Chapter 1 : Basic Concepts

Goals

- To provide the tools and techniques necessary to design and implement large-scale computer systems.
    - solid foundation in data abstraction, algorithm specification and performance analysis and measurement provides the necessary methodology.

1.1 SYSTEM LIFE CYCLE

- **Requirement**
  - a set of specifications that define the purpose of the project.
  - input/output
- **Analysis**
  - break the problems down into manageable pieces.
  - bottom-up / top-down
- **Design**
  - creation of abstract data types
  - specification of algorithms and consideration of algorithm design strategies.

    (* language independent *)

- **Refinement and Coding**
  - choose representations for data objects and write algorithms for each operation on them.
  - data object's representation can determine the efficiency of the algorithms related to it.
- **Verification**
  - Developing correctness proof for the program
  - Testing the program with a variety of input data
  - Error removal
  - Performance analysis
    - running time
    - amount of memory used

1.3 ALGORITHM SPECIFICATION

1.3.1 Introduction

- ***Definition:***

  An algorithm is a finite set of instructions that, if followed, accomplishes particular task.

  All algorithms must satisfy the following criteria:

  (1) **Input**: There exist quantities that are externally supplied. (zero or more)

  (2) **Output**: At least one quantity is produced.

  (3) **Definiteness**: Each instruction is <u>clear</u> and <u>unambiguous</u>.

  (4) **Finiteness**: The algorithm <u>terminates</u> after a <u>finite number of steps</u>.

  (5) **Effectiveness**: Every instruction must be <u>feasible</u>.

  (6) **Correctness**: The algorithm is correct with respect to a specification.

1.3 ALGORITHM SPECIFICATION

1.3.1 Introduction

- Algorithm vs. Program (procedure)

→ Algorithm always terminates (for all cases).

→ Program does not have to satisfy “finiteness”

- **How to describe an algorithm**
  - natural language
  - flowchart
  - programming language

- **Example 1.1** **[Selection Sort]**

  Sorting a set of n ≥ 1 integers

  From those integers that are currently unsorted,
  find the smallest and place it next in the sorted list.

- **[Program 1.2   Selection sort algorithm]**

```
for (i=0; i<n; i++) {
     Examine list[i] to list[n-1]
      and suppose that the smallest integer is at list[min];
     Interchange list[i] and list[min];
}
```

- **first task : finding the smallest integer;**
- **second task : exchange;**

  either a function or a macro

[Program 1.3   swap function]

```
void swap(int *x, int *y)
/* both parameters are pointers to ints */
{
        int temp = *x; /* declares temp as an int and assigns to it
                           the contents of what x points to */
        *x = *y;       /* stores what y points to into the location
                          where x points  */
        *y = temp;     /* place the contents of temp in the location
                          pointed to by y */
}
```

Call --   swap(&a, &b);

- **macro version of swap -**

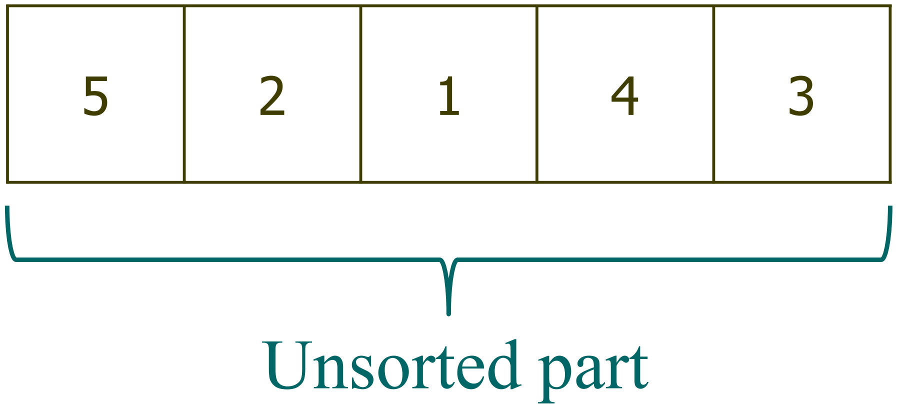
```
#define SWAP(x,y,t)    ((t) = (x), (x) = (y), (y) = (t))
```

- **[Program 1.4 Selection sort]**

```
void sort (int list[], int n)
{
        int i, j, min, temp;
        for (i=0; i<n-1; i++)  {
            min = i;
            for (j = i+1; j < n; j++)
                if (list[j] < list[min])
                    min = j;
            SWAP(list[i], list[min], temp);
        }
}
```

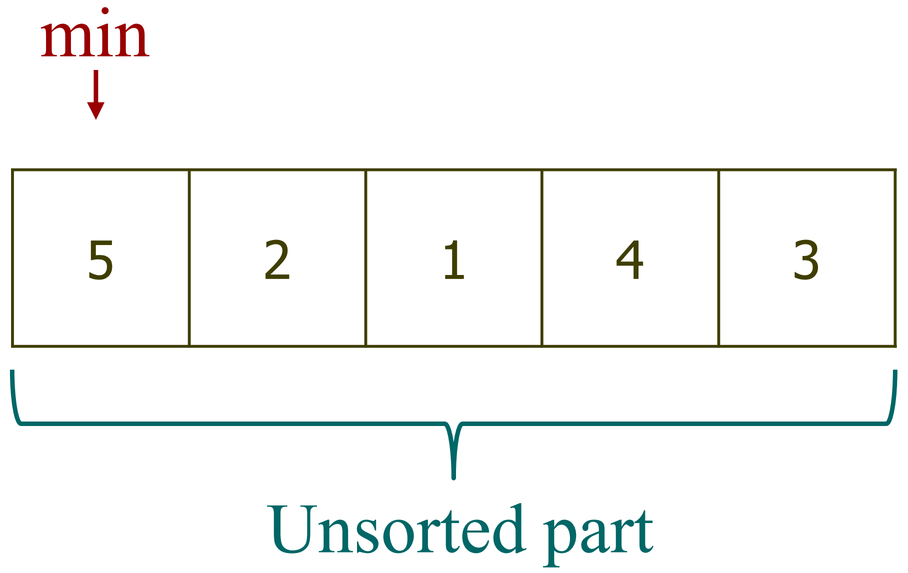
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

Initial state: unsorted array



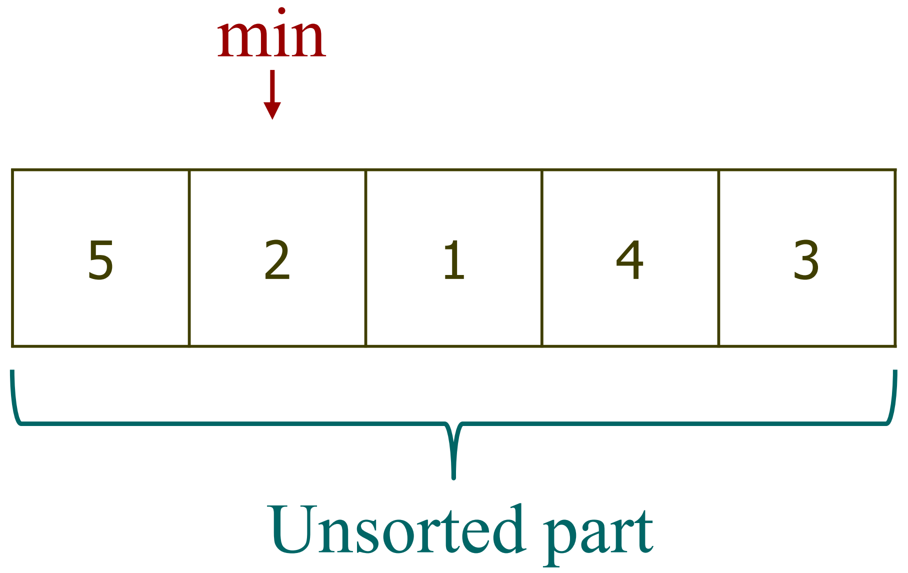
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 0, j = ?, min = 0



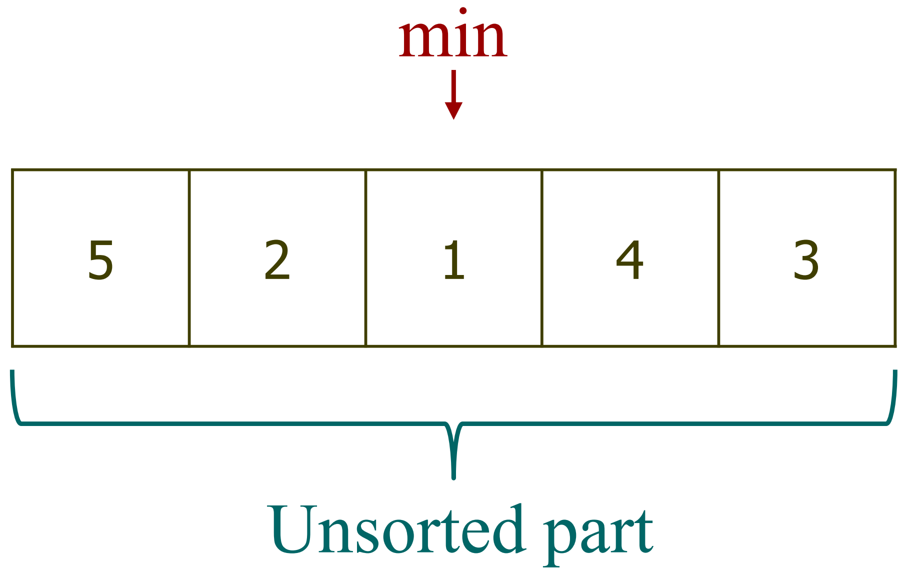
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 0, j = 1, min = 1



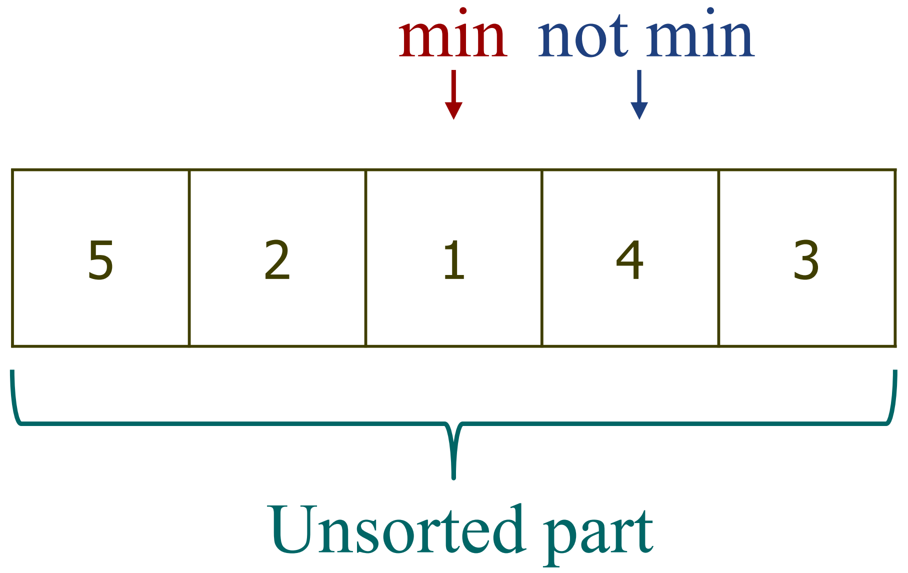
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                  temp);
    }
}
```

i = 0, j = 2, min = 2



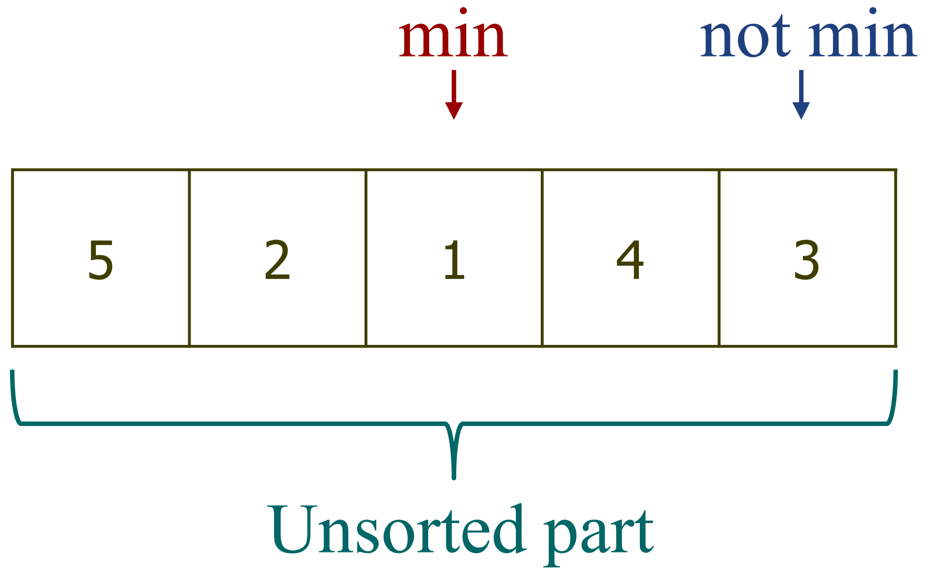
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                    temp);
    }
}
```

i = 0, j = 3, min = 2



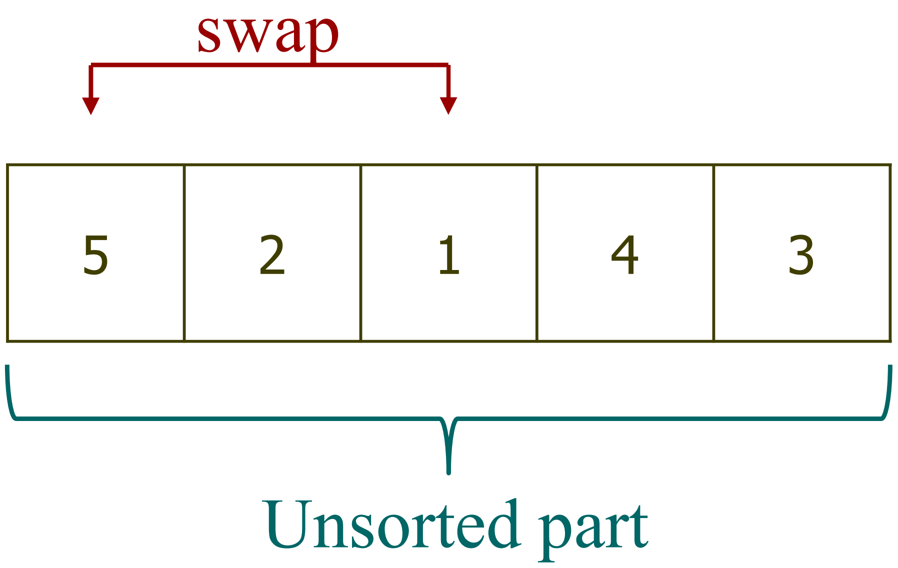
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                    temp);
    }
}
```

i = 0, j = 4, min = 2



```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 0, j = 4, min = 2

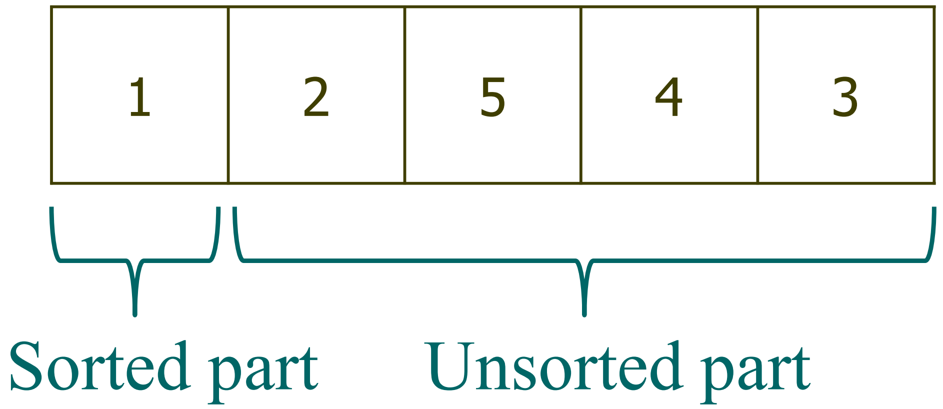


```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                  temp);
    }
}
```

After i=0:

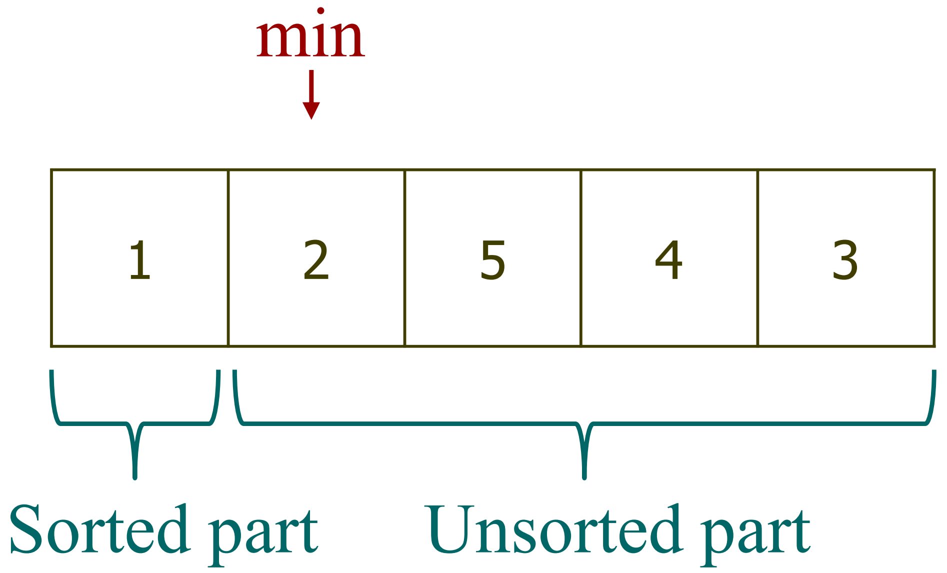
Sorted part: [1]

Unsorted part: [2][5][4][3]



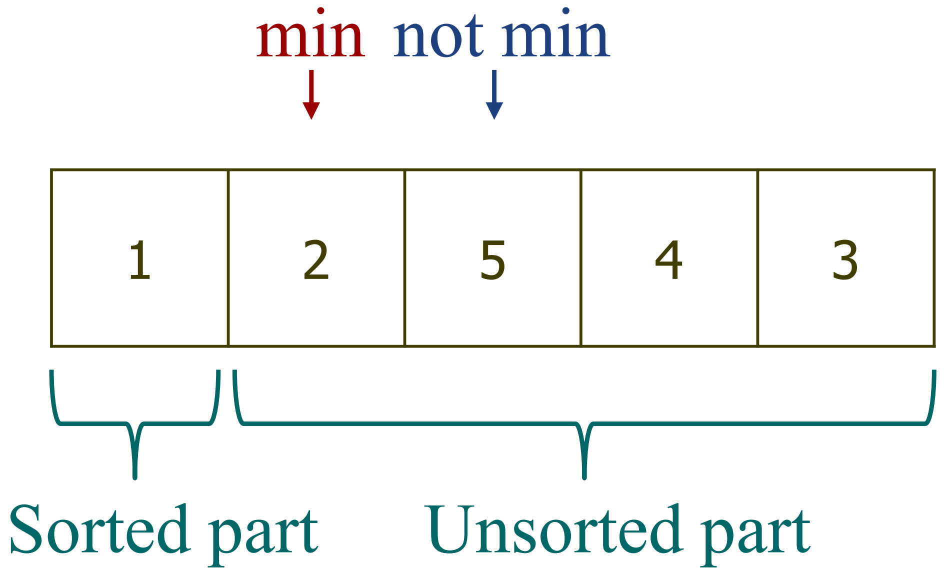
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 1, j = ?, min = 1



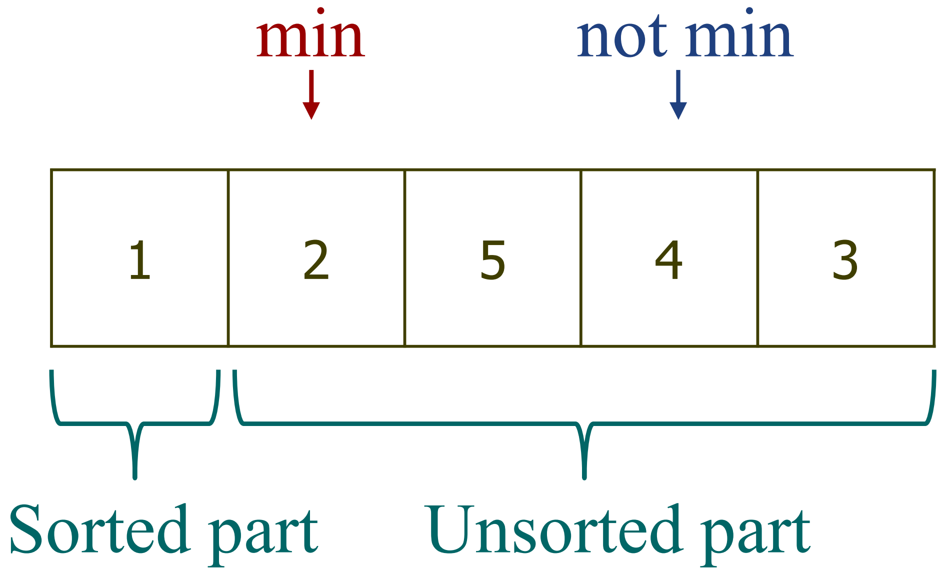
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                 temp);
    }
}
```

i = 1, j = 2, min = 1



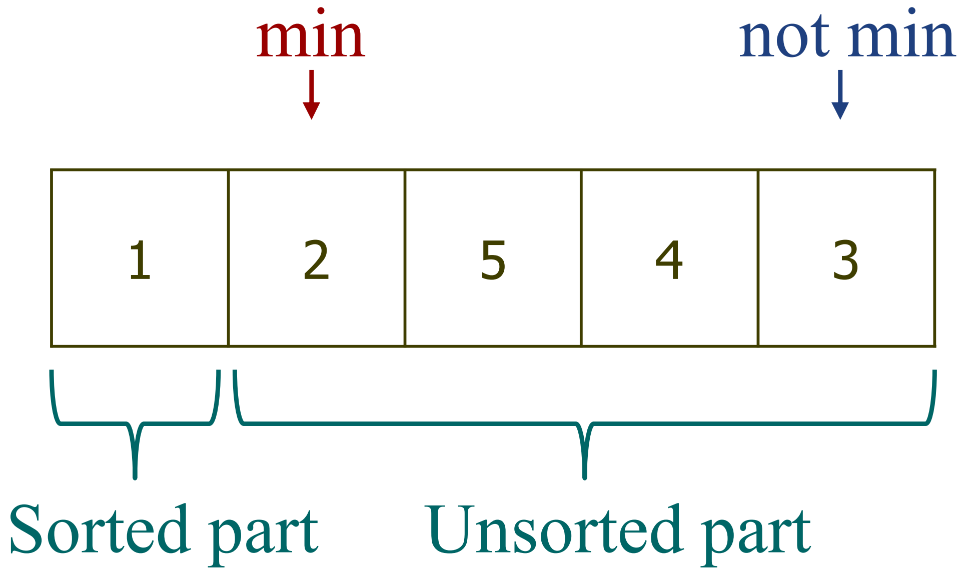
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 1, j = 3, min = 1



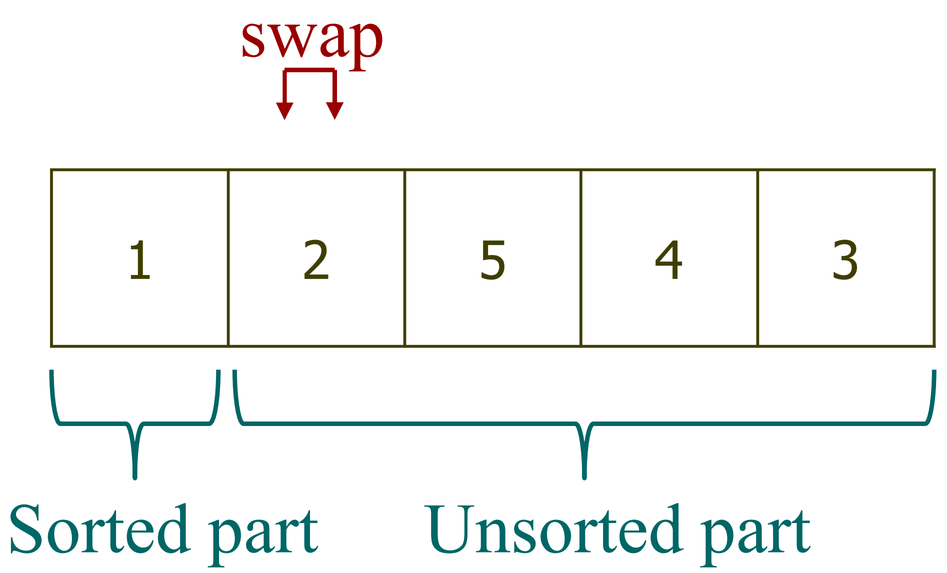
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                 temp);
    }
}
```

i = 1, j = 4, min = 1



```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 1, j = 4, min = 1

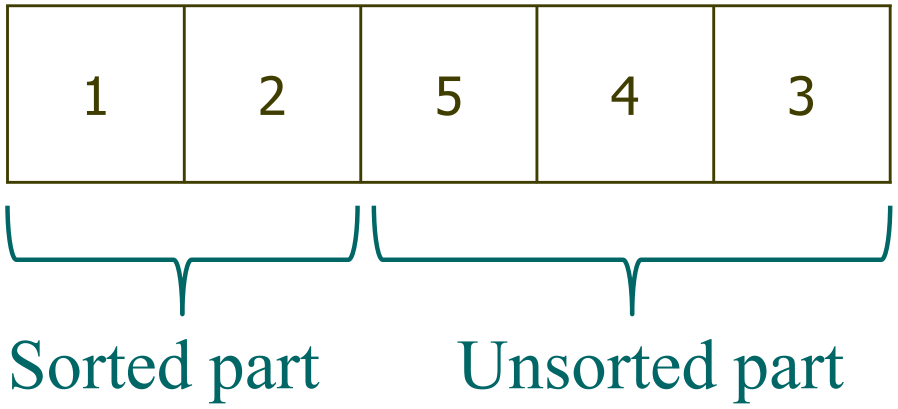


```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

After i=1:

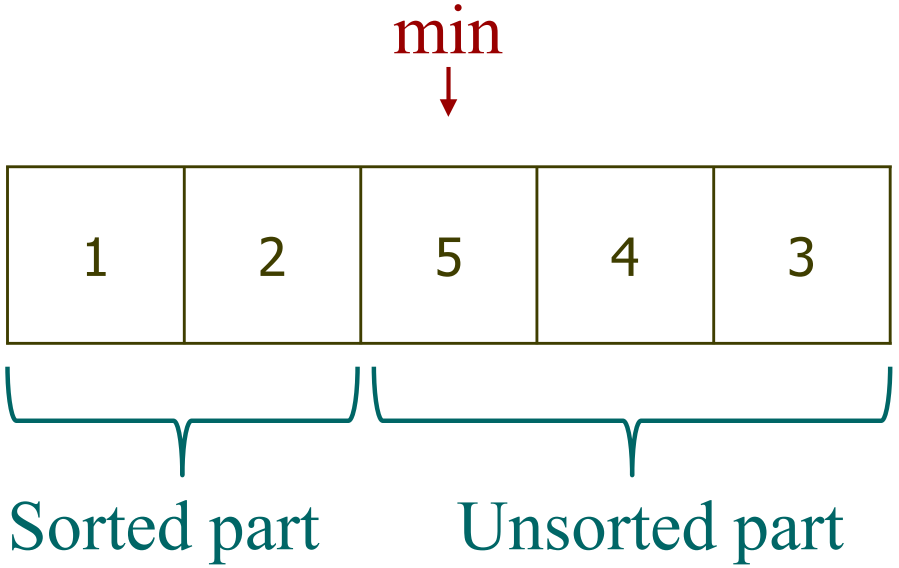
Sorted part: [1][2]

Unsorted part: [5][4][3]



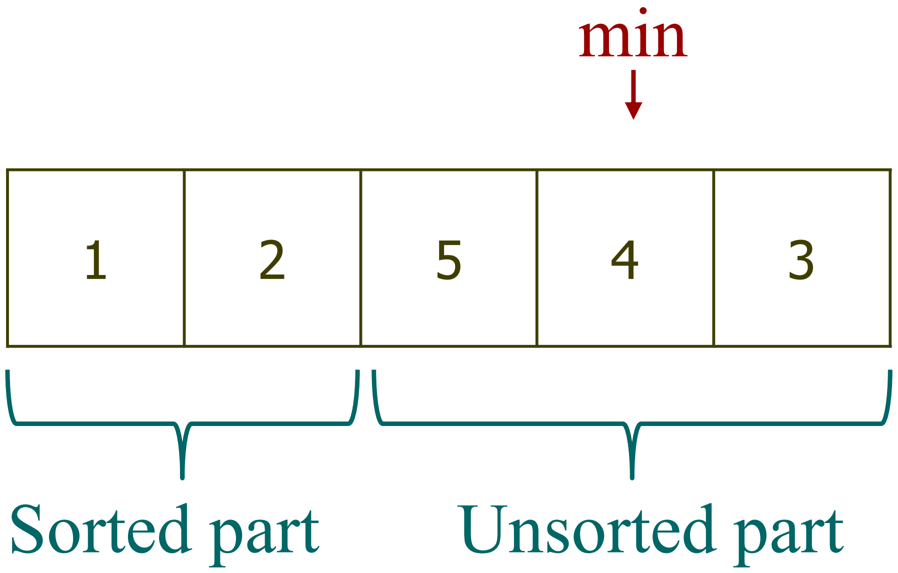
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 2, j = ?, min = 2



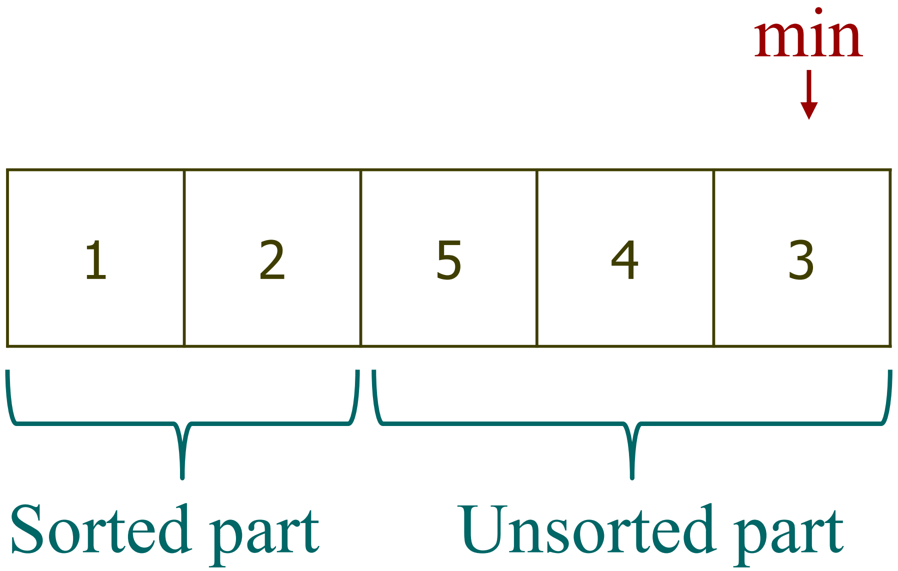
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 2, j = 3, min = 3



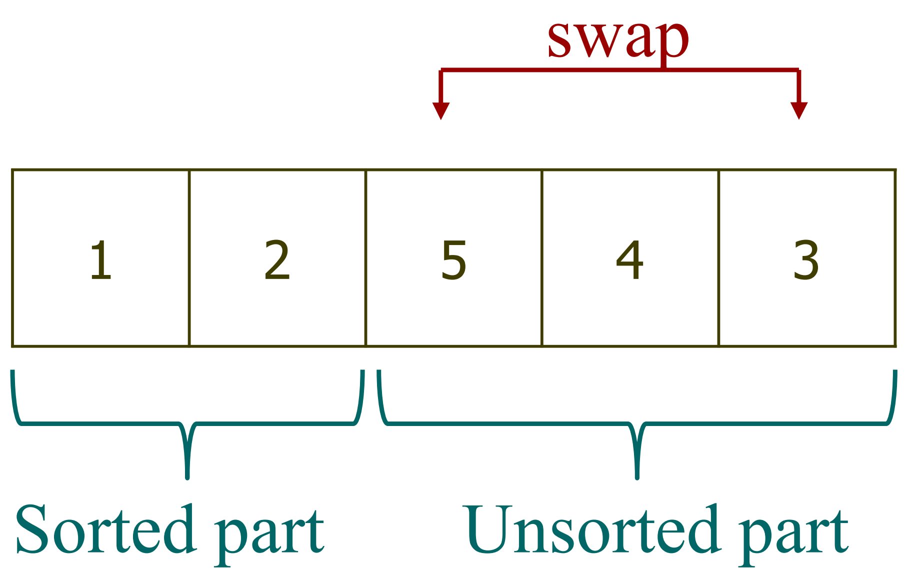
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 2, j = 4, min = 4



```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 2, j = 4, min = 4

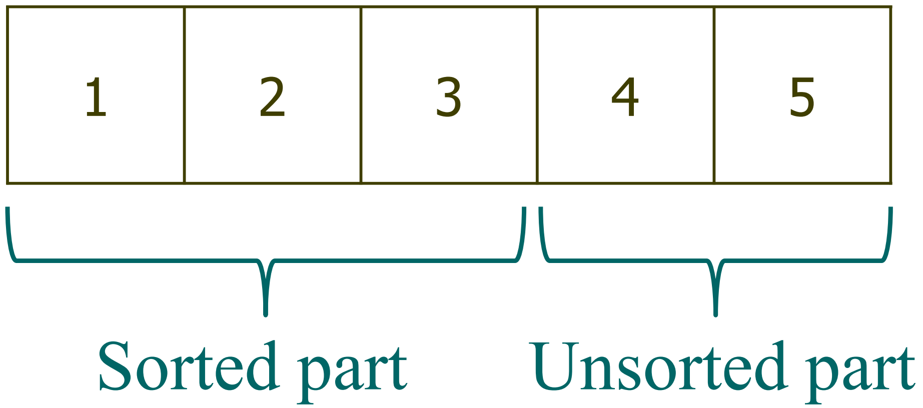


```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

After i=2:

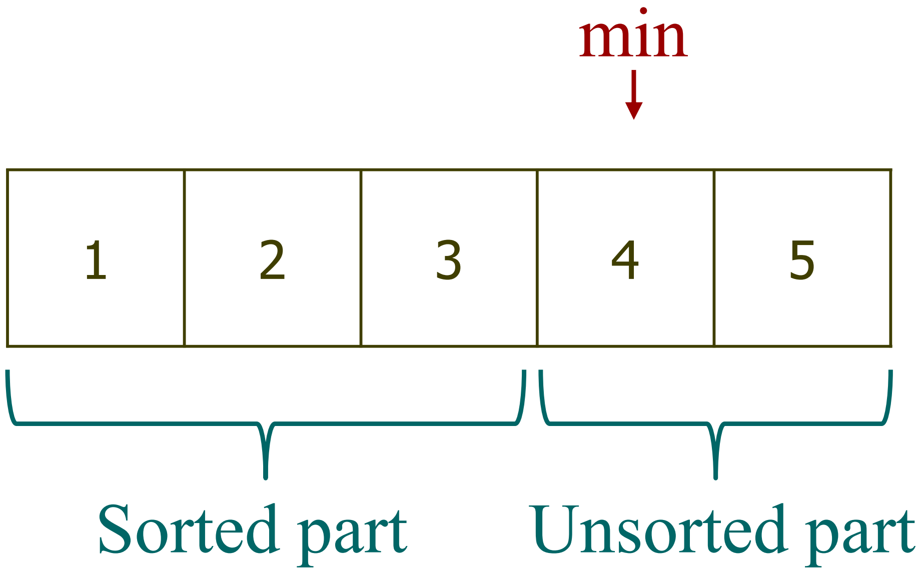
Sorted part: [1][2][3]

Unsorted part: [4][5]



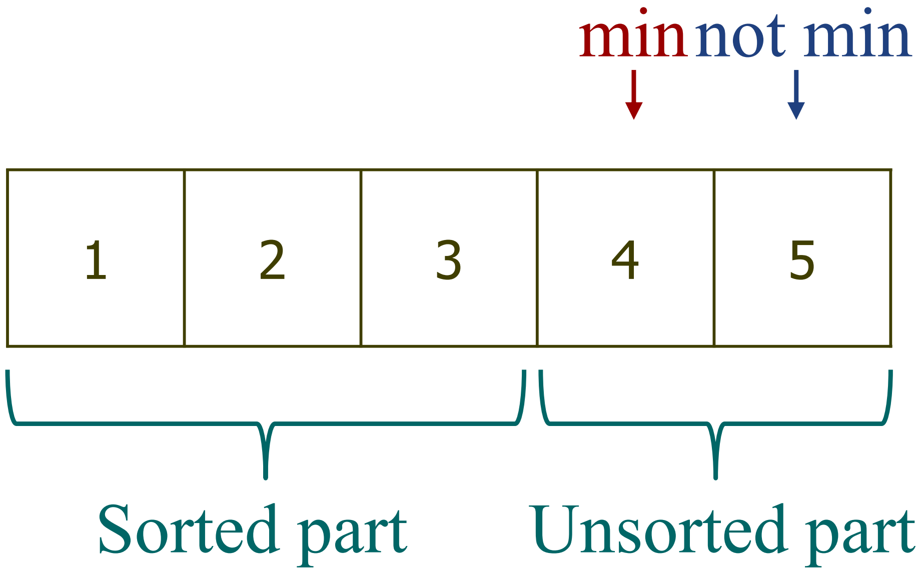
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 3, j = ?, min = 3



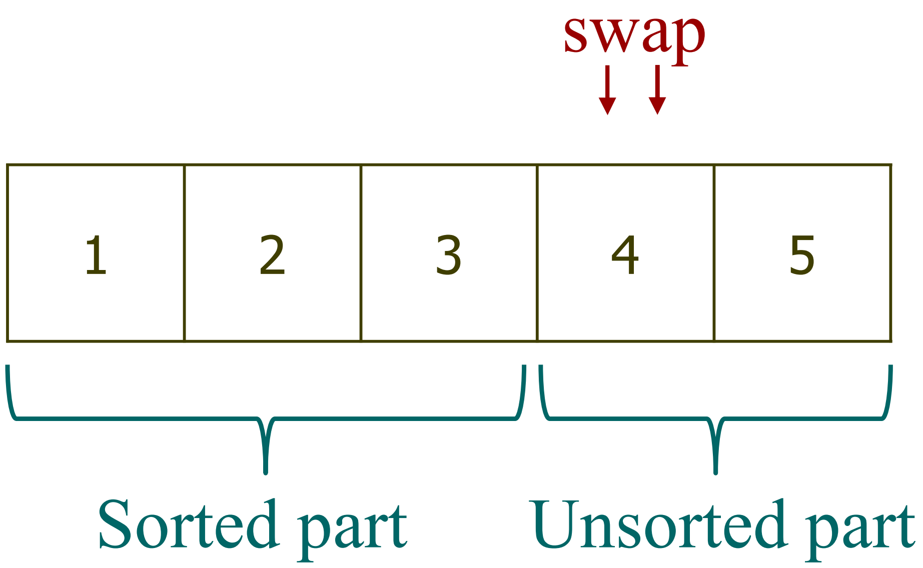
```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                  temp);
    }
}
```

i = 3, j = 4, min = 3



```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

i = 3, j = 4, min = 3

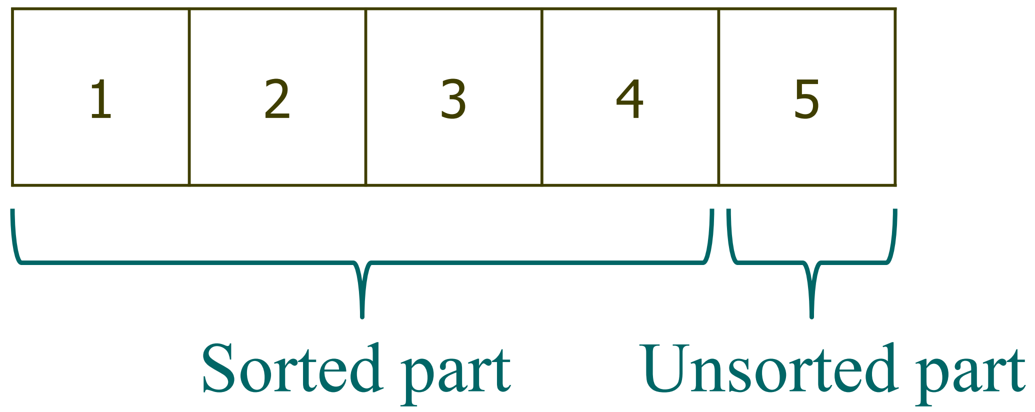


```
void sort (int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++)  {
        min = i;
        for (j = i+1; j < n; j++)
            if (list[j] < list[min])
                min = j;
        SWAP(list[i], list[min],
                temp);
    }
}
```

After i=3:

Sorted part: [1][2][3][4]

Unsorted part: [5]



Theorem 1.1:

Function *sort*(*list,n*) correctly sorts a set of n≥1 integers. The result remains in list[0], . . ., list[n-1] such that list[0]≤list[1]≤ . . . ≤list[n-1].

proof :

When the outer **for** loop completes its iteration for $i = q$, we have $list[q] \leq list[r], q < r < n$.

Further, on subsequent iterations, $i > q$ and $list[0]$ through $list[q]$ are unchanged.

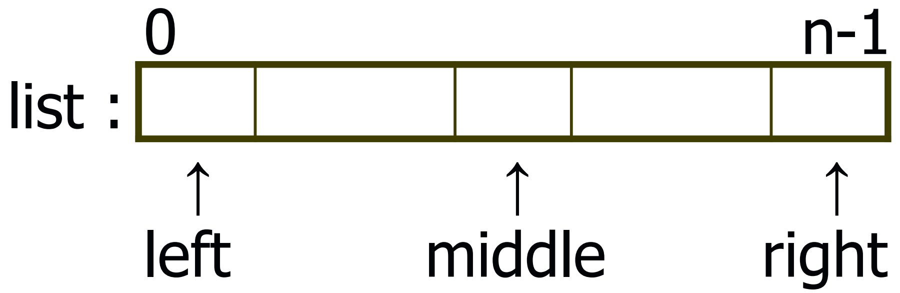
Hence following the last iteration of the outer **for** loop (i.e., $i = n - 2$), we have $list[0] \leq list[1] \leq \cdots \leq list[n-1]$. □

Example 1.2 [Binary Search]

Find out if an integer *searchnum* is in a list.

If so, return i such that list[i] = *searchnum,*
Otherwise, return -1.

For a sorted list (in nondecreasing order)



middle = (left + right) / 2

- **Compare list[*middle*] with *searchnum***
  - ***searchnum* < list[*middle*]**
    if *searchnum* is present, it must be in the positions between *left* and *middle-1.*
    set *right* to *middle-1.*
  - ***searchnum* = list[*middle*]**
    return *middle.*
  - ***searchnum* > list[*middle*]**
    if *searchnum* is present, it must be in the positions between *middle+1* and *right.*
    set *left* to *middle+1*

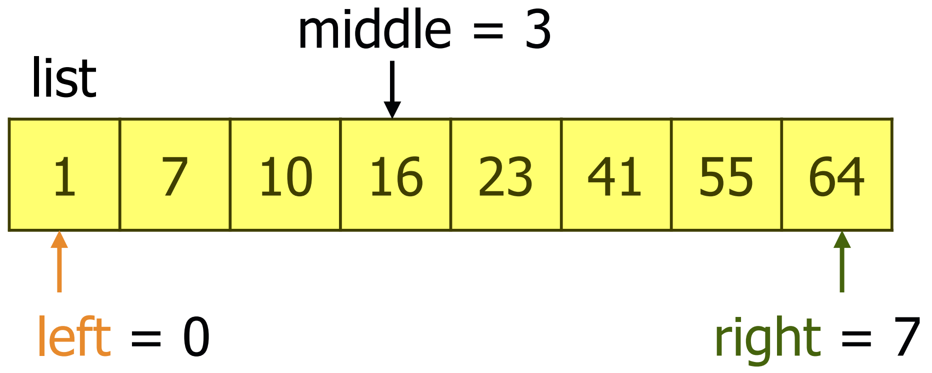
- **Implementing this search strategy :**

  [Program 1.5]

```
while (there are more integers to check) {
        middle = (left + right) / 2;
        if (searchnum < list[middle])
                right = middle - 1;
        else if (searchnum == list[middle])
                return middle;
        else  left = middle + 1;
}
```

```
while(there are more integers to
          check)
{
   middle = (left + right) / 2;
   if(searchnum < list[middle])
      right = middle - 1;
   else if(searchnum==list[middle])
      return middle;
   else  left = middle + 1;
}
```

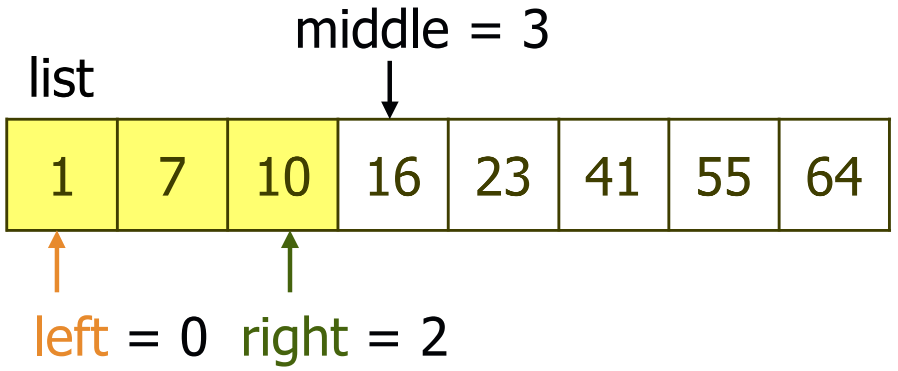
searchnum = 10



middle = (left+right)/2
= (0+7)/2
= 3

```
while(there are more integers to
          check)
{
   middle = (left + right) / 2;
   if(searchnum < list[middle])
      right = middle - 1;
   else if(searchnum==list[middle])
      return middle;
   else  left = middle + 1;
}
```

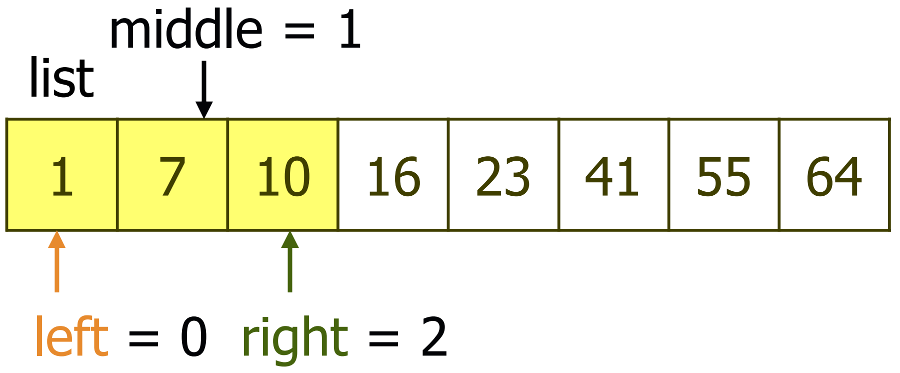
searchnum = 10



right = middle – 1
      = 3-1
      = 2

```
while(there are more integers to
          check)
{
   middle = (left + right) / 2;
   if(searchnum < list[middle])
      right = middle - 1;
   else if(searchnum==list[middle])
      return middle;
   else  left = middle + 1;
}
```

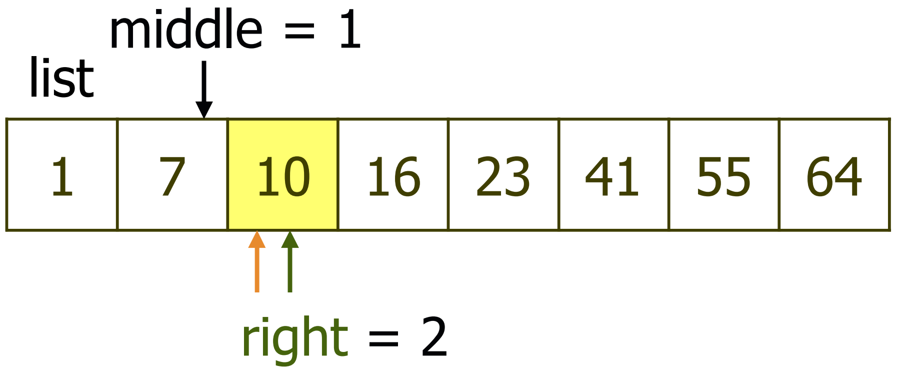
searchnum = 10



middle = (left+right)/2
= (0+2)/2
= 1

```
while(there are more integers to
        check)
{
    middle = (left + right) / 2;
    if(searchnum < list[middle])
        right = middle - 1;
    else if(searchnum==list[middle])
        return middle;
    else  left = middle + 1;
}
```

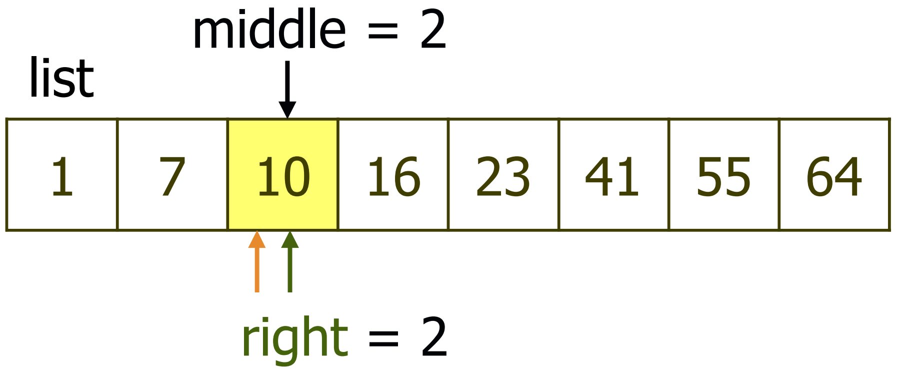
searchnum = 10



left = middle + 1
= 1+1
= 2

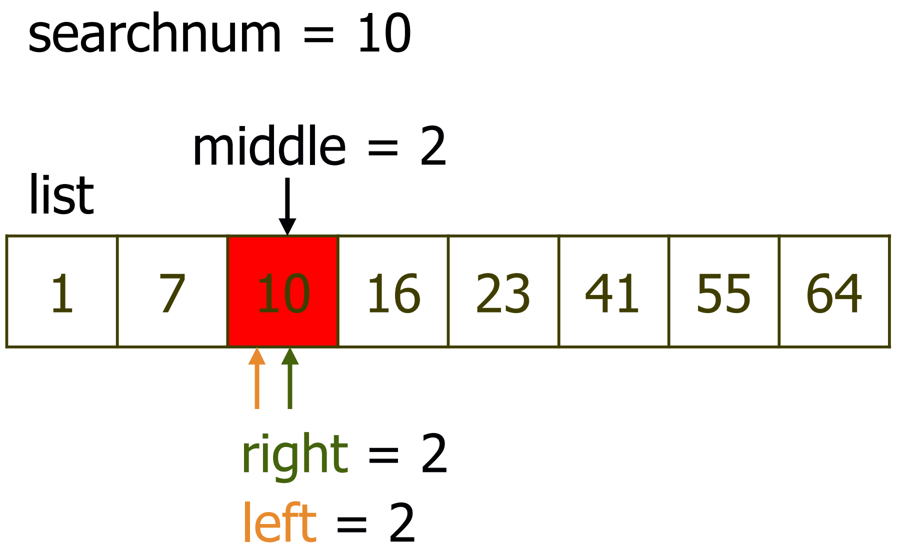
```
while(there are more integers to
            check)
{
    middle = (left + right) / 2;
    if(searchnum < list[middle])
        right = middle - 1;
    else if(searchnum==list[middle])
        return middle;
    else  left = middle + 1;
}
```

searchnum = 10



middle = (left+right)/2
= (2+2)/2
= 2

```
while(there are more integers to
        check)
{
    middle = (left + right) / 2;
    if(searchnum < list[middle])
        right = middle - 1;
    else if(searchnum==list[middle])
        return middle;
    else  left = middle + 1;
}
```



- **Handling the comparisons:**

<	**returns**	**-1**
=		**0**
>		**1**

- ***function -***

```
int compare (int x, int y)
{
        /* compare x and y, return -1 for less than,
        0 for equal, 1 for greater */
        if (x < y)  return -1;
        else  if  (x == y)  return 0;
        else  return 1;
}
```

- ***macro -***

```
# define COMPARE (x,y) ((x) < (y)) ? -1: ((x) == (y)) ? 0: 1)
```

[Program 1.7]

```
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
        Return its position if found.  Otherwise return -1 */
    int middle;
    while (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum))
        {
            case -1 : left = middle + 1;
                      break;
            case 0 : return middle;
            case 1 : right = middle - 1;
        }
    }
    return -1;
}
```

cf. Without COMPARE macro

```
while(left<=right) {
    middle = (left + right)/2;
    if(searchnum < list[middle])
        right = middle - 1;
    else if(searchnum==list[middle])
        return middle;
    else  left = middle + 1;
}
```

1.3.2 Recursive Algorithms

- **Direct recursion:** Functions that call themselves.
- **Indirect recursion:** Functions that invoke the calling function again.

- Recursion is a general control scheme.
- Often recursive function is easier to understand than its iterative counterpart.
- Many problems can be defined recursively in natural way.

- **[Binomial Coefficients]**

$$\begin{bmatrix} n \\ m \end{bmatrix} = \frac{n!}{m!(n-m)!}$$

can be recursively computed by the formula:

$$\begin{bmatrix} n \\ m \end{bmatrix} = \begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix}$$

- **Examples :**
  - **[factorial]**

$$n! = \begin{cases} n * (n-1)! & \text{if } n>1 \\ 1 & \text{if } n=1 \end{cases}$$

  - **[Binary search]**

$$\text{Bsrch}[key, left, right] = \begin{cases} \text{Bsrch}[key, left, middle-1] & \text{if } key < list[middle] \\ middle & \text{if } key = list[middle] \\ \text{Bsrch}[key, middle+1, right] & \text{if } key > list[middle] \end{cases}$$

- **[Fibonacci numbers]**

$$f_n = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ f_{n-1} + f_{n-2} & \text{if } n>1 \end{cases}$$

- **[Permutations]**

We can construct the set of permutations by printing:

(1) *a* followed by all permutations of (*b, c, d*)
(2) *b* followed by all permutations of (*a, c, d*)
(3) *c* followed by all permutations of (*a, b, d*)
(4) *d* followed by all permutations of (*a, b, c*)

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

Initial state

n	5
g	0
h	1
f	?
i	?

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
           f = g+h;
           g = h;
           h = f;
       }
   }
   else   f = n;
   return  f;
}
```

i=2

n	5
g	0
h	1 +
f	? → 1
i	2

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

i=2

n	5
g	0→1
h	1→1
f	1
i	2

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

i=3

n	5
g	1
h	1 +
f	1→2
i	3

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

i=3

n	5
g	1→1
h	1→2
f	2
i	3

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

i=4

n	5
g	1
h	2 +
f	2→3
i	4

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

i=4

n	5
g	$1 \rightarrow 2$
h	$2 \rightarrow 3$
f	3
i	4

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

i=5

n	5
g	2
h	3 +
f	$3 \rightarrow 5$
i	5

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
           f = g+h;
           g = h;
           h = f;
       }
   }
   else   f = n;
   return  f;
}
```

i=5

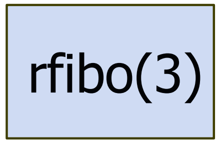
n	5
g	2→3
h	3→5
f	5
i	5

Sogang University

Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

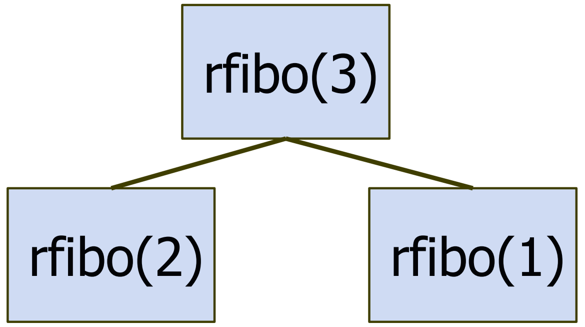
n=3



Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

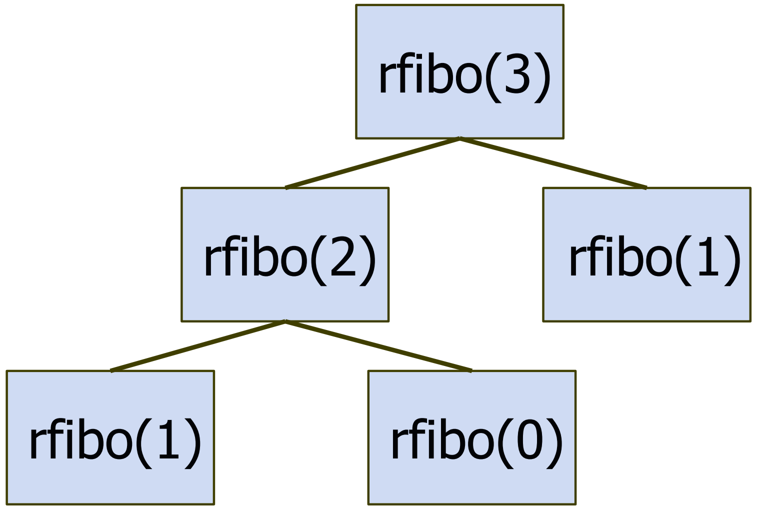
n=3



Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

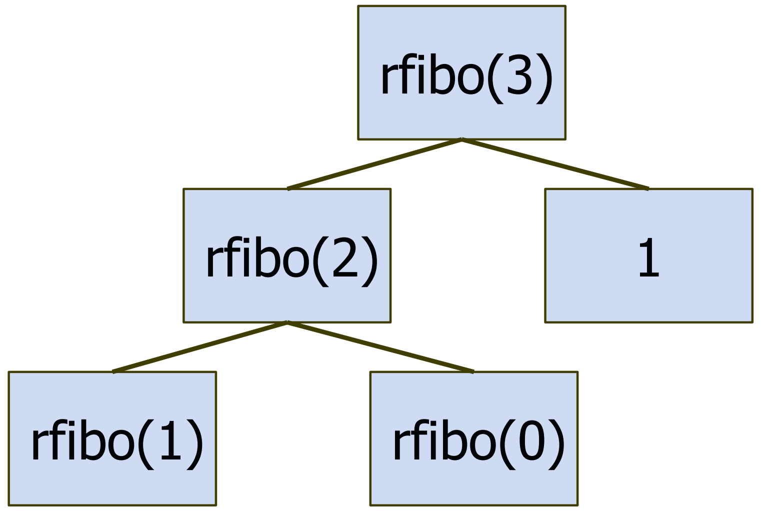
n=3



Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

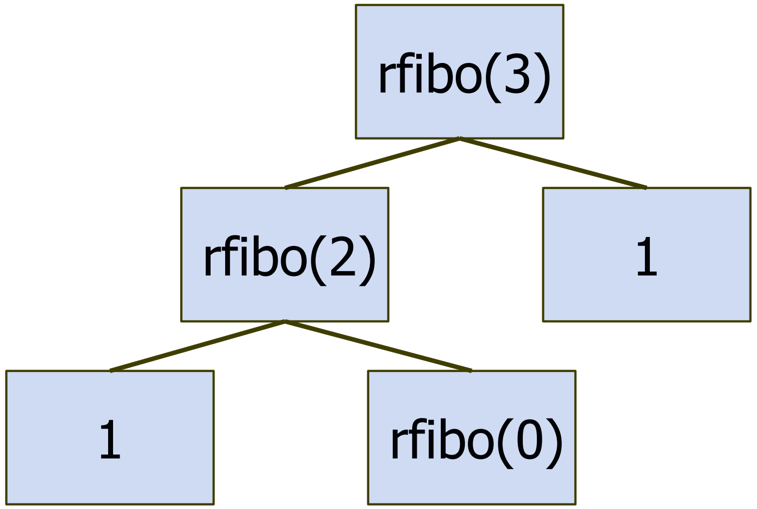
n=3



Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

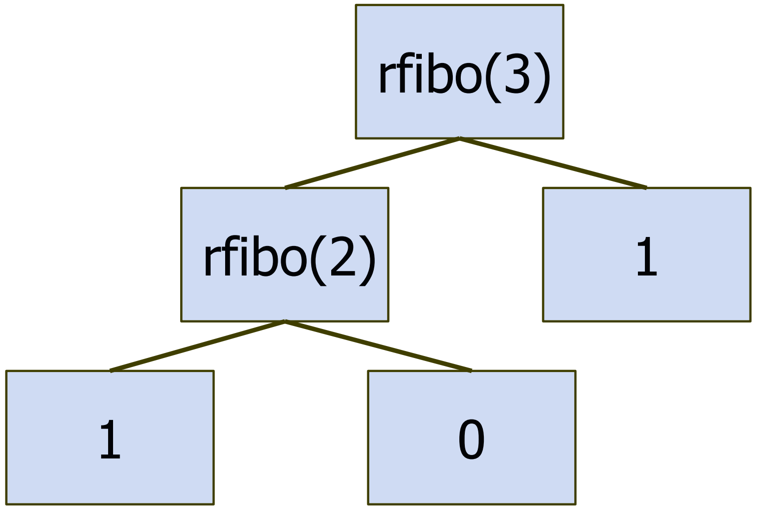
n=3



Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

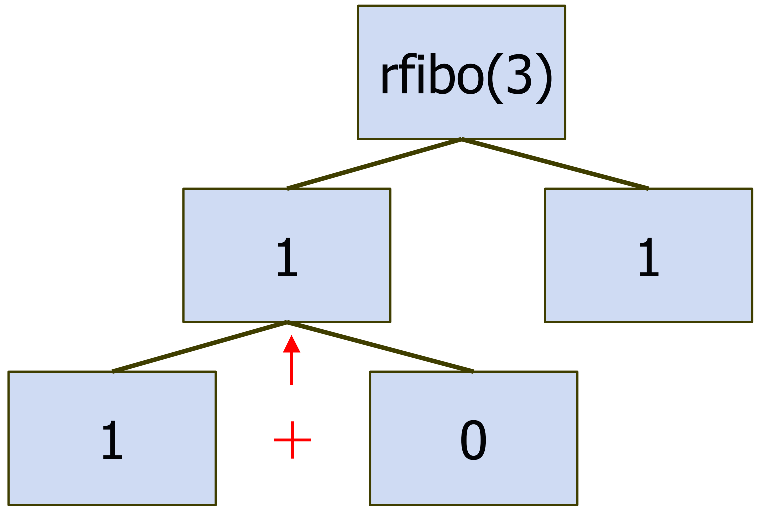
n=3



Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

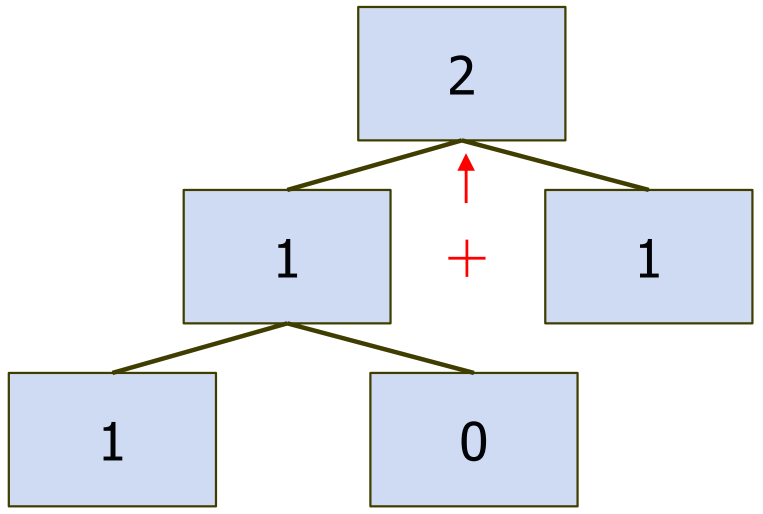
n=3



Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

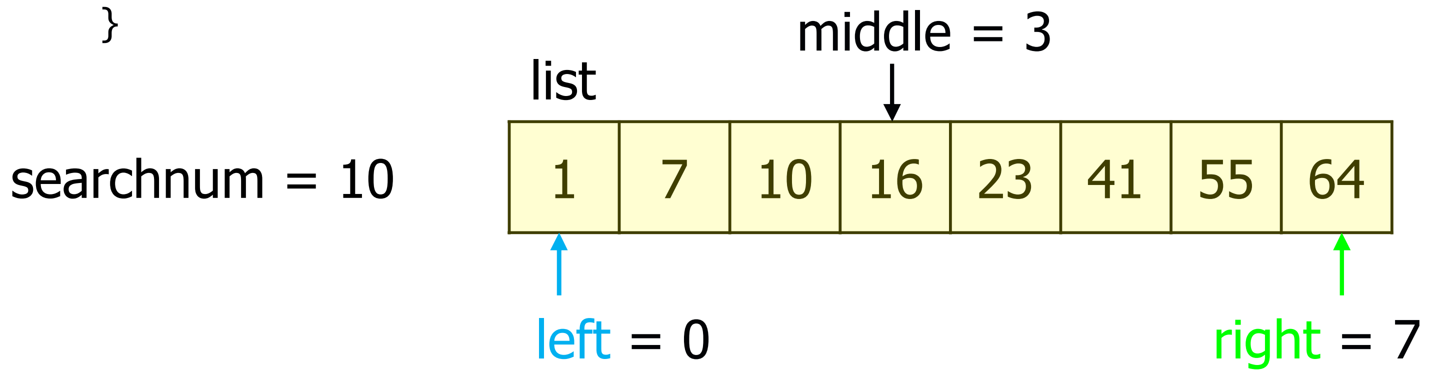
n=3



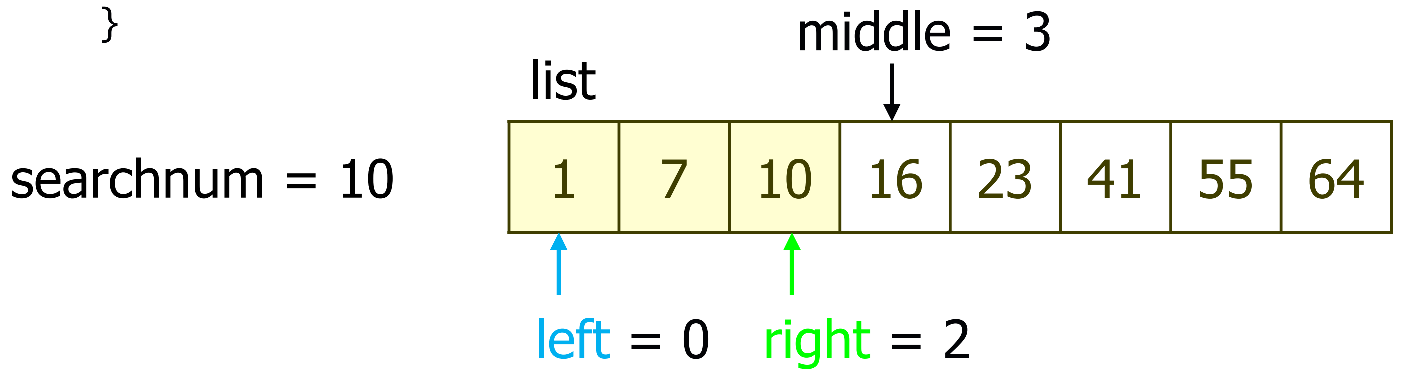
- **[Program 1.8]**

```
int binsearch(int list[], int searchnum, int left, int right)
{
 /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
     Return its position if found.  Otherwise return -1 */

 int middle;
 if (left <= right)  {
    middle = (left + right)/2;
    switch (COMPARE(list[middle], searchnum)) {
         case -1 : return  binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return  binsearch(list, searchnum, left, middle - 1);
    }
  }
  return -1;
}
```

```
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
        Return its position if found.  Otherwise return -1 */

    int middle;
    if (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1 : return  binsearch(list, searchnum, middle + 1, right);
            case 0 : return middle;
            case 1 : return  binsearch(list, searchnum, left, middle - 1);
        }
    }
    return -1;
}
```



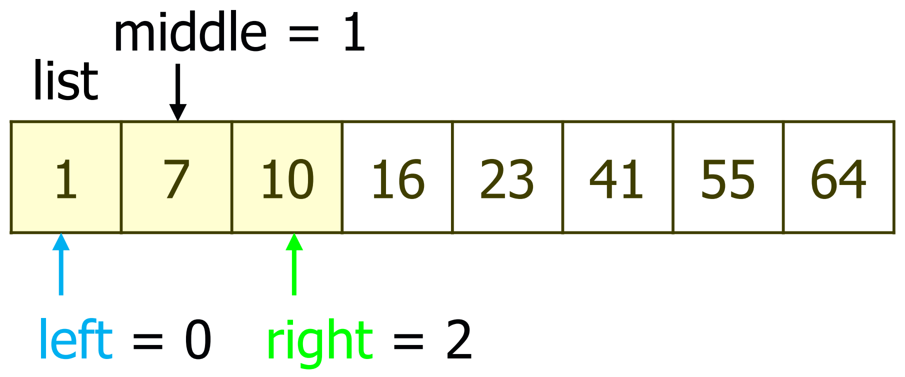
```
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
        Return its position if found.  Otherwise return -1 */

    int middle;
    if (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1 : return  binsearch(list, searchnum, middle + 1, right);
            case 0 : return middle;
            case 1 : return  binsearch(list, searchnum, left, middle - 1);
        }
    }
    return -1;
}
```



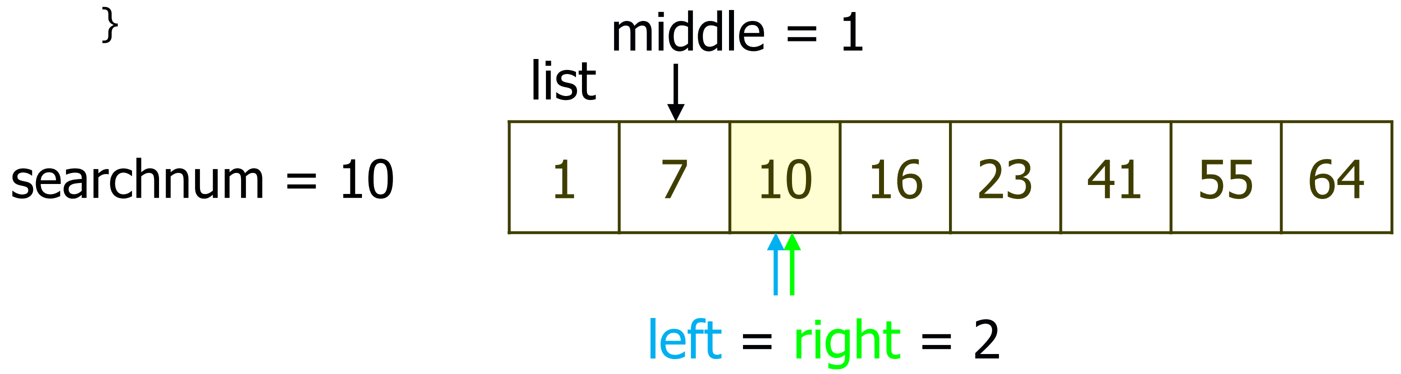
Sogang University

```
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
        Return its position if found.  Otherwise return -1 */

    int middle;
    if (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1 : return  binsearch(list, searchnum, middle + 1, right);
            case 0 : return middle;
            case 1 : return  binsearch(list, searchnum, left, middle - 1);
        }
    }
    return -1;
}
```

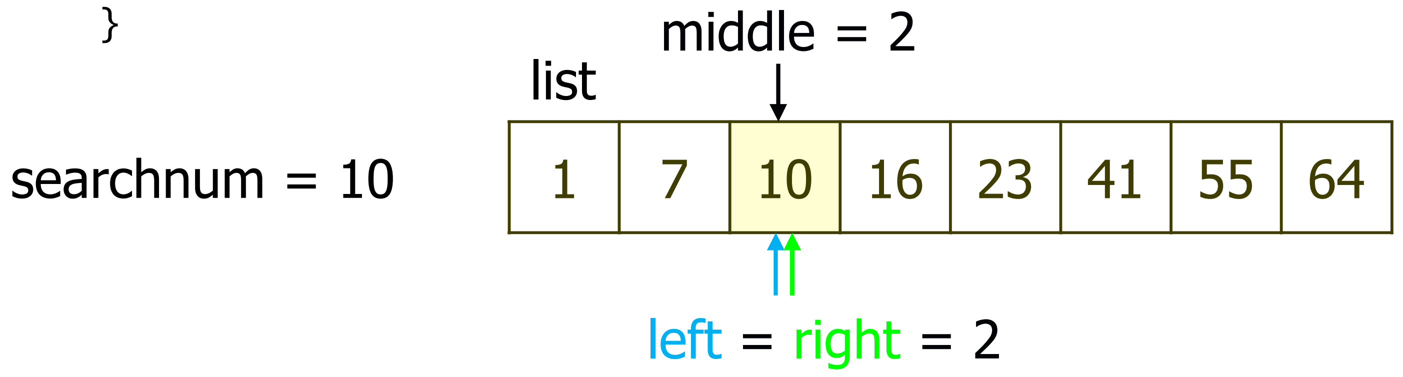
searchnum = 10



```
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
        Return its position if found.  Otherwise return -1 */

    int middle;
    if (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1 : return  binsearch(list, searchnum, middle + 1, right);
            case 0 : return middle;
            case 1 : return  binsearch(list, searchnum, left, middle - 1);
        }
    }
    return -1;
}
```

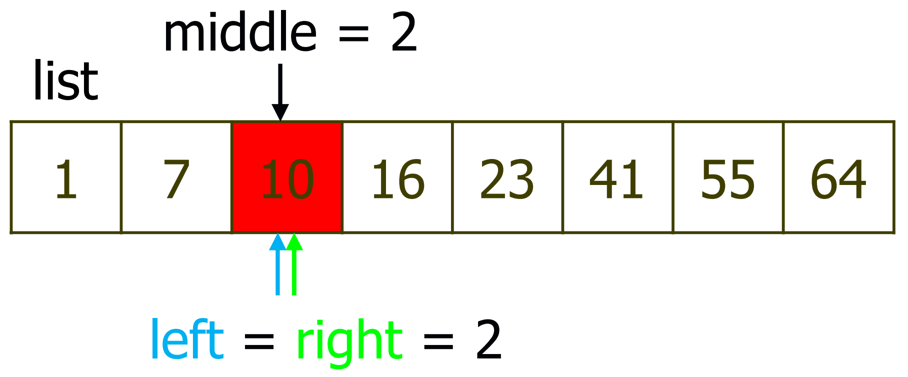


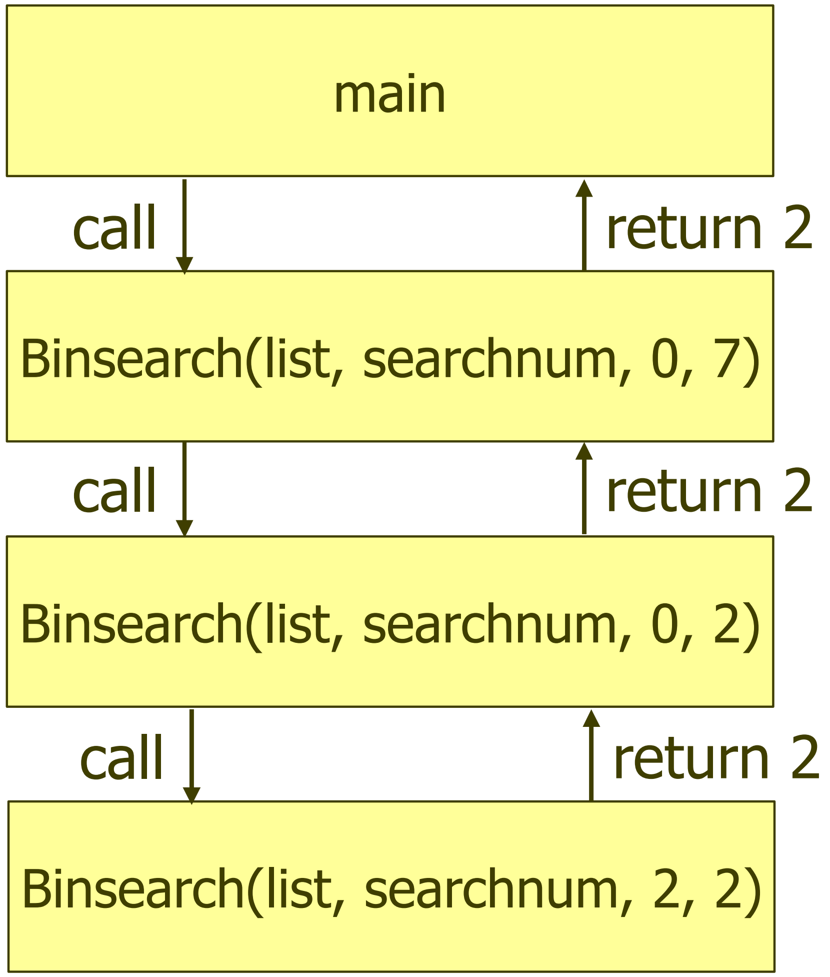
```
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
        Return its position if found.  Otherwise return -1 */

    int middle;
    if (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1 : return  binsearch(list, searchnum, middle + 1, right);
            case 0 : return middle;
            case 1 : return  binsearch(list, searchnum, left, middle - 1);
        }
    }
    return -1;
}
```



```
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
        Return its position if found.  Otherwise return -1 */

    int middle;
    if (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1 : return  binsearch(list, searchnum, middle + 1, right);
            case 0 : return middle;
            case 1 : return  binsearch(list, searchnum, left, middle - 1);
        }
    }
    return -1;
}
```

searchnum = 10




main
call
return 2
Binsearch(list, searchnum, 0, 7)
call
return 2
Binsearch(list, searchnum, 0, 2)
call
return 2
Binsearch(list, searchnum, 2, 2)

- **[Program 1.9]**

```
void perm(char *list, int i, int n)
{
   /* generate all the permutations of list[i] to list[n] */
   int j, temp;
   if (i == n)  {
       for (j=0; j<=n; j++) printf("%c", list[j]);
       printf("     ");
   }
   else  {
   /* list[i] to list[n] has more than one permutation,
       generate these recursively  */
         for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
         }
   }
}
```

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

perm([a][b][c], 0, 2) // initial call

list:

a	b	c

perm([a][b][c], 0, 2)

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);     // j=0, i=0
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

list:

a	b	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

perm([a][b][c], 0, 2)

$\xrightarrow[j=0,\ i=0]{}$ perm([a][b][c], 1, 2)

list:

a	b	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);     // j=0, i=0
        }
    }
}
```

perm([a][b][c], 0, 2)

$\xrightarrow[j=0,\ i=0]{}$ perm([a][b][c], 1, 2)

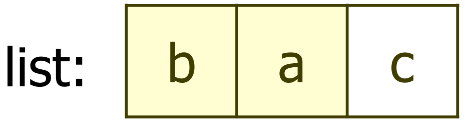
list:

a	b	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);     // j=1, i=0
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

perm([a][b][c], 0, 2)

$\xrightarrow[j=0,\ i=0]{}$ perm([a][b][c], 1, 2)



```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

perm([a][b][c], 0, 2)

$\xrightarrow[\text{j=0, i=0}]{}$ perm([a][b][c], 1, 2)

$\xrightarrow[\text{j=1, i=0}]{}$ perm([b][a][c], 1, 2)

list:

b	a	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);     // j=1, i=0
        }
    }
}
```

$$\begin{tikzcd}
\text{perm([a][b][c], 0, 2)} \\
{} \arrow[r, "\text{j=0, i=0}"'] & \text{perm([a][b][c], 1, 2)} \\
{} \arrow[r, "\text{j=1, i=0}"'] & \text{perm([b][a][c], 1, 2)}
\end{tikzcd}$$

list:

a	b	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);     // j=2, i=0
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

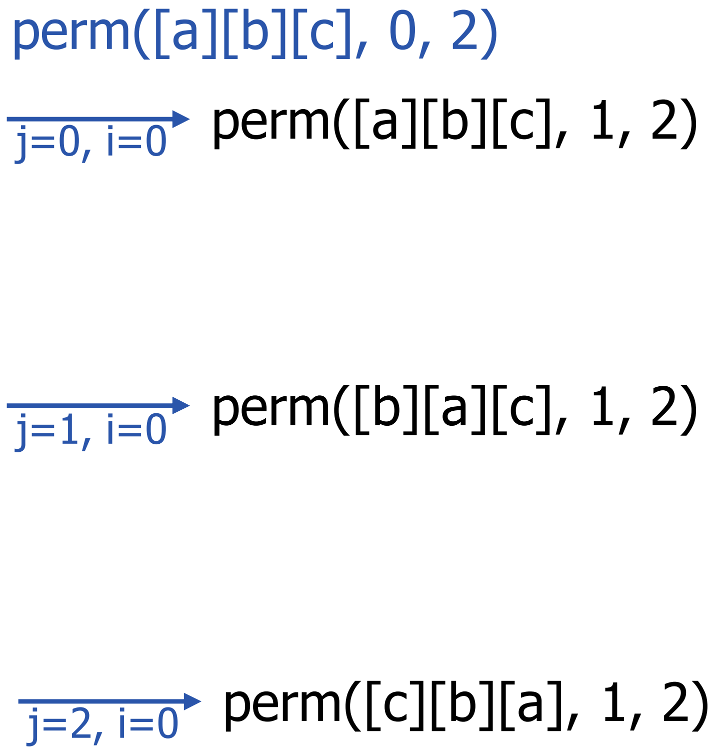
perm([a][b][c], 0, 2)

$\xrightarrow[j=0,\ i=0]{}$ perm([a][b][c], 1, 2)

$\xrightarrow[j=1,\ i=0]{}$ perm([b][a][c], 1, 2)

list:	c	b	a

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```



list:

c	b	a

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively   */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);     // j=2, i=0
        }
    }
}
```

perm([a][b][c], 0, 2)

$\xrightarrow[j=0,\ i=0]{}$ perm([a][b][c], 1, 2)

$\xrightarrow[j=1,\ i=0]{}$ perm([b][a][c], 1, 2)

$\xrightarrow[j=2,\ i=0]{}$ perm([c][b][a], 1, 2)

list:

a	b	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);     // j=1, i=1
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

list:

a	b	c

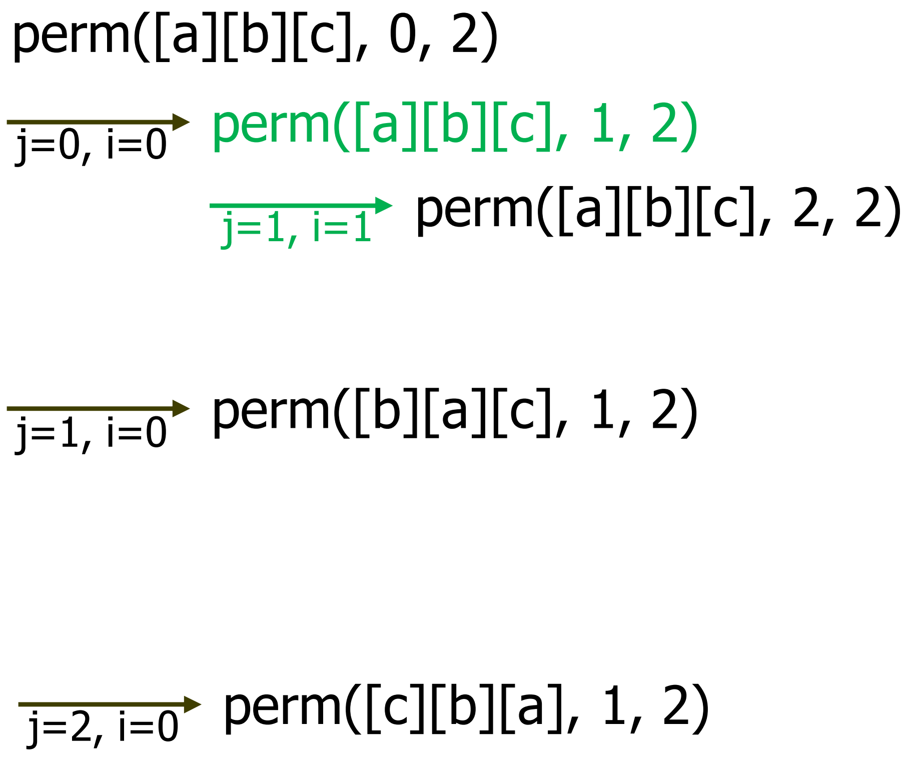
perm([a][b][c], 0, 2)

$\xrightarrow[j=0,\ i=0]{}$ perm([a][b][c], 1, 2)

$\xrightarrow[j=1,\ i=0]{}$ perm([b][a][c], 1, 2)

$\xrightarrow[j=2,\ i=0]{}$ perm([c][b][a], 1, 2)

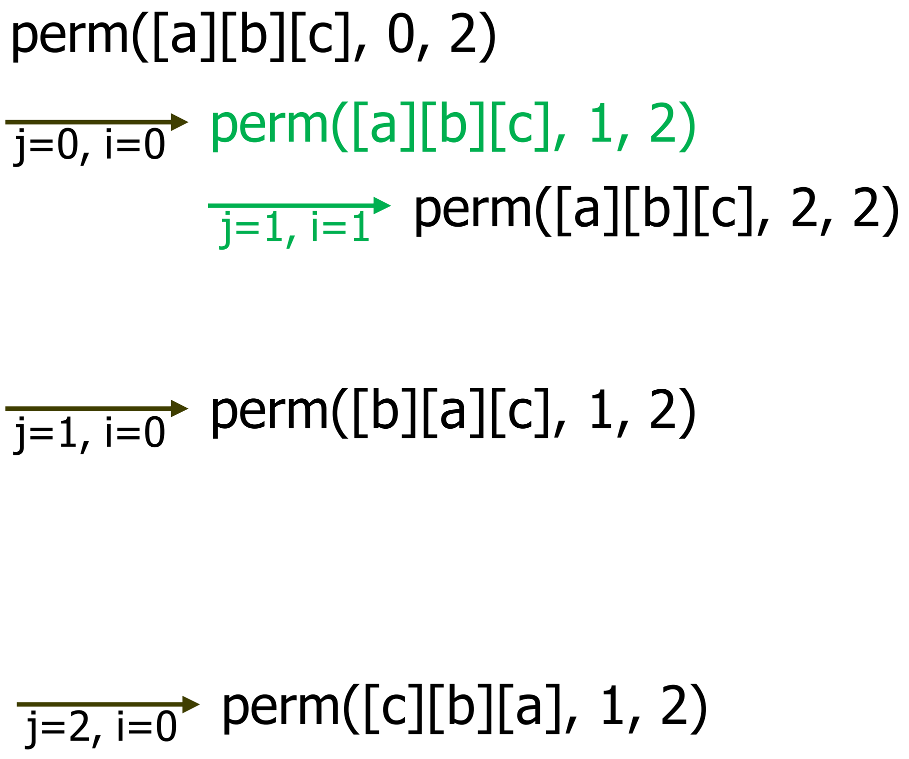
```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```



list:

