematically

– *Algorithm analysis*

Analysis of Algorithms

- Efficiency measure
  - how long the program runs   time complexity
  - how much memory it uses   space complexity
    - For today, we'll focus on time complexity only

Algorithm Analysis

- In general, we will always analyze algorithms with respect to one or more variables of input data. In this lecture, we focus on 1 variable
- Given an algorithm:
  - We need to describe running time as a function T(N) (**Time complexity**) mathematically with N is data input.
  - We need to do this in a machine-independent way

- We count number of abstract simple steps
  - Not physical runtime in seconds
  - Not every machine instruction

Time Complexity: Sum of Array

```
int sum_array( int *array, int N ) {
    int sum;                                   No time
                              1
    sum = 0;                        1      N+1      N

    for ( int i = 0; i < N; ++i ) {
        sum = sum + array[i];
    }          N                         N

    return sum;                   1
}
```

Time complexity: T(N) = 4*N + 4

Asymptotic Analysis

- Complexity as a function of input size n

  $T(n) = 4n + 5$

  $T(n) = 0.5\ n \log n - 2n + 7$

  $T(n) = 2^n + n^3 + 3n$

- *What happens as n grows?*

Rate of Growth

- Most algorithms are fast for small n
  - Time difference too small to be noticeable
  - External things dominate (OS, disk I/O, …)
- n is typically large in practice
  - Databases, internet, graphics, …

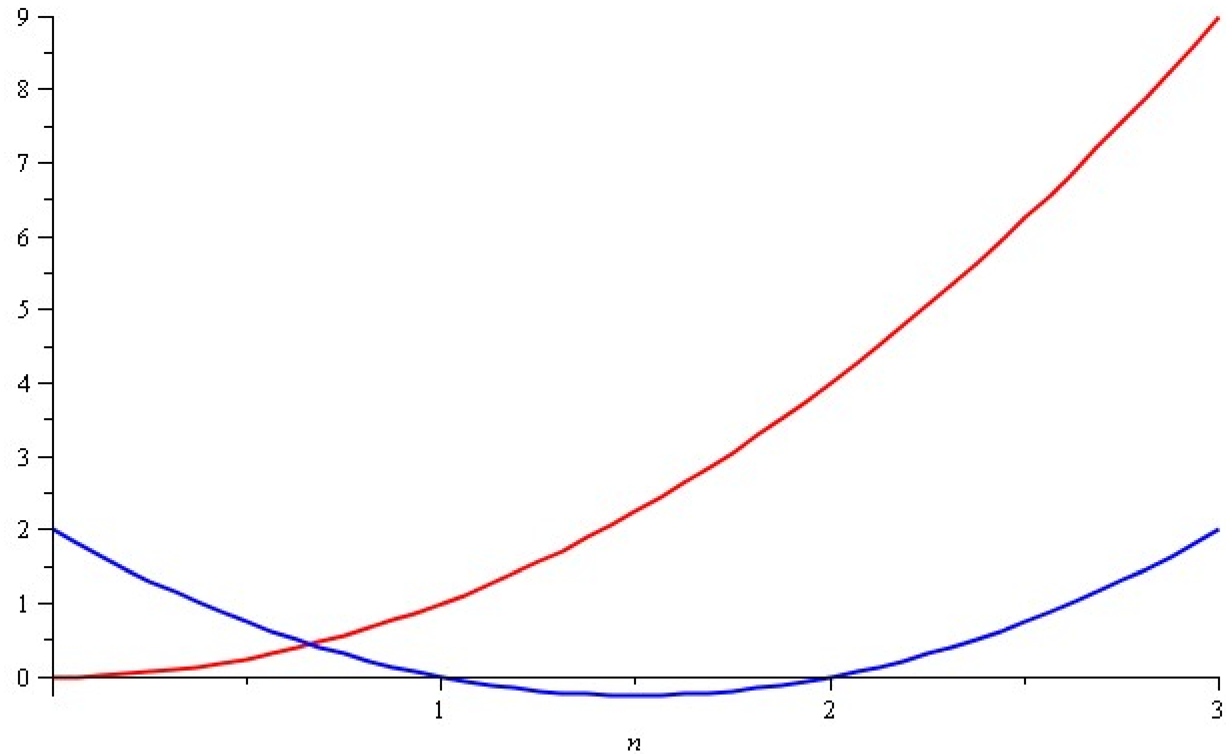
- **Time difference really shows up as n grows!**

