IS 709/809: Computational Methods in IS Research

Algorithm Analysis

Nirmalya Roy
Department of Information Systems
University of Maryland Baltimore County
What is an Algorithm?

- An algorithm is a clearly specified set of instructions to be followed to solve a problem
  - Solves a problem but requires a year is hardly of any use
  - Requires several gigabytes of main memory is not useful on most machines

Problem

- Specifies the desired input-output relationship

Correct algorithm

- Produces the correct output for every possible input in finite time
- Solves the problem
Purpose

- Why bother analyzing algorithm or 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 and space
  - Speed up critical algorithms or make them more efficient
Algorithm Analysis

- Predict resource utilization of an algorithm
  - Running time
  - Memory usage

- Dependent on architecture
  - Serial
  - Parallel
  - Quantum
What to Analyze

- Our main focus is on running time
  - Memory/time tradeoff
  - Memory is cheap

- Our assumption: simple serial computing model
  - Single processor, infinite memory
What to Analyze (cont’d)

- Let $T(N)$ be the running time
  - $N$ (sometimes $n$) is typically the size of the input
    - Linear or binary search?
    - Sorting?
    - Multiplying two integers?
    - Multiplying two matrices?
    - Traversing a graph?

- $T(N)$ measures number of primitive operations performed
  - E.g., addition, multiplication, comparison, assignment
An Example

```c
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;
}
```

\[ T(n) = ? \]
Running Time Calculations

- The declarations count for no time
- Simple operations (e.g. +, *, <=, =) count for one unit each
- Return statement counts for one unit
Revisit the Example

```
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;
}
```

\[ T(N) = 6N+4 \]
Running Time Calculations (cont’d)

- General rules
  - Rule 1 – Loops
    - The running time of a loop is at most the running time of the statements inside the loop (including tests) times the number of iterations of the loop
  - Rule 2 – Nested loops
    - Analyze these inside out
    - The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops

  Number of iterations
Running Time Calculations (cont’d)

- **Examples**
  - **Rule 1 – Loops**
    ```
    for (int i = 0; i < N; i++)
        sum += i * i;
    # of operations
    ? (1 + N + N)
    ? 3*N
    T(N) = ?
    ```
  - **Rule 2 – Nested loops**
    ```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            sum++;
    # of operations
    ? (1 + N + N)
    ? N*(1 + N + N)
    ? N * N
    T(N) = ?
    ```
Running Time Calculations (cont’d)

- **General rules**
  - **Rule 3 – Consecutive statements**
    - These just add
    - Only the maximum is the one that counts
  - **Rule 4 – Conditional statements (e.g. if/else)**
    - The running time of a conditional statement is never more than the running time of the test plus the largest of the running times of the various blocks of conditionally executed statements
  - **Rule 5 – Function calls**
    - These must be analyzed first
Running Time Calculations (cont’d)

Examples

- Rule 3 – Consecutive statements

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

# of operations

```
?  
?  
?  
?  
T(n) = ?
```
Examples

Rule 4 – Conditional statements

```java
if (a > b && c < d) {
    for (int j = 0; j < n; j++)
        a[i] += j;
}
else {
    for (int j = 0; j < n; j++)
        for (int k = 1; k <= n; k++)
            a[i] += j * k;
}
```

```
\# of operations

\begin{align*}
\text{if (a > b && c < d)} \quad & ? \\
\text{for (int j = 0; j < n; j++)} \quad & ? \\
\text{a[i] += j;} \quad & ? \\
\end{align*}
```

```
\begin{align*}
\text{else} \quad & ? \\
\text{for (int j = 0; j < n; j++)} \quad & ? \\
\text{for (int k = 1; k <= n; k++)} \quad & ? \\
\text{a[i] += j * k;} \quad & ? \\
\end{align*}
```

```
\begin{align*}
T(n) = ?
\end{align*}
```
Average and Worst-Case Running Times