a	b	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);     // j=1, i=1
        }
    }
}
```



list:

a	b	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);     // j=2, i=1
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

perm([a][b][c], 0, 2)

$\xrightarrow[j=0,\ i=0]{}$ perm([a][b][c], 1, 2)

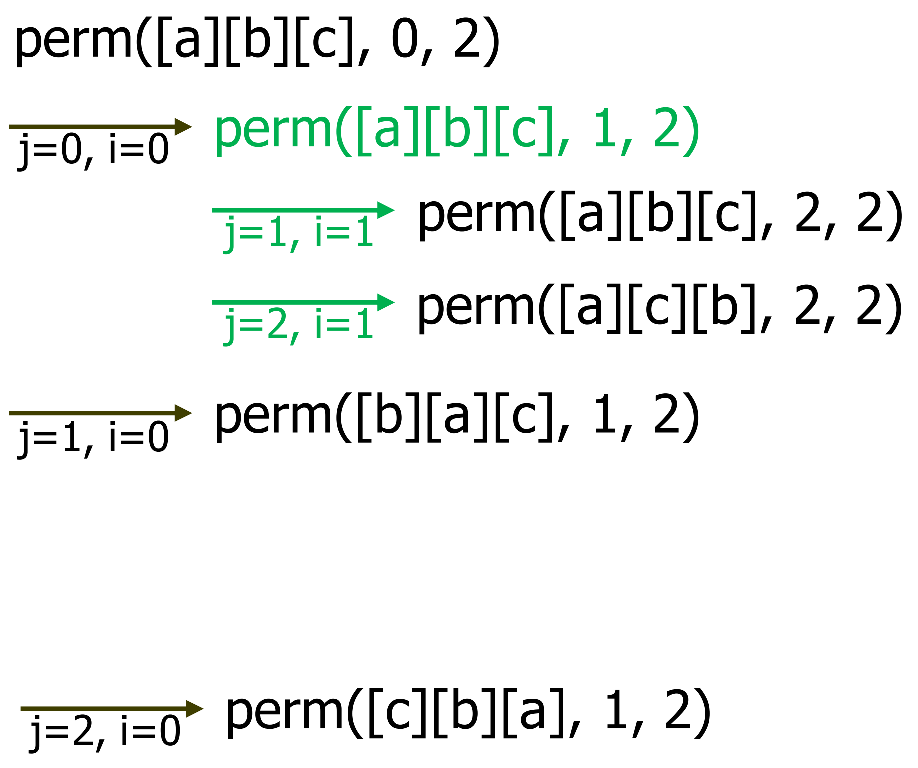
$\qquad \xrightarrow[j=1,\ i=1]{}$ perm([a][b][c], 2, 2)

$\xrightarrow[j=1,\ i=0]{}$ perm([b][a][c], 1, 2)

$\xrightarrow[j=2,\ i=0]{}$ perm([c][b][a], 1, 2)

list:			
	a	c	b

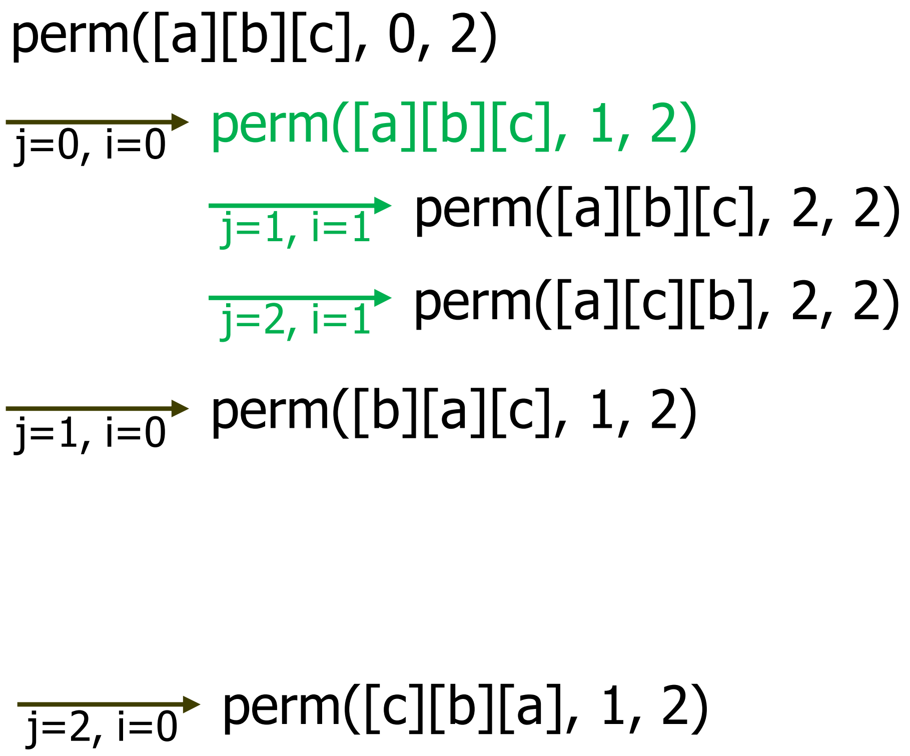
```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```



list:

a	c	b

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);     // j=2, i=1
        }
    }
}
```



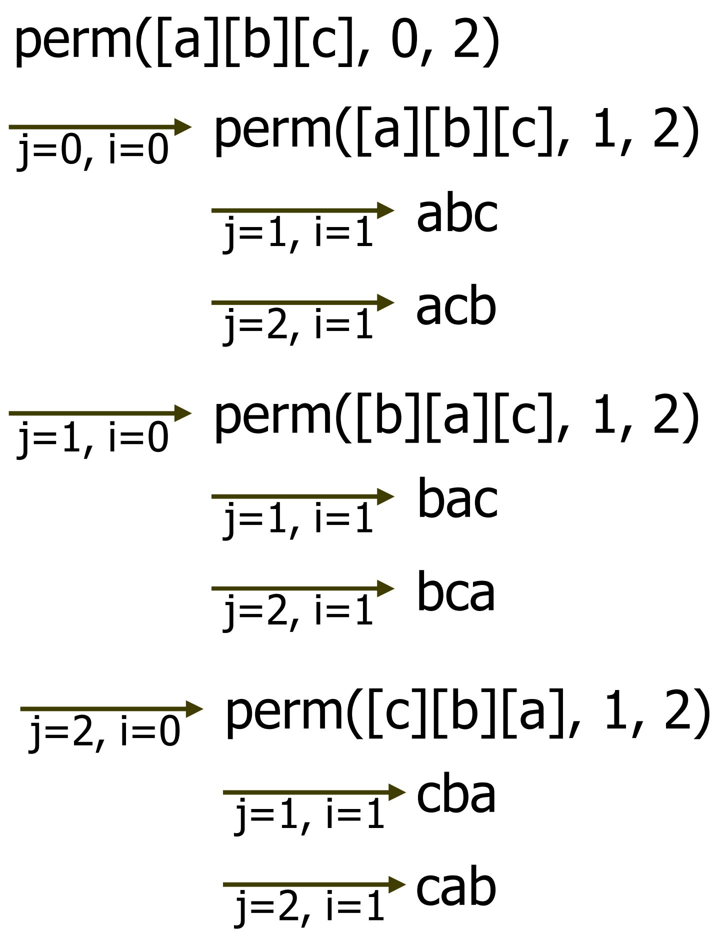
list:

a	b	c

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```

```
perm([a][b][c], 0, 2)
  --j=0, i=0--> perm([a][b][c], 1, 2)
                  --j=1, i=1--> perm([a][b][c], 2, 2)
                  --j=2, i=1--> perm([a][c][b], 2, 2)
  --j=1, i=0--> perm([b][a][c], 1, 2)
                  --j=1, i=1--> perm([b][a][c], 2, 2)
                  --j=2, i=1--> perm([b][c][a], 2, 2)
  --j=2, i=0--> perm([c][b][a], 1, 2)
                  --j=1, i=1--> perm([c][b][a], 2, 2)
                  --j=2, i=1--> perm([c][a][b], 2, 2)
```

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n)  {
        for (j=0; j<=n; j++) printf("%c", list[j]);
        printf("     ");
    }
    else  {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively  */
        for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
```



1.4 DATA ABSTRACTION

- **basic data types of C :**
  char, int, float, double, . . .
  short, long, unsigned

- **mechanisms for grouping data together :**
  Arrays and Structs

```
int  list[5];

struct student {
        char last_name[10];
        int  student_id;
        char grade;
};
```

- **pointer data type :**

  for every basic data type
  there is a corresponding pointer data type, such as
  pointer-to-an-int,
  pointer-to-a-real,
  pointer-to-a-char,
  and pointer-to-a-float.

  ```
  int   i,  *pi;
  ```

  predefined data types / user-defined data types

"What is a data type?"

- **Definition :**
  A *data type* is a collection of *objects* and a set of *operations* that act on those objects.

▶ **specification of objects**
e.g., type *int*,
{0, +1, -1, +2, -2, . . ., INT_MAX, INT_MIN}
specification of operations
e.g., type *int*,
+, −, *, /, and %.

▶ Algorithms should NOT make use of the representation of objects, but the functions (operations) that are provided.

- **Definition :**

  An *abstract data type* (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations.

- ***an abstract data type is implementation independent.***

  Specification of operations consists of the names of operations, the type of its arguments, and the type of its result. Also a description what the function does without appealing to internal representation details.

  e.g., *package* in Ada, *class* in C++

- **Categories to classify the operations of a data types:**
  - Constructor/creator: These functions create a new instance of the designated type.
  - Transformers: These functions create an instance of the designated type, generally by using one or more other instances.
  - Observers/reporters: These functions provide information about an instance of the type.

Example 1.5 [Abstract data type *Natural_Number*]

```
ADT Natural_Number is
        object: an ordered subrange of the integers starting at zero and ending at the
        maximum integer (INT_MAX) on the computer
        functions:
          for all x, y ∈ Natural_Number, TRUE, FALSE ∈ Boolean
          and where +, -, <, and == are usual integer operations

constructor →  Natural_Number Zero()           ::= 0
               Boolean Is_Zero(x)              ::= if (x) return FALSE
observer    {                                      else return TRUE
               Boolean Equal(x, y)             ::= if  (x==y) return TRUE
                                                   else return FALSE
               Natural_Number Successor(x)     ::= if (x == INT_MAX) return x
                                                   else return x+1
transformer {  Natural_Number Add(x, y)        ::= if ((x+y)<= INT_MAX) return  x+y
                                                   else return INT_MAX
               Natural_Number Subtract(x,y)    ::= if (x<y) return 0
                                                   else return x-y
end  Natural_Number
```

1.5 PERFORMANCE ANALYSIS

- ***Criteria of judging a program:***
  1. Does the program <u>meet the original specifications</u> of the task?
  2. Does it <u>work correctly</u>?
  3. Is the program <u>well documented</u>?
  4. Does the program <u>effectively use functions</u> to create logical units?
  5. Is the program's code <u>readable</u>?

  [Performance Evaluation]

  6. Does the program <u>efficiently use primary and secondary storage</u>?
  7. Is the program's <u>running time acceptable</u> for the task?

- ***Performance Analysis*** **:**
  estimates of time and space that are <u>machine independent</u>.

- ***Performance Measurement*** **:**
  obtaining <u>machine-dependent</u> running times.
  used to identify inefficient code segments.

- **<u>Definition</u> :**
  The *space complexity* of a program is the amount of memory that it needs to run to completion.
  The *time complexity* of a program is the amount of computer time that it needs to run to completion.

1.5.1 Space Complexity

- ***Fixed space requirements*** :
  independent from the number and size of the program's inputs and outputs, e.g., the instruction space, space for simple variables, fixed-size structured variables, and constants.

- ***Variable space requirements*** :
  space needed by structured variables whose size depends on the particular instance, *I*, of the problem being solved.

  The variable space requirement of a program *P* working on an instance I is denoted $S_P(I)$.
  We can express the total space requirement $S(P)$ of any program as:

$$S(P) = c + S_P(I)$$

where c is a constant representing the fixed space requirements.

Example 1.6 : [simple arithmetic function]

$S_{abc}(I) = 0.$

[Program 1.10]

```
float abc (float a, float b, float c)
{
    return a+b+b*c + (a+b-c)/(a+b) + 4.00;
}
```

- **Example 1.7 : [*adding a list of numbers iteratively*]**

[Program 1.11]

```
float sum(float list[], int n)
{
        float  tempsum = 0;
        int i;
        for (i=0; i<n; i++)
            tempsum += list[i];
        return tempsum;
}
```

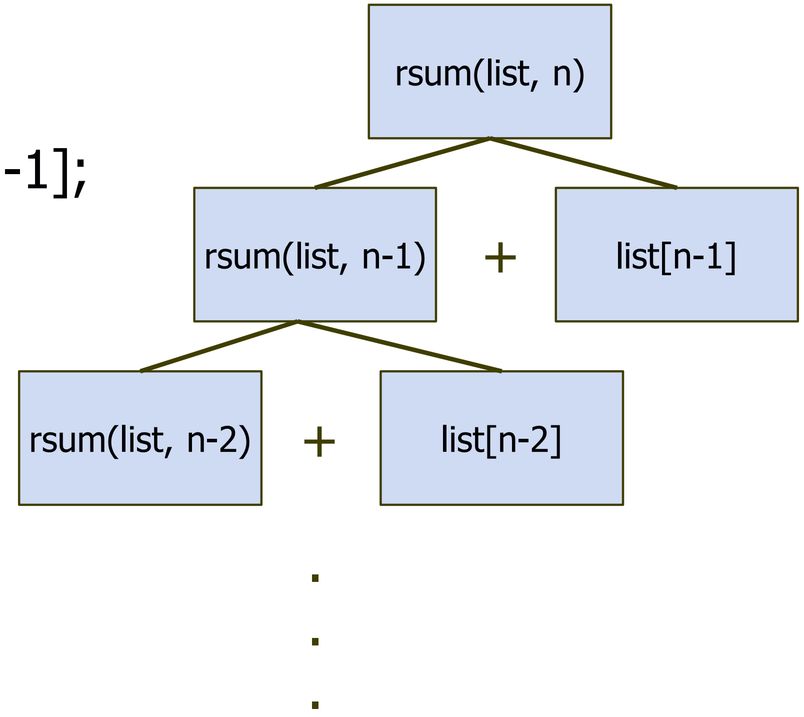
$S_{sum}(n) = n$ if parameters are passed by value.
$S_{sum}(n) = 0$ if parameters are passed by reference.

- **Example 1.8 : [*adding a list of numbers recursively*]**

[Program 1.12]

```
float rsum(float list[], int n)
{
        if (n) return rsum(list, n-1) + list[n-1];
        return 0;
}
```

$$S_{rsum}(n) = 12 * n$$



Type	Name	Number of bytes
parameter: array pointer parameter: integer return address: (used internally)	list[] n	4 4 4
TOTAL per recursive call		12

Figure 1.1 : Space needed for one recursive call of program 1.12

1.5.2 Time Complexity

- **The time, T_p, taken by a program, P, is the sum of:**

  (1) ***Compile Time***

  (2) ***Execution (Running) Time***

We are really concerned only with the program's execution time, T_p

- ***Determining the execution time*** **requires a knowledge of:**
  - the times needed to perform each operation.
  - the number of each operation performed for the given instance (dependent on the compiler).

$$T_P(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$$

- n: the instance characteristic
- c_a, c_s, c_l, c_{st} : the time needed to perform each operations.
- ADD, SUB, LDA, STA : the number of additions, subtractions, loads and stores

- Obtaining such a detailed estimate of running time is rarely worth the effort.

- Counting the number of operations the program performs gives us a machine-independent estimate.

- <u>**Definition**</u> **:**

  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

  Determining the number of steps that a program or a function needs to solve a particular problem instance by creating a global variable, *count*, and inserting statements that increment count.

- **[Example 1.9] [*Iterative summing of a list of numbers*]**

[Program 1.13]

```
float sum(float list[], int n)
{
     float  tempsum = 0;  count++;    /*for assignment*/
     int i;
     for (i=0; i<n; i++)  {
         count++;   /*for the for loop */
         tempsum += list[i];  count++;  /*for assignment*/
     }
     count++;  /* last execution of for */
     count++;  /* for return */  return tempsum;
}
```

```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;    /*for assignment*/
    int i;
    for (i=0; i<n; i++)  {
        count++;   /*for the for loop */
        tempsum += list[i];  count++; /*for assignment*/
    }
    count++;  /* last execution of for */
    count++;  /* for return */    return tempsum;
}
```

tempsum=0	count++

```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;    /*for assignment*/
    int i;
    for (i=0; i<n; i++)  {
        count++;   /*for the for loop */
        tempsum += list[i];  count++; /*for assignment*/
    }
    count++;  /* last execution of for */
    count++;  /* for return */    return tempsum;
}
```

tempsum=0	count++
for(i=0;i<n;i++)	count++

```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;    /*for assignment*/
    int i;
    for (i=0; i<n; i++)  {
        count++;   /*for the for loop */
        tempsum += list[i];  count++; /*for assignment*/
    }
    count++;  /* last execution of for */
    count++;  /* for return */    return tempsum;
}
```

tempsum=0	count++
for(i=0;i<n;i++)	count++
tempsum+=list[0]	count++

```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;    /*for assignment*/
    int i;
    for (i=0; i<n; i++)  {
        count++;   /*for the for loop */
        tempsum += list[i];  count++; /*for assignment*/
    }
    count++;  /* last execution of for */
    count++;  /* for return */    return tempsum;
}
```

tempsum=0	count++
for(i=0;i<n;i++)	count++
tempsum+=list[0]	count++
…	…
for(i=0;i<n;i++)	count++

```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;    /*for assignment*/
    int i;
    for (i=0; i<n; i++)  {
        count++;   /*for the for loop */
        tempsum += list[i];  count++; /*for assignment*/
    }
    count++;  /* last execution of for */
    count++;  /* for return */    return tempsum;
}
```

tempsum=0	count++	
for(i=0;i<n;i++)	count++	
tempsum+=list[0]	count++	
…		n
for(i=0;i<n;i++)	count++	
tempsum+=list[n-1]	count++	

```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;    /*for assignment*/
    int i;
    for (i=0; i<n; i++)  {
        count++;   /*for the for loop */
        tempsum += list[i];  count++; /*for assignment*/
    }
    count++;  /* last execution of for */
    count++;  /* for return */    return tempsum;
}
```

tempsum=0	count++	
for(i=0;i<n;i++)	count++	
tempsum+=list[0]	count++	
…		n
for(i=0;i<n;i++)	count++	
tempsum+=list[n-1]	count++	
for(i=0;i<n;i++)	count++	

i becomes n+1,
then breaks the loop

```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;    /*for assignment*/
    int i;
    for (i=0; i<n; i++)  {
        count++;   /*for the for loop */
        tempsum += list[i];  count++; /*for assignment*/
    }
    count++;  /* last execution of for */
    count++;  /* for return */    return tempsum;
}
```

tempsum=0	count++	
for(i=0;i<n;i++)	count++	
tempsum+=list[0]	count++	
…		n
for(i=0;i<n;i++)	count++	
tempsum+=list[n-1]	count++	
for(i=0;i<n;i++)	count++	
return tempsum	count++	

[Program 1.14 Simplified version of Program 1.13]

```
float sum(float list[], int n)
{
    float  tempsum = 0;
    int i;
    for (i=0; i<n; i++)
        /*for the for loop and assignment*/
        count += 2;
    count += 3;
    return tempsum;
}
```

cf.

