Algorithm Analysis

CptS 223 – Advanced Data Structures

Larry Holder
School of Electrical Engineering and Computer Science
Washington State University
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
  - Speed up critical algorithms
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
  - Serial
  - Parallel
  - Quantum
What to Analyze

- Main focus is on running time
  - Memory/time tradeoff
  - Memory is cheap ($20 per GB)

- Simple serial computing model
  - Single processor, infinite memory

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

- Running time $T(N)$
  - $N$ is typically 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
Example

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

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

\[ T(N) = 6N+4 \]
What to Analyze

- Worst-case running time $T_{\text{worst}}(N)$
- Average-case running time $T_{\text{avg}}(N)$
- $T_{\text{avg}}(N) \leq T_{\text{worst}}(N)$
- Typically analyze worst-case behavior
  - Average case hard to compute
  - Worst-case gives guaranteed upper bound
Rate of Growth

- Exact expressions for $T(N)$ meaningless and hard to compare
- Rate of growth
  - Asymptotic behavior of $T(N)$ as $N$ gets big
  - Usually expressed as fastest growing term in $T(N)$, dropping constant coefficients
  - E.g., $T(N) = 3N^2 + N + 1 \rightarrow \Theta(N^2)$
Rate of Growth

- $T(N) = O(f(N))$ if there are positive constants $c$ and $n_0$ such that $T(N) \leq cf(N)$ when $N \geq n_0$
  - Asymptotic upper bound
  - “Big-Oh” notation

- $T(N) = \Omega(g(N))$ if there are positive constants $c$ and $n_0$ such that $T(N) \geq cg(N)$ when $N \geq n_0$
  - Asymptotic lower bound
  - “Big-Omega” notation
Rate of Growth

\[ g(N) = O(f(N)) \]

\[ f(N) = \Omega(g(N)) \]
Rate of Growth

- \( T(N) = \Theta(h(N)) \) if and only if \( T(N) = O(h(N)) \) and \( T(N) = \Omega(h(N)) \)
  - Asymptotic tight bound
- \( T(N) = o(p(N)) \) if for all constants \( c \) there exists an \( n_0 \) such that \( T(N) < cp(N) \) when \( N>n_0 \)
  - I.e., \( T(N) = o(p(N)) \) if \( T(N) = O(p(N)) \) and \( T(N) \neq \Theta(p(N)) \)
  - “Little-oh” notation
Rate of Growth

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

- Rule 1: If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = O(f(N) + g(N))$
  - $T_1(N) \times T_2(N) = O(f(N) \times g(N))$

- 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 $k$
# 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>

![Graph showing different growth rates](graph.png)
Example

- Maximum subsequence sum problem
  - Given N integers $A_1, A_2, ..., A_N$, find the maximum value ($\geq 0$) of:
    \[
    \sum_{k=i}^{j} A_k
    \]
  - Don’t need the actual sequence $(i,j)$
  - E.g., $<1, -4, 4, 2, -3, 5, 8, -2> \rightarrow 16$
MaxSubSum: Solution 1

- Compute each possible subsequence independently

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
18

/**
 * 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

- Three nested for loops, each iterating at most N times
- Operations inside for loops take constant time
- Thus, T(N) = ?
- But, for loops don’t always iterate N times
Solution 1: Analysis

- More precisely

\[ T(N) = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} \Theta(1) \]
MaxSubSum: Solution 2

- Note that $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$
- So, no reason to re-compute sum each time

```
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
```
/**
 * 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

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

\[ T(N) = \Theta(N^2) \]
MaxSubSum: Solution 3

- 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 \( A_{(\text{center}+1)} \)

- E.g., \(<1, -4, 4, 2, -3, 5, 8, -2>\)
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 );
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: Analysis

- $T(1) = \Theta(1)$
- $T(N) = 2T(N/2) + \Theta(N)$
- $T(N) = \Theta(?)$
MaxSubSum: Solution 4

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

- \( T(N) = ? \)

MaxSubSum4 \((A)\)
\[
\begin{align*}
\text{maxSum} &= 0 \\
\text{sum} &= 0 \\
\text{for } j &= 1 \text{ to } N \\
\text{sum} &= \text{sum} + A[j] \\
\text{if } (\text{sum} > \text{maxSum}) \\
\text{then } \text{maxSum} &= \text{sum} \\
\text{else if } (\text{sum} < 0) \\
\text{then } \text{sum} &= 0 \\
\text{return } \text{maxSum}
\end{align*}
\]
/**
 * 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;
}
MaxSubSum Running Times

<table>
<thead>
<tr>
<th>Input Size</th>
<th>1 (O(N^3))</th>
<th>2 (O(N^2))</th>
<th>3 (O(N \log N))</th>
<th>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

![Graph showing running times for MaxSubSum algorithms]

- Linear
- \(O(N \log N)\)
- Quadratic
- Cubic

Input Size (N) vs. Running Time
MaxSubSum Running Times

![Graph showing the running times for linear, $O(N \log N)$, quadratic, and cubic complexities.](image)
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
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.

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

- $T(N) = \Theta(\log_2 N)$?
/**
 * 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

```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: 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
  - $X^N = X^{N/2} \times X^{N/2}$ (for even $N$)
  - $X^N = X^{(N-1)/2} \times X^{(N-1)/2} \times X$ (for odd $N$)

- Minimize multiplications $T(N)$

- $T(N) = 2 \log_2 N = O(\log_2 N)$
Exponentiation

1 long pow( long x, int n )

T(N) = Θ(1), N ≤ 1
T(N) = T(N/2) + Θ(1), N > 1
T(N) = O(log₂ N)
T(N) = Θ(log₂ N)

2 {

T(N) = Θ(1), N ≤ 1
T(N) = T(N/2) + Θ(1), N > 1
T(N) = O(log₂ N)
T(N) = Θ(log₂ N) ?

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 }
Summary

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