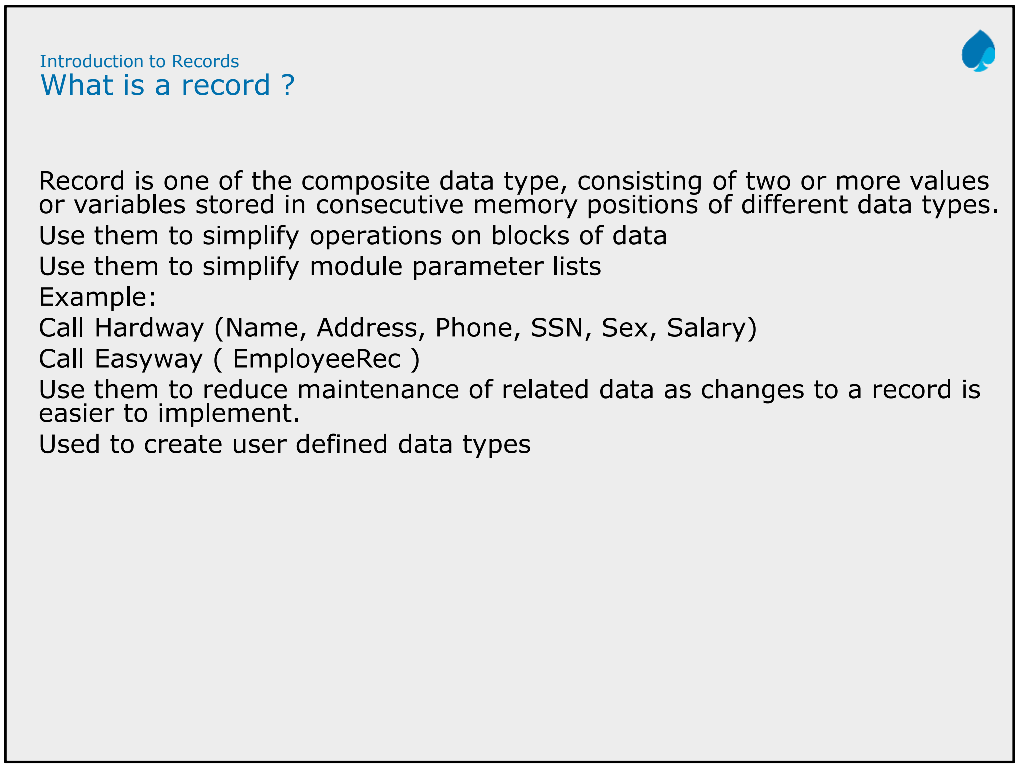


Introduction to Records

What is a record ?

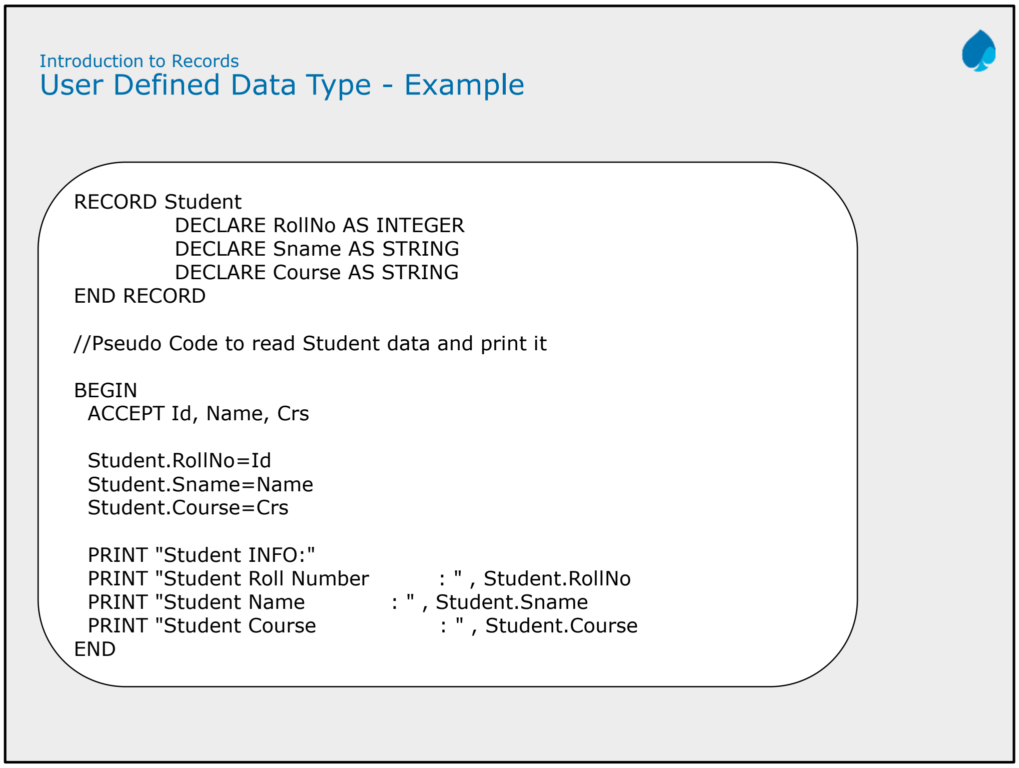
Record is one of the composite data type, consisting of two or more values or variables stored in consecutive memory positions of different data types.
Use them to simplify operations on blocks of data
Use them to simplify module parameter lists
Example:
Call Hardway (Name, Address, Phone, SSN, Sex, Salary)
Call Easyway (EmployeeRec)
Use them to reduce maintenance of related data as changes to a record is easier to implement.
Used to create user defined data types

Record: Record is one of the composite data type, consisting of two or more values or variables stored in consecutive memory positions .

Example for record:

```
RECORD Employee
        DECLARE ecode AS INTEGER
        DECLARE ename AS STRING
        DECLARE esal AS INTEGER
        DECLARE edept AS STRING
END RECORD
```

The above record is used to hold details about an employee such as employee code, employee name, employee salary and department in which employee is working.



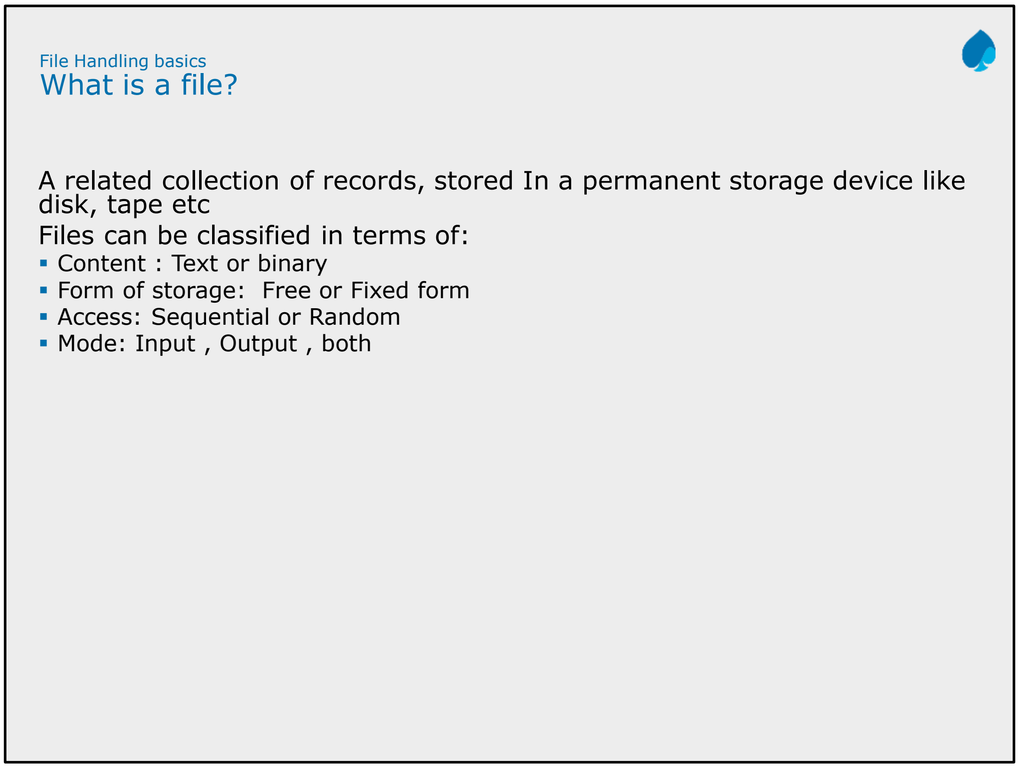
Introduction to Records

User Defined Data Type - Example

```
RECORD Student
        DECLARE RollNo AS INTEGER
        DECLARE Sname AS STRING
        DECLARE Course AS STRING
END RECORD

//Pseudo Code to read Student data and print it

BEGIN
  ACCEPT Id, Name, Crs

  Student.RollNo=Id
  Student.Sname=Name
  Student.Course=Crs

  PRINT "Student INFO:"
  PRINT "Student Roll Number      : " , Student.RollNo
  PRINT "Student Name       : " , Student.Sname
  PRINT "Student Course         : " , Student.Course
END
```

The code given in slide is used to maintain information about a student like rollno, sname and course details.

Functionalities implemented in the above code are
    Accepting student details like id, name and course
    Printing the same.



File Handling basics

What is a file?

A related collection of records, stored In a permanent storage device like disk, tape etc

Files can be classified in terms of:

- Content : Text or binary
- Form of storage: Free or Fixed form
- Access: Sequential or Random
- Mode: Input , Output , both

Files:

So far the input data was read from standard input and the output was displayed on the standard output. These programs are adequate if the volume of data involved is not large. However many business-related applications require that a large amount of data be read, processed, and saved for later use. In such a case, the data is stored on storage device, usually a disk in the form of file.

There are two types of files:

Binary file:

It is efficient for large amount of numeric data.

Text file:

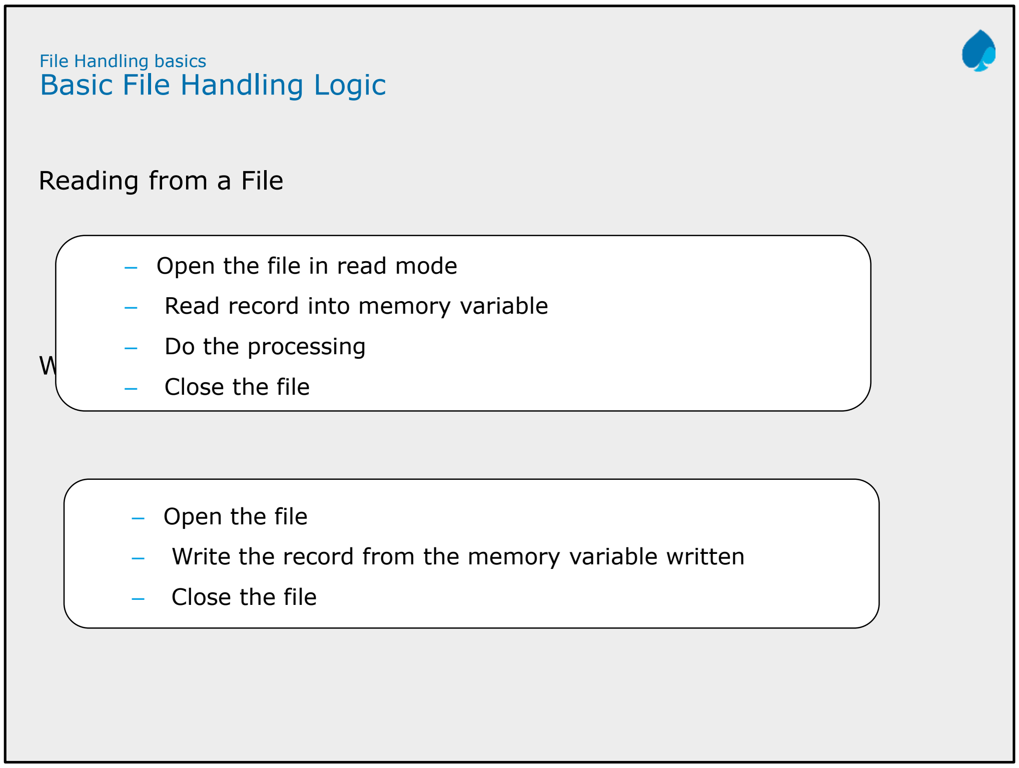
It stores text and numbers as one character per byte.

It consumes large amount of storage for numbers.

It is useful for printing reports and text documents.

Access: Data from the file is accessible either sequentially or randomly.

Mode: Data from the file can be readable(Output) or data can be writable to the file(Input).



File Handling basics

Basic File Handling Logic

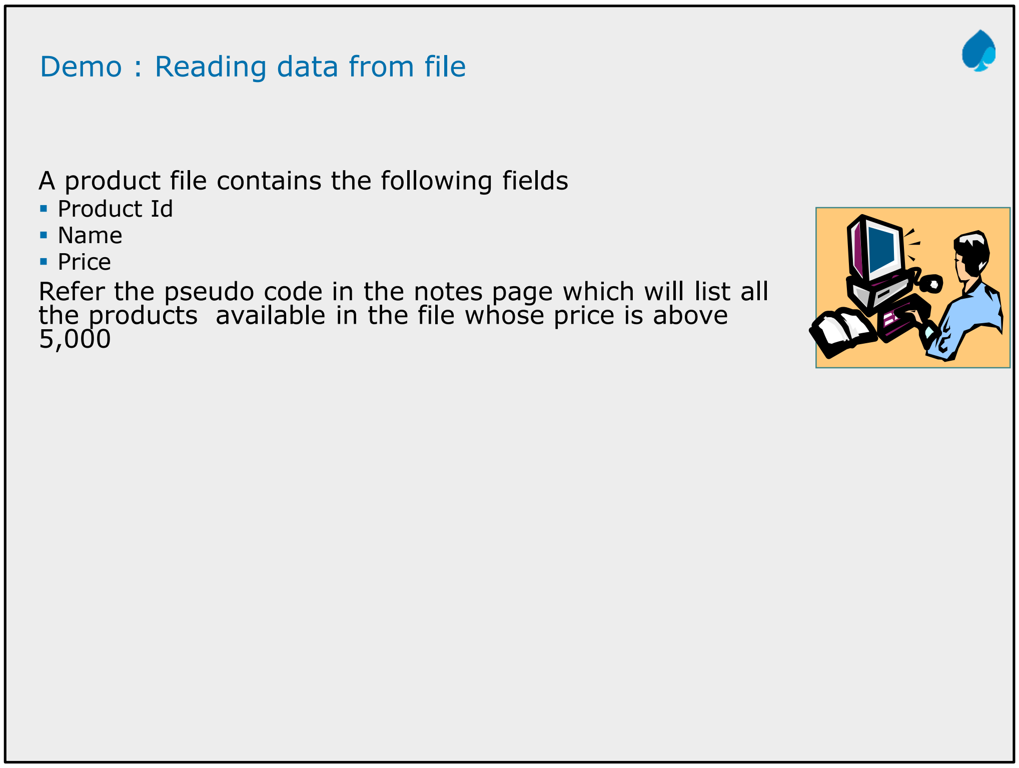
Reading from a File

- Open the file in read mode
- Read record into memory variable
- Do the processing
- Close the file

W

- Open the file
- Write the record from the memory variable written
- Close the file

File I/O requires four functions:
open: It allows access to file.
close: It ends access to a file.
read: It gets data from file.
write: It adds data into file.



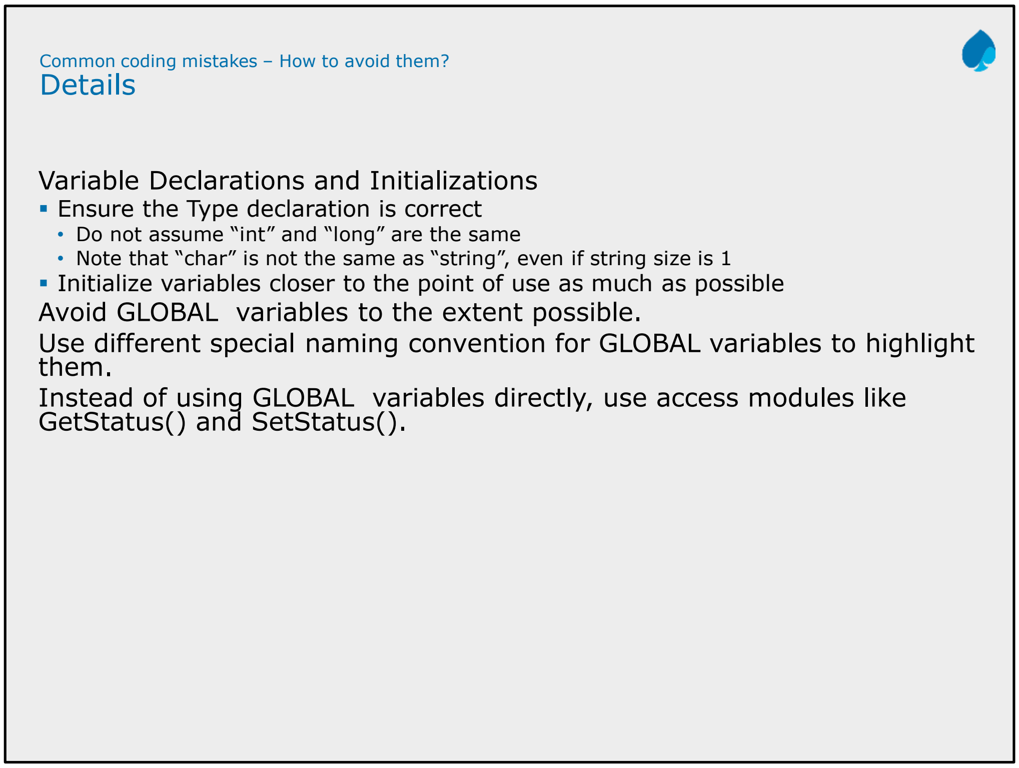
Demo : Reading data from file

A product file contains the following fields

- Product Id
- Name
- Price

Refer the pseudo code in the notes page which will list all the products available in the file whose price is above 5,000

```
RECORD ProductRec
        DECLARE productId AS INTEGER
        DECLARE name AS STRING
        DECLARE price AS INTEGER
END RECORD
BEGIN
        DECLARE file AS FILE
        DECLARE product AS ProductRec
        PROMPT "Enter the filename" AND STORE IN file
        IF(fileExists(file)) THEN
             OPEN file
             READ data from the file AND STORE IN product
             IF(product.price>5000) THEN
                          DISPLAY "Product Id" + product.productId
                          DISPLAY "Product Name" + product.name
                          DISPLAY "Product Price" + product.price
             END IF
        END IF
END
```



Common coding mistakes – How to avoid them?

Details

Variable Declarations and Initializations
- Ensure the Type declaration is correct
  - Do not assume "int" and "long" are the same
  - Note that "char" is not the same as "string", even if string size is 1
- Initialize variables closer to the point of use as much as possible

Avoid GLOBAL variables to the extent possible.

Use different special naming convention for GLOBAL variables to highlight them.

Instead of using GLOBAL variables directly, use access modules like GetStatus() and SetStatus().

Variable Declarations and Initializations:

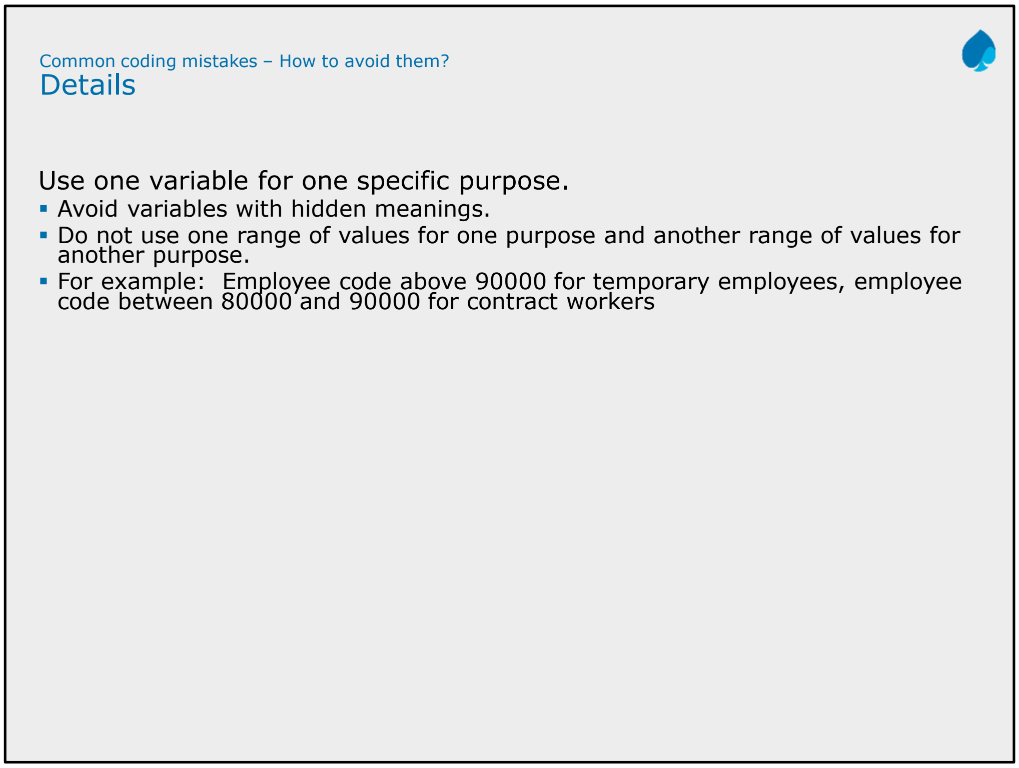
Code Snippet - Example:

```
Boolean more-records = TRUE;
…
while ( more-records ) do …
```

Better version of the above code snippet:

```
Boolean more-records;
…
more-records = TRUE;
while ( more-records ) do …
```

Avoid GLOBAL variables to the extent possible. : Note that it is difficult to debug, because a GLOBAL variable can be changed from any module. It may create problems with a recursive module.
Use different special naming convention for GLOBAL variables to highlight them. For example: MAX_USERS_G for global
Instead of using GLOBAL variables directly, use access modules like GetStatus() and SetStatus(). Note that this ensures that the variable is used only at once place inside those modules. This becomes easier to change or debug.



Common coding mistakes – How to avoid them?

Details

Use one variable for one specific purpose.

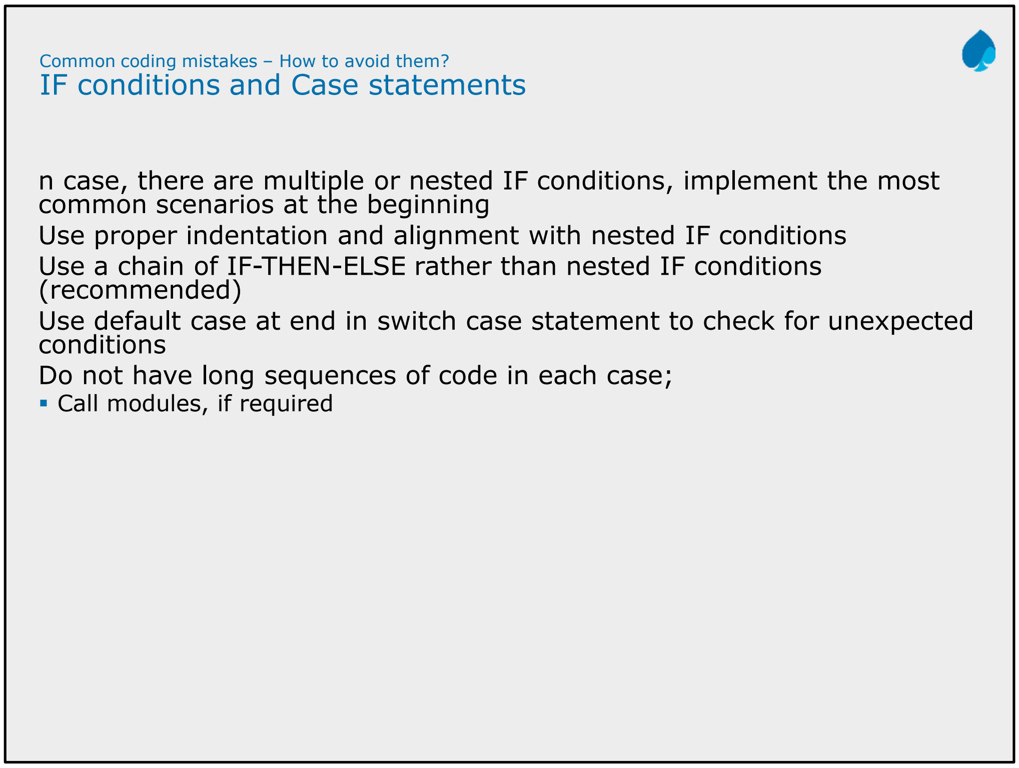
- Avoid variables with hidden meanings.
- Do not use one range of values for one purpose and another range of values for another purpose.
- For example: Employee code above 90000 for temporary employees, employee code between 80000 and 90000 for contract workers

Use one variable for only one purpose.

For an example, if a variable is used for multiple different what can we provide? Consider a variable that is used to count the number of students, and count their grades. How would you name this variable: count, studentCount/gradeCount?

So create one variable for one purpose, otherwise data management related to the requirement will be complex.

Don’t use variable with hidden meanings because it might not be understandable by others.



Common coding mistakes – How to avoid them?

IF conditions and Case statements

n case, there are multiple or nested IF conditions, implement the most common scenarios at the beginning

Use proper indentation and alignment with nested IF conditions

Use a chain of IF-THEN-ELSE rather than nested IF conditions (recommended)

Use default case at end in switch case statement to check for unexpected conditions

Do not have long sequences of code in each case;

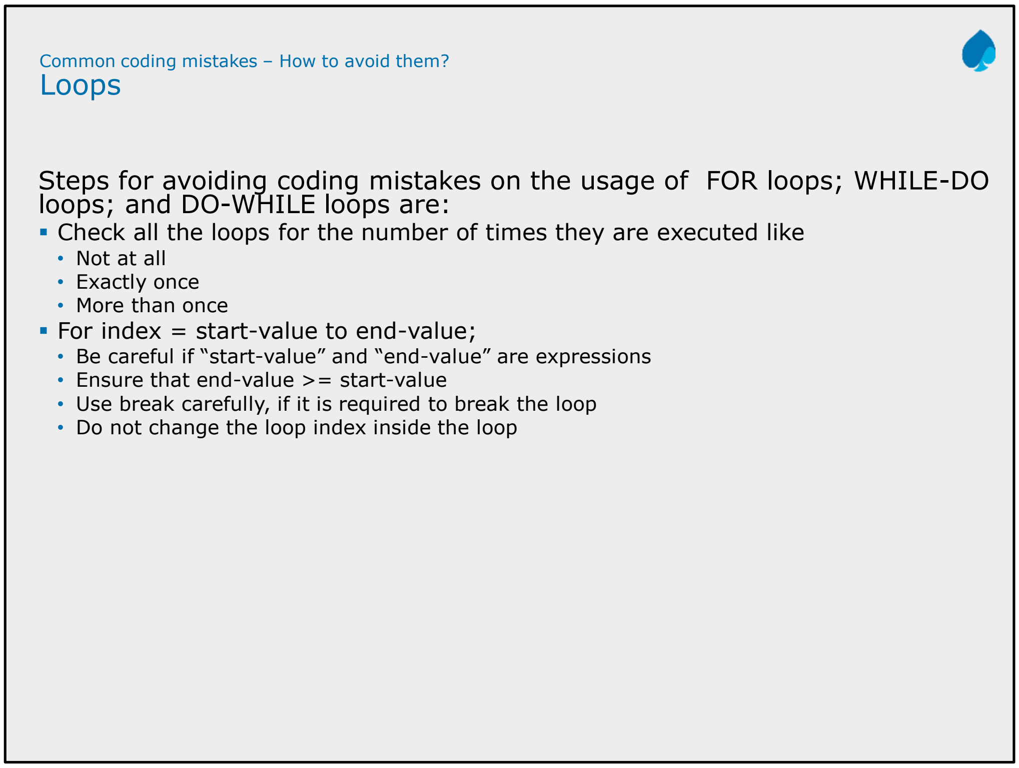
- Call modules, if required

How to avoid Common Coding Mistakes while using IF and case statements:

If there are multiple nested IF conditions, then find out the condition which has high priority and specify the same in first if condition.
Adopt a standard indentation style for your code, and stick with it throughout the program so that program will be easily readable by other programmers.
For an Example:

```
BEGIN
    DECLARE file AS FILE
    PROMPT "Enter the filename" AND STORE IN file
    IF (theFileExits) THEN
            determine the length of the file
             IF(FileLength>o) THEN
                readFile(file); // Invoke readFile module
             ELSE
                PRINT "File doesn't contain data"
              END IF
    ELSE
                PRINT "File doesn't exist"
    END IF
END
```

In the above code, indentation is followed, if-else is used and fileexists condition is checked first before finding length of a file for avoiding common mistakes.



Common coding mistakes – How to avoid them?

Loops

Steps for avoiding coding mistakes on the usage of FOR loops; WHILE-DO loops; and DO-WHILE loops are:

- Check all the loops for the number of times they are executed like
  - Not at all
  - Exactly once
  - More than once
- For index = start-value to end-value;
  - Be careful if "start-value" and "end-value" are expressions
  - Ensure that end-value >= start-value
  - Use break carefully, if it is required to break the loop
  - Do not change the loop index inside the loop

Steps for avoiding Common Coding Mistakes while using Loops are:

Check for the number of times the loop executed: For an example, execute the below loop thrice, by considering num values as 5, 4 and 2.

```
WHILE (num<5)
DO
 ....
END WHILE
```

The above loop will be executed once for the value of 4, and the loop will not be executed at all for the num value of 5 and the same loop will be executed more than once if num value is 2.

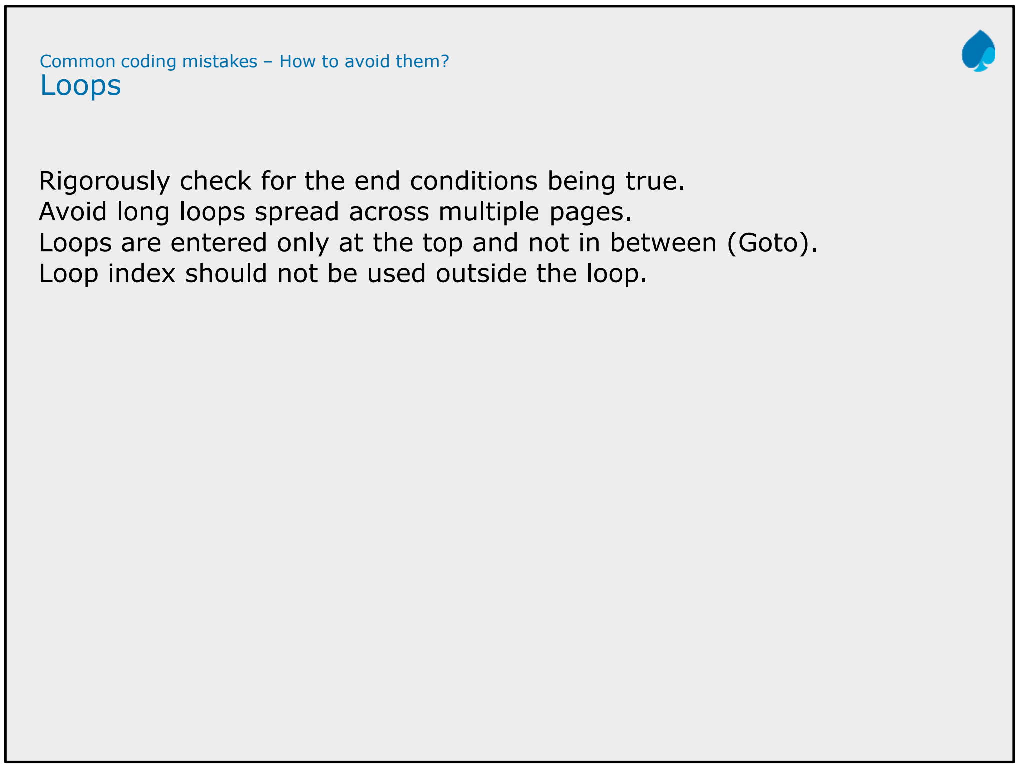
For loop:

Ensure that endvalue>=startvalue.

```
Better version                  Not recommended to be used
FOR index= 0 to 10               FOR index=10 to 0
..                               ...
END LOOP                         END LOOP
```

Do not change the loop index within the loop as shown in the below example. In the below example, loop execution will be stopped after the re-initialization of loop index inside the loop.

```
FOR index = 0 to 10
      index=13
END LOOP
```



Common coding mistakes – How to avoid them?

Loops

Rigorously check for the end conditions being true.
Avoid long loops spread across multiple pages.
Loops are entered only at the top and not in between (Goto).
Loop index should not be used outside the loop.

Steps for avoiding Common Coding Mistakes while using Loops are:

Rigorously check for the end conditions being true : Rigorously avoid coding infinite loops.

Avoid long loops spread across multiple pages : The loop should be short enough to view all loop code at once.

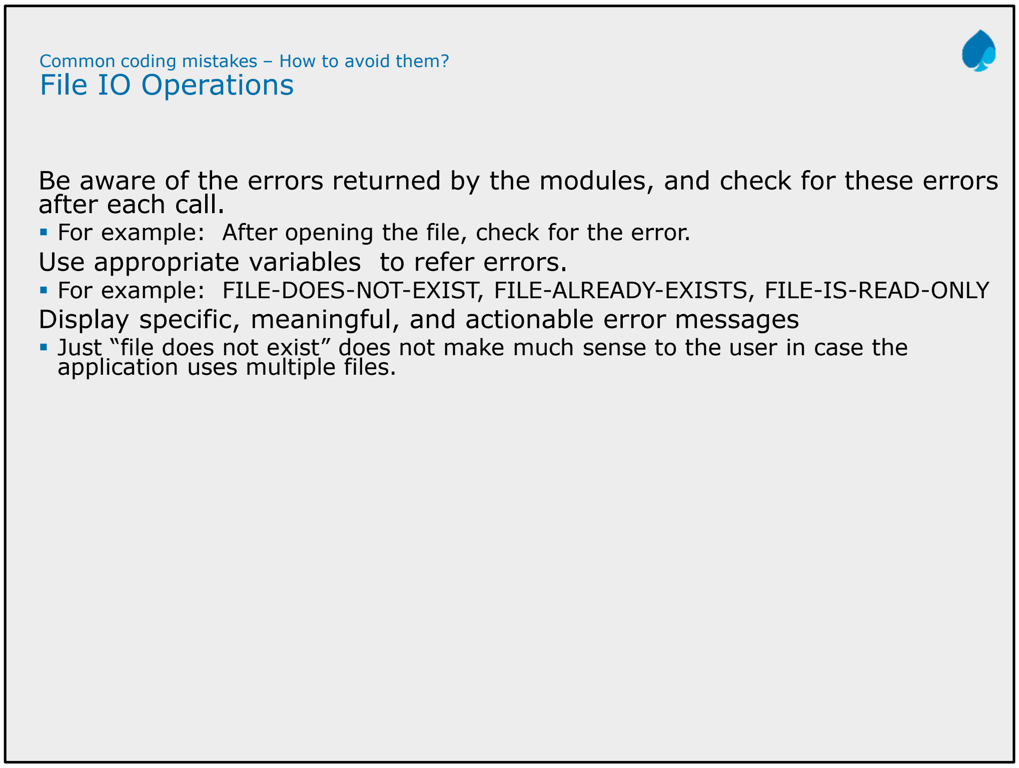
Loops are entered only at the top and not in between (Goto) : "goto" statement is always not recommended to be used inside loop as the loop execution always re start from the lower limit.

Loop index should not be used outside the loop. Consider the below code

```
FOR index=1 to 5
END LOOP
index=3;
```

Don't use index outside of the loop. Use another variable, if required as shown below:

```
FOR index=1 to 5
END LOOP
num=3;
```



Common coding mistakes – How to avoid them?

File IO Operations

Be aware of the errors returned by the modules, and check for these errors after each call.

- For example: After opening the file, check for the error.

Use appropriate variables to refer errors.

- For example: FILE-DOES-NOT-EXIST, FILE-ALREADY-EXISTS, FILE-IS-READ-ONLY

Display specific, meaningful, and actionable error messages

- Just "file does not exist" does not make much sense to the user in case the application uses multiple files.

Points to be considered for avoiding Common Coding Mistakes while performing file IO operations are:

Be aware of the errors returned by the modules, and check for these errors after each call like

Check for the error, if file doesn't exists

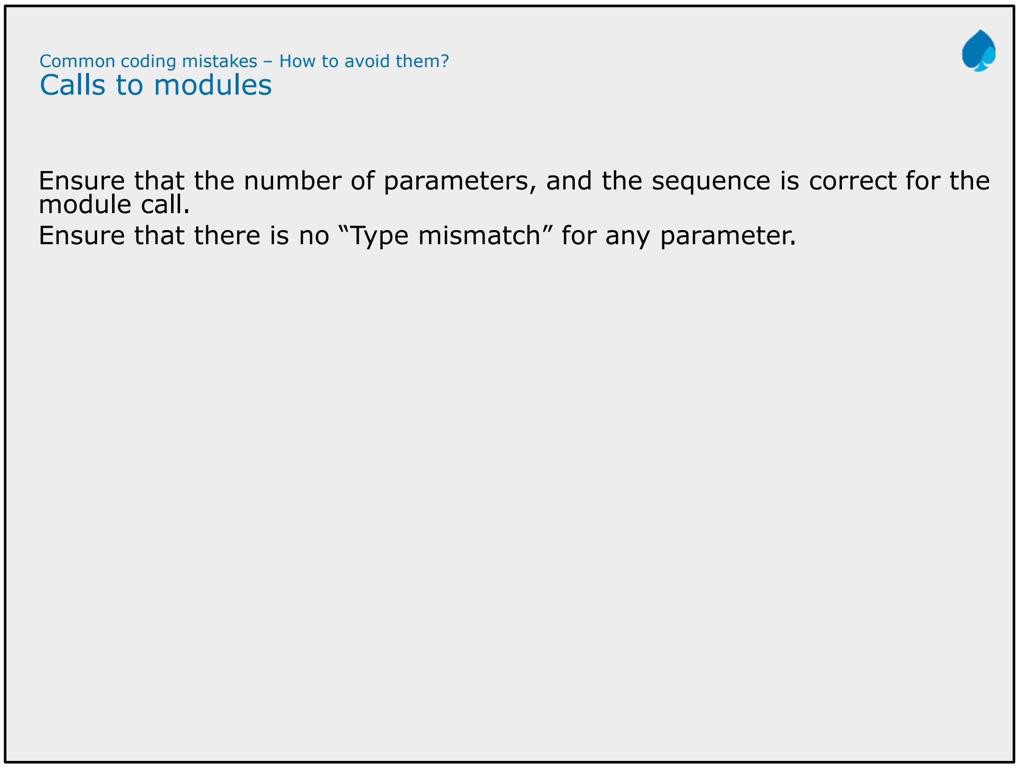
After opening the file, check for the error.