Quadratic Growth

Consider the two functions

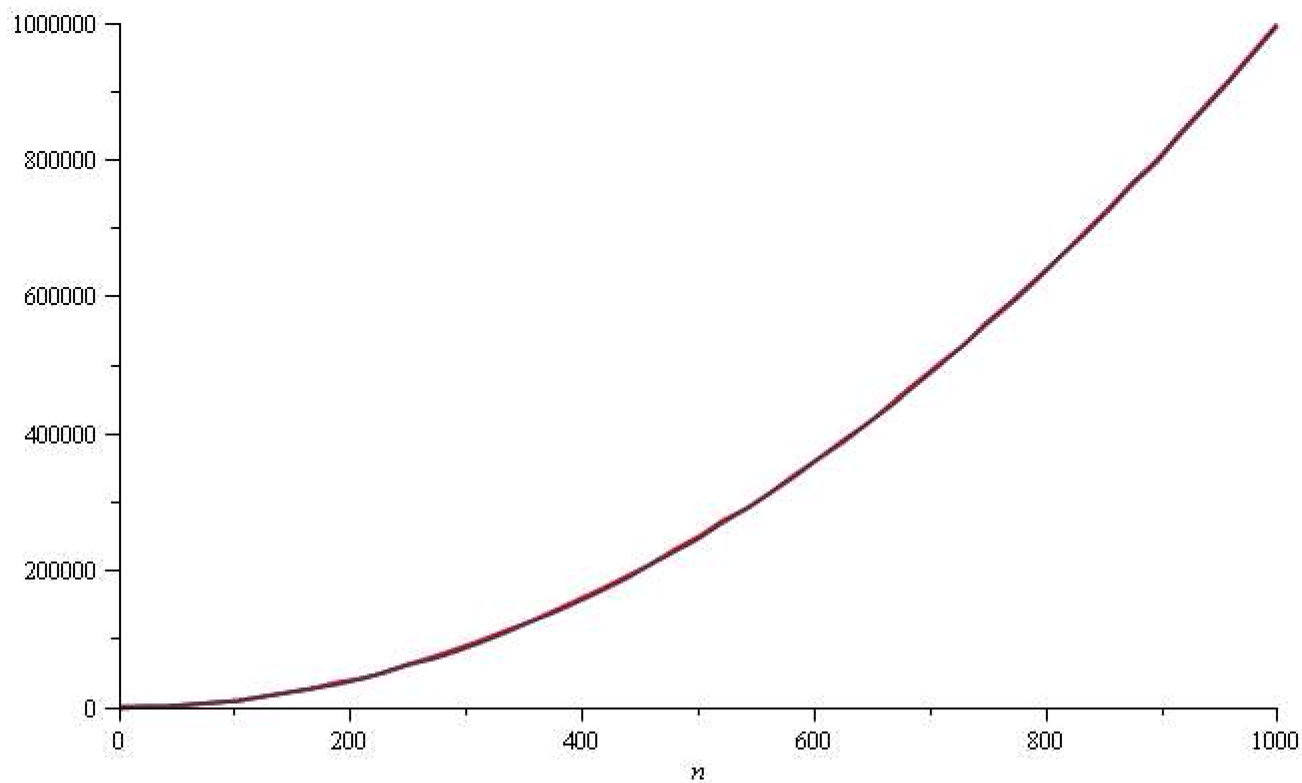
$f(n) = n^2$ and $g(n) = n^2 - 3n + 2$

Around $n = 0$, they look very different



Quadratic Growth

Yet on the range $n = [0, 1000]$, they are (relatively) indistinguishable:



Quadratic Growth

The absolute difference is large, for example,

$$\mathrm{f}(1000) = 1\ 000\ 000$$

$$\mathrm{g}(1000) = \ \ 997\ 002$$

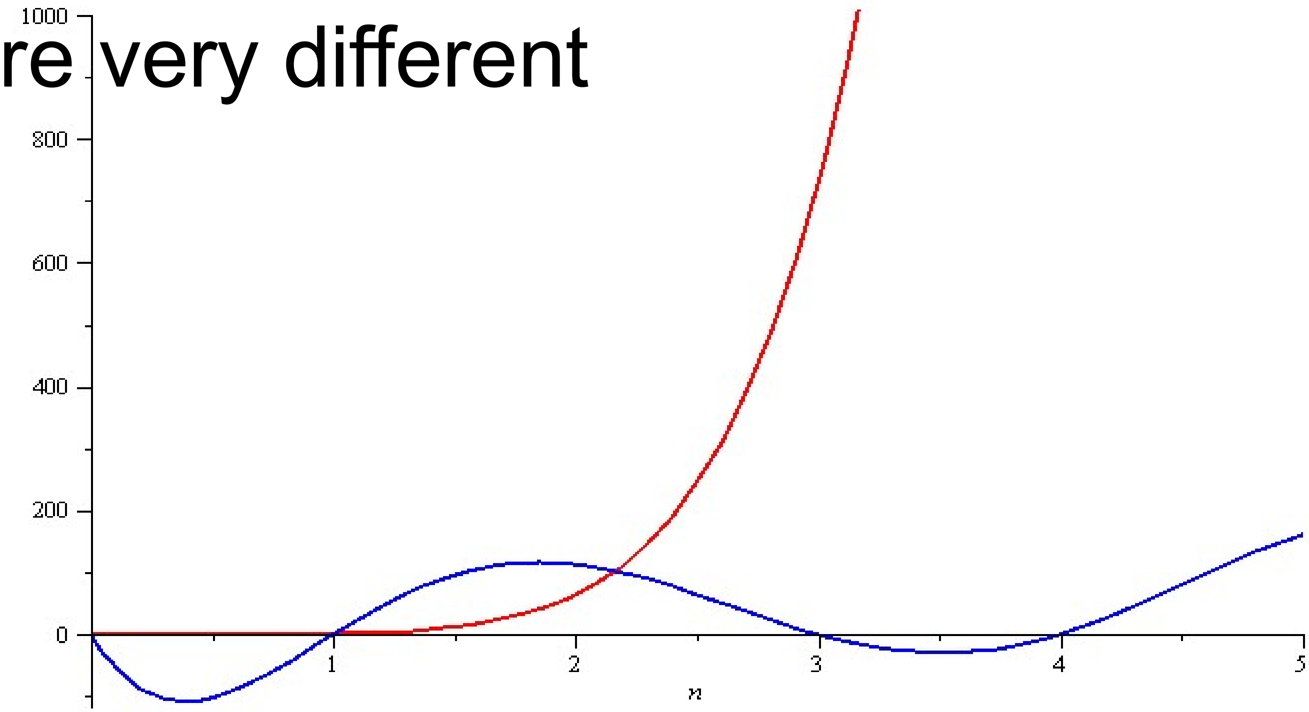
but the relative difference is very small

$$\left|\frac{\mathrm{f}(1000) - \mathrm{g}(1000)}{\mathrm{f}(1000)}\right| = 0.002998 < 0.3\%$$

