Algorithms (Introduction)

Readings: [SG] Ch. 2

- **Chapter Outline:**
  1. Chapter Goals
  2. What are Algorithms
  3. Pseudo-Code to Express Algorithms
  4. Some Simple Algorithms
  5. Examples of Algorithmic Problem Solving [Ch. 2.3]
     1. ***Searching Example,***
     2. ***Finding Maximum/Largest***
     3. ***Modular Program Design***
     4. ***Pattern Matching***

Last Revised: 31 August 2016.

Recurring Principles in CS & IT

RP1: Multiple Levels of Abstraction
(very high to very low)

RP2: One Data, Multiple Views
(thru diff interfaces)

RP3: Define a (small) set of basic primitives
(building blocks)

RP4: Divide & Conquer aka
(Decomposition)

RP5: "The Power of Iteration"
(aka Recursion)

Next…

Enhancing your Computational Toolkit

Algorithmic Problem Solving

- **Examples of algorithmic problem solving**

  1. Sequential search: find a particular value in an unordered collection
  2. Find maximum: find the largest value in a collection of data
  3. Pattern matching: determine if and where a particular pattern occurs in a piece of text

Re-using the Array-Sum template…

Array-Sum is a classic linear-scan algorithm;

Use it as template for similar problem.

- Counting how many positive numbers
  Keeping Scores in games
  Embedded counting (apps & games)
- Finding Maximum, Minimum, Rank
- Computing Histograms
- Computing Fibonacci Numbers
- Computing Sum of series
- … and many, many others

Re-using the Array-Sum template…

Array-Sum as template

Modify it to solve similar linear-scan type computational problems

Count-Pos

Find-Min

etc, etc

Find-Max

Fibonacci

Hon Wai Leong, NUS

Template Linear-Scan Algorithm

```
Algorithm Array-Sum(A, n);
(* Find the sum of A1, A2,…,An. *)
begin
  Sum_SF ← 0;
  k ← 1;
  while (k <= n) do
     Sum_SF ← Sum_SF + A[k];
     k ← k + 1;
  endwhile
  Sum ← Sum_SF;
  return Sum
end;
```

Initialization block

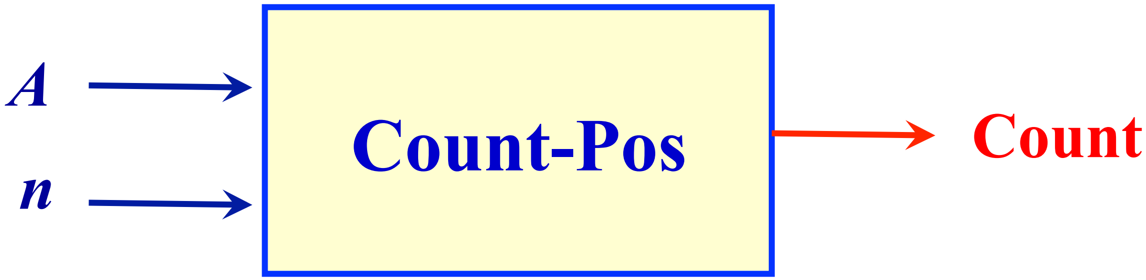
Iteration block;
the key step where
most of the work is done

Post-Processing block

Algorithm Problem Solving: Count-Pos

Counting Positive Numbers

Task: Count the number of positive numbers in a list $A[1..n]$ of numbers



Definition: Count-Pos (A, n)
The high-level primitive Count-Pos takes as input a variable n and an array $A[1..n]$, then it computes & returns variable Count that represents the number of positive numbers in $A[1..n]$

Counting Positive Numbers

Task: Count the number of positive numbers in a list $A[1..n]$ of numbers

PQ: Reuse the algorithm for Array-Sum(A, n)

IDEA: Use variable Count-SF to count the number of positive numbers encountered "so far"

THINK:	What to do with Count-SF during *Initialize* *Iteration* *Post-processing*

Algorithm Count-Pos

Preconditions: The variable *n* and the arrays *A* [1..*n*] has been read in.

```
Count-Pos(A,n);
begin
  Count-SF ← 0;
  k ← 1;

  while (k <= n) do
    if (A[k] > 0) then
      Count-SF ← Count-SF + 1;
    k ← k + 1
  endwhile

  Count ← Count-SF;
  return Count
end;
```

Initialization **block**

Iteration **block; the key step where most of the work is done**

Post-Processing **block**

Algorithm Problem Solving:
Find-Max

Finding the Maximum

Task: Finding *a maximum number* in a list $A[1..n]$ of numbers

Definition: Find-Max (A, n)
The high-level primitive Find-Max takes in as input any array A, a variable n, and it finds and returns variable *Max*, the maximum element in the array $A[1..n]$, found in location *Loc*.

