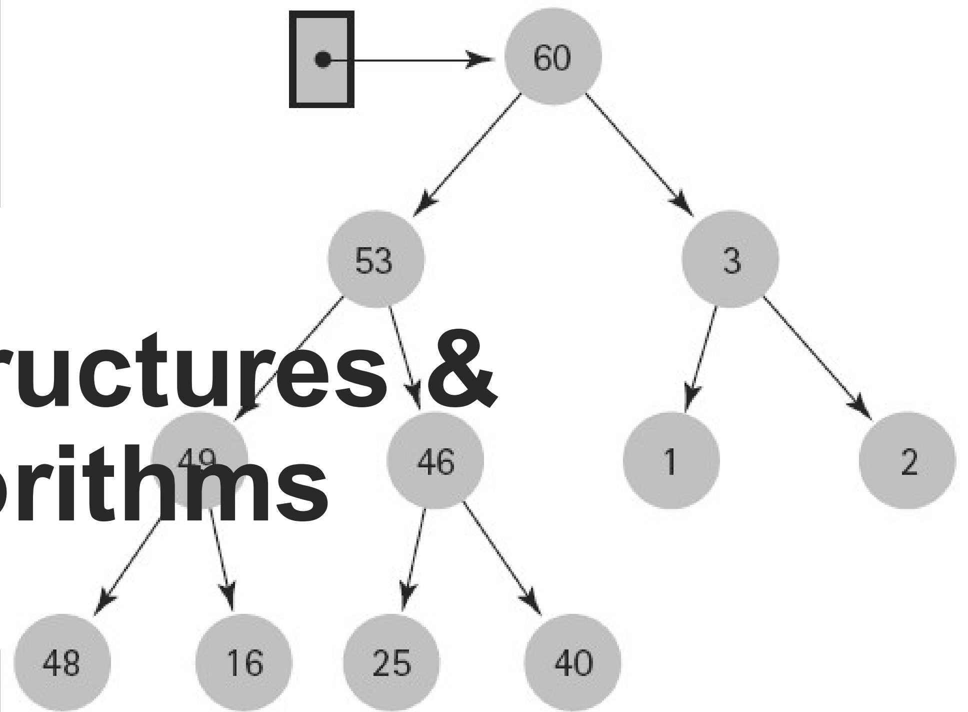
Data Structures & Algorithms



Week1

Contents

- Textbook
- Grade
- Software

Textbook

- C & Data Structures
  - P. S. Deshpande, O. G. Kakde
  - CHARLES RIVER MEDIA, INC.
    Hingham, Massachusetts

Grade

- Midterm test (Lab)
- Final test (Lab)
- Project (working on group)
- Multiple choice test
- How to Grade

HỌC PHẦN

Học lại từ đầu

KHÔNG ĐẠT *lần 1*

TIỂU LUẬN (Đối với học phần lý thuyết)

BÀI TẬP TỔNG HỢP (Đối với học phần thực hành)

KHÔNG ĐẠT *lần 2*

ĐẠT

THI GIỮA HỌC PHẦN

KHÔNG ĐẠT

Thi lại

ĐẠT

ĐẠT

KHÔNG ĐẠT

Xét

ĐