Chapter 2
Algorithm Analysis

Introduction

- algorithm
- set of simple ___________ to solve a problem
- analyzed in terms, such as _____ and ________, 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 ___________

- big-O  \( O(f(N)) \)
- big-omega  \( \Omega(f(N)) \)
- big-theta  \( \Theta(f(N)) \)

Asymptotic Notation

- formal definitions
  \[
  f(n) = O(g(n)) \\
  f(n) = \Omega(g(n)) \\
  f(n) = \Theta(g(n))
  \]

\( f(n) \) and \( g(n) \) are ___________ 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 _________________

Asymptotic Notation: Big-\( O \)

\[
f(n) = O(g(n)) \text{ if there exist constants } c \text{ and } N \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n > N
\]

loosely speaking, \( f(n) = O(g(n)) \) is an analog of _______

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: $\Omega$

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

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

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

Asymptotic Notation: $\Theta$

$f(n) = \Theta(g(n))$ 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, $f(n) = \Theta(g(n))$ is an analog of ______

Relationship of $O$, $\Omega$, and $\Theta$

- note that

$$f(n) = O(g(n)) \iff g(n) = \_$$

since

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

- also

$$f(n) = \Theta(g(n)) \iff f(n) = \_ \text{ and } f(n) = \_$$

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 ______ bound on $f$ in terms of $g$

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

$f(n) = \Theta(g(n))$ gives both an ______ and ______ bound on $f$ in terms of $g$
Asymptotic Notation: Second Observation

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

\[
\begin{align*}
n^2 + n &< 4n^2 \text{ for all } n > 1 \\
n^2 + n &< 2n^2 \text{ for all } n > 1 \\
n^2 + n &< 1.01n^2 \text{ for all } n > 100 \\
n^2 + n &< 1.000001n^2 \text{ for all } n > 1000
\end{align*}
\]

Asymptotic Notation: Third Observation

- we typically _________ 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:

\[
\begin{align*}
f(n) &= O(1,000n^2) \quad \text{true, but bad form} \\
f(n) &= O(n^2 + n) \quad \text{true, but bad form} \\
f(n) &< O(n^2) \quad \text{__________}
\end{align*}
\]

Asymptotic Notation: Fourth Observation

\( O(g(n)) \) assures that you can’t do any _____ than \( g(n) \), but it can be unduly pessimistic

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

\( \Theta(g(n)) \) is the _________ 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 $\Omega$

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 $n$, _______ – 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 _______ as possible!

$\Theta$ is the Most Informative

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

$$c_1 \cdot g(n) < f(n) < c_2 \cdot g(n)$$

for all $n > N$

example: $2n^2 + n = ______$

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

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 _______)

(b) $T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$

Rule 2

if $T(n)$ is a polynomial of degree $k$, then $T(n) = ______$

Rule 3

$log^k n = ______$ for any constant $k$ (logarithms grow very slowly)
Typical Growth Rates

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>_______</td>
</tr>
<tr>
<td>$\log n$</td>
<td>logarithmic</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>_______</td>
</tr>
<tr>
<td>$n^3$</td>
<td>cubic</td>
</tr>
<tr>
<td>$2^n$</td>
<td>_______</td>
</tr>
</tbody>
</table>

Complexity and Tractability

The order of an algorithm is useful for 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?
  - Quadratic?
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 ________ of computation
  - standard computer with ______________ 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 ____ as

  $$x^n \leftarrow x \times x \times \cdots \times x \times x$$

  there are $n - 1$ ____________, so there are $n - 1$ basic operations to compute $x^n$ plus one more basic operation to ______________ the result

What to Analyze

different types of performance are analyzed
  - best case
    - may not represent ________ behavior
  - average case
    - often reflects typical behavior
    - difficult to determine what is average
  - worst case
    - ____________ 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, \( A_1, A_2, \ldots, A_N \), find the maximum value of
\[
\sum_{k=i}^{j} A_k
\]
- e.g., for input -2, 11, -4, 13, -5, -2, the answer is ____ (\( A_2 \) through \( A_4 \))

many algorithms to solve this problem
-we will focus on four with run time in the table below

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Algorithm Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N = 10 )</td>
<td>( O(N^3) )</td>
</tr>
<tr>
<td>( N = 100 )</td>
<td>0.000009</td>
</tr>
<tr>
<td>( N = 1,000 )</td>
<td>2.281013</td>
</tr>
<tr>
<td>( N = 10,000 )</td>
<td>NA</td>
</tr>
<tr>
<td>( N = 100,000 )</td>
<td>NA</td>
</tr>
</tbody>
</table>

notes
- computer actually run on __________ since we’re looking to compare algorithms
- for _________ inputs, all algorithms perform well
- times do not include ______________
  - 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 (_______)
- algorithm 2 is __________; therefore an increase of input by a factor of 10 yields a run time factor of 100
- algorithm 1 (_______): run time factor of 1000

