Efficiency of Algorithms

- Readings: [SG] Ch. 3

- Chapter Outline:
  - Attributes of Algorithms
  - Measuring Efficiency of Algorithms
  - Simple Analysis of Algorithms
  - Polynomial vs Exponential Time Algorithms

Still Updating! (07 Sep 2016)
Efficiency of Algorithms

- **Readings:** [SG] Ch. 3

- **Chapter Outline**
  - Attributes of Algorithms
    - *What makes a Good Algorithm*
    - *Key Efficiency considerations*
  - Measuring Efficiency of Algorithms
  - Simple Analysis of Algorithms
  - Polynomial vs Exponential Time Algorithms
What are good algorithms?

- Desirable attributes in an algorithm
  - Correctness
  - Simplicity (Ease of understanding)
  - Elegance
  - Efficiency
  - *Embrace Multiple Levels of Abstraction*
  - *Well Documented, Multi-Platform*
Attributes: Correctness, Simplicity

- **Correctness**
  - Does algorithm *solve the problem* it is designed for?
  - Does the algorithm solve *all instances* of the problem correctly?

- **Simplicity (Ease of understanding)**
  - Is it *easy to understand*,
  - Is it *clear, concise* (not tricky)
  - Is it *easy to alter* the algorithm?
  - Important for program maintenance
Attributes: Abstraction, Elegance

- Multiple Levels of Abstraction
  - Decomposes problem into sub-problems
  - Easier to understand at different levels of abstraction?
  - Usable as modules (libraries) by others

- Elegance
  - How clever or sophisticated is the algorithm?
  - Is pleasing and satisfying to the designer.
Attributes: *Efficiency*, etc…

- **Efficiency**
  - The *amount of time* the algorithm requires?
  - The *amount of space* the algorithm requires?
  - The *most important* attribute
  - Especially for large-scale problems.

- **Well documented, multi-platform**
  - Is *well-documented* with sufficient details
  - Not OS-dependent, company-dependent, computer-dependent
When designing algorithms, all computer scientists strive to achieve

Simplicity, Elegance

plus

Efficiency

However, they are often **contradictory**

- Simple algorithms are often slow
- Efficient algorithms tend to be complicated

To really master this, take an **algorithms course**.
Given an algorithmic problem, there can be many different algorithms to solve it.

**Problem:** Searching for a name in a list;

**Algorithms:**
- Sequential Search: Slow, Takes **linear** time
- Binary Search: Fast, Takes **logarithmic** time
- Interpolation Search, etc
- Hashing

Not covered in UIT2201
Sequential Search: Idea

- Search for $NAME$ among a list of $n$ names

- Start at the beginning and compare $NAME$ to each entry in the list until a match is found
Sequential Search: Pseudo-Code

1. Get values for NAME, n, N₁, . . . , Nₙ and T₁, . . . , Tₙ
2. Set the value of i to 1 and set the value of Found to NO
3. While (Found = NO) and (i ≤ n) do steps 4 through 7
4. If NAME is equal to the i-th name on the list, Nᵢ, then
5. Print the telephone number of that person, Tᵢ
6. Set the value of Found to YES
   Else (NAME is not equal to Nᵢ)
7. Add 1 to the value of i
8. If (Found = NO) then
9. Print the message ‘Sorry, this name is not in the directory’
10. Stop

Figure 3.1: Sequential Search Algorithm
Recall: Algorithm Sequential Search

- **Precondition:** The variables n, NAME and the arrays N and T have been read into memory.

```plaintext
Seq-Search(N, T, n, NAME);
begin
    i ß 1;
    Found ß No;
    while (Found=No) and (i <= n) do
        if (NAME = N[i])
            then Print T[i]; Found ß Yes;
        else i ß i + 1;
        endif
    endwhile
    if (Found=No) then
        Print NAME “is not found” endif
    return Found, i;
end;
```
Analysis of Algorithm (introduction)

Analysis of Algorithms

Analyze an algorithm to predict its efficiency – namely, the resources \((\text{time and space})\) that an algorithm need during its execution.

Time complexity \(T_A(n)\)

• the \textit{time} taken by an algorithm \(A\) on problems with input size \(n\)

Space complexity \(S_A(n)\)

• the \textit{space} taken by an algorithm \(A\) on problems with input size \(n\)
Sequential Search: Analysis

- Comparing \textit{NAME} with the \textit{i}^{th} name on the list \textit{N}
  - Central unit of work (\textit{dominant} operation)
  - Used for efficiency analysis

- For lists with \textit{n} entries
  - Best case (best case is \textit{usually not important})
    - \textit{NAME} is the first name in the list
    - 1 comparison
    - \(\Theta(1)\)
    - Roughly means a constant
Efficiency of Algorithms

- **Readings:** [SG] Ch. 3

- **Chapter Outline:**
  - Attributes of Algorithms
  - **Measuring Efficiency of Algorithms**
    - One Problem, Many *algorithmic* solutions
    - Time complexity, Space complexity
    - $\Theta$ notation, order of growth of functions
  - Simple Analysis of Algorithms
  - Polynomial vs Exponential Time Algorithms
Sequential Search: Analysis

- For lists with \( n \) entries
  - **Worst case (usually the most important)**
    - NAME is the last name in the list
    - NAME is not in the list
    - \( n \) comparisons
    - \( \Theta(n) \)
  
  - **Average case (sometimes used)**
    - Roughly \( n/2 \) comparisons
    - \( \Theta(n) \)

Roughly means “proportional to \( n \)”

Here \( n/2 \) is also proportional to \( n \). The constant \( c \) in \( cn \) is not important. Usually, we let \( c=1 \).
Sequential Search: Analysis

- **Space Complexity**
  - Uses $2n$ memory storage for *input data*
    (for the input names and telephone numbers)
  - A few more memory storage for *variables*
    (NAME, i, FOUND)
  - Space is $\Theta(n)$
  - Very space efficient

*Total Space*: Linear in $n$  
("proportional to $n$")
Viewing the Rate of Growth of $T(n) = cn$

**Figure 3.4.** Work = $cn$, for various values of $c$
Order of Growth: Order $n$  [Linear]

- All functions that have a linear “shape” are considered equivalent to each other.
- As $n$ grows large, the order of magnitude dominates the running time.
  - minimizing effect of coefficients
  - and lower-order terms
- **Order of magnitude $n$**
  - Written as $\Theta(n)$ (read as “theta-$n$”)
  - Functions vary as $cn$, for some constant $c$
  - *Linear time*
A Side Track:
To explain why we can analyze just the dominant operation.

But, it’s OK even if you cannot understand this Side-Track.
In the analysis above,

- We only analyze the “dominant operation”

This sub-section gives why.

- Namely, why we can take this short-cuts

This may help you better appreciate “analysis of algorithm”

- but, if you have difficulties with this part, you can skip it, without affecting anything else.
Analysis of Algorithm

- To estimate running time of algorithm
  - Without actually running the algorithm

- Method...
  - Estimate cost (work done) for each elementary operation
  - Analyze the number of times each operation is executed during the algorithm
  - Then, sum up the total cost (work done) by the algo

- AIM: To conclude that we
  - Only need to analyze the dominant operation
## Analyzing Algorithm Sequential Search

### Suppose we assume the following estimated costs

<table>
<thead>
<tr>
<th>Statement</th>
<th>Cost</th>
<th>Statement</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>20</td>
<td>Print</td>
<td>4</td>
</tr>
<tr>
<td>while, if</td>
<td>5</td>
<td>endwhile</td>
<td>1</td>
</tr>
</tbody>
</table>

### Statement Costs

<table>
<thead>
<tr>
<th>Statement</th>
<th>#times</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq-Search(N, T, n, NAME);</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i ← 1; Found ← No;</td>
<td>1+1</td>
<td>20</td>
</tr>
<tr>
<td>while (Found=No) and (i &lt;= n) do</td>
<td>n+1</td>
<td>5</td>
</tr>
<tr>
<td>if (NAME = N[i])</td>
<td>n</td>
<td>5</td>
</tr>
<tr>
<td>then Print T[i]; Found ← Yes;</td>
<td>*1</td>
<td>4+20</td>
</tr>
<tr>
<td>else i ← i + 1;</td>
<td>*n</td>
<td>20</td>
</tr>
<tr>
<td>endwhile</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>if (Found=No) then</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Print NAME “is not found” endif</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>end;</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Running Time

\[ T(n) = (1+20+20) + (n+1)5 + n(5+20+1) + (4+20)+(5+4+1) \]

\[ = 31n + 80 = \Theta(n) \]

\[ \leftrightarrow \text{[proportional to } n]\]
Analyzing Algorithm Sequential Search