and this difference goes to zero as $n \longrightarrow \infty$

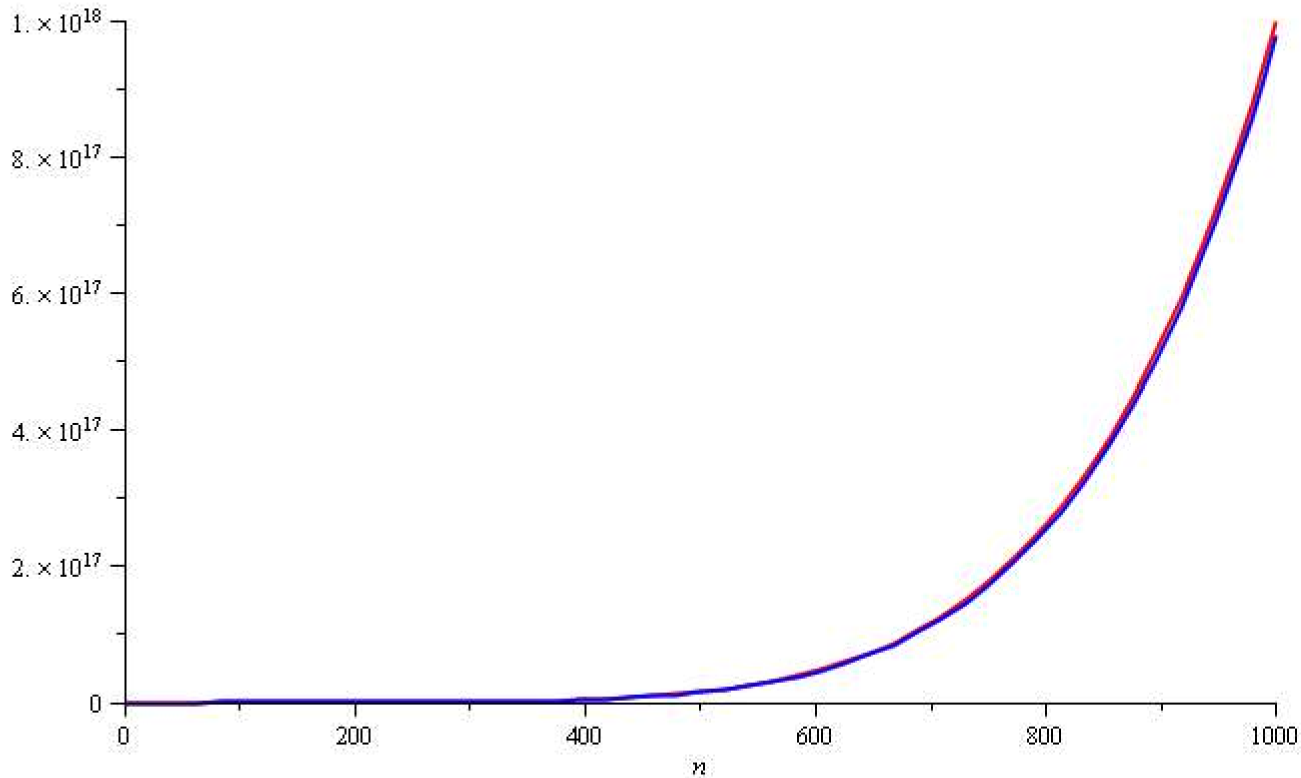
Polynomial Growth

To demonstrate with another example, $\mathrm{f}(n) = n^6$ and $\mathrm{g}(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$. Around $n = 0$, they are very different



Polynomial Growth

Still, around $n = 1000$, the relative difference is less than 3%



Comparison of two functions

Which is better: $50N^2 + 31N^3 + 24N + 15$ or
$3N^2 + N + 21 + 4*3^N$

Answer depends on value of N:

N	$50N^2 + 31N^3 + 24N + 15$	$3N^2 + N + 21 + 4*3^N$
1	120	37
2	511	71
3	1374	159
4	2895	397
5	5260	1073
6	8655	3051
7	13266	8923
8	19279	26465
9	26880	79005
10	36255	236527

What Happened?

N	$3N^2 + N + 21 + 4*3^N$	$4*3^N$	$\%ofTotal$
1	37	12	32.4
2	71	36	50.7
3	159	108	67.9
4	397	324	81.6
5	1073	972	90.6
6	3051	2916	95.6
7	8923	8748	98.0
8	26465	26244	99.2
9	79005	78732	99.7
10	236527	236196	99.9

– One term dominated the sum

As N Grows, Some Terms Dominate

Function	10	100	1000	10000	100000
$\log_2 N$	3	6	9	13	16
N	10	100	1000	10000	100000
$N \log_2 N$	30	664	9965	10^5	10^6
N^2	10^2	10^4	10^6	10^8	10^{10}
N^3	10^3	10^6	10^9	10^{12}	10^{15}
2^N	10^3	10^{30}	10^{301}	10^{3010}	10^{30103}

Order of Magnitude Analysis

Measure speed with respect to the part of the sum that grows quickest

$$50N^2 + \boxed{31N^3} + 24N + 15$$

$$3N^2 + N + 21 + \boxed{4 * 3^N}$$

Ordering:

$$1 < \log_2 N < N < N \log_2 N < N^2 < N^3 < 2^N < 3^N$$

Order of Magnitude Analysis (cont)

Furthermore, simply ignore any constants in front of term and simply report general class of the term:

$31\boxed{N^3}$ $\quad 4 * \boxed{3^N}$ $\quad 15\boxed{N \log_2 N}$

$50N^2 + 31N^3 + 24N + 15$ grows proportionally to N^3

$3N^2 + N + 21 + 4 * 3^N$ grows proportionally to 3^N

When comparing algorithms, determine formulas to count operation(s) of interest, then compare dominant terms of formulas

Obtaining Asymptotic Bounds

- Eliminate low order terms
  - $4n + 5 \Rightarrow 4n$
  - $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$
  - $2^n + n^3 + 3n \Rightarrow 2^n$

- Eliminate coefficients
  - $4n \Rightarrow n$
  - $0.5\ n \log n \Rightarrow n \log n$
  - $n \log n^2 = 2\ n \log n \Rightarrow n \log n$

Big O Notation

- Algorithm A requires time proportional to f(N) - algorithm is said to be of order f(N) or O(f(N))
- **Definition**: an algorithm is said to take time proportional to O(f(N)) if there is some constant C such that for all but a finite number of values of N, the time taken by the algorithm is less than C*f(N)
- $T(n) = O(f(n))$: growth rate of $T(n) \leq$ that of $f(n)$
  - $\exists$ constants c and n_0 s.t. $T(n) \leq c\, f(n)\quad \forall n \geq n_0$
  - Or if $\lim_{n\to\infty} T(n)/f(n)$ exists and is finite, then T(n) is O(f(n))

Examples:

$$50N^2 + 31N^3 + 24N + 15 \text{ is } O(N^3)$$

$$3N^2 + N + 21 + 4*3^N \text{ is } O(3^N)$$

Big O Notation(2)

- If an algorithm is O(f(N)), f(N) is said to be the *growth-rate* function of the algorithm.
- Or: f(N) is an ***upper bound*** on T(N):
- $T(N)= 2N^2 \Rightarrow T(N)=O(N^2)=O(N^3)=O(N^4)$
- But $O(N^2)$ is the best answer. The answer should be as tight (good) as possible.

Other Terminologies

- $T(n) = \Omega(f(n))$ (growth rate of $T(n)$ >= that of $f(n)$)
  - $\exists$ constants c and n_0 s.t. $T(n) \geq c\, f(n) \quad \forall n \geq n_0$
- $T(n) = \theta(f(n))$ (growth rate of $T(n)$ = that of $f(n)$)
  - $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$
- $T(n) = o(f(n))$ (growth rate of $T(n)$ < that of $f(n)$)
  - $T(n) = O(f(n))$ and $T(n) \mathrel{!=} \theta(f(n))$
- $T(n) \in \omega(f(n))$ (growth rate of $T(n)$ > that of $f(n)$)
  - $T(n) = \Omega(f(n))$ and $T(n) \mathrel{!=} \theta(f(n))$

Typical Growth Rates

– c:	Constant	
– $\log N$	Logarithmic	
– $\log^k N$	Poly-log	(k is a constant)
– N	Linear	
– $N \log N$	Log-linear	
– N^2	Quadratic	
– N^3	Cubic	
– N^k	Polynomial	(k is a constant)
– C^n	Exponential	(C is a constant)

Types of Analysis

Three orthogonal axes:

- bound flavor
  - upper bound (O, o)
  - lower bound (Ω, ω)
  - asymptotically tight (θ)
- analysis case
  - worst case (adversary)
  - average case
  - best case
  - “common” case
- analysis quality
  - loose bound (most true analyses)
  - tight bound (no better bound which is asymptotically different)

Analyzing Code

- General guidelines

Simple C++ operations	- constant time
consecutive stmts	- sum of times per stmt
conditionals	- sum of branches and condition
loops	- sum over iterations
function calls	- cost of function body

Simple loop

- **Rule 1- single loops:** The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.
  Ex:

```
for i = 1 to n do
   k++
```

Simple loops (2)

Rule 2-Nested loops: Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

Simple loops (3)

```
Ex1. for i = 1 to n do
          for j = i to n do
               sum = sum + 1

     T(n)= ?

Ex2. for i = 1 to n do
          for j = i to n do
               for k= i to j do
                    sum = sum + 1
     T(n)= ?
```

Conditions

- Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).
- Conditional

```
if C then S1 else S2
```

T(n) <= time of C + MAX (S1, S2)

<= time of C + S1 + S2

- Ex:
- If (N==0) return 0;
- else {

for (i=0; i<N; i++) sum++;}

Consecutive statements

- Rule 3: Add the time complexities of each statement.
- If $T_1(N) = O\big(f(N)\big)$ and $T_2(N) = O\big(g(N)\big)$
  - $T_1(N) + T_2(N) = \max(O\big(f(N)\big), O\big(g(N)\big))$
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$
- Ex:

```
 for i = 1 to n do
    k=k+1
for i = 1 to n do
for j = 1 to n do
```

T(N)= ?

Quiz

Logarithms in running time

- Example 1:

```
for (i = N; i>=1;)
    i=i/2;
```

`T(N)= O(logN)`

- Example 2:

Greatest common divisor

O(log(min(m, n))

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

Recursion

- Recursion
  - *Function calls its self*
  - *Some cases are very difficult to analyze*
- Example: Factorial
- Normal case O(N)
- Recursion:

```
fac(n)
  if n = 0 return 1
  else return n * fac(n - 1)
```

$$T(0) = 1$$

$$T(n) \leq c + T(n - 1) \qquad \textit{if } n > 0$$

Example: Factorial

Analysis by substitution method

$T(n) \le c + c + T(n - 2)$

(by substitution)

$T(n) \le c + c + c + T(n - 3)$

(by substitution, again)

$T(n) \le kc + T(n - k)$

(extrapolating $0 < k \le n$)

$T(n) \le nc + T(0) = nc + b$

(setting $k = n$)

- $T(n)$ **= O(n) (the same as normal case)**

Bad example of Recursion: Fibonacci

```
        long int
        Fib( int N )
        {
/* 1*/      if( N <= 1 )
/* 2*/          return 1;
            else
/* 3*/          return Fib( N - 1 ) + Fib( N - 2 );
        }
```

$T(N) = T(N-1) + T(N-2) +2$

$\mathbf{(3/2)^N <= T(N) < (5/3)^N}$

Good example of Recursion: 3^N

- Normal case: O(N)
- Using recursion, we can achieve a faster program: O(logN)

```
Exponentiator (n) // n is an integer
        if (n==1)
                return 1
        else if (n%2==0){
                        x = Exponentiator (n/2);
                        return x*x;}
            else {
                        x = Exponentiator ((n-1)/2);
                        return 3*x*x;}
```

Proof

- $T(N)= T(N/2) + c$ and $T(1)=1$;
- $T(N) = T(N/4) + c + c$
- …
- $T(N) = T(N/2^i) + c + \ldots. + c$
- $T(N) = T(1) + c\log N = O(\log N)$

= T(N/2^i) + i * c
= T(1) + i * c (2^i = N => i = logN)