Do not include array read times.
Maximum Subsequence Sum Problem

Running Time Calculations

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

- ____________
  - as in previous results
  - realistic measurement

- ____________
  - helps to eliminate bad algorithms early when several algorithms are being considered
  - analysis itself provides insight into ___________ efficient algorithms
  - pinpoints ____________, which need special coding

A Simple Example

calculate ___________

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

A Simple Example

analysis

- declarations count for ______; ignore call costs
  - lines 1 and 4 count for 1 time unit each
  - line 3 counts for 4 time units
    - 2 ____________
      - 1 addition
      - 1 assignment
    - line 2 has hidden costs
      - ____________ i (1 time unit)
      - testing i ≤ n (N + 1)
      - incrementing i (N)
    - total: ___________ 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 _______ to help
- line 3 counts for 4 time units, but is $O(1)$
- line 1 is _______________ compared to the for loop

General Rules

concentrate on _______ and __________________
- Rule 1 – FOR loops
  - running time of the statements inside (including tests) times the number of iterations
- Rule 2 – Nested loops
  - analyze from the _______________
  - running time of the statements times the product of the sizes of all the loops
  - watch out for loops that
    - contribute only a ____________ amount of computation
    - contain ________ statements

Single Loops

```java
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

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

- ____________ amount of computation inside nested loop
- must multiply number of times _______ 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
  ____________
Nested Loops

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

- outer loop is executed a fixed number of times
  - ______

General Rules (cont.)

concentrate on loops and recursive calls
- Rule 3 – Consecutive statements
  - simply _____ the running times

```c
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 ______
```

General Rules (cont.)

concentrate on loops and recursive calls
- Rule 4 – if/else

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

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

```c
long factorial (int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial (n - 1);
}
```
previous example not a good use of ____________
- if recursion used properly, difficult to convert to a simple ____________ structure

what about this example?
```c
long fib (int n)
{
    if (n <= 1)
        return 1;
    else
        return fib (n - 1) + fib (n - 2);
}
```
even worse!
- extremely inefficient, especially for values ________
- 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
  
  $T(N) = T(N - 1) + T(N - 2) + 2$ for $N \geq 2$
  - we have seen that this algorithm ________________

  - similarly, we could show it is $> \left(\frac{3}{2}\right)^N$
  - ________________

with current recursion
- lots of redundant work
  - violating ________________ rule
  - ________________ thrown away

running time can be reduced substantially with simple for loop

example: Maximum Subsequence Sum Problem
- given (possibly negative) integers, $A_1, A_2, \ldots, A_N$, 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

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

Do not include array read times.

Algorithm 1

exhaustively tries all ________________
– first loop iterates __ times
– second loop iterates ____ 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
**Algorithm 1**

more precise calculation

- use ____________ 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 ____________
  \[ \sum_{k=i}^{j} 1 = j - i + 1 \]

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

**Algorithm 2**

to speed up the algorithm, we can ____________ 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 ____________

**Algorithm 1**

more precise calculation (cont.)

- to complete the calculation

\[
\sum_{i=0}^{N-1} \frac{(N - i + 1)(N - i)}{2} = \sum_{i=1}^{N} \frac{(N - i + 1)(N - i + 2)}{2} \\
= \frac{1}{2} \sum_{i=1}^{N} i^2 - \frac{(N + 3)}{2} \sum_{i=1}^{N} i + \frac{1}{2}(N^2 + 3N + 2) \sum_{i=1}^{N} 1 \\
= \frac{1}{2} \frac{N(N+1)(2N+1)}{6} - \frac{(N + 3)}{2} \frac{N(N+1)}{2} + \frac{N^2 + 3N + 2}{6} \frac{N}{N} \\
= \frac{N^3 + 3N^2 + 2N}{6}
\]

**Algorithm 2**

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

Algorithm runs even faster
- divide and conquer strategy
  - divide: problem into two roughly equal subproblems
  - conquer: solutions to subproblems
- maximum subsequence can be in one of three places
  - entirely in half of input
  - entirely in half of input
  - in both halves, 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 (cont.)

Algorithm 3 (cont.)

Consider the following example

<table>
<thead>
<tr>
<th>First Half</th>
<th>Second Half</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 -3 -5 -2</td>
<td>-1 2 6 -2</td>
</tr>
</tbody>
</table>

- 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: __
  - maximum subsequence in the second half that includes the first element in the second half: __
  - total: __

Algorithm 3 (cont.)

Algorithm 3 (cont.)