Finding Max: Big, Bigger, Biggest

Task: Finding *a maximum number*
in a list $A[1..n]$ of numbers

PQ: Reuse the algorithm for Array-Sum(A, n)

IDEA: Use variable Max-SF to remember the maximum encountered "so far".
And variable Loc to remember location of Max-SF.

THINK:	What to do with Max-sf, Loc during *Initialize* *Iteration* *Post-processing*

Algorithm Find-Max

Preconditions: The variable *n* and the arrays *A* [1..*n*] has been read in.

```
Find-Max(A,n);   (* find max of A[1..n] *)
begin
  Max-SF ← A[1];   Location ← 1;
  i ← 2;    (* why 2, not 1? *)

  while (i <= n) do
    if (A[i] > Max-SF) then
      Max-SF ← A[i];
      Location ← i;
    endif
    i ← i + 1
  endwhile

  Max ← Max-SF;
  return Max, Location
end;
```

Initialization **block**

Iteration **block; the key step where most of the work is done**

Post-Processing **block**

Algorithm: Finding the Largest [SG3]

Find Largest Algorithm

Get a value for n, the size of the list
Get values for $A_1, A_2, \ldots, A_n$, the list to be searched
Set the value of *largest so far* to A_1
Set the value of *location* to 1
Set the value of i to 2
While ($i \leq n$) do
    If A_i > *largest so far* then
        Set *largest so far* to A_i
        Set *location* to i
    Add 1 to the value of i
End of the loop
Print out the values of *largest so far* and *location*
Stop

Initialization **block**

Iteration **block; the key step where most of the work is done**

Post-Processing **block**

Figure 2.10: Algorithm to Find the Largest Value in a List

Algorithm Problem Solving:

A Lookup Problem

A Lookup Problem

	N	T
1	RICHARD Son	6666-8989
2	HENZ Marvin	7575-7575
3	TEO Alfred	1212-4343
…	…	…
5001	LEONG Hon Wai	8888-8888
5002	HOU Manuel	7555-7555
…	…	…
…	…	…
…	…	…
…	…	…
10000	ZZZ Zorro	4545-6767

An *unordered* telephone Directory with 10,000 names and phone numbers

TASK:

Look up the telephone number of a particular person.

ALGORITHMIC TASK:

Give an algorithm to Look up the telephone number of a particular person.

Task: A Lookup Problem

Given: An unordered phone directory of subscribers, names stored in N[1..10,000] and telephone numbers stored in T[1..10,000]

Task: Lookup (search for) the telephone number of a given person.

PQ: Reuse the algorithm for Array-Sum(A, n)

IDEA: Use Linear-Scan algorithm,
Test name entries in N, one-by-one

Task: A Lookup Problem

PQ: Reuse the algorithm for Array-Sum(A, n)

Use a Linear-Search Algorithm:

- Use a "pointer" i to seach name N[i]
- Use a variable called Found (set to true when the given name is found, and terminate search asap)

THINK:	What to do with Found during *Initialize* *Iteration* *Post-processing*

Hon Wai Leong, NUS

Task 1: Linear Search Algorithm

Sequential Search Algorithm

STEP	OPERATION
1	Get values for *NAME*, $N_1, \ldots, N_{10,000}$, and $T_1, \ldots, T_{10,000}$
2	Set the value of *i* to 1 and set the value of *Found* to NO
3	While both (*Found* = NO) and ($i \leq 10,000$) do steps 4 through 7
4	If *NAME* is equal to the *i*th name on the list N_i then
5	Print the telephone number of that person, T_i
6	Set the value of *Found* to YES
	Else (*NAME* is not equal to N_i)
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

Initialization **block** (step 2)

Iteration **block; the key step where most of the work is done** (steps 3–7)

Post-Processing **block** (steps 8–9)