Now, let’s assume a different set of estimated costs

<table>
<thead>
<tr>
<th>Statement</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>10</td>
</tr>
<tr>
<td>while, if</td>
<td>15</td>
</tr>
<tr>
<td>endwhile</td>
<td>0</td>
</tr>
</tbody>
</table>
From the two examples above…

- **Actual total cost different for**
  - The different sets of estimated costs for basic ops

- **But… Order of growth is the same**
  - for the different sets of estimated costs for basic ops
  - All linear (but with different constants)

- **So… to simplify analysis**
  - Assign a constant cost $\Theta(1)$ for basic operation
  - Can analyze only the dominant operations
    - Namely, the operation that is done “most often”
    - Can also ignore “lower order” terms
    - Such as operations that are done only once
## Simplified Analysis

### Only dominant ops, $\Theta(1)$ cost per basic op

( $\Theta(1)$ means a constant)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Cost</th>
<th>Statement</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>$\Theta(1)$</td>
<td>Print</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>while, if</td>
<td>$\Theta(1)$</td>
<td>endwhile</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

#### Seq-Search(N, T, n, NAME);

begin

\[
i \leftarrow 1; \quad \text{Found} \leftarrow \text{No};
\]

while (Found=No) and (i <= n) do

\[
\text{if (NAME = N[i])}
\]

then Print T[i]; Found $\leftarrow$ Yes;

else \(i \leftarrow i + 1;\)

endwhile

if (Found=No) then

Print NAME “is not found”

endif

end;

#### $T(n) = 4n \times \Theta(1)$

(counting only dominant ops)

\[
= \Theta(4n) = \Theta(n) \leftrightarrow [\text{proportional to } n]
\]
Identifying the dominant operation

Seq-Search(N, T, n, NAME);
begin
    i \leftarrow 1; \; Found \leftarrow No;
    while (Found=No) and (i \leq n) do
        if (NAME = N[i])
            then Print T[i]; Found \leftarrow Yes;
            else i \leftarrow i + 1;
        endwhile
    if (Found=No) then
        Print NAME “is not found” endif
end;

\[ T(n) = n \times \Theta(1) \]
\[ = \Theta(n) \] \leftrightarrow [proportional to n]
As the above examples show,

- If we analyze only the *dominant* operation(s)
- It gives the *same* running time in $\Theta$ notations
- But, it is MUCH simpler.

**Conclusion:**

- Sufficient to analyze only *dominant operation*

**END of SideTrack: and remember…**

- *If you have difficulties with this sub-section, you can skip it, without affecting anything else.*
End of the Side Track

Remember:
It’s OK even if you cannot understand this Side-Track.
Efficiency of Algorithms

- Readings: [SG] Ch. 3

- Chapter Outline:
  - Attributes of Algorithms
  - Measuring Efficiency of Algorithms
  - Simple Analysis of Algorithms
    - Selection Sort Algorithm
    - Pattern Match Algorithm
    - Binary Search Algorithm
  - Polynomial vs Exponential Time Algorithms
Analysis of Algorithms

- To analyze algorithms,
  - Analyze dominant operations
  - Use $\Theta$-notations to simplify analysis
  - Determine order of the running time

- Can apply this at high-level pseudo-codes
  - If high-level primitive are used
    - Analyze running time of high-level primitive
    - expressed in $\Theta$ notations
    - multiply by number of times it is called
  - Next, will apply to analysis of
    Selection Sort and Pattern-Matching
Sorting: Problem and Algorithms

Problem: Sorting

- Take a list of $n$ numbers and rearrange them in increasing order

Algorithms:

- Selection Sort $\Theta(n^2)$
- Insertion Sort $\Theta(n^2)$
- Bubble Sort $\Theta(n^2)$
- Merge Sort $\Theta(n \log n)$
- Quicksort $\Theta(n \log n)**$

**average case**

Not covered in the course
Selection sort

Idea behind the algorithm...

