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

Many algorithms can exist to solve one problem.

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

Many algorithms can exist to solve one problem.

Problem – Algorithm – Data Structures & Techniques

Problem (e.g., searching) solves Algorithm (e.g., binary search)

specification (Input => Output)

contains

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
  - ...

Copyright © 1997 United Feature Syndicate, Inc. Redistribution in whole or in part prohibited
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
Supercomputer speed is measured in number of floating point operations per sec (or FLOPS).

Other Models

- 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 $T(n)$?

![Graph showing running time vs input size with different growth rates: Linear, $O(N \log N)$, Quadratic, Cubic.](image)
Example for calculating $T(n)$

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

$T(n) = 6n+4$
Example: Another less-precise but equally informative analysis

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

<table>
<thead>
<tr>
<th>#operations</th>
<th>T(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\propto n$</td>
</tr>
<tr>
<td>0</td>
<td>$\propto n$</td>
</tr>
<tr>
<td>1</td>
<td>$\propto n^2$</td>
</tr>
<tr>
<td>$n$</td>
<td>$\propto n$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$\propto n^2$</td>
</tr>
<tr>
<td>1</td>
<td>$\propto n$</td>
</tr>
</tbody>
</table>

$T(n) \propto n$
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 $\Theta()$

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

Describes the input

- “Algorithms”
- Lower-bound
- Upper-bound
- Tight-bound
- “Optimality”

Describes the problem & its solution

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

<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>

(curves not to scale)

Initial aberrations do NOT matter!
Big-oh : \( O() \)

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

- Asymptotic upper bound, possibly tight
Example for big-Oh

- E.g., let \( f(n) = 10n \)
  - \( \Rightarrow f(n) = O(n^2) \)

**Proof:**
- If \( f(n) = O(g(n)) \)
  - \( \Rightarrow f(n) \leq c \cdot g(n) \), for all \( n > n_0 \)
  - (show such a \( \langle c, n_0 \rangle \) combination exists)
  - \( \Rightarrow c = 1, n_0 = 9 \)
- (Remember: always try to find the lowest possible \( n_0 \))

Breakeven point \( (n_0) \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 10n )</th>
<th>( c \cdot n^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
<td>81</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>121</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\( c = 1 \)
Ponder this

If \( f(n) = 10n \), 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-o : o()

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

- E.g., $f(n) = 10n$; $g(n) = n^2$
  - $\Rightarrow f(n) = o(g(n))$
Big-omega: \( \Omega() \)

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

Lower bound for \( f(N) \), possibly tight

- E.g., \( f(n) = 2n^2; \ g(n) = n \log n \)
  \[ \implies f(n) = \Omega(g(n)) \]
Little-omega : $\omega()$

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

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

- \( f(N) = \Theta(h(N)) \) 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)

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

Tight bound for \( f(N) \)
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:
  - $c_1 n^2 \leq f(n) \leq c_2 n^2$
  - $c_1 n^2 \leq n^2 - 2n \leq c_2 n^2$
  - ...
## The Asymptotic Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Purpose</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>Upper bound, possibly tight</td>
<td>$f(n) \leq c \cdot g(n)$</td>
</tr>
<tr>
<td>$\Omega(n)$</td>
<td>Lower bound, possibly tight</td>
<td>$f(n) \geq c \cdot g(n)$</td>
</tr>
<tr>
<td>$\Theta(n)$</td>
<td>Tight bound</td>
<td>$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$</td>
</tr>
<tr>
<td>$o(n)$</td>
<td>Upper bound, strict</td>
<td>$f(n) &lt; c \cdot g(n)$</td>
</tr>
<tr>
<td>$\omega(n)$</td>
<td>Lower bound, strict</td>
<td>$f(n) &gt; c \cdot g(n)$</td>
</tr>
</tbody>
</table>
BLAH, BLAH, BLAH, BLAH... I HAVE NO IDEA WHAT HE'S TALKING ABOUT.

DILBERT
By Scott Adams
Asymptotic Growths

Asymptotic Notations:

- \( f(n) = O( g(n) ) \)
- \( f(n) = \Omega( g(n) ) \)
- \( f(n) = o( g(n) ) \)
- \( f(n) = \omega( g(n) ) \)
- \( f(n) = \Theta( g(n) ) \)
Rules of Thumb while using Asymptotic Notations

Algorithm’s complexity:

- When asked to analyze an algorithm’s complexities:
  - 1st Preference: Whenever possible, use \( \Theta() \)
  - 2nd Preference: If not, use \( O() \) - or \( o() \)
  - 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

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

- Is \( f(n) = \Theta(f(n)) \)?
  - yes
- Is \( f(n) = O(f(n)) \)?
  - yes
- Is \( f(n) = \Omega(f(n)) \)?
  - yes
- Is \( f(n) = o(f(n)) \)?
  - no
- Is \( f(n) = \omega(f(n)) \)?
  - no
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

- \( f(n) = \Theta(g(n)) \) and \( g(n) = \Theta(h(n)) \) \( \Rightarrow \) \( f(n) = \Theta(h(n)) \)

- \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \) \( \Rightarrow \) ?

- \( f(n) = \Omega(g(n)) \) and \( g(n) = \Omega(h(n)) \) \( \Rightarrow \) ?

- ...
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: $T_1(n) \times T_2(n) = O(f(n) \times g(n))$

- **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 $k$

- **Rule 4:** $\log_a n = \Theta(\log_b n)$ for any constants $a$ & $b$
Some More Proofs

- **Prove that:** \( n \log n = O(n^2) \)
  - We know \( \log n \leq n \), for \( n \geq 1 \) (\( \Rightarrow n_0 = 1 \))
  - Multiplying \( n \) on both sides:
    - \( n \log n \leq n^2 \)
    - \( \Rightarrow n \log n \leq 1 \cdot n^2 \)
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 ❌
Maximum subsequence sum problem

- Given $N$ integers $A_1, A_2, \ldots, A_N$, find the maximum value ($\geq 0$) 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., $[1, -4, 4, 2, -3, 5, 8, -2]$ 
  16 is the answer
MaxSubSum: Solution 1

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

\[
\text{MaxSubSum1} \ (A) \\
\text{maxSum} = 0 \\
\text{for } i = 1 \text{ to } N \\
\quad \text{for } j = i \text{ to } N \\
\quad \quad \text{sum} = 0 \\
\quad \quad \text{for } k = i \text{ to } j \\
\quad \quad \quad \text{sum} = \text{sum} + A[k] \\
\quad \quad \text{if } (\text{sum} > \text{maxSum}) \\
\quad \quad \quad \text{then maxSum} = \text{sum} \\
\quad \text{return maxSum}
\]

- All possible start points
- All possible end points
- Calculate sum for range \([i..j]\)

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

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

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

Observation: \[ \sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k \]

\[ \Rightarrow \] 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
```

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
```

Sum (new k) = sum (old k) + A[k]

\[ \Rightarrow \] So NO need to recompute sum for range A[i..k-1]

Cpt S 223. School of EECS, WSU
/**
 * 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 $\max\{\text{left\_max}, \text{right\_max}, \text{center\_max}\}$

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

Cpt S 223. School of EECS, WSU
MaxSubSum: Solution 3

\[
\text{MaxSubSum3} (A, i, j) \\
\text{maxSum} = 0 \\
\text{if} \ (i = j) \\
\text{then if} \ A[i] > 0 \\
\text{then maxSum} = A[i] \\
\text{else} \ k = \text{floor}((i+j)/2) \\
\text{maxSumLeft} = \text{MaxSubSum3}(A, i, k) \\
\text{maxSumRight} = \text{MaxSubSum3}(A, k+1, j) \\
\text{compute maxSumThruCenter} \\
\text{maxSum} = \text{Maximum} (\text{maxSumLeft}, \text{maxSumRight}, \text{maxSumThruCenter}) \\
\text{return maxSum}
\]
/**
 * 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 );
38    /**
39    * Driver for divide-and-conquer maximum contiguous
40    * subsequence sum algorithm.
41    */
42    int maxSubSum3( const vector<int> & a )
43    {
44        return maxSumRec( a, 0, a.size()-1 );
45    }
Solution 3: Analysis

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

Can we do even better?

$T(N) = \Theta(N \log N)$
MaxSubSum: Solution 4

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>-4</th>
<th>4</th>
<th>2</th>
<th>-3</th>
<th>5</th>
<th>8</th>
<th>-2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sum</strong></td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>8</td>
<td><strong>16</strong></td>
<td>14</td>
</tr>
</tbody>
</table>

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

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: $\text{Max}(i) \leq \text{maxsubsum}$ for range $A[1..i]$
  - $\implies \text{Max}(N)$ is the final answer
- Let; $\text{Max}’(i) \leq \text{maxsubsum}$ ending at $A[i]$
- Base case:
  - $\text{Max}(1) = = \text{Max}’(1) = \max \{ 0, A[1] \}$
- Recurrence (i.e., FOR $i=2$ to $N$):
  - $\text{Max}(i) = ?$
    - $= \max \{ \text{Max}’(i), \text{Max}(i-1) \}$
  - $\text{Max}’(i) = ?$
    - $= \max \{ \text{Max}’(i-1) + A[i], A[i], 0 \}$

This technique is called “dynamic programming”
Algorithm 5 pseudocode

Dynamic programming: (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?

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 and 5 $O(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = 10$</td>
<td>0.0000009</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>

Do not include array read times.

38 min

26 days
MaxSubSum Running Times

![Graph showing running times for different complexities: Linear, O(N log N), Quadratic, and Cubic.](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

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 presorted and already in memory, find \( i \) such that \( A_i = X \), or return \( i = -1 \) if \( X \) is not in the input.

- \( T(N) = O(\log_2 N) \)
/**
 * 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 $M \mod N < M/2$
- $T(N) = 2 \log_2 N = O(\log_2 N)$
  - $\log_2 225 \approx 7.8$, $T(225) = 16$ (?)
- 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:
  ```python
  def 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

```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;
}
```

- $T(N) = \Theta(1)$, $N \leq 1$
- $T(N) = T(N/2) + \Theta(1)$, $N > 1$
- $T(N) = O(\log_2 N)$
- $T(N) = \Theta(\log_2 N)$?
Summary

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