- Estimating the resource use of an algorithm is generally a theoretical framework and therefore a formal framework is required.
- Define some mathematical definitions.
- Average-case running time $T_{\text{avg}}(N)$
- Worst-case running time $T_{\text{worst}}(N)$
- $T_{\text{avg}}(N) \leq T_{\text{worst}}(N)$
- Average-case performance often reflects typical behavior of an algorithm.
- Worst-case performance represents a guarantee for performance on any possible input.
Average and Worst-Case Running Times (cont’d)

- Typically, we analyze worst-case performance
  - Worst-case provides a guaranteed upper bound for all input
  - Average-case is usually much more difficult to compute
Asymptotic Analysis of Algorithms

- We are mostly interested in the performance or behavior of algorithms for very large input (i.e., as $N \to \infty$)
  - For example, let $T(N) = 10,000 + 10N$ be the running time of an algorithm that processes $N$ transactions
  - As $N$ grows large ($N \to \infty$), the term $10N$ will dominate
  - Therefore, the smaller looking term $10N$ is more important if $N$ is large

- Asymptotic efficiency of the algorithms
  - How the running time of an algorithm increases with the size of the input \textit{in the limit}, as the size of the input increases without bound
Asymptotic Analysis of Algorithms (cont’d)

- Asymptotic behavior of $T(N)$ as $N$ gets big
- Exact expression for $T(N)$ is meaningless and hard to compare
- Usually expressed as fastest growing term in $T(N)$, dropping constant coefficients
  - For example, $T(N) = 3N^2 + 5N + 1$
  - Therefore, the term $N^2$ describes the behavior of $T(N)$ as $N$ gets big
Mathematical Background

- Let $T(N)$ be the running time of an algorithm
- Let $f(N)$ be another function (preferably simple) that we will use as a bound for $T(N)$
- Asymptotic notations
  - “Big-Oh” notation $O()$
  - “Big-Omega” notation $\Omega()$
  - “Big-Theta” notation $\Theta()$
  - “Little-oh” notation $o()$
“Big-Oh” notation

- Definition: \( T(N) = O(f(N)) \) if there are positive constants \( c \) and \( n_0 \) such that \( T(N) \leq cf(N) \) when \( N \geq n_0 \)
- Asymptotic upper bound on a function \( T(N) \)
- “The growth rate of \( T(N) \) is \( \leq \) that of \( f(N) \)”
  - Compare the relative rates of growth
- For example: \( T(N) = 10,000 + 10N \)
- Is \( T(N) \) bounded by Big-Oh notation by some simple function \( f(N) \)? Try \( f(N) = N \) and \( c = 20 \)
- See graphs on the next slide
Mathematical Background (cont’d)

\[ T(N) = 10,000 + 10N \]

\[ cf(N), \text{ where } f(N) = N \text{ and } c = 20 \]

Therefore, \( T(N) = O(f(N)) \), where \( f(N) = N \), \( c = 20 \), \( N \geq n_0 \), and \( n_0 = 1,000 \)

Simply, \( T(N) = O(N) \)

Check if \( T(N) \leq cf(N) \Rightarrow T(N) \leq 20N \) for large \( N \)
Mathematical Background (cont’d)

- “Big-Oh” notation
  - $O(f(N))$ is the set of all functions $T(N)$ that satisfy:
    - There exist positive constants $c$ and $n_0$ such that, for all $N \geq n_0$, $T(N) \leq cf(N)$
  - $O(f(N))$ is an uncountably infinite set of functions
“Big-Oh” notation

Examples

1,000,000N ∈ O(N)

Proof: Choose c = 1,000,000 and n₀ = 1

Thus, big-oh notation doesn’t care about (most) constant factors

It is unnecessary to write O(2N). We can just simply write O(N)

N ∈ O(N³)

Proof: Set c = 1, n₀ = 1

See graphs on the next slide

Big-Oh is an upper bound
Mathematical Background (cont’d)

- Graph of $N$ vs. $N^3$
Mathematical Background (cont’d)

“Big-Oh” notation

- Example
  - $N^3 + N^2 + N \in O(N^3)$
    - Proof: Set $c = 3$, and $n_0 = 1$

Big-Oh notation is usually used to indicate dominating (fastest-growing) term
Mathematical Background (cont’d)