- Repeatedly
  - find the largest number in unsorted section
  - Swap it to the end (the sorted section)

Re-uses the Find-Max & Swap algorithms

$A[m]$ is largest among $A[1..j]$

$A$:  

\[ 1 \quad m \quad j \quad n \]

swap

sorted
Selection Sort Algorithm (pseudo-code)

\[
\text{Selection-Sort}(A, n); \\
\text{begin} \\
\quad j \leftarrow n; \\
\quad \text{while } (j > 1) \text{ do} \\
\quad \quad m \leftarrow \text{Find-Max}(A, j); \\
\quad \quad \text{swap}(A[m], A[j]); \\
\quad \quad j \leftarrow j - 1; \\
\quad \text{endwhile} \\
\text{end;}
\]

\(A[m]\) is largest among \(A[1..j]\)

\[A:\]

\[
\begin{array}{cccccc}
1 & m & j & \ldots & n \\
\end{array}
\]

\(\text{sorted}\)
Example of selection sort

\[
\begin{array}{ccccc}
6 & 10 & 13 & 5 & 8 \\
\end{array}
\]

\[
\begin{array}{c}
 m \\
 j \\
\end{array}
\]

\text{swap}
Example of selection sort

\[
\begin{array}{cccccc}
6 & 10 & 13 & 5 & 8 \\
6 & 10 & 8 & 5 & 13 \\
\end{array}
\]

\[j \rightarrow\]
Example of selection sort

\[
\begin{array}{cccccc}
6 & 10 & 13 & 5 & 8 \\
6 & 10 & 8 & 5 & 13 \\
\end{array}
\]

\[m\]
\[j\]

swap
Example of selection sort

\[ j \]

\[
\begin{array}{cccc}
6 & 10 & 13 & 5 \\
6 & 10 & 8 & 5 \\
6 & 5 & 8 & 10 & 13 \\
\end{array}
\]
Example of selection sort

\[
\begin{array}{cccccc}
6 & 10 & 13 & 5 & 8 \\
6 & 10 & 8 & 5 & 13 \\
6 & 5 & 8 & 10 & 13 \\
\end{array}
\]

\[ j \]
\[ m \]
swap
Example of selection sort

\[
\begin{array}{cccccc}
6 & 10 & 13 & 5 & 8 \\
6 & 10 & 8 & 5 & 13 \\
6 & 5 & 8 & 10 & 13 \\
6 & 5 & 8 & 10 & 13 \\
\end{array}
\]
Example of selection sort

\[
\begin{array}{c|c|c|c|c|c}
6 & 10 & 13 & 5 & 8 \\
\hline
6 & 10 & 8 & 5 & 13 \\
\hline
6 & 5 & 8 & 10 & 13 \\
\hline
6 & 5 & 8 & 10 & 13 \\
\end{array}
\]

\[m \quad j\]

\[\text{swap}\]
Example of selection sort

6 | 10 | 13 | 5 | 8

6 | 10 | 8 | 5 | 13

6 | 5 | 8 | 10 | 13

6 | 5 | 8 | 10 | 13

5 | 6 | 8 | 10 | 13

Done. $j$
Selection Sort Algorithm [SG]

1. Get values for \( n \) and the \( n \) list items
2. Set the marker for the unsorted section at the end of the list
3. While the sorted section of the list is not empty, do steps 4 through 6
   4. Select the largest number in the unsorted section of the list
   5. Exchange this number with the last number in the unsorted section of the list
   6. Move the marker for the unsorted section left one position
7. Stop

Figure 3.6: Selection Sort Algorithm
What about the time complexity?

- **Dominant operation:** *comparisons*

  - 6 10 13 5 8
    - 4 comparisons

  - 6 10 8 5 13
    - 3 comparisons

  - 6 5 8 10 13
    - 2 comparisons

  - 6 5 8 10 13
    - 1 comparisons

Done.  \( j \)
Analysis of Selection Sort

- When sorting \( n \) numbers,
  - \((n-1)\) comparisons in iteration 1 (when \( j = n \))
  - \((n-2)\) comparisons in iteration 2 (when \( j = n-1 \))
  - \((n-3)\) comparisons in iteration 3 (when \( j = n-2 \))
  - \(\ldots\)
  - 2 comparisons in iteration \((n-2)\) (when \( j = 3 \))
  - 1 comparisons in iteration \((n-1)\) (when \( j = 2 \))

Find-Max for \( j \) numbers takes \((j - 1)\) comparisons

- Total number of comparisons:
  - \(\text{Cost} = (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2 = \Theta(n^2)\)
Selection Sort: Summary