Algorithm Linear Search (*revised*)

- ***Preconditions:*** **The variables NAME, *m*, and the arrays N[1..m] and T[1..m] have been read into memory.**

```
Algorithm Seq-Search (N, T, m, NAME);
begin
   i ← 1;
   Found ← No;
   while (Found = No) and (i <= m) do
      if (NAME = N[i])
         then Print T[i]; Found ← Yes;
         else i ← i + 1;
      endif
   endwhile
   if (Found=No) then
      Print NAME "is not found" endif
   return Found, i;
end;
```

Initialization **block**

Iteration **block; the key step where most of the work is done**

Post-Processing **block**

Abstraction: Define new primitive

- Then Seq-Search becomes a *high- level primitive defined as* Seq-Search (*N*, *T, m, Name*)

Definition: Seq-Search (*N*, *T*, *m*, *Name*)
The high-level primitive Seq-Search takes in two input arrays *N* (*storing name*), and *T* (*storing telephone #s*), *m* the size of the arrays, and *Name*, the name to search; and return the variables *Found* and *Loc*.

Using a High-level Primitive

Definition: Seq-Search (*N*, *T*, *m*, *Name*)
The high-level primitive Seq-Search takes in two input arrays *N* (*storing name*), and *T* (*storing telephone #s*), *m* the size of the arrays, and *Name*, the name to search; and return the variable *Found* and *Loc*.

To use the high-level primitive (or just primitive, in short) we just issue a call to that high-level primitive

Example 1: Seq-Search (*N*, *T*, 100, "John Olson")
call the Seq-Search to find "John Olson" in array *N*[1 .. 100].

Example 2: Top ← Array-Sum (*B*, 8)
"compute the sum of *B*[1 .. 8], and store that in variable Top

More linear-scan algorithms

Linear-scan is a powerful algorithm

Can solve many other problems:

- Searching for a telephone number
- Reversing an Array
- Partitioning an Array
- Removing Duplicates
- Reversing digits of an integer
- Converting the base
- Character ←→ number conversion;

Problem Solving by Decomposition

Problem Decomposition

- **Software are Complex**
  - **Huge** (Millions of lines of code): Linux, Powerpoint, Firefox, Outlook
  - **Complex:** Flight-simulator, Wolfram-Alpha
- **How to manage this Complexity?**

RP4: Decomposition

Hon Wai Leong, NUS

Modular Software Design

RP4: Decomposition

- **Divide large software into small modules**
  - Each module solve a sub-task,
  - Modules are design, built & tested separately
  - Combined to give solution to overall problem
  - Achieves good division of labour
  - Reduces complexity of the software dev.

Algorithm Problem Solving:
Pattern Matching

Task 3: Pattern Matching

- **Algorithm search for a pattern in a source text**

  Given: A source text $S[1..n]$ and a pattern $P[1..m]$

  Question: Find all occurrence of pattern P in text S?

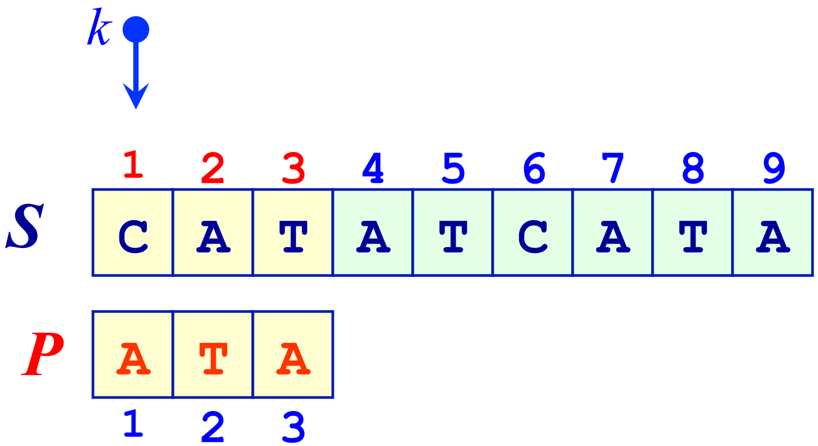
	1	2	3	4	5	6	7	8	9
S	C	A	T	A	T	C	A	T	A

P	A	T	A
	1	2	3

Output of Pattern Matching Algorithm:

```
There is a match at position 2
There is a match at position 7
```