Check for the error, if file size is zero.

Check for the error, if file is not readable/writable.

Use appropriate variables to refer errors. For example: FILE-DOES-NOT-EXIST, FILE-ALREADY-EXISTS, FILE-IS-READ-ONLY

Display specific, meaningful, and actionable error messages. For an Example, Just "file does not exist" does not make much sense to the user in case the application uses multiple files. Instead use the error message as "Employee.txt file does not exist", considering "Employee.txt" is a filename.



Common coding mistakes – How to avoid them?

Calls to modules

Ensure that the number of parameters, and the sequence is correct for the module call.

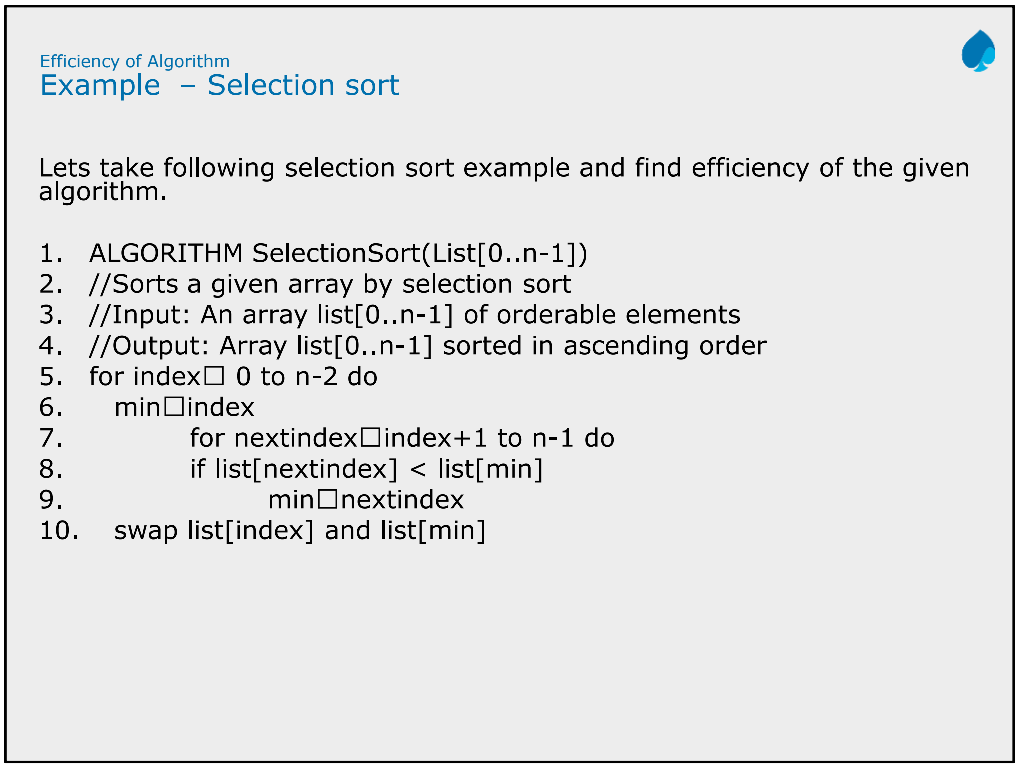
Ensure that there is no “Type mismatch” for any parameter.

Consider the below signature of a module to calculate total price.

```
calculateTotal(Integer price, Integer quantity)
```

Refer the valid and invalid statements to invoke a module

```
calculateTotal(3,5); //Valid

calculateTotal(4,3,4); //Invalid

calculateTotal(‘Test’,3); //Invalid
```

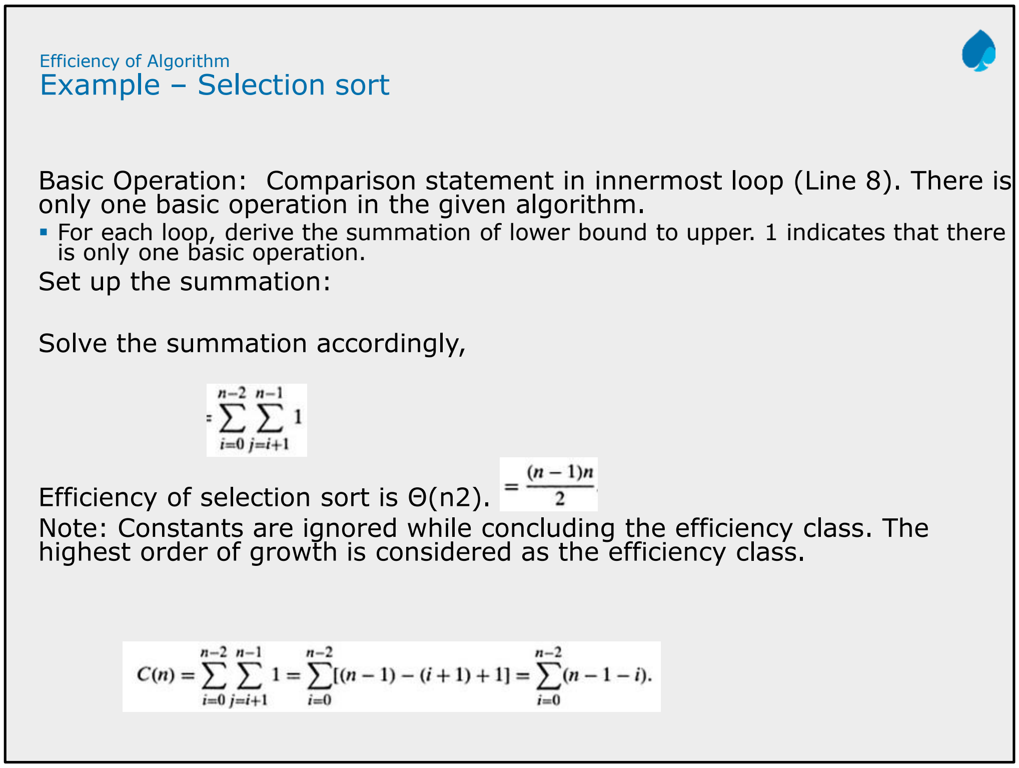


Efficiency of Algorithm

Example – Selection sort

Lets take following selection sort example and find efficiency of the given algorithm.

```
1.  ALGORITHM SelectionSort(List[0..n-1])
2.  //Sorts a given array by selection sort
3.  //Input: An array list[0..n-1] of orderable elements
4.  //Output: Array list[0..n-1] sorted in ascending order
5.  for index□ 0 to n-2 do
6.     min□index
7.           for nextindex□index+1 to n-1 do
8.           if list[nextindex] < list[min]
9.                 min□nextindex
10.    swap list[index] and list[min]
```



Efficiency of Algorithm

Example – Selection sort

Basic Operation: Comparison statement in innermost loop (Line 8). There is only one basic operation in the given algorithm.

- For each loop, derive the summation of lower bound to upper. 1 indicates that there is only one basic operation.

Set up the summation:

Solve the summation accordingly,

$$= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \frac{(n-1)n}{2}$$

Efficiency of selection sort is Θ(n2).

Note: Constants are ignored while concluding the efficiency class. The highest order of growth is considered as the efficiency class.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

How it works?

Basic Operation: Comparison statement in the innermost loop. Summation of lower bound to upper bound. For each for loop derive the summation. Hence as given algorithm has 2 loops, we end up with 2 summations. 1 indicates that there is only one basic operation. By this we get the expression.

Now solving the above expression with proper formula's, we end up with (n-1)n/2, which results in (n2-n)/2. Highest order of growth from the expression will be considered as the efficiency class. Hence we say, efficiency is O(n2). If we have a series of statements within a loop, it would not matter because we are considering the basic operation which will take care of finding the highest order of growth. And we are only interested in highest order of growth, neither lesser order nor constants.

Efficiency of Algorithm

Example – Selection sort

Notes Page

[For easy understanding lets use index as i and nextindex as j in solving the efficiency]

Summation formula used:

$$\sum_{i=l}^{u} 1 = u-l+1$$ where l<=u are lower and upper integer limits.

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1+2+3+\ldots+n = n(n+1)/2 \approx (1/2)n2 \in O(n2)$$

Solution in detail:

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

Efficiency of Algorithm

Example – Selection sort

Notes Page

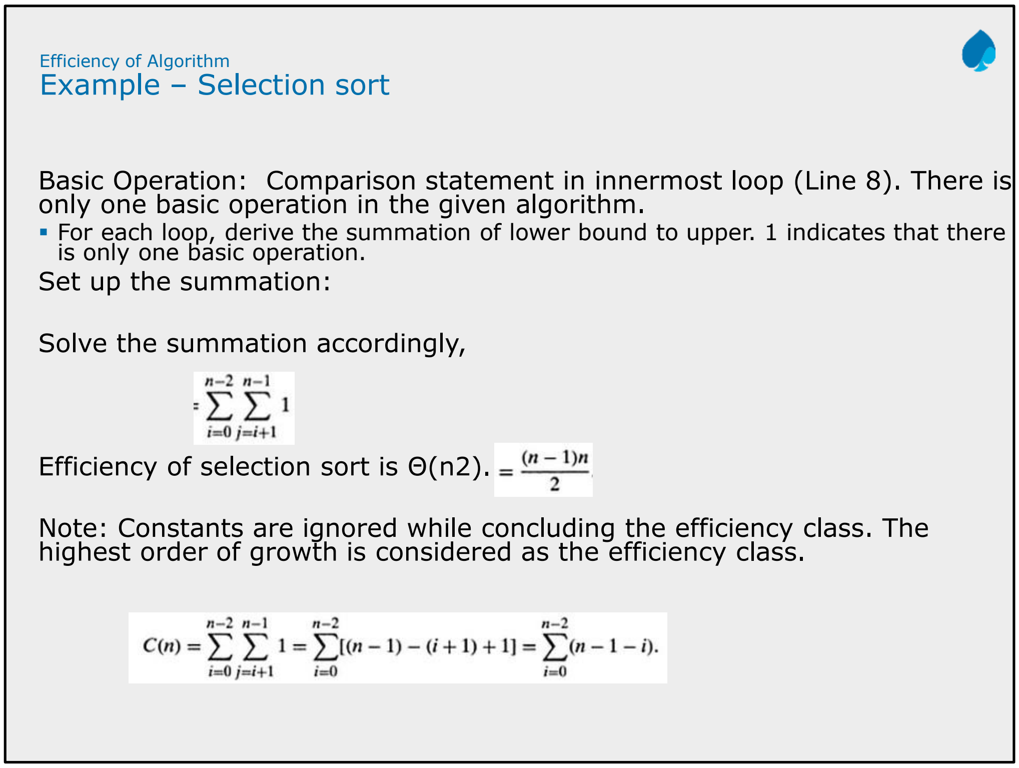
$$=(n-1)\sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$=(n-1)\sum_{i=0}^{n-2} 1 - (n-2)(n-1)/2$$

$$=(n-1)(n-1) - (n-2)(n-1)/2$$

$$=(n-1)((n-1) - (n-2)/2)$$

$$=(n-1)(2n-2-n+2)/2$$
$$=(n-1)(n)/2$$
$$=(n2-n)/2$$
$$=n2/2-n/2$$
$$\in O(n2)$$



Efficiency of Algorithm

Example – Selection sort

Basic Operation: Comparison statement in innermost loop (Line 8). There is only one basic operation in the given algorithm.

- For each loop, derive the summation of lower bound to upper. 1 indicates that there is only one basic operation.

Set up the summation:

Solve the summation accordingly,

$$= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Efficiency of selection sort is Θ(n2). $= \frac{(n-1)n}{2}$

Note: Constants are ignored while concluding the efficiency class. The highest order of growth is considered as the efficiency class.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

How it works?

Basic Operation: Comparison statement in the innermost loop. Summation of lower bound to upper bound. For each for loop derive the summation. Hence as given algorithm has 2 loops, we end up with 2 summations. 1 indicates that there is only one basic operation. By this we get the expression.

Now solving the above expression with proper formula's, we end up with (n-1)n/2, which results in (n2-n)/2. Highest order of growth from the expression will be considered as the efficiency class. Hence we say, efficiency is O(n2). If we have a series of statements within a loop, it would not matter because we are considering the basic operation which will take care of finding the highest order of growth. And we are only interested in highest order of growth, neither lesser order nor constants.

[For easy understanding lets use index as i and nextindex as j in solving the efficiency]

Summation formula used:

$\sum_{i=l}^{u} 1 = u-l+1$ where $l<=u$ are lower and upper integer limits.

$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1+2+3+\ldots+n = n(n+1)/2 \approx (1/2)n^2 \in O(n^2)$

Solution in detail:

$=\sum_{i=0}^{n-2} (n-1-i)$

$=\sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$

$=(n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$

$=(n-1) \sum_{i=0}^{n-2} 1 - (n-2)(n-1)/2$

$=(n-1)(n-1) – (n-2)(n-1)/2$

$=(n-1)((n-1) – (n-2)/2)$

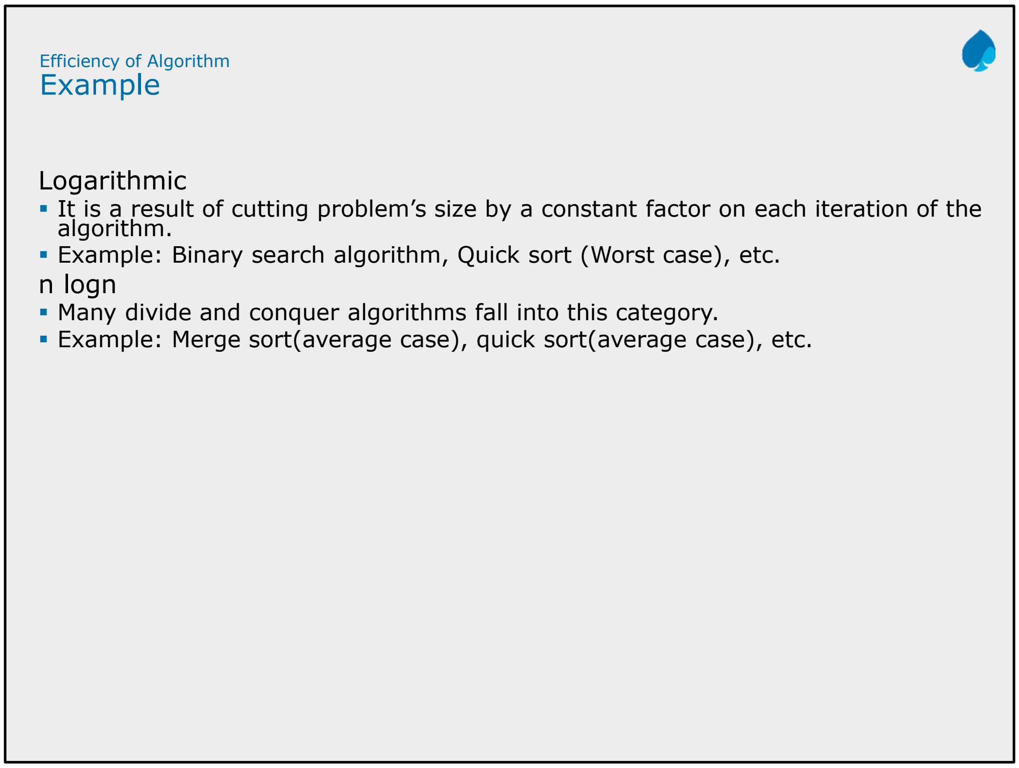
$=(n-1)(2n-2-n+2)/2$

$=(n-1)(n)/2$

$=(n^2-n)/2$

$=n^2/2-n/2$

$\in O(n^2)$



Efficiency of Algorithm

Example

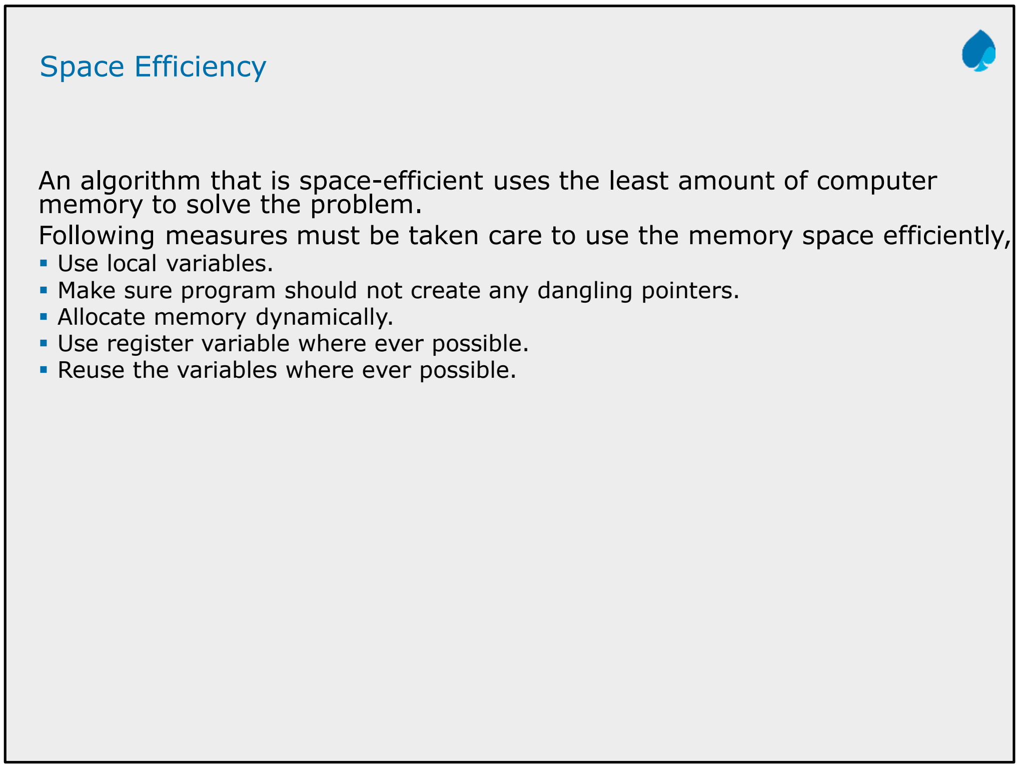
Logarithmic

- It is a result of cutting problem's size by a constant factor on each iteration of the algorithm.
- Example: Binary search algorithm, Quick sort (Worst case), etc.

n logn

- Many divide and conquer algorithms fall into this category.
- Example: Merge sort(average case), quick sort(average case), etc.

After constant, log n is the next best efficiency class. Logarithmic algorithm will not take all into account. Any algorithm that does so will at least have Linear running time. Examples of algorithms which fall under this category are Binary search, Quick sort etc.

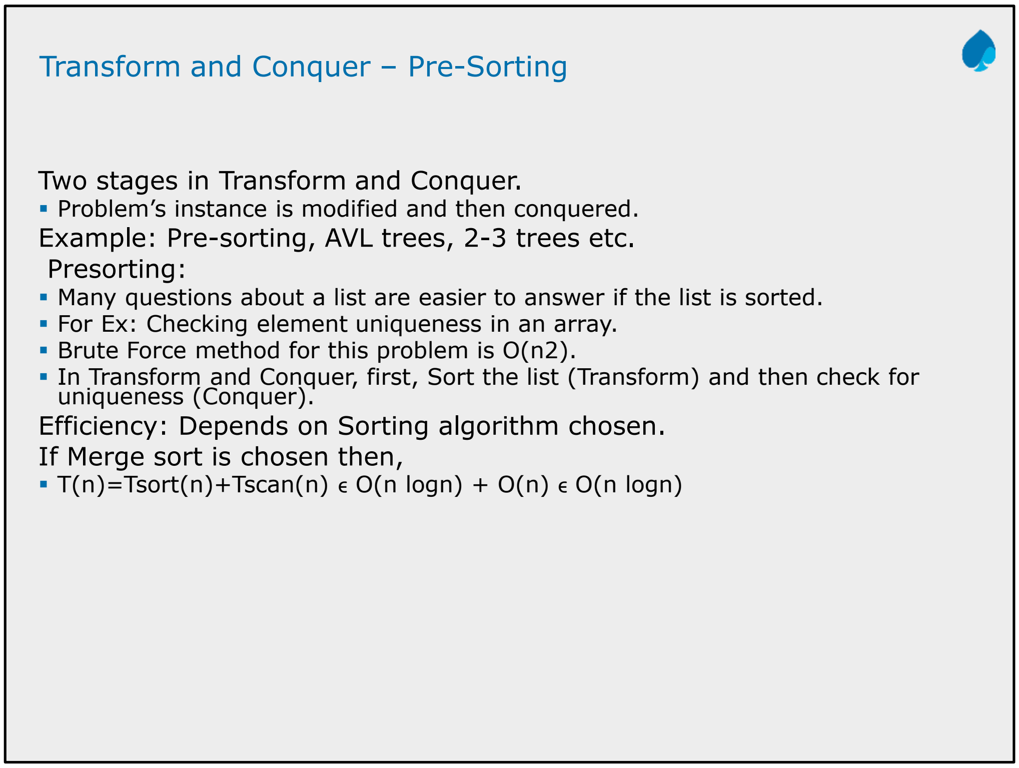


Space Efficiency

An algorithm that is space-efficient uses the least amount of computer memory to solve the problem.

Following measures must be taken care to use the memory space efficiently,

- Use local variables.
- Make sure program should not create any dangling pointers.
- Allocate memory dynamically.
- Use register variable where ever possible.
- Reuse the variables where ever possible.



Transform and Conquer – Pre-Sorting

Two stages in Transform and Conquer.

- Problem's instance is modified and then conquered.

Example: Pre-sorting, AVL trees, 2-3 trees etc.

Presorting:

- Many questions about a list are easier to answer if the list is sorted.
- For Ex: Checking element uniqueness in an array.
- Brute Force method for this problem is O(n2).
- In Transform and Conquer, first, Sort the list (Transform) and then check for uniqueness (Conquer).

Efficiency: Depends on Sorting algorithm chosen.

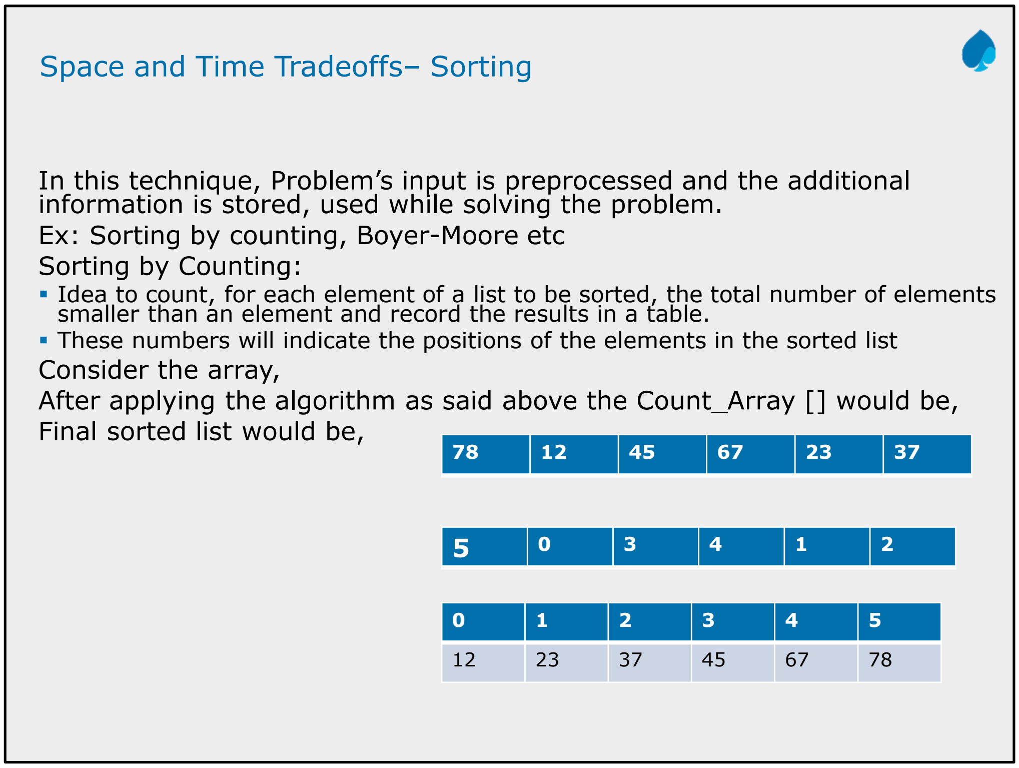
If Merge sort is chosen then,

- T(n)=Tsort(n)+Tscan(n) ϵ O(n logn) + O(n) ϵ O(n logn)

This technique is based on the idea of transformation. These methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.
Pre-sorting as an example is discussed in detail in the above given slides. Many questions about a list are easier to answer if the list is already sorted. One such example is the problem to find element uniqueness. This can be easily solved if the list/array is already sorted. Hence sorting the array is transformation to the simpler instance and then solving the problem.

The brute-force algorithm compares pairs of the array's elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency would be O(n2).
Alternatively, we can sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each
other, and vice versa.

```
ALGORITHM ElementUniqueness(List[0..n − 1])
//Solves the element uniqueness problem by sorting the array first
//Input: An array List[0..n − 1] of orderable elements
//Output: Returns “true” if List has no equal elements, “false” otherwise
sort the array List
for index←0 to n − 2 do
    if A[index]= A[index + 1] return false
return true
```



Space and Time Tradeoffs– Sorting

In this technique, Problem’s input is preprocessed and the additional information is stored, used while solving the problem.
Ex: Sorting by counting, Boyer-Moore etc
Sorting by Counting:

- Idea to count, for each element of a list to be sorted, the total number of elements smaller than an element and record the results in a table.
- These numbers will indicate the positions of the elements in the sorted list

Consider the array,
After applying the algorithm as said above the Count_Array [] would be,
Final sorted list would be,

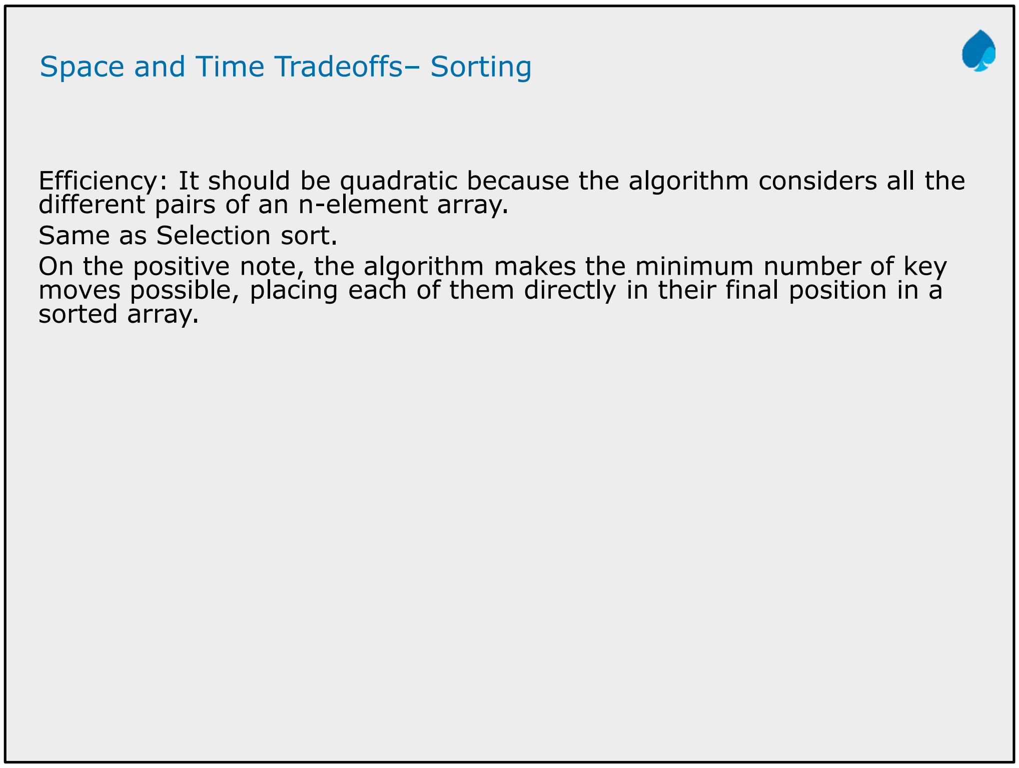
78	12	45	67	23	37

5	0	3	4	1	2

0	1	2	3	4	5
12	23	37	45	67	78

In this technique, Problem’s input is preprocessed and the additional information is stored, used while solving the problem. Idea to count, for each element of a list to be sorted, the total number of elements smaller than an element and record the results in a table. These numbers will indicate the positions of the elements in the sorted list e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.
Algorithm for your reference is given below.

```
ALGORITHM ComparisonCountingSort(List[0..n − 1])
//Sorts an array by comparison counting
//Input: An array List[0..n − 1] of orderable elements
//Output: Array Sort[0..n − 1] of A’s elements sorted in nondecreasing order
for index←0 to n − 1 do Count[index]←0
for index ←0 to n − 2 do
    for nextindex←index + 1 to n − 1 do
        if List[index]<List[jnextindex]
            Count[nextindex ]←Count[nextindex]+ 1
        else Count[index]←Count[index]+ 1
for index ←0 to n − 1 do Sort[Count[index]]←List[index]
return Sort
```

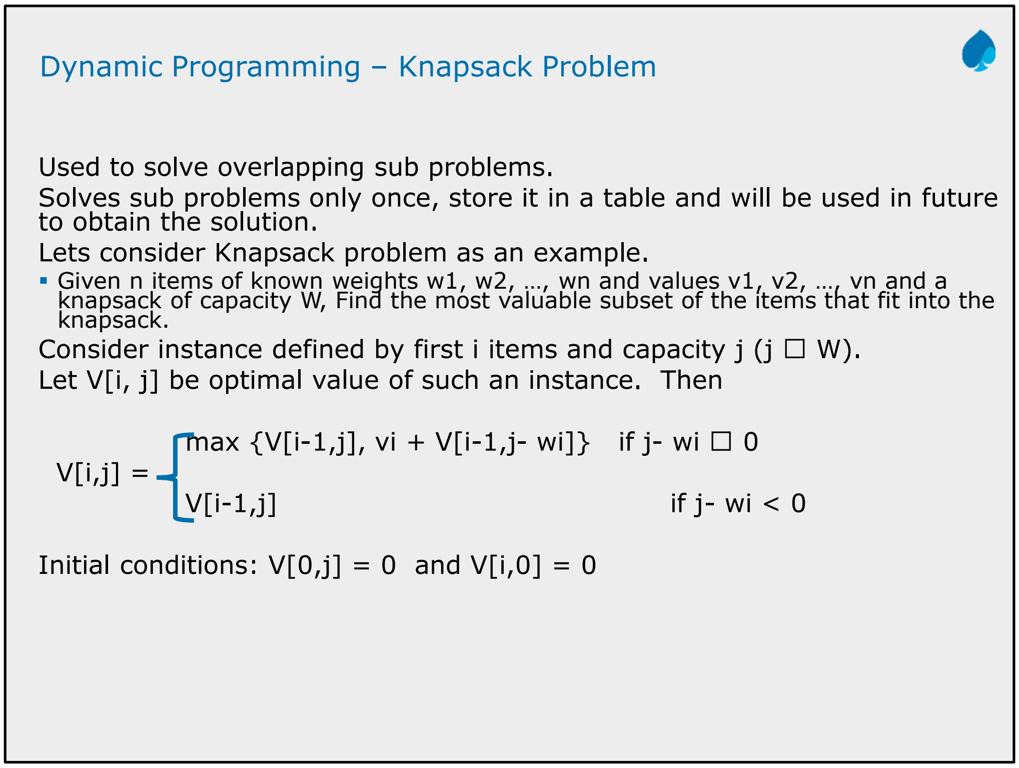


Space and Time Tradeoffs– Sorting

Efficiency: It should be quadratic because the algorithm considers all the different pairs of an n-element array.
Same as Selection sort.
On the positive note, the algorithm makes the minimum number of key moves possible, placing each of them directly in their final position in a sorted array.

e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.

```
ALGORITHM ComparisonCountingSort(List[0..n − 1])
//Sorts an array by comparison counting
//Input: An array List[0..n − 1] of orderable elements
//Output: Array Sort[0..n − 1] of A's elements sorted in nondecreasing order
for index←0 to n − 1 do Count[index]←0
for index ←0 to n − 2 do
    for nextindex←index + 1 to n − 1 do
        if List[index]<List[jnextindex]
            Count[nextindex ]←Count[nextindex]+ 1
        else Count[index]←Count[index]+ 1
for index ←0 to n − 1 do Sort[Count[index]]←List[index]
return Sort
```



Dynamic Programming – Knapsack Problem

Used to solve overlapping sub problems.
Solves sub problems only once, store it in a table and will be used in future to obtain the solution.
Lets consider Knapsack problem as an example.

- Given n items of known weights w1, w2, …, wn and values v1, v2, …, vn and a knapsack of capacity W, Find the most valuable subset of the items that fit into the knapsack.

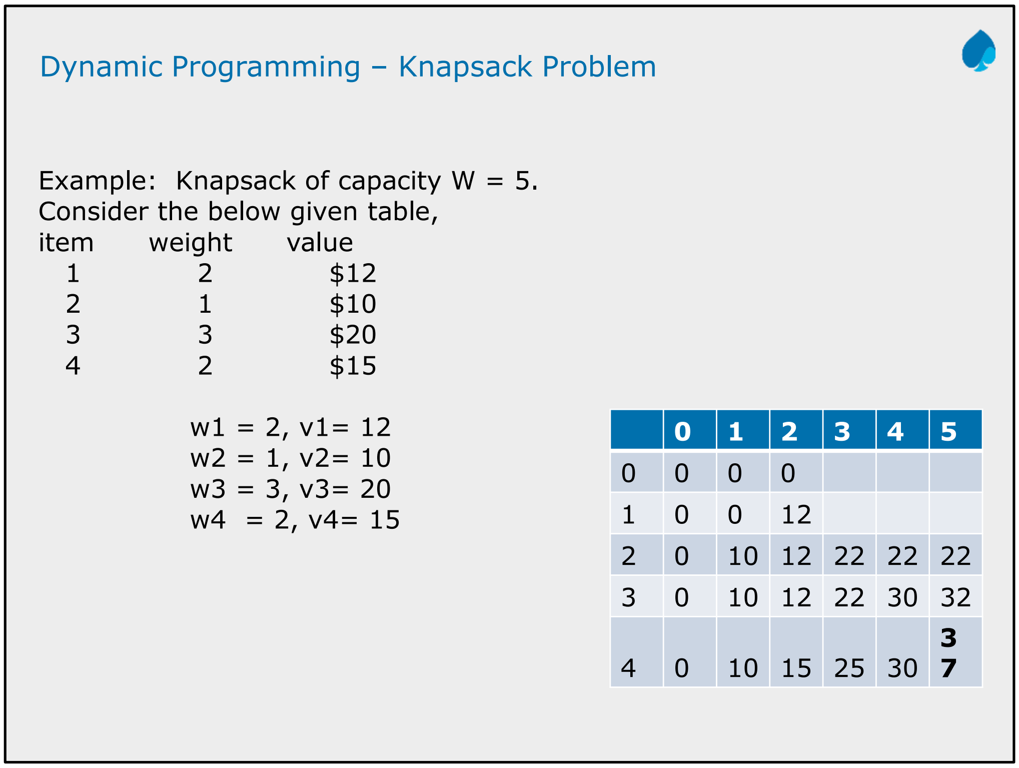
Consider instance defined by first i items and capacity j (j ☐ W).
Let V[i, j] be optimal value of such an instance. Then

$$V[i,j] = \begin{cases} \max \{V[i-1,j],\ vi + V[i-1,j- wi]\} & \text{if } j- wi \ \square\ 0 \\ V[i-1,j] & \text{if } j- wi < 0 \end{cases}$$

Initial conditions: V[0,j] = 0 and V[i,0] = 0

Dynamic programming is a technique for solving problems with overlapping subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Knapsack Problem: Given n items of known weights w1, w2, . . . , wn and values v1, v2, . . . , vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. Brute force approach for this problem leads to O(2n).



Dynamic Programming – Knapsack Problem

Example: Knapsack of capacity W = 5.
Consider the below given table,

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

w1 = 2, v1= 12
w2 = 1, v2= 10
w3 = 3, v3= 20
w4 = 2, v4= 15

	0	1	2	3	4	5
0	0	0	0			
1	0	0	12			
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**

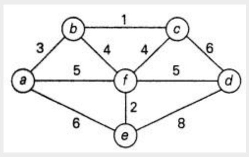
Using the formula given, compute and fill the table. After computing all values, the maximal value is F(4, 5) = $37.
Optimal solution is {1, 2, 4}. We can find the composition of an optimal subset by backtracing the computations of this entry in the table.
Since F(4, 5) > F(3, 5), item 4 has to be included in an optimal solution along with an optimal subset for filling 5 − 2 = 3 remaining units of the knapsack capacity. The value of the latter is F(3, 3). Since F(3, 3) = F(2, 3), item 3 need not be in an optimal subset. Since F(2, 3) > F(1, 3), item 2 is a part of an optimal selection, which leaves element F(1, 3 − 1) to specify its remaining composition. Similarly, since F(1, 2) > F(0, 2), item 1 is the final part of the optimal solution {item 1,item 2, item 4}.
The time efficiency and space efficiency of this algorithm are both in O(nW). The time needed to find the composition of an optimal solution is in O(n).

Greedy Technique – Kruskal’s Algorithm

It constructs the solution through a sequence of steps.
Example - Kruskal’s Algorithm.

- Algorithm begins by sorting the graph’s edges in non decreasing order of their weights.
- Start with empty sub graph.
- Scan the sorted list and add next edge on the list to the current sub graph. If the edge is creating a cycle then simply skip the edge.

Consider the following graph,



Sorted Edge List: (bc,1) (ef,2) (ab,3) (bf,4) (cf,4) (af,5) (df,5) (ae,6) (cd,6) (de, 8)

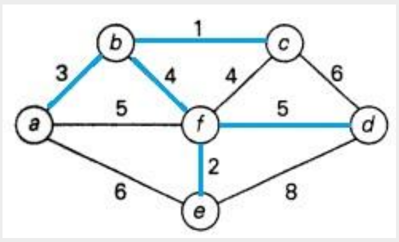
The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. Kruskal’s algorithm looks at a minimum spanning tree of a weighted connected graph G = V, E as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest.
Aspanning tree of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a
graph has weights assigned to its edges, a minimum spanning tree is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

Greedy Technique – Kruskal's Algorithm

Sub tree which forms minimum spanning tree is according to Kruskal's algorithm is

- (bc,1) (ef,2) (ab,3) (bf,4) (df,5)

Below figure shows the subgraph which is minimum spanning tree.



Algorithm begins by sorting the graph's edges in non decreasing order of their weights.
Start with empty sub graph.
Scan the sorted list and add next edge on the list to the current sub graph. If the edge is creating a cycle then simply skip the edge.