**Time complexity:** $\Theta(n^2)$
- Comparisons: $n(n-1)/2$
- Exchanges: $n$ (swapping largest into place)
- Overall time complexity: $\Theta(n^2)$

**Space complexity:** $\Theta(n)$
- $\Theta(n)$ – space for input list and space for a few variables

Selection Sort:
- Time complexity: $T(n) = \Theta(n^2)$
- Space complexity: $S(n) = \Theta(n)$
Rate of Growth of $T(n) = cn^2$

Figure 3.10: Work = $cn^2$ for various values of $c$
Quadratic Order of Magnitude – $\Theta(n^2)$

- All functions of form $c n^2$
  - have similar shape
  - have same order of growth

- Quadratic algorithm -- $\Theta(n^2)$
  - an algorithm that does $c n^2$ work
    - for some constant $c$
  - order of magnitude is $n^2$
  - $\Theta(n^2)$ (read as “theta $n$-square”)
Comparison: Order $n$ vs Order $n^2$

Have seen...

- **Algorithms of order $n$** *(Linear)*
  - Sum, Find-Max, Find-Min, Seq-Search

- **Algorithm of order $n^2$** *(Quadratic)*
  - Selection sort
  - Printing an $n \times n$ table
Rate of Growth Comparison: $n^2$ vs $n$

Figure 3.11: A Comparison of $n$ and $n^2$
Comparison: $\Theta(n^2)$ vs $\Theta(n)$

- Anything that is $\Theta(n^2)$ will eventually be bigger than anything that is $\Theta(n)$, no matter what the constants are.
- An algorithm that runs in time $\Theta(n)$ will be faster one that runs in $\Theta(n^2)$ when $n$ is large.

Eg: compare $T_1(n) = 1000n$ and $T_2(n) = 10n^2$

See also tutorial problem.
Advantages of Using \( \Theta \)-notations

- Gives you a lot of short-cuts!!!
- Focus only on the “highest” order terms
- Can ignore lower order terms.
- Can ignore constant factors…

If \( T(n) = 3n^2 + 14n + 20 \) (ignore \( 14n + 20 \))

Then \( T(n) = \Theta(3n^2) \) (ignore constant 3)

Then \( T(n) = \Theta(n^2) \)
**Θ-notations more examples**

Ex2: If \( T(n) = 7n^2 + 25(3n+4) + 4(2n+5) \), then \( T(n) = \Theta(n^2) \).

But, must be careful.

Ex3: If \( G(n) = (5n^2+4)(3n) + 2n + 43 \), then can “throw away” \( 2n + 43 \) but cannot “throw away” \( (3n) \).

**DIY Question:** What is \( G(n) \) in \( Θ \)-notation?
Why worry about Time Complexity?

Eg: $\Theta(n^2)$ versus $\Theta(n \lg n)$

Because it separates feasible from impossible
Testing * operation in a CPU

Q: How to test that the “*” operation of your CPU is correct?

A: Check exhaustively. For all \( a, b \) check \( a \times b = c \)

Q: How long will it take?

A: Any guesses?
Testing * operation in a CPU

Q: How long will it take?

Assume we use a 100G-Flop CPU.
Can do 100B operations per sec.

\[ a \text{ is a 32-bit number } (2^{32} \text{ cases}) \]
\[ b \text{ is a 32-bit number } (2^{32} \text{ cases}) \]

So, \((a * b)\) there are \((2^{64} \text{ cases})\)

Time taken = \((2^{64} / 100 \times 10^9) \text{ sec}\)
\[
\frac{2^{64}}{100 \times 10^9} \text{ seconds}
\]

Assuming seconds of time for "seconds" | Use seconds of arc instead

http://www.wolframalpha.com/input/?i=2^{64}+\%2F+100x10^9+seconds

- URL

Unit conversions:

3.074 \times 10^6 \text{ minutes}

51,241 \text{ hours}

2,135 \text{ days}

305 \text{ weeks}

70.19 \text{ months}
Q: How long will it take?

Assume we use a 100G-Flop CPU.
Can do 100B operations per sec.

\[ a \text{ is a 32-bit number (}2^{32}\text{ cases)} \]
\[ b \text{ is a 32-bit number (}2^{32}\text{ cases)} \]
So, \((a \times b)\) there are \((2^{64}\text{ cases)}\)

Time taken = \((2^{64} / 100 \times 10^9)\) sec

\[ \approx 6 \text{ years!} \]

Impossible!
(Cannot wait so long to test just one operation)
Testing * operation in a CPU

Q: What if we have \((n \lg n)\) algorithm?