Quiz

- $T(N) = 2T(N/2) + 1$

$T(1) = 1$

$= 2 * (2T(N/4) + 1) + 1$
$= 4 * T(N/4) + 3$
$= 4 * (2T(N/8) + 1) + 3$
$= 8 * T(N/8) + 7$
$= 8 * (2T(N/16) + 1) + 7$
$= 16 * T(N/16) + 15$
...
$= 2^i * T(N/ 2^i) + 2^i - 1$
$2^i = N \Rightarrow i = \log N$
$\Rightarrow T(N) = 2N - 1$

- $T(N) = 2T(N/2) + n$

$= 2 * (2T(N/4) + n) + n$
$= 4 * T(N/4) + 3n$
$= 4 * (2T(N/8) + n) + 3n$
$= 8 * T(N/8) + 7n$
$= 8 * (2T(N/16) + n) + 7n$
$= 16 * T(N/16) + 15n$
...
$= 2^i * T(N/ 2^i) + (2^i - 1)n$
$2^i = N \Rightarrow i = \log N$
$\Rightarrow T(N) = N + (N-1)*n = (n+1)N - n$

Master Theorem for Divide and Conquer

$T(n) = aT(n/b) + \theta(n^k \log^p n)$ kep giua

T(n) = f(n)

$a >= 1$, $b > 1$, $k >= 0$ and p is a real number

Case (1): $a > b^k$ then $T(n) = \theta(n^{\log_b a})$

Case(2): $a = b^k$:

if $p > -1$ then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

if $p = -1$ then $T(n) = \theta(n^{\log_b a} \log \log n)$

if $p < -1$ then $T(n) = \theta(n^{\log_b a})$

Case(3): $a < b^k$:

if $p >= 0$ then $T(n) = \theta(n^k \log^p n)$

if $p = 0$ then $T(n) = O(n^k)$

Examples

Admissible equations:

Example 1: $T(n) = 9T(n/3) + n$ $\quad\quad T(n) = ?$

Example 2: $T(n) = T(2n/3) + 1$ $\quad\quad$ T(n) = ?

Example 3: $T(n) = 3T(n/4) + n \log n$ $\quad\quad T(n) = ?$

Example 4: $T(n) = 2T(n/2) + n/\log n$ $\quad\quad T(n) = ?$

Inadmissible equations*:*

Example 5: $T(n) = 2^n T(n/2)+1$

Example 6: $T(n) = T(n/2) - n^2 \log n$

Master Theorem for Subtraction and Conquer

$T(n) = aT(n-b) + f(n)$

for some constants $a > 0$, $b > 0$, $k \geq 0$, and function $f(n)$.

If $f(n)$ is in $O(n^k)$, then:

$$T(n) = \begin{cases} O(n^k), & \text{if } a < 1 \\ O(n^{k+1}), & \text{if } a = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if } a > 1 \end{cases}$$

Maximum Subsequence Problem

There is an array of N integers (possible negative)

Find the maximum sum of all elements between the ith and jth position.
For example: -2, 11, -4, 13, -5, -2, the answer is 20 (from A[1] to A[3])

Algorithm 1

Figure 2.5 Algorithm 1

```
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, i, j, k;

/* 1*/      MaxSum = 0;
/* 2*/      for( i = 0; i < N; i++ )
/* 3*/          for( j = i; j < N; j++ )
                {
/* 4*/              ThisSum = 0;
/* 5*/              for( k = i; k <= j; k++ )
/* 6*/                  ThisSum += A[ k ];

/* 7*/              if( ThisSum > MaxSum )
/* 8*/                  MaxSum = ThisSum;
                }
/* 9*/      return MaxSum;
        }
```

Analysis

Inner loop:

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

Outer Loop:

$$\sum_{i=0}^{N-1} (N-i+1)(N-i)/2 = (N^3 + 3N^2 + 2N)/6$$

Overall: $O(N^3)$

Algorithm 2

Figure 2.6 Algorithm 2

```
        int
        MaxSubSequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, i, j;

/* 1*/      MaxSum = 0
/* 2*/      for( i = 0; i < N; i++ )
            {
/* 3*/          ThisSum = 0;
/* 4*/          for( j = i; j < N; j++ )
                {
/* 5*/              ThisSum += A[ j ];

/* 6*/              if( ThisSum > MaxSum )
/* 7*/                  MaxSum = ThisSum;
                }
            }
/* 8*/      return MaxSum;
        }
```

Divide and Conquer

1. Break a big problem into two small sub-problems

2. Solve each of them efficiently.

3. Combine the two solutions

Algorithm 3: Divide and conquer

Divide the array into two parts: left part, right part

Max. subsequence lies completely in left, or completely in right or spans the middle.