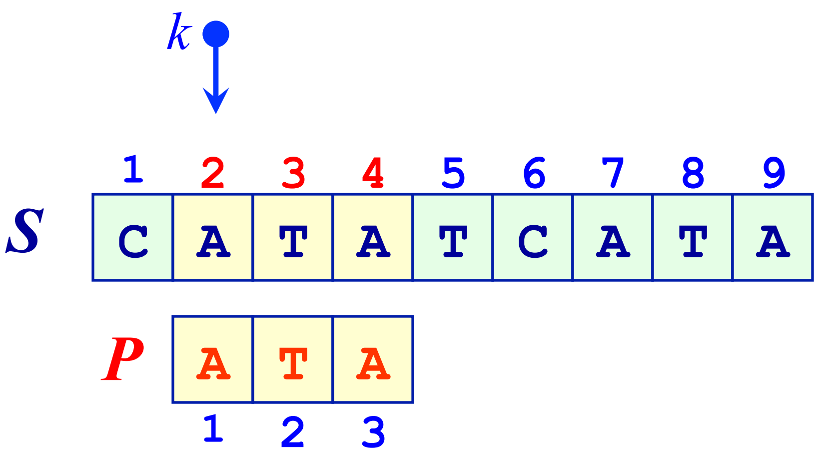
Example of Pattern Matching 1



- Align pattern ***P*** with text ***S*** starting at pos *k* = 1;
- Check for match (between ***S***[1..3] and ***P***[1..3])
- Result – no match

Hon Wai Leong, NUS

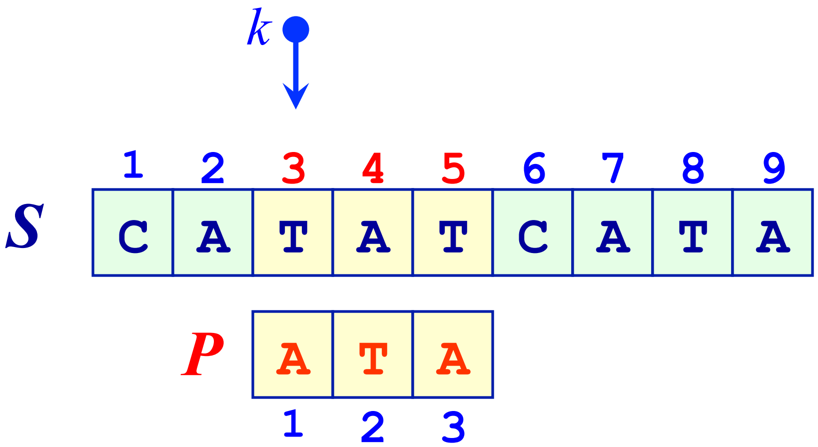
Example of Pattern Matching 2



- Align pattern ***P*** with text ***S*** starting at pos *k* = 2;
- Check for match (between ***S***[2..4] and ***P***[1..3])
- Result – match!

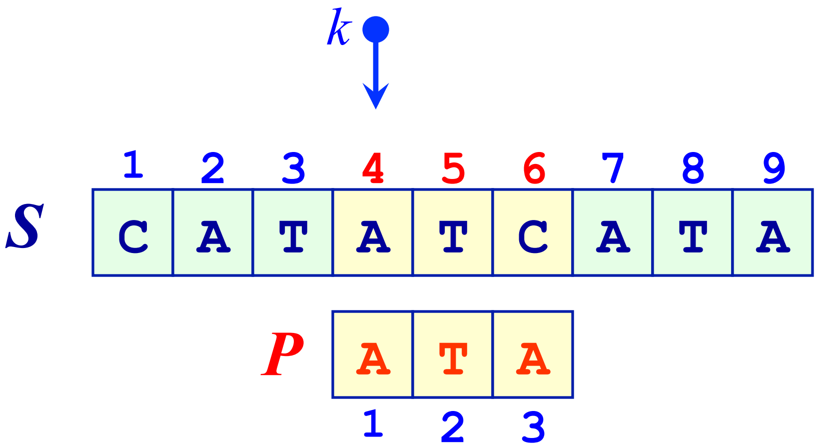
  Output: There is a match at position 2

Example of Pattern Matching 3



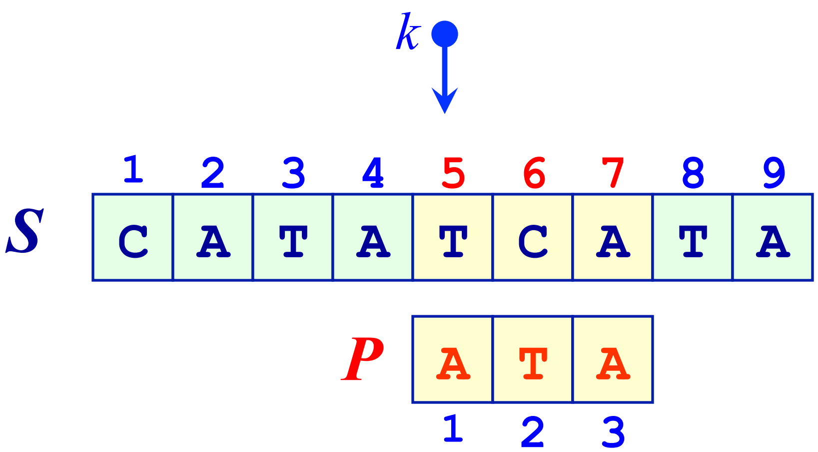
- Align pattern ***P*** with text ***S*** starting at pos *k* = 3;
- Check for match (between ***S***[3..5] and ***P***[1..3])
- Result – No match.

Example of Pattern Matching 4



- Align pattern ***P*** with text ***S*** starting at pos *k* = 4;
- Check for match (between ***S***[4..6] and ***P***[1..3])
- Result – No match.

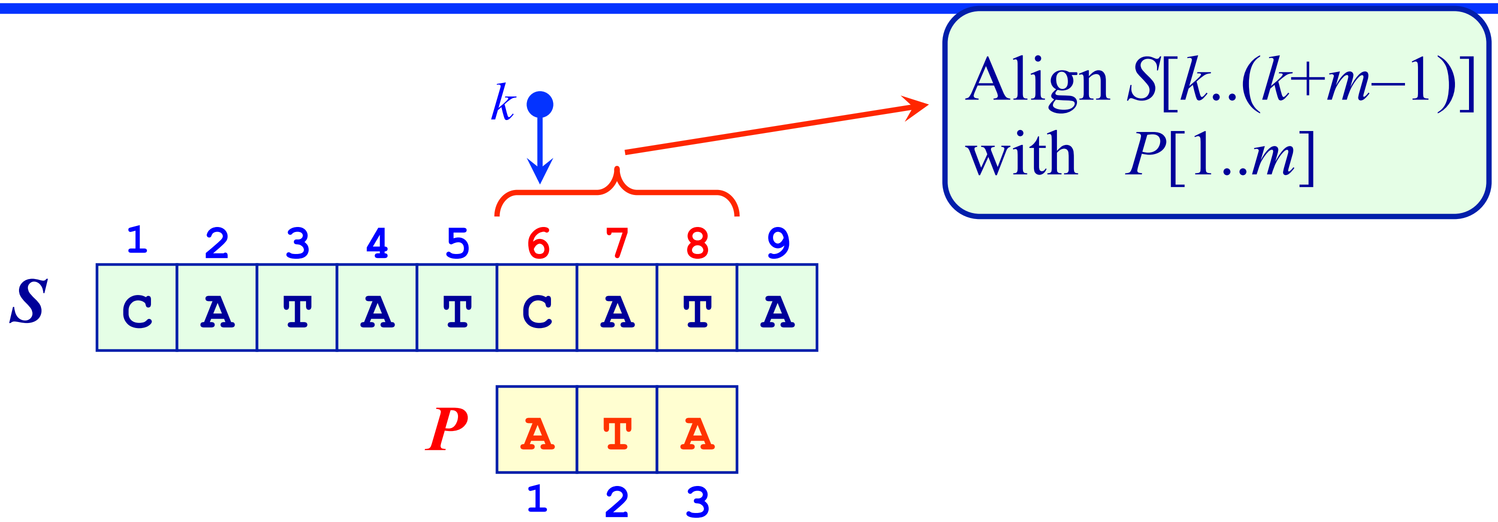
Example of Pattern Matching 5



- Align pattern ***P*** with text ***S*** starting at pos *k* = 5;
- Check for match (between ***S***[5..7] and ***P***[1..3])
- Result – No match.