Assume we use a 100G-Flop CPU can take 100B operations per sec.

\(a\) is a 32-bit number \((2^{32} \text{ cases})\)
\(b\) is a 32-bit number \((2^{32} \text{ cases})\)

When \(n = 2^{32}\), with \((n \lg n)\) algorithm

Time taken = \((2^{32} \times 32 / 100 \times 10^9)\) sec

< 2 sec!

Feasible.
(even with much slower computers)
Testing * operation in a CPU

Summary:

If we use $O(n^2)$ algorithm?

$\approx 6 \text{ years!}$

Impossible.
(Not practical)

If we use $O(n \log n)$ algorithm?

$< 2 \text{ sec!}$

Fast, Feasible!
(Practical)
Moral of the story

Time Complexity Analysis help us make predictions.

Time Complexity Analysis help us prepare for the worst case.
Analysis of Pattern Match Algorithm
Analysis of Pat-Match Algorithm

Our pattern matching alg. consists of two modules

- **Pat-Match** \((S, n, P, m)\)  
  "high-level" view

- **Match** \((S, k, P, m)\)  
  "high-level" primitive

To analyze, we do “bottom-up” analysis

- First, analyze time complexity of Match \((S, k, P, m)\)
  - *Note: This is not an \(\Theta(1)\) operation!!*
  - *Express its running time in \(\Theta\) notation (simplified).*

- Then analyze Pat-Match \((S, n, P, m)\)
First, analyze the Match primitive

Align $S[k..k+m-1]$ with $P[1..m]$ (Here, $k = 4$)

### Match primitive

**Algorithm:**

```plaintext
Match(S, k, P, m);
begin
    i \leftarrow 1; \text{MisMatch} \leftarrow \text{No};
    while (i <= m) and (\text{MisMatch}=\text{No}) do
        if (S[k+i-1] not equal to P[i])
            then \text{MisMatch}=\text{Yes}
            else i \leftarrow i + 1
        endif
    endwhile
    \text{Match} \leftarrow \text{not(MisMatch)}; \text{(* Opposite of *)}
end;
```

**Dominant Op is comparison. In worst case, m comparisons. So, $\Theta(m)$**
Next, analyze the Pat-Match Algorithm

\[
\text{Pat-Match}(S,n,P,m); \\
(* \text{Finds all occurrences of } P \text{ in } S *) \\
\text{begin} \\
\quad k \leftarrow 1; \\
\quad \text{while } (k \leq n-m+1) \text{ do} \\
\quad \quad \text{if } \text{Match}(S,k,P,m) = \text{Yes} \text{ then Print \"Match at pos \", k;} \\
\quad \quad \text{endif} \\
\quad k \leftarrow k+1; \\
\text{endwhile} \\
\text{end;} \\
\]

\textbf{Dominant Operation:} \textit{high level op} \text{ Match}(S,k,P,m); \\
\text{Match is called } (n+1-m) \text{ times, each call cost } \Theta(m) \text{ times} \\
\textbf{Total: } \Theta((n+1-m)m) = \Theta(nm)
A Fast Algorithm:

Binary Search

on Sorted Arrays
A Very Fast Search Algorithm

If the list is sorted, \((A_1 \leq A_2 \leq A_3 \leq \ldots \leq A_n)\)

then we can do better when searching

- actually a lot better….

Can use “Binary Search”

**Example**: Find 9

3 5 7 8 9 12 15
Binary search algorithm

Find an element in a sorted array:

IDEA:

• Check middle element.
• Recursively search 1 subarray.

Example: Find 9

3  5  7  8  9  12  15
Binary search algorithm

Find an element in a sorted array:

**IDEA:**

- Check middle element.
- Recursively search 1 subarray.

**Example:** Find 9

3  5  7  8  9  12  15
Binary search algorithm

Find an element in a sorted array:

**IDEA:**
- Check middle element.
- Recursively search 1 subarray.

**Example:** Find 9

3  5  7  8  9  12  15
Binary search algorithm

Find an element in a sorted array:

**IDEA:**

- Check middle element.
- Recursively search 1 subarray.