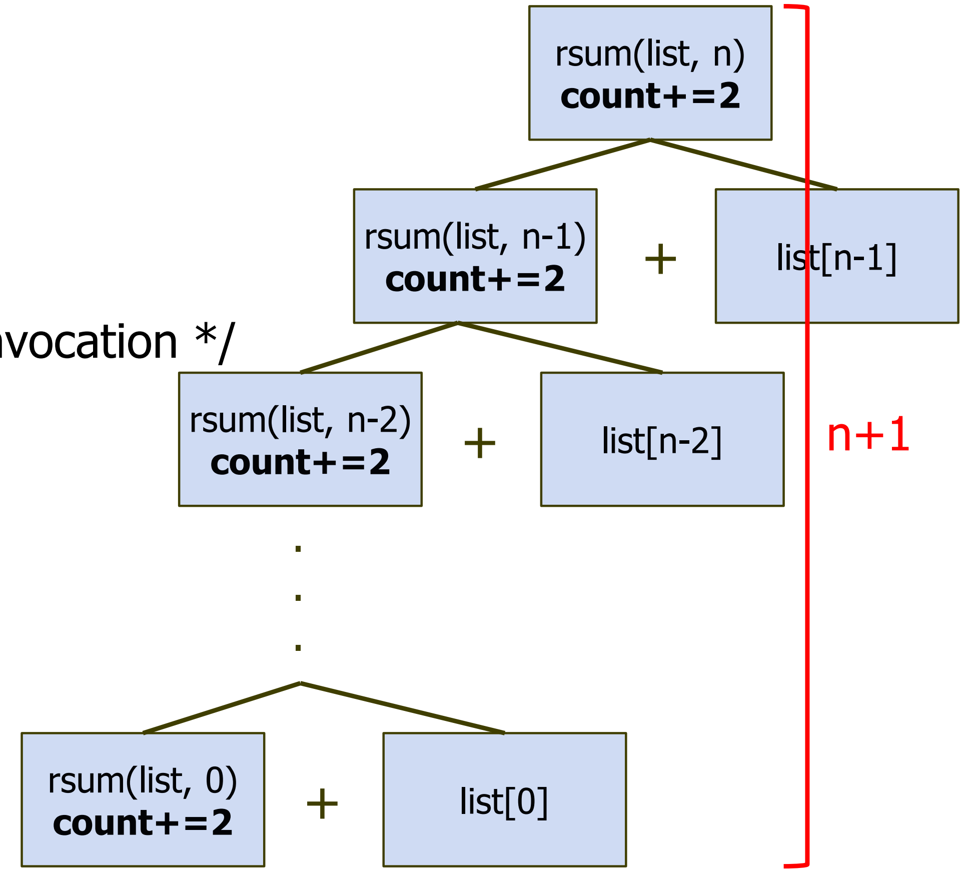
```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;
    int i;
    for (i=0; i<n; i++)  {
        count++;  /*for the for loop */
        tempsum += list[i];  count++;
    }
    count++;
    count++;
    return tempsum;
}
```

If the initial value of count is 0, its final value will be **2*n*+3**.

■ [Example 1.10] [*Recursive summing of a list of numbers*]

[Program 1.15]

```
float rsum(float list[], int n)
{
    count++;  /* for if conditional */
    if (n) {
        count++;  /* for return and rsum invocation */
        return rsum(list, n-1) + list[n-1];
    }
    count++;
    return list[0];
}
```



the step count is **2*n*+2**.

■ [Example 1.11] : [*Matrix addition*]

[Program 1.16]

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
              int c[][MAX_SIZE], int rows, int cols)
{
      int i, j;
      for (i=0; i<rows; i++)
         for (j=0; j<cols; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

[Program 1.17]

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
            int c[][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i=0; i<rows; i++)   {
        count++;    /* for i for loop */
        for (j=0; j<cols; j++)  {
            count++;     /* for j for loop */
            c[i][j] = a[i][j] + b[i][j];
            count++;     /* for assignment statement */
        }
        count++;  /* last time of j for loop */
    }
    count++;  /* last time of i for loop */
}
```

[Program 1.18 Simplification of Program 1.17]

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
            int c[][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i=0; i<rows; i++)   {
        for (j=0; j<cols; j++)
            count += 2;
        count += 2;
    }
    count++;
}
```

cf.

```
void add(…)
{
    int i, j;
    for (i=0; i<rows; i++) {
        count++;
        for (j=0; j<cols; j++)  {
            count++;
            c[i][j] = a[i][j] + b[i][j];
            count++;
        }
        count++;
    }
    count++;
}
```

The step count will be 2*rows*cols* + 2*rows* + 1

- **Tabular method: *steps/execution***

[Figure 1.2] Step count table for Program 1.11

Statement	s/e	Frequency	Total steps
float sum(float list[], int n)	0	0	0
{	0	0	0
float tempsum=0;	1	1	1
int i;	0	0	0
for (i=0; i<n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

- **[Example 1.13]**

[Figure 1.3] Step count table for Program 1.12

Statement	s/e	Frequency	Total steps
`float rsum(float list[], int n)`	0	0	0
`{`	0	0	0
`    if (n)`	1	n+1	n+1
`        return rsum(list, n-1)+list[n-1];`	1	n	n
`    return 0;`	1	1	1
`}`	0	0	0
Total			2n+2

[Example 1.14]

[Figure 1.4] Step count table for Program 1.16

Statement	s/e	Frequency	Total Steps
void add(int a[][MAX_SIZE], ...)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i=0; i<rows; i++)	1	rows + 1	rows + 1
for (j=0; j<cols; j++)	1	rows·(cols+1)	rows·cols + rows
c[i][j]=a[i][j]+b[i][j];	1	rows·cols	rows·cols
}	0	0	0
Total			2rows·cols + 2rows + 1

Summary

- Time complexity of a program is given by the number of steps taken by the program to compute the function it was written for.
- The number of steps is itself a function of the instance characteristics.

  e.g., the number of inputs, the number of outputs,

  the magnitudes of the inputs and outputs, etc.
- Before the step count of a program can be determined, we need to know exactly which characteristics of the problem are to be used.

- For many programs, the time complexity is not dependent solely on the characteristics specified.

- The step count varies for different inputs of the same size.

  - Best case: the minimum number of steps that can be executed for the given parameters.

  - **Worst case:** the maximum number of steps that can be executed for the given parameters.

  - **Average:** the average number of steps executed on instances with the given parameters.

1.5.3 Asymptotic Notation (O, Ω, Θ)

- **Our motivation to determine step counts:**

  to compare the time complexities of two programs for the same function, and

  to predict the growth in run time as the instance characteristics change.

- Determining the exact step count (either worst case or average) of a program can prove to be an exceedingly difficult task.

- Expending immense effort to determine the step count exactly isn't a worthwhile endeavor as the notion of a step is itself inexact.

  (e.g., x = y and x = y+z+(x/y)+(x*y*z-x/t) count as one step)

- Because of the inexactness of what a step stands for, the exact step count isn't very useful for comparative purposes.

- For most situations, step counts can be represented as a function of instance characteristics, such as
  $C_1 n^2 \leq T_P(n) \leq C_2 n^2$ or $T_Q(n,m) = C_1 n + C_2 m$.

  What if the difference of two step counts are large?
  e.g., $3n + 3$ versus $100n + 100$.

  What if two step counts are of different orders?
  e.g., $c_1 n^2 + c_2 n$ versus $c_3 n$.

  $\rightarrow$ If n is very large, the program with complexity $c_3 n$ will be faster than the one with complexity $c_1 n^2 + c_2 n$, no matter what the values of c_1, c_2 and c_3.

- ***break even point:***
  - The value of n beyond which one program will be faster than the other program.
  - The exact break even point cannot be determined analytically.
  - The programs have to be run on a computer in order to determine the break even point.

Some terminology :

- **Definition : [Big "*oh*"]**

$$\boldsymbol{f(n) = O(g(n))}$$

iff there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all $n, n \ge n_0$.

$$3n + 2 = O(n)$$

$$3n + 3 = O(n)$$

$$100n + 6 = O(n)$$

$$10n^2 + 4n + 2 = O(n^2)$$

$$1000n^2 + 100n - 6 = O(n^2)$$

$$6 * 2^n + n^2 = O(2^n)$$

$$3n + 3 = O(n^2)$$

$$10n^2 + 4n + 2 = O(n^4)$$

$$3n + 2 \neq O(1)$$

$$10n^2 + 4n + 2 \neq O(n)$$

$O(1)$	a constant	$O(n^2)$	quadratic
$O(\log n)$	logarithm	$O(n^3)$	cubic
$O(n)$	linear	$O(2^n)$	exponential

In order for the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$.

- **Theorem 1.2 :**

  If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

- **Proof :**

$$
\begin{aligned}
f(n) &\leq \sum_{i=0}^{m} |a_i| n^i \\
&\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \\
&\leq n^m \sum_{i=0}^{m} |a_i|, \text{ for } n \geq 1.
\end{aligned}
$$

So, $f(n) = O(n^m)$. □

- **Definition : [Omega]**

$$\boldsymbol{f(n) = \Omega(g(n))}$$

iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$.

Example 1.16 :

$3n + 2 = \Omega(n)$

$3n + 3 = \Omega(n)$

$100n + 6 = \Omega(n)$

$10n^2 + 4n + 2 = \Omega(n^2)$

$6*2^n + n^2 = \Omega(2^n)$

$10n^2 + 4n + 2 = \Omega(n)$

$10n^2 + 4n + 2 = \Omega(1)$

$6*2^n + n^2 = \Omega(n^{100})$

$6*2^n + n^2 = \Omega(n^2)$

$6*2^n + n^2 = \Omega(n)$

$6*2^n + n^2 = \Omega(1)$

- In order for the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which $f(n) = \Omega(g(n))$ is true.

- **<u>Theorem 1.3</u>** :
  If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$,
  then $f(n) = \Omega(n^m)$.

- **Proof :**

From **Theorem 1.2** and **Definition of big-O**,

$\exists c, n_0$ with $c > 0$ such that $\sum_{i=0}^{m-1} |a_i| n^i \le cn^{m-1}$ for $\forall n \ge n_0$.

Note that

$$\sum_{i=0}^{m-1} -a_i n^i \le \sum_{i=0}^{m-1} |a_i| n^i \le cn^{m-1} \text{ and}$$

$$\sum_{i=0}^{m-1} a_i n^i \ge -\sum_{i=0}^{m-1} |a_i| n^i \ge -cn^{m-1} \text{ for } \forall n \ge n_0.$$

- **Proof (cont.)** :

Thus,

$$f(n) = a_m n^m + \sum_{i=0}^{m-1} a_i n^i \geq a_m n^m - cn^{m-1}$$

$$\geq \left(a_m - \frac{c}{n}\right) n^m \text{ for } \forall n \geq n_0.$$

Let n_1 be $\max\left\{n_0, ceiling\left(\frac{c}{a_m}\right)\right\} + 1$ and let c_1 be $a_m - \frac{c}{n_1}$.

Then $c_1 > 0$ and $f(n) \geq c_1 n^m$ for $\forall n \geq n_1$. □

- **Definition : [Theta]**

$$\boldsymbol{f(n) = \Theta(g(n))}$$

iff there exist positive constants c_1, c_2 and n_0
such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$.

Example 1.17 :

$3n + 2 = \Theta(n)$

$3n + 3 = \Theta(n)$

$10n^2 + 4n + 2 = \Theta(n^2)$

$6 * 2^n + n^2 = \Theta(2^n)$

$10 * \log n + 4 = \Theta(\log n)$

$3n + 2 \neq \Theta(1)$

$3n + 2 \neq \Theta(n^2)$

$10n^2 + 4n + 2 \neq \Theta(n)$

$10n^2 + 4n + 2 \neq \Theta(1)$

$6 * 2^n + n^2 \neq \Theta(n^{100})$

$6 * 2^n + n^2 \neq \Theta(n^2)$

$6 * 2^n + n^2 \neq \Theta(1)$

- **Theorem 1.4** :
  If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$,
  then $f(n) = \Theta(n^m)$.

- **Proof:**
  By **Theorem 1.2**,
  $\exists c_2, n_2$ such that $c_2 > 0$ and $f(n) \leq c_2 n^m$ for $\forall n \geq n_2$.
  Likewise, by **Theorem 1.3**,
  $\exists c_1, n_1$ such that $c_1 > 0$ and $f(n) \geq c_1 n^m$ for $\forall n \geq n_1$.
  Let n_0 be $\max(\mathrm{n}_1, n_2)$, then
  $c_1 n^m \leq f(n) \leq c_2 n^m$ for $\forall n \geq n_0$. □

- **Example 1.18*: [Complexity of matrix addition]***

Statement	Asymptotic complexity
`void add(int a[][MAX_SIZE] ...)`	0
`{`	0
`  int i, j;`	0
`  for (i=0; i<rows; i++)`	$\Theta(rows)$
`      for (j=0; j<cols; j++)`	$\Theta(rows \cdot cols)$
`          c[i][j] = a[i][j] + b[i][j];`	$\Theta(rows \cdot cols)$
`}`	0
Total	$\Theta(rows \cdot cols)$

- **Example 1.19 :** ***[Binary Search]***

  [Program 1.7] (Refer to next page)
  The instance characteristic -- number of elements in the list.
  Each iteration of *while* loop takes Θ(1) time.
  The *while* loop is iterated at most $\lceil \log_2(n+1) \rceil$ times.

  Worst case - the loop is iterated **Θ(log n)** times
  Best case - **Θ(1)**.

■ [Program 1.7]

```
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
        Return its position if found.  Otherwise return -1 */
    int middle;
    while (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum))
        {
            case -1 : left = middle + 1;
                        break;
            case 0 : return middle;
            case 1 : right = middle - 1;
        }
    }
    return -1;
}
```

- **Example 1.21 : *[Magic square]***

  The magic square is an $n \times n$ matrix of integers from 1 to n^2 such that the sum of each row and column and two major diagonals is the same.

  When n=5 : the common sum is 65.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

- **Coxeter's rule :**

  Put a one in the middle of the top row. Go up and left assigning numbers in increasing order to empty boxes. If your move cause you to jump off the square (that is, you go beyond the square's boundaries), figure out where you would be if you landed on a box on the opposite side of the square. Continue with this box. If a box is occupied, go down instead of up and continue.

[Program 1.23]

```
#include <stdio.h>
#define MAX_SIZE  15  /* maximum size of square */

void main(void)
/* construct a magic square, iteratively */
{
  static int square[MAX_SIZE] [MAX_SIZE];
  int i, j, row, column;        /* indices */
  int count;                    /* counter */
  int size;                     /* Square size */
```

```
printf (“Enter the size of the square: ”);
scanf(“%d”, &size);
 /* check for input errors */
 if (size<1 || size>MAX_SIZE+1) {
             fprintf(stderr, “Error! Size is out of range₩n”);
             exit(1);
    }
 if (!(size % 2))   {
             fprintf(stderr, “Error! Size is even₩n”);
             exit(1);
    }
 for (i=0; i<size; i++)
    for (j=0; j<size; j++)
             square[i][j] = 0;
```

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

```
square[0][(size-1)/2] = 1;  /* middle of first row */
/* i and j are current position  */
   i = 0;
   j = (size-1) / 2;
   for (count = 2; count <= size * size; count++)  {
          row = (i-1 < 0) ? (size-1) : (i-1);    /* up */
          column = (j-1 < 0) ? (size-1) : (j-1);  /* left */
          if (square[row][column])          /* down  */
             i = (++i) % size;
          else  {                      /* square is unoccupied */
             i = row;
             j = column;
          }
          square[i][j] = count;
     }
```

```
i = 0
j = (5-1)/2 = 2
row = ?
column = ?
```

0	0	1	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

```
square[0][(size-1)/2] = 1;  /* middle of first row */
/* i and j are current position  */
   i = 0;
   j = (size-1) / 2;
   for (count = 2; count <= size * size; count++)  {
          row = (i-1 < 0) ? (size-1) : (i-1);     /* up */
          column = (j-1 < 0) ? (size-1) : (j-1);  /* left */
          if (square[row][column])            /* down  */
            i = (++i) % size;
          else  {                       /* square is unoccupied */
            i = row;
            j = column;
          }
          square[i][j] = count;
     }
```

i = 0
j = 2
row = 5-1 = 4
column = j-1 = 1

0	0	1	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

```
square[0][(size-1)/2] = 1;  /* middle of first row */
/* i and j are current position  */
   i = 0;
   j = (size-1) / 2;
   for (count = 2; count <= size * size; count++)  {
            row = (i-1 < 0) ? (size-1) : (i-1);     /* up */
            column = (j-1 < 0) ? (size-1) : (j-1);  /* left */
            if (square[row][column])            /* down  */
               i = (++i) % size;
            else  {                        /* square is unoccupied */
               i = row;
               j = column;
            }
            square[i][j] = count;
     }
```

```
i = row = 4
j = column = 1
row = 4
column = 1
```

0	0	1	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	2	0	0	0

```
square[0][(size-1)/2] = 1;  /* middle of first row */
/* i and j are current position  */
   i = 0;
   j = (size-1) / 2;
   for (count = 2; count <= size * size; count++)  {
            row = (i-1 < 0) ? (size-1) : (i-1);     /* up */
            column = (j-1 < 0) ? (size-1) : (j-1);  /* left */
            if (square[row][column])            /* down  */
               i = (++i) % size;
            else  {                         /* square is unoccupied */
               i = row;
               j = column;
            }
            square[i][j] = count;
      }
```

i = 4
j = 1
row = 4-1 = 3
column = 1-1 =0

0	0	1	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	2	0	0	0

```
square[0][(size-1)/2] = 1;  /* middle of first row */
/* i and j are current position  */
   i = 0;
   j = (size-1) / 2;
   for (count = 2; count <= size * size; count++)  {
           row = (i-1 < 0) ? (size-1) : (i-1);     /* up */
           column = (j-1 < 0) ? (size-1) : (j-1);  /* left */
           if (square[row][column])            /* down  */
             i = (++i) % size;
           else  {                     /* square is unoccupied */
             i = row;
             j = column;
           }
           square[i][j] = count;
     }
```

```
i = row = 3
j = column = 0
row = 3
column = 0
```

0	0	1	0	0
0	0	0	0	0
0	0	0	0	0
3	0	0	0	0
0	2	0	0	0

```
square[0][(size-1)/2] = 1;  /* middle of first row */
/* i and j are current position  */
   i = 0;
   j = (size-1) / 2;
   for (count = 2; count <= size * size; count++)  {
          row = (i-1 < 0) ? (size-1) : (i-1);     /* up */
          column = (j-1 < 0) ? (size-1) : (j-1);  /* left */
          if (square[row][column])           /* down  */
            i = (++i) % size;
          else  {                      /* square is unoccupied */
            i = row;
            j = column;
          }
          square[i][j] = count;
    }
```

i = 3
j = 0
row = 3-1 = 2
column = 5-1 = 4

0	0	1	0	0
0	0	0	0	0
0	0	0	0	0
3	0	0	0	0
0	2	0	0	0

```
square[0][(size-1)/2] = 1;  /* middle of first row */
/* i and j are current position  */
   i = 0;
   j = (size-1) / 2;
   for (count = 2; count <= size * size; count++)  {
            row = (i-1 < 0) ? (size-1) : (i-1);    /* up */
            column = (j-1 < 0) ? (size-1) : (j-1);  /* left */
            if (square[row][column])           /* down  */
               i = (++i) % size;
            else  {                       /* square is unoccupied */
               i = row;
               j = column;
            }
            square[i][j] = count;
     }
```

```
i = row = 2
j = column = 4
row = 2
column = 4
```

0	0	1	0	0
0	0	0	0	0
0	0	0	0	4
3	0	0	0	0
0	2	0	0	0

```
/* output the magic square */
   printf("Magic Square of the size %d : ₩n₩n", size);
   for (i = 0; i < size; i++)   {
     for (j = 0; j < size; j++)
              printf ("%5d", square[i][j];
     printf("₩n");
    }
  printf("₩n ₩ n");
}
```

Result magic square:

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

instance characteristic -- *n* denoting the size of the magic square.

the nested for loops -- $\Theta(n^2)$

next for loop -- $\Theta(n^2)$

Others --- $\Theta(1)$

Total asymptotic complexity is $\Theta(n^2)$.

```
for (i=0; i<size; i++)
   for (j=0; j<size; j++)
      square[i][j] = 0;
```

```
for (count = 2; count <= size * size; count++) {
   row = (i-1 < 0) ? (size-1) : (i-1);    /* up */
   column = (j-1 < 0) ? (size-1) : (j-1);    /* left */
   if (square[row][column])    /* down  */
      i = (++i) % size;
   else  {    /* square is unoccupied */
      i = row;
      j = column;
   }
   square[i][j] = count;
}
```

1.5.4 Practical Complexities

- The time complexity of a program is generally some function of the instance characteristics.