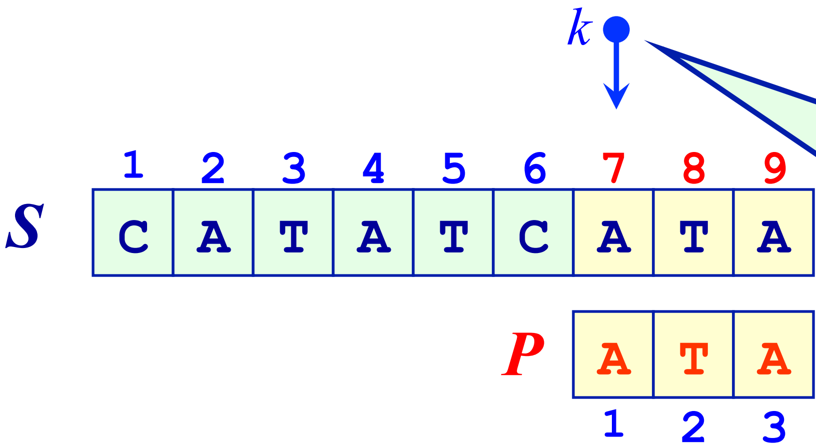
Hon Wai Leong, NUS

Example of Pattern Matching 6



- Align pattern ***P*** with text ***S*** starting at pos $k = 6$;
- Check for match (between ***S***[6..8] and ***P***[1..3])
- Result – No match.

Example of Pattern Matching 7



Note:
k = 7 is the ***last position*** to test;
After that ***S*** is "too short".
In general, it is ***k* = *n*–*m*+1**

- Align pattern ***P*** with text ***S*** starting at pos *k* = 7;
- Check for match (between ***S***[7..9] and ***P***[1..3])
- Result – match!

Output: There is a match at position 7

Pattern Matching: Decomposition

Task: Find all occurrences of the pattern *P* in text *S*;

- **Algorithm Design: Top Down Decomposition**
  - Modify from basic-iterative-algorithm (index k)
- **At each iterative step (for each k)**
  - Align pattern P with S at position k and
  - Test for match between $P[1..m]$ and $S[k \,..\, k+m-1]$
- **Define an abstraction ("high level operation")**

$$\text{Match}(S, k, P, m) = \begin{cases} \text{Yes} & \text{if } S[k..k+m-1] = P[1..m] \\ \text{No} & \text{otherwise} \end{cases}$$

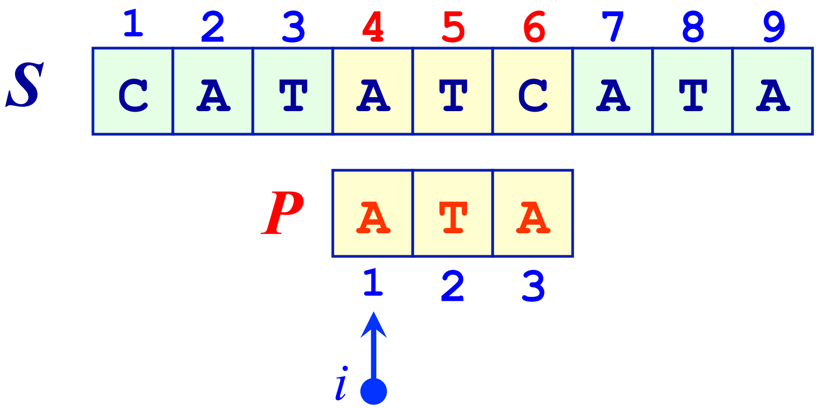
Pattern Matching: Pat-Match

- ***Preconditions:*** **The variables *n*, *m*, and the arrays *S* and *P* have been read into memory.**

```
Pat-Match(S,n,P,m);
(* Finds all occurrences of P in S *)
begin
   k ← 1;
   while (k <= n-m+1) do
      if Match(S,k,P,m) = Yes
         then Print "Match at pos ", k;
      endif
      k ← k+1;
   endwhile
end;
```

Use the ***"high level operation"*** **Match(*S*,*k*,*P*,*m*)** which can be refined later.