**Example:** Find 9

3  5  7  8  9  12  15
Binary search algorithm

Find an element in a sorted array:

IDEA:

• Check middle element.
• Recursively search 1 subarray.

Example: Find 9

3  5  7  8  9  12  15
Binary search algorithm

Find an element in a sorted array:

IDEA:

• Check middle element.
• Recursively search 1 subarray.

Example: Find 9

3 5 7 8 9 12 15

Found!
Visualization of Binary Search

To Come Soon!
Binary search – overview

Find an element in a sorted array:

Have two pointers first, last on two ends of the sub-array being search

1. Check middle element.
   Here, \( \text{mid} = (\text{first} + \text{last}) / 2; \)

2. Recursively search 1 subarray.
   Move one of the two pointers to update the sub-array being search.
   Either \( \text{last} \leftarrow \text{mid} - 1; \)
   Or \( \text{first} \leftarrow \text{mid} + 1; \)
Binary Search Algorithm (code)

BinarySearch(A,n,x);
(* search for x in a sorted array A[1..n] *)
begin
  first \leftarrow 1; \quad last \leftarrow n;
  while (first \leq last) do
    mid \leftarrow (first + last) \div 2;
    if (x = A[mid])
      then print “Found x in pos”, mid; Stop
    else if (x < A[mid])
      then last \leftarrow mid-1;
    else first \leftarrow mid+1;
  endif
  endwhile
  print “x not found”;
end;
Binary Search – How fast is it? (1/3)

- **Starting with $n$ numbers,**
  - Binary search repeatedly halves the size
  - Until it reaches 1 element – to check

**Eg: When $n = 100**
- After 1 step, size is $\leq 50$
- After 2 steps, size is $\leq 25$
- After 3 steps, size is $\leq 12$
- After 4 steps, size is $\leq 6$
- After 5 steps, size is $\leq 3$
- After 6 steps, size is $\leq 1$
- One last comparison, DONE!!

$7 = \log_2 100$ steps; [Déjà vu? repeated-halving]
Starting with \( n \) numbers,
- Binary search repeatedly halves the size
- Until it reaches 1 element – to check

Binary Search has complexity
- \( T(n) = \Theta(\ lg n) \)

Recall facts about \( T(n) = \lg n \)
When \( 2^k = n \) [after taking log-base-2]
\[ k = \log_2 n \text{ or } \lg n \]
\[ = \# \text{ of steps of repeated-halving} \]
Starting with \( n \) numbers,
- Binary search repeatedly halves the size
- Until it reaches 1 element – to check

Binary Search has complexity
- \( T(n) = \Theta(\log_2 n) \)

\( T(n) = \Theta(\log n) \) is a very fast algorithm!

<table>
<thead>
<tr>
<th>( n ) (# of element)</th>
<th>( T(n) = \log n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>
Summary: Searching Algorithms

- **Sequential Search (Alg):**
  - Worst Case: \( n \) comparisons
  - Best Case: 1 comparison
  - Avg Case: \( n/2 \) comparisons

- **Binary Search (Alg):**
  - Worst Case: \( \lg n \) comparisons
  - Best Case: 1 comparison
  - Avg Case: \( \lg n \) comparisons*

* How to get the Average Case? Answer: using mathematical analysis.
  - This is OK (do-able) for small \( n \) (see example in tutorial).
  - (Read Sect 3.4.2 of [SG3])
  - For general \( n \), analysis is complex (beyond the scope of this course)
Comparison: order $n$ vs order $\log n$

Figure 3.21. A Comparison of $n$ and $\log n$
Complexity of Algorithms...

- **Logarithmic Time Algorithm**
  - Binary Search – $\Theta(\lg n)$ time

- **A Linear Time Algorithm**
  - Array-Sum – $\Theta(n)$ time
  - Sequential-Search – $\Theta(n)$ time