If it spans the middle, then it includes the max subsequence in the left ending at the last element and the max subsequence in the right starting from the center

Divide and conquer

4 –3 5 –2 -1 2 6 -2

Max subsequence sum for first half = 6

second half = 8

Max subsequence sum for first half ending at the last element is 4

Max subsequence sum for sum second half starting at the first element is 7

Max subsequence sum spanning the middle is ? 4+7 = 11

Max subsequence is 11 and the best subarray is to include elements from both half.

```
          static int
          MaxSubSum( const int A[ ], int Left, int Right )
          {
              int MaxLeftSum, MaxRightSum;
              int MaxLeftBorderSum, MaxRightBorderSum;
              int LeftBorderSum, RightBorderSum;
              int Center, i;

/* 1*/        if( Left == Right )  /* Base Case */
/* 2*/            if( A[ Left ] > 0 )
/* 3*/                return A[ Left ];
                  else
/* 4*/                return 0;

/* 5*/        Center = ( Left + Right ) / 2;
/* 6*/        MaxLeftSum = MaxSubSum( A, Left, Center );
/* 7*/        MaxRightSum = MaxSubSum( A, Center + 1, Right );

/* 8*/        MaxLeftBorderSum = 0; LeftBorderSum = 0
/* 9*/        for( i = Center; i >= Left; i-- )
              {
/*10*/            LeftBorderSum += A[ i ];
/*11*/            if( LeftBorderSum > MaxLeftBorderSum )
/*12*/                MaxLeftBorderSum = LeftBorderSum;
              }

/*13*/        MaxRightBorderSum = 0; RightBorderSum = 0;
/*14*/        for( i = Center + 1; i <= Right; i++ )
              {
/*15*/            RightBorderSum += A[ i ];
/*16*/            if( RightBorderSum > MaxRightBorderSum )
/*17*/                MaxRightBorderSum = RightBorderSum;
              }

/*18*/        return Max3( MaxLeftSum, MaxRightSum,
/*19*/                     MaxLeftBorderSum + MaxRightBorderSum );
          }

          int
          MaxSubsequenceSum( const int A[ ], int N )
          {
              return MaxSubSum( A, 0, N - 1 );
          }
```

Complexity Analysis

$T(n) = \underbrace{2T(n/2)}_{\text{left and right}} + \underbrace{cn}_{\text{middle}} = O(n\log n)$ (Master theorem)

Proof:

$= 2.cn/2 + 4T(n/4) + cn$

$= 4T(n/4) + 2cn$

$= 8T(n/8) + 3cn$

$= \ldots\ldots\ldots\ldots\ldots$

$= 2^iT(n/2^i) + icn$

$= \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$ (reach a point when $n = 2^i$ $i=\log n$)

$= n.T(1) + cn\log n = O(n\log n)$

Algorithm 4

```
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, j;

/* 1*/      ThisSum = MaxSum = 0;
/* 2*/      for( j = 0; j < N; j++ )
            {
/* 3*/          ThisSum += A[ j ];

/* 4*/          if( ThisSum > MaxSum )
/* 5*/              MaxSum = ThisSum;
/* 6*/          else if( ThisSum < 0 )
/* 7*/              ThisSum = 0;
            }
/* 8*/      return MaxSum;
        }
```

Figure 2.8 Algorithm 4

O(n) complexity

Summary

- Describe running time of an algorithm as a mathematical function of input size.
- Terminologies: $O(\bullet)$, $\Omega(\bullet)$, $\theta(\bullet)$.
- How to analyze a program
- Example: Maximum Subsequence Problem
- Next week: Sorting