- This complexity function:
  - is very useful in determining how the time requirements vary as the instance characteristics changes, and
  - may also be used to compare two programs P and Q that perform the same task.

Assume that program P has complexity $\Theta(n)$ and
program Q has complexity $\Theta(n^2)$.

We can assert that
program P is faster than program Q for *sufficiently large* n.

How the various functions grow with n?

Instance characteristic *n*							
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
log n	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
nlog n	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
n!	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

Figure 1.7 Function values

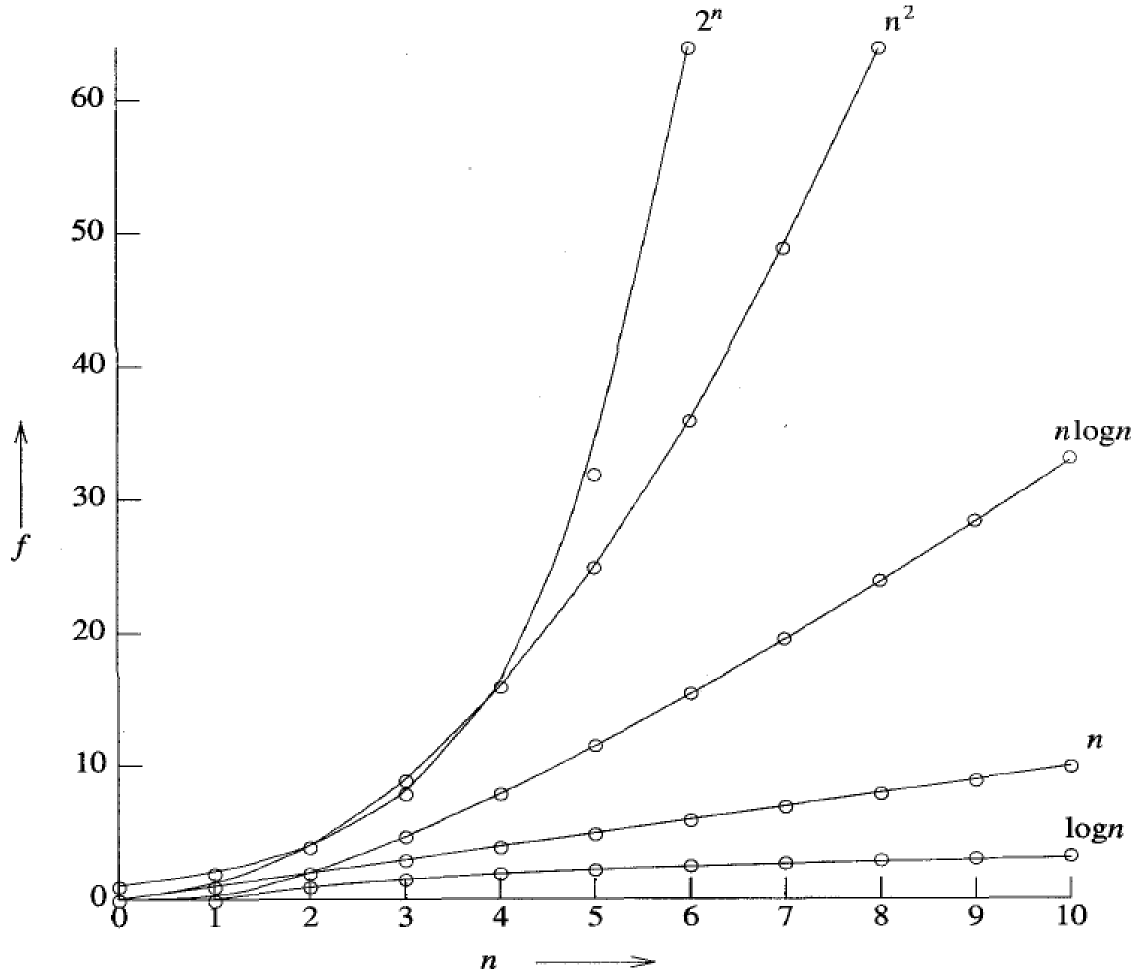
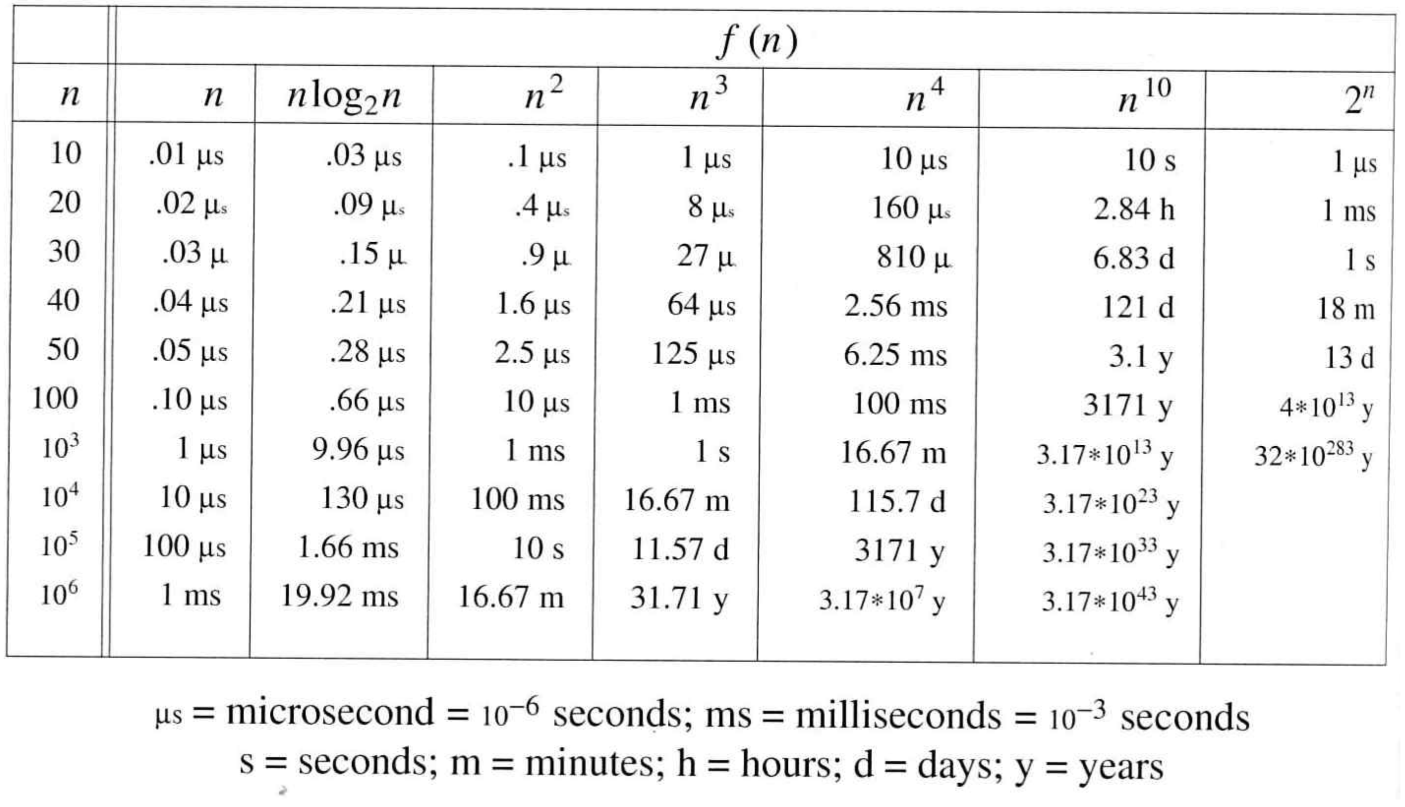


Figure 1.8 Plot of function values



	$f(n)$						
n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 µs	.03 µs	.1 µs	1 µs	10 µs	10 s	1 µs
20	.02 µs	.09 µs	.4 µs	8 µs	160 µs	2.84 h	1 ms
30	.03 µ	.15 µ	.9 µ	27 µ	810 µ	6.83 d	1 s
40	.04 µs	.21 µs	1.6 µs	64 µs	2.56 ms	121 d	18 m
50	.05 µs	.28 µs	2.5 µs	125 µs	6.25 ms	3.1 y	13 d
100	.10 µs	.66 µs	10 µs	1 ms	100 ms	3171 y	$4*10^{13}$ y
10^3	1 µs	9.96 µs	1 ms	1 s	16.67 m	$3.17*10^{13}$ y	$32*10^{283}$ y
10^4	10 µs	130 µs	100 ms	16.67 m	115.7 d	$3.17*10^{23}$ y	
10^5	100 µs	1.66 ms	10 s	11.57 d	3171 y	$3.17*10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17*10^7$ y	$3.17*10^{43}$ y	

µs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

Figure 1.9 Times on a 1 billion instruction per second computer

1.6 PERFORMANCE MEASUREMENT

- **How to measure real execution time.**

  - Use of C's standard library.
    Functions are accessed through the statement:
    #include <time.h>.

  - Inaccurate results can be produced for small data
    (e.g. if the value of CLOCKS_PER_SEC is 18 on our computer, and the number of clock ticks is less than 10)

	Method 1	Method 2
Start timing	start=clock();	start=time(NULL);
Stop timing	stop=clock();	stop=time(NULL);
Type returned	clock_t	time_t
Result in seconds	duration= ((double)(stop-start))/ CLOCKS_PER_SEC;	duration= (double) difftime(stop, start);

Figure 1.10: Event timing in C

[Program 1.24 First timing program for selection sort]

```
#include <stdio.h>
#include <time.h>
#include “selectionSort.h”
#define MAX_SIZE 1001

void main(void)
{
      int i, n, step=10;
      int a[MAX_SIZE];
      double duration;
      clock_t start;
```

```
    /* times for n=0, 10, …, 100, 200, …, 1000 */
    printf("          n          time₩n")
    for(n=0; n<=1000; n+=step)
    { /* get time for size n */

        /* initialize with worst-case data */
        for(i=0; i<n; i++)
            a[i] = n-i;

        start = clock();
        sort(a, n);
        duration = ((double)(clock()-start)) / CLOCKS_PER_SEC;
        printf("%6d  %f₩n", n, duration);
        if(n==100) step=100;
    }
}
```

- Although **Program 1.24** is logically correct, it can fail to measure run times accurately because the events we are trying to time are too short.

- In **Program 1.25**, for each n, we do the sort as many times as needed to bring the total time up to 1 second.

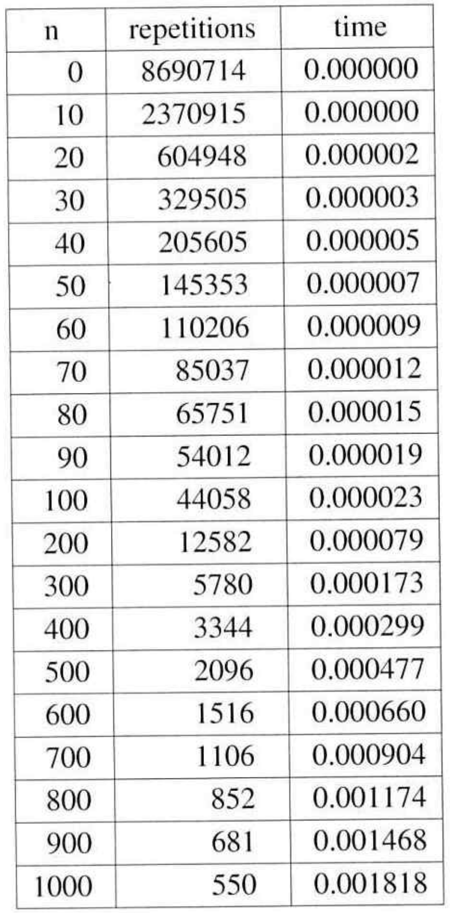
  (refer to the following page)

[Program 1.25 More accurate timing program for selection sort]

```
#include <stdio.h>
#include <time.h>
#include “selectionSort.h”
#define MAX_SIZE 1001

void main(void)
{
    int i, n, step=10;
    int a[MAX_SIZE];
    double duration;
```

```
/* times for n=0, 10, …, 100, 200, …, 1000 */
printf("          n          repetitions          time₩n")
for(n=0; n<=1000; n+=step)
{ /* get time for size n */
    long repetitions = 0;
    clock_t start = clock();
    do
    {
        repetitions++;

        /* initialize with worst-case data */
        for(i=0; i<n; i++)
            a[i] = n - i;

        sort(a, n);
    } while(clock()-start < 1000); /* repeat until enough time has elapsed */
```

```
            duration = ((double)(clock()-start)) / CLOCKS_PER_SEC;
            duration /= repetitions;
            printf("%6d  %9d        %f₩n", n, repetitions, duration);
            if(n==100) step = 100;
        }
}
```



n	repetitions	time
0	8690714	0.000000
10	2370915	0.000000
20	604948	0.000002
30	329505	0.000003
40	205605	0.000005
50	145353	0.000007
60	110206	0.000009
70	85037	0.000012
80	65751	0.000015
90	54012	0.000019
100	44058	0.000023
200	12582	0.000079
300	5780	0.000173
400	3344	0.000299
500	2096	0.000477
600	1516	0.000660
700	1106	0.000904
800	852	0.001174
900	681	0.001468
1000	550	0.001818

Figure 1.11: Worst case performance of selection sort (in seconds)

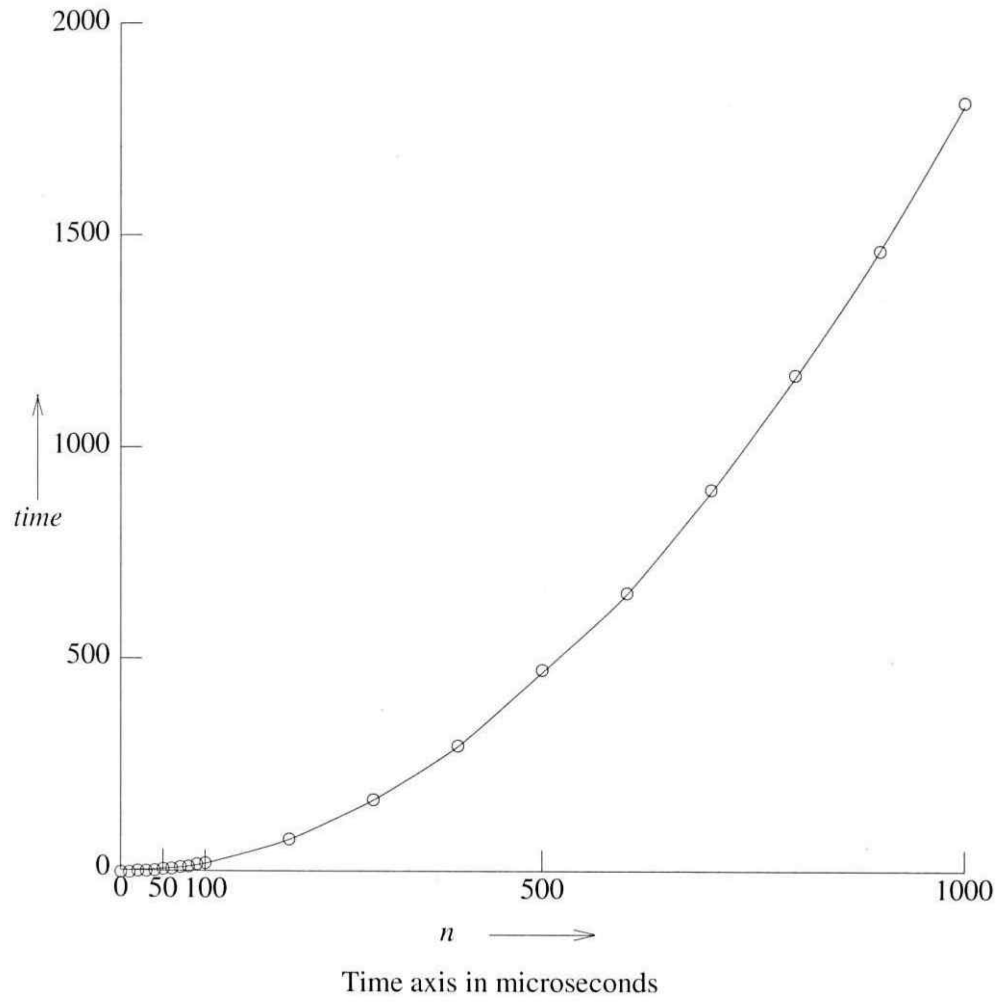


Figure 1.12: Graph of worst case performance of selection sort

Generating Test Data

- Generating a data set that results in the worst case performance of a program isn't always easy.
- We may generate a suitably large number of random test data.
- Obtaining average case data is usually much harder.
- It is desirable to analyze the algorithm being tested to determine classes of data that should be generated for the experiment - algorithm specific task.