- “Big-Oh” notation
  - Another example: $1,000N \in O(N^2)$
    - Proof: Set $n_0 = 1,000$ and $c = 1$
    - We could also use $n_0 = 10$ and $c = 100$
      - There are many possible pairs $c$ and $n_0$

  - Another example: If $T(N) = 2N^2$
    - $T(N) = O(N^4)$
    - $T(N) = O(N^3)$
    - $T(N) = O(N^2)$
      - All are technically correct, but the last one is the best answer
"Big-Omega" notation

- Definition: \( T(N) = \Omega(g(N)) \) if there are positive constants \( c \) and \( n_0 \) such that \( T(N) \geq cg(N) \) when \( N \geq n_0 \)

- Asymptotic lower bound

- “The growth rate of \( T(N) \) is \( \geq \) that of \( g(N) \)”

- Examples
  - \( N^3 = \Omega(N^2) \) (Proof: \( c = ? \), \( n_0 = ? \))
  - \( N^3 = \Omega(N) \) (Proof: \( c = 1 \), \( n_0 = 1 \))
Mathematical Background (cont’d)

- $g(N)$ is asymptotically upper bounded by $f(N)$
- $f(N)$ is asymptotically lower bounded by $g(N)$

$g(N) = O(f(N))$

$f(N) = \Omega(g(N))$
“Big-Theta” notation

- Definition: \( T(N) = \Theta(h(N)) \) if and only if \( T(N) = O(h(N)) \) and \( T(N) = \Omega(h(N)) \)
- Asymptotic tight bound
- “The growth rate of \( T(N) \) equals the growth rate of \( h(N) \)”
- Examples
  - \( 2N^2 = \Theta(N^2) \)
  - Suppose \( T(N) = 2N^2 \) then \( T(N) = O(N^4) \); \( T(N) = O(N^3) \); \( T(N) = O(N^2) \)
    all are technically correct, but last one is the best answer. Now
    writing \( T(N) = \Theta(N^2) \) says not only that \( T(N) = O(N^2) \), but also the
    result is as good (tight) as possible
Mathematical Background (cont’d)

- “Little-oh” notation
  - Definition: \( T(N) = o(g(N)) \) if for all constants \( c \) there exists an \( n_0 \) such that \( T(N) < cg(N) \) when \( N > n_0 \)
  - That is, \( T(N) = o(g(N)) \) if \( T(N) = O(g(N)) \) and \( T(N) \neq \Theta(g(N)) \)
  - The growth rate of \( T(N) \) less than (<) the growth rate of \( g(N) \)
  - Denote an upper bound that is not asymptotically tight

- The definition of \( O \)-notation and \( o \)-notation are similar
  - The main difference is that in \( T(N) = O(g(N)) \), the bound \( 0 \leq T(N) \leq cg(N) \) holds for some constant \( c > 0 \), but in \( T(N) = o(g(N)) \), the bound \( 0 \leq T(N) < cg(N) \) holds for all constants \( c > 0 \)
  - For example, \( N = o(N^2) \), but \( 2N^2 \neq o(N^2) \)
Examples

- \( N^2 = O(N^2) = O(N^3) = O(2^N) \)
- \( N^2 = \Omega(1) = \Omega(N) = \Omega(N^2) \)
- \( N^2 = \Theta(N^2) \)
- \( N^2 = o(N^3) \)
- \( 2N^2 + 1 = \Theta(?) \)
- \( N^2 + N = \Theta(?) \)
Mathematical Background (cont’d)

- $O()$ – upper bound
- $\Omega()$ – lower bound
- $\Theta()$ – tight bound
- $o()$ – strict upper bound
Mathematical Background (cont’d)

- **O-notation** gives an upper bound for a function to within a constant factor.
- **Ω-notation** gives a lower bound for a function to within a constant factor.
- **Θ-notation** bounds a function to within a constant factor.
  - The value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.
Rules of thumb when using asymptotic notations

- When asked to analyze an algorithm’s complexity
  - 1st preference: Use $\Theta()$  
  - 2nd preference: Use $O()$ or $o()$  
  - 3rd preference: Use $\Omega()$
Rules of thumb when using asymptotic notations

- Always express an algorithm’s complexity in terms of its worst-case, unless specified otherwise

  Note: Worst-case can be expressed in any of the asymptotic notations: \( O() \), \( \Omega() \), \( \Theta() \), or \( o() \)
Rules of thumb when using asymptotic notations

- Way’s 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
Mathematical Background (cont’d)

Some rules

- Rule 1: If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = O(f(N) + g(N))$ (less formally it is max $(O(f(N)), O(g(N)))$)
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$

- 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 $k$
  - Logarithm grows very slowly as $\log N \leq N$ for $N \geq 1$

- Rule 4: $\log_a N = \Theta(\log_b N)$ for any constants $a$ and $b$
Mathematical Background (cont’d)

- **Rate of Growth**

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td>Constant</td>
</tr>
<tr>
<td>(\log N)</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>(\log^2 N)</td>
<td>Log-squared</td>
</tr>
<tr>
<td>(N)</td>
<td>Linear</td>
</tr>
<tr>
<td>(N \log N)</td>
<td></td>
</tr>
<tr>
<td>(N^2)</td>
<td>Quadratic</td>
</tr>
<tr>
<td>(N^3)</td>
<td>Cubic</td>
</tr>
<tr>
<td>(2^N)</td>
<td>Exponential</td>
</tr>
</tbody>
</table>

The graph illustrates the growth rates of different functions. Functions labeled with a blue box are considered “useful,” while those labeled with a red box are considered “inefficient.”

- **Considered “efficient”**
- **Considered “useless”**
Some examples

- Prove that: \( n \log n = O(n^2) \).
  - We know that \( \log n \leq n \) for \( n \geq 1 \) (here, \( n_0 = 1 \)).
  - Multiplying both sides by \( n \): \( n \log n \leq n^2 \)

- Prove that: \( 6n^3 \neq O(n^2) \).
  - Proof by contradiction
  - If \( 6n^3 = O(n^2) \), then \( 6n^3 \leq cn^2 \)
Maximum subsequence sum problem

- Maximum subsequence sum problem
  - Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find the maximum value ($\geq 0$) of:

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

  - We don’t need the actual sequence $(i, j)$, just the sum
  - If the final sum is negative, the maximum sum is 0
  - E.g. $<1, -4, 4, 2, -3, 5, 8, -2>$

  The maximum sum is 16
Solution 1

MaxSubSum: Solution 1

- Idea: Compute the sum for all possible subsequence ranges \((i, j)\) and pick the maximum sum

```plaintext
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)
      maxSum = sum
return maxSum
```

All possible starting point
All possible ending point
Calculate sum for range \((i, j)\)

\(T(N) = O(N^3)\)
Algorithm 1

```c++
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     }
```
Solution 1 (cont’d)

- Analysis of Solution 1
  - Three nested *for* loops, each iterating at most N times
  - Operations inside *for* loops take constant time
  - But, *for* loops don’t always iterate N times
  - More precisely;

\[
T(N) = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} 1
\]
Solution 1 (cont’d)

- Analysis of Solution 1
  - Detailed calculation of $T(N)$
    $$T(N) = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^j 1$$
  - Will be derived in the class;
    $T(N) = (N^3 + 3N^2 + 2N)/6 = O(N^3)$
Solution 2

- MaxSubSum: Solution 2
  - Observation: \[ \sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k \]
  - So, we can re-use the sum from previous range

MaxSubSum2 (A)