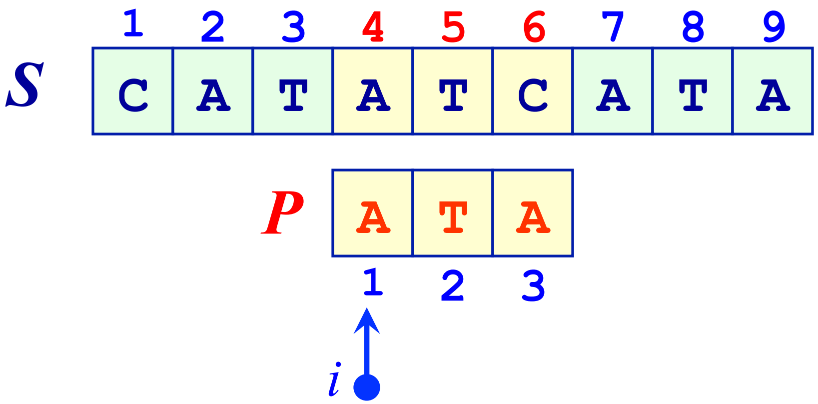
Match of S[k..k+m-1] and P[1..m]



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

```
Match(S,k,P,m);
begin
  i ← 1; MisMatch ← No;
  while (i <= m) and (MisMatch=No) do
    if (S[k+i-1] not equal to P[i])
      then MisMatch=Yes
      else i ← i + 1
    endif
  endwhile
  Match ← not(MisMatch);  return Match
end;
```

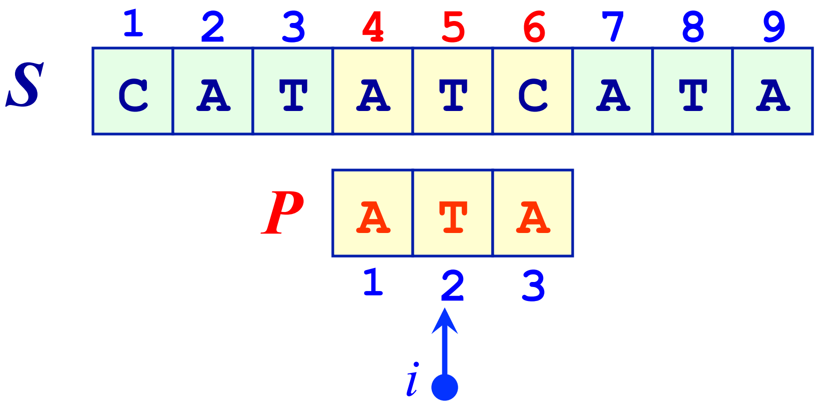
Example: Match of S[4..6] and P[1..3]



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

- $[k = 4]$ With $i = 1$, MisMatch = No
- Compare ***S***[4] and ***P***[1] (S[k+i-1] and P[i])
- They are equal, so increment i

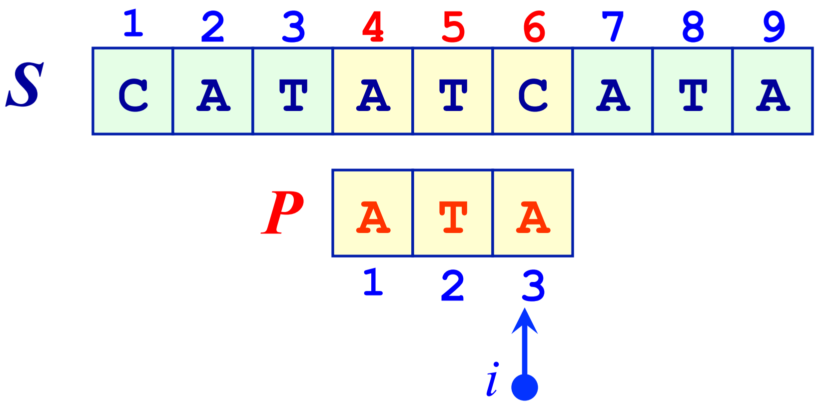
Example: Match of S[4..6] and P[1..3]



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

- [$k = 4$] With $i = 2$, MisMatch = No
- Compare ***S***[5] and ***P***[2] (S[$k+i$-1] and P[i])
- They are equal, so increment i

Example: Match of S[4..6] and P[1..3]