```c++
/**
 * 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 leftBorderSum = 0, leftBorderMax = 0;
  for( int i = center; i >= left; i-- )
    if( leftBorderSum + a[i] > leftBorderMax )
      leftBorderMax = leftBorderSum + a[i];
  int rightBorderSum = 0, rightBorderMax = 0;
  for( int i = center + 1; i <= right; i++ )
    if( rightBorderSum + a[i] > rightBorderMax )
      rightBorderMax = rightBorderSum + a[i];
  int maxRightBorderSum = rightBorderMax;
  maxRightBorderSum = maxRightSum = max( maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum );
  return max3( maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum );
}
```

/**
 * Driver for divide-and-conquer maximum contiguous subsequence sum algorithm.
 */
int maxSubSum3( const vector<int> &a )
{
  int maxLeftBorderSum = 0, rightBorderSum = 0;
  for( int j = center + 1; j <= right; ++j )
    for( int i = center; i >= left; i-- )
      if( rightBorderSum + a[j] > rightBorderMax )
        rightBorderMax = rightBorderSum + a[j];
  int maxRightBorderSum = 0, leftBorderSum = 0;
  for( int i = center; i <= right; i++ )
    if( leftBorderSum + a[i] > leftBorderMax )
      leftBorderMax = leftBorderSum + a[i];
  int maxRightBorderSum = rightBorderMax;
  maxRightBorderSum = maxRightSum = max( maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum );
  return max3( maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum );
}
Algorithm 3

notes
- driver function needed
- gets solution started by calling function with initial parameters that _______ entire array
- if left == right
  - _______ case
- _______ 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 _______
- otherwise, the program must perform two recursive calls
- the two for loops access each element in the subarray, with _______ work; thus \( O(N) \)
- all other non-recursive work in the function is constant and can be _________
- 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 _______

where \( T(1) = 1 \)

- note that

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

and

\[
T(n) = T(n-1) + cn \quad \text{is} \quad T(n) = cn(n) \quad \text{or just} \quad O(n^2)
\]

we can solve the recurrence relation directly (later)

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

- for now, note that

\[
T(1) = 1
\]

\[
T(2) = 4 = 2 \times 2
\]

\[
T(4) = 12 = 4 \times 3
\]

\[
T(8) = 32 = 8 \times 4
\]

\[
T(16) = 80 = 16 \times 5
\]

thus, if ________,

\[
T(N) = N \times (k + 1) = N \log N + N = O(N \log N)
\]
Algorithm 4

Algorithm 4

notes

Algorithm 4

logarithms will show up regularly in analyzing algorithms

Algorithm 4

Logarithms in the Running Time

logarithms will show up regularly in analyzing algorithms

Algorithm 4

Logarithms in the Running Time

logarithms will show up regularly in analyzing algorithms

Algorithm 4

Logarithms in the Running Time

logarithms will show up regularly in analyzing algorithms
**Binary Search**

binary search
- given an integer $X$ and integers $A_0, A_1, ..., A_{N-1}$, which are _________ 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 ________ time
  - does not take into account that elements are presorted
  - better solution: check if $X$ is in the _________ element
    - if yes, done
    - if smaller, apply same strategy to sorted _____ subarray
    - if greater, use _________ subarray

**Analysis**

- all the work done inside the loop takes ______ time per iteration
  - first iteration of the loop is for $N - 1$
    - subsequent iterations of the loop ______ 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 ___ accesses would be required

**Euclid’s Algorithm**

Euclid’s algorithm
- computes gcd (greatest common divisor) of two integers
  - largest integer that ____________ both
Euclid's Algorithm

notes
- algorithm computes gcd$(M, N)$ assuming $M \geq N$
  - if $N > M$, the values are ___________
- algorithm works by continually computing __________ until 0 is reached
  - the last __________ 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 ________ in a uniform way
- we can prove that the remainder is at most ______ of its original value after ______ 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 $M$ ______ 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 (____________) is $(12 \ln 2 \ln N)/\pi^2 + 1.47$
Exponentiation

exponentiation
- raising an integer to an __________ power
- results are often large, so machine must be able to support such values
- number of __________________ 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 ________, $X^N = X^{N/2} \cdot X^{N/2}$
  - if $N$ is ________, $X^N = X^{(N-1)/2} \cdot X^{(N-1)/2} \cdot X$
  - example: $X^{62} = (X^{31})^2$
  - $X^3 = (X^2)X$, $X^7 = (X^3)^2X$, $X^{15} = (X^7)^2X$, $X^{31} = (X^{15})^2X$

Exponentiation

analysis
- number of multiplications: $2\log N$
  - at most, ___ multiplications are required to ______ 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 ______________
- analysis can be tightened
- average running time may be significantly less than worst-case running time
  - often, the worst-case is achievable by very __________,
    but still a large overestimate in the __________ case
  - unfortunately, average-case analysis is extremely complex (and in many cases currently unsolved), so worst-case is the best that we’ve got

```c
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 }
```