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

- Time complexity: \( T(N) = O(N^2) \)
Algorithm 2

```cpp
/**
 * 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 (cont’d)

- Analysis of Solution 2
  - Two nested for loops, each iterating at most N times
  - Operations inside for loops take constant time
  - More precisely:

\[
T(N) = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} 1
\]
Solution 2 (cont’d)

- Analysis of Solution 2
  - Detailed calculation of T(N)
    \[ T(N) = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} 1 \]
  - Will be derived in the class;
    \[ T(N) = N(N+1)/2 = O(N^2) \]
Solution 3

- MaxSubSum: Solution 3
  - Idea: Recursive, “divide and conquer”
    - Divide sequence in half: $A_{1..\text{center}}$ and $A_{(\text{center} + 1)..N}$
    - Recursively compute MaxSubSum of left half
    - Recursively compute MaxSubSum of right half
    - Compute MaxSubSum of sequence constrained to use $A_{\text{center}}$ and $A_{(\text{center} + 1)}$
  - For example

\[ <1, -4, 4, 2, -3, 5, 8, -2> \]

- Compute maxsubsum\text{left}
- Compute maxsubsum\text{right}
- Compute maxsubsum\text{center}
Solution 3 (cont’d)

- MaxSubSum: Solution 3
  - Idea: Recursive, “divide and conquer”
  - Divide: split the problem into two roughly equal subproblems, which are then solved recursively
  - Conquer: patching together the two solutions of the subproblems, and possibly doing a small amount of additional work to arrive at a solution for the whole problem
  - The maximum subsequence sum can be in one of three places
    - Entirely in the left half of the input
    - Entirely in the right half
    - Or it crosses the middle and is in both halves
    - First two cases can be solved recursively
    - Last case: find the largest sum in the first half that includes the last element in the first half and the largest sum in the second half that includes the first element in the second half. These two sums then can be added together.
Example

For example, consider the sequence
4, -3, 5, -2 || -1, 2, 6, -4, where || marks the half-way point

- The maximum subsequence sum of the left half is 6: 4 + -3 + 5.
- The maximum subsequence sum of the right half is 8: 2 + 6.
- The maximum subsequence sum of sequences having -2 as the right edge is 4: 4 + -3 + 5 + -2; and the maximum subsequence sum of sequences having -1 as the left edge is 7: -1 + 2 + 6.
- Comparing 6, 8 and 11 (4 + 7), the maximum subsequence sum is 11 where the subsequence spans both halves: 4 + -3 + 5 + -2 + -1 + 2 + 6.
Solution 3 (cont’d)

MaxSubSum: Solution 3

MaxSubSum3(A, i, j)
    maxSum = 0
    if (i == j)
        if (A[i] > 0)
            maxSum = A[i]
        else
            k = floor((i + j) / 2)
            maxSumLeft = MaxSubSum3(A, i, k)
            maxSumRight = MaxSubSum4(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 maximum subsequence sum 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 );
/**
 * 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 (cont’d)

- Analysis of Solution 3
  - $T(1) = O(1)$
  - $T(N) = 2T(N / 2) + O(N)$
  - $T(N) = O(?)$
    - Will be derived in the class
Solution 4

MaxSubSum: Solution 4

Observations

- Any negative subsequence cannot be a prefix to the maximum subsequence
- Or, only a positive, contiguous subsequence is worth adding
- Example: <1, -4, 4, 2, -3, 5, 8, -2>

MaxSubSum4(A)

```c
/**
   * 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;
            else if( thisSum < 0 )
                sum = 0;
        }
    }
    return maxSum;
}
```

T(N) = O(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;
}
Solution 4 (cont’d)

- Online Algorithm
  - constant space and runs in linear time
  - just about as good as possible
# MaxSubSum Running Times

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Algorithm 1 ($O(N^3)$)</th>
<th>Algorithm 2 ($O(N^2)$)</th>
<th>Algorithm 3 ($O(N \log N)$)</th>
<th>Algorithm 4 ($O(N)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = 10$</td>
<td>0.000009</td>
<td>0.000004</td>
<td>0.000006</td>
<td>0.000003</td>
</tr>
<tr>
<td>$N = 100$</td>
<td>0.002580</td>
<td>0.000109</td>
<td>0.000045</td>
<td>0.000006</td>
</tr>
<tr>
<td>$N = 1,000$</td>
<td>2.281013</td>
<td>0.010203</td>
<td>0.000485</td>
<td>0.000031</td>
</tr>
<tr>
<td>$N = 10,000$</td>
<td>NA</td>
<td>1.2329</td>
<td>0.005712</td>
<td>0.000317</td>
</tr>
<tr>
<td>$N = 100,000$</td>
<td>NA</td>
<td>135</td>
<td>0.064618</td>
<td>0.003206</td>
</tr>
</tbody>
</table>

Time in seconds. Does not include time to read array.
MaxSubSum Running Times (cont’d)
MaxSubSum Running Times (cont’d)
Logarithmic Behavior

- $T(N) = O(\log_2 N)$

- An algorithm is $O(\log_2 N)$ if it takes constant $O(1)$ time to cut the problem size by a fraction (which is usually $\frac{1}{2}$)

- 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
**Binary Search**

- Given an integer $X$ and integers $A_0, A_1, \ldots, A_{N-1}$, which are \textit{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: scanning through the list from left to right and runs in linear time
  - Does not take advantage of the fact that list is sorted
  - Not likely to be best

- Better Strategy: Check if $X$ is the middle element
  - If so, the answer is at hand
  - If $X$ is smaller than middle element, apply the same strategy to the sorted subarray to the left of the middle element
  - Likewise, if $X$ is larger than middle element, we look at the right half

- $T(N) = O(\log_2 N)$
/**
 * 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 $\text{gcd}(M, N)$ between the integers $M$ and $N$
  - That is, the largest integer that divides both
  - Example: $\text{gcd} \ (50, 15) = 5$
  - Used in encryption
Euclid’s Algorithm (cont’d)

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

Example: gcd(3360,225)
- m = 3360, n = 225
- m = 225, n = 210
- m = 210, n = 15
- m = 15, n = 0
Euclid’s Algorithm (cont’d)

- Estimating the running time: how long the sequence of remainders is?
  - \( \log N \) is a good answer, but value of the remainder does not decrease by a constant factor
  - Indeed the remainder does not decrease by a constant factor in one iteration, however we can prove that after two iterations the remainder is at most half of its original value
  - Number of iterations is at most \( 2 \log N = O(\log N) \)

- \( T(N) = O(\log_2 N) \)
Euclid’s Algorithm (cont’d)

- **Analysis**
  - Note: After two iterations, remainder is at most half its original value
    - Theorem 2.1: If $M > N$, then $M \mod N < M / 2$
  - $T(N) = 2 \log_2 N = O(\log_2 N)$
    - $\log_2 225 = 7.8$, $T(225) = 16$ (overestimate)
  - 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 $X^N = X \times X \times \ldots \times X$ (N times), integer N
- Obvious algorithm:
  - To compute $X^N$ uses $(N-1)$ multiplications
- Observations
  - A recursive algorithm can do better
  - $N \leq 1$ is the base case
  - $X^N = X^{N/2} \times X^{N/2}$ (for even N)
  - $X^N = X^{(N-1)/2} \times X^{(N-1)/2} \times X$ (for odd N)
- Minimize number of multiplications
- $T(N) = 2 \log_2 N = O(\log_2 N)$

```python
def pow(x, n):
    result = 1
    for i in range(1, n):
        result *= x
    return result
```
Exponentiation (cont’d)

```c
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;
}
```

- \( T(N) = \Theta(1), N \leq 1 \)
- \( T(N) = T(N/2) + \Theta(1), N > 1 \)
- \( T(N) = O(\log_2 N) \)
- \( T(N) = \Theta(\log_2 N) \)
Exponentiation (cont’d)

To compute $X^{62}$, the algorithm does the following calculations, which involve only 9 multiplications.

- $X^3 = (X^2) \cdot X$ then $X^7 = (X^3)^2 \cdot X$ then $X^{15} = (X^7)^2 \cdot X$ then $X^{31} = (X^{15})^2 \cdot X$ then $X^{62} = (X^{31})^2$

- The number of multiplications required is at most $2 \log N$, because at most 2 multiplications (if $N$ is odd) are required to halve the problem.
Summary

- Algorithm analysis
- Bound running time as input gets big
- Rate of growth: $O()$ and $\Theta()$
- Compare algorithms
- Recursion and logarithmic behavior