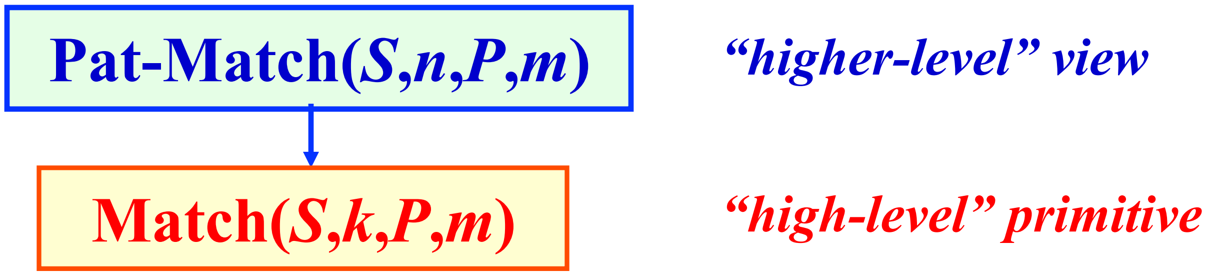
Align $S[k..k+m-1]$
with $P[1..m]$
(Here, $k = 4$)

- [$k = 4$] With $i = 3$, MisMatch = No
- Compare ***S***[6] and ***P***[3] (S[$k+i-1$] and P[i])
- They are *not* equal, so set MisMatch=Yes

Hon Wai Leong, NUS

Our Top-Down Design

- **Our pattern matching alg. consists of two modules**



  - Achieves good division-of-labour

- **Made use of top-down design and abstraction**
  - Separate "high-level" view from "low-level" details
  - Make difficult problems more manageable
  - Allows piece-by-piece development of algorithms
  - Key concept in computer science

Pattern Matching: Pat-Match (*1st draft*)

- ***Preconditions:*** **The variables *n*, *m*, and the arrays *S* and *P* have been read into memory.**

```
Pat-Match(S,n,P,m);
(* Finds all occurrences of P in S *)
begin
   k ← 1;
   while (k <= n-m+1) do
      if Match(S,k,P,m) = Yes
         then Print "Match at pos ", k;
      endif
      k ← k+1;
   endwhile
end;
```

Use the ***"high level primitive operation"*** **Match(*S*,*k*,*P*,*m*)** which can be de/refined later.

Pattern Matching Algorithm of [SG]

Pattern-Matching Algorithm

```
Get values for n and m, the size of the text and the pattern, respectively
Get values for both the text T1 T2 … Tn and the pattern P1 P2 … Pm
Set k, the starting location for the attempted match, to 1
While (k ≤ (n - m +1)) do
    Set the value of i to 1
    Set the value of Mismatch to NO
    While both (i ≤ m) and (Mismatch = NO) do
        If  Pi ≠ Tk+(i-1)  then
            Set Mismatch to YES
        Else
            Increment i by 1 (to move to the next character)
    End of the loop
    If Mismatch = NO then
        Print the message 'There is a match at position'
        Print the value of k
    Increment k by 1
End of the loop
Stop, we are finished
```

THINK:: How can *Mismatch*=NO here?

This part compute Match(*T*,*k*,*P*,*m*)

Figure 2.12: Final Draft of the Pattern-Matching Algorithm

Hon Wai Leong, NUS

Pattern Matching Algorithm of [SG]

- **Pattern-matching algorithm**
  - Contains a loop *within a loop*
    - ***External loop iterates through possible locations of matches to pattern***
    - ***Internal loop iterates through corresponding characters of pattern and string to evaluate match***

Summary of Chapter 2 [SG3]

- **Specify algorithms using pseudo-code**
  - Unambiguous, readable, analyzable
- **Algorithm specified by three types of operations**
  - Sequential, conditional, and repetitive operations
- **Seen several examples of algorithm design**
  - Designing algorithm is not so hard
  - Re-use, Modify/Adapt, Abstract your algorithms
- **Algorithm design is also a creative process**
  - Top-down design helps manage complexity
  - Process-oriented thinking helps too

Summary

- **Importance of “doing it”**
  - Test out each algorithm to find out “what is really happening”
  - Run some of the animations in the lecture notes
- **If you are new to algorithms**
  - read the textbook
  - try out the algorithms
  - do the exercises

… The End …

Thank you!