- **A Quadratic Time Algorithm**
  - Selection Sort – $\Theta(n^2)$ time

- **An Exponential Time Algorithm**
  - All-Subsets – $\Theta(2^n)$ time

Covered later in the course
# Complexity of Time Efficiency

<table>
<thead>
<tr>
<th>Problem</th>
<th>Unit of Work</th>
<th>Algorithm</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching</td>
<td>Comparisons</td>
<td>Sequential search</td>
<td>1</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Binary search</td>
<td>1</td>
<td>$\Theta(lg\ n)$</td>
<td>$\Theta(lg\ n)$</td>
</tr>
<tr>
<td>Sorting</td>
<td>Comparisons and exchanges</td>
<td>Selection sort</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Data cleanup</td>
<td>Examinations and copies</td>
<td>Shuffle-left</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy-over</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Converging-pointers</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>Character comparisons</td>
<td>Forward march</td>
<td>$\Theta(n)$</td>
<td>$\Theta(m \times n)$</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.22: Summary of Time Efficiency**
Efficiency of Algorithms

- **Readings:** [SG] Ch. 3

- **Chapter Outline:**
  - Attributes of Algorithms
  - Measuring Efficiency of Algorithms
  - Simple Analysis of Algorithms
  - Polynomial vs Exponential Time Algorithms
    - *Polynomial Time Algorithms* (Tractable)
    - *Exponential Time Algorithms* (Intractable)
    - Approximation Algorithms
      (e.g., Bin Packing)
Polynomial and Exponential

- **Polynomial functions of** \( n \)
  - Some linear combinations of \( n, n^2, n^3 \), etc
  - \( f(n) = 2n + 7, \quad = \Theta(n) \) linear
  - \( g(n) = 10n^2 + 9n + 23, \quad = \Theta(n^2) \) quadratic
  - \( h(n) = 3n^4 + 5n^2 + 2n + 8, \quad = \Theta(n^4) \) deg-4 polynomial

- **Exponential functions of** \( n \)
  - Some power of \( n \) (\( n \) in exponent), or factorial
  - \( f(n) = 3(2^n), \quad = \Theta(2^n) \) exponential
  - \( g(n) = 5(n!), \quad = \Theta(n!) \) factorial
Algorithms: Polynomial vs Exponential

- **Logarithmic Time Algorithm**
  - Binary Search – $\Theta(\lg n)$ time

- **A Linear Time Algorithm**
  - Array-Sum – $\Theta(n)$ time
  - Sequential-Search – $\Theta(n)$ time

- **A Quadratic Time Algorithm**
  - Selection Sort – $\Theta(n^2)$ time

- **An Exponential Time Algorithm**
  - All-Subsets – $\Theta(2^n)$ time

Covered later in the course
When Things Get Out of Hand

- **Polynomially bound algorithms**
  - Time complexity is some polynomial order
  - Example: $T(n)$ is of order of $n^2$

- **Intractable algorithms**
  - Run time is exponential (worse than polynomial time)
  - Examples:
    - *Hamiltonian circuit*
    - *Bin-packing*
## Comparison of Time Complexities

### Figure 3.27: A Comparison of Four Orders of Magnitude

<table>
<thead>
<tr>
<th>ORDER</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lg n )</td>
<td>0.0003 sec</td>
<td>0.0006 sec</td>
<td>0.0007 sec</td>
<td>0.001 sec</td>
</tr>
<tr>
<td>( n )</td>
<td>0.001 sec</td>
<td>0.005 sec</td>
<td>0.01 sec</td>
<td>0.1 sec</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>0.01 sec</td>
<td>0.25 sec</td>
<td>1 sec</td>
<td>1.67 min</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>0.1024 sec</td>
<td>3,570 years</td>
<td>4 ( \times ) 10^{16} centuries</td>
<td>Too big to compute!!</td>
</tr>
</tbody>
</table>

See extended table in tutorial problem.
Figure 3.25: Comparisons of $\lg n$, $n$, $n^2$, and $2^n$
Exponential algorithm

- $\Theta(2^n)$
- More work than any polynomial in $n$

Approximation algorithms

- Run in polynomial time
- But *do not* give optimal solutions
- Example: Bin Packing Algorithms
Summary of Chapter 3

- **Desirable attributes in algorithms:**
  - Correctness
  - Ease of understanding
  - Elegance
  - Efficiency

- **Efficiency of an algorithm is extremely important**
  - Time Complexity
  - Space Complexity
Summary of Chapter 3 (2)

- To compare the efficiency of two algorithms that do the same task
  - Measure their time complexities

- Efficiency focuses on order of magnitude
  - Time complexity in $\Theta$-notations.
  - In its simplest form (e.g., $\Theta(n)$, $\Theta(n^2)$)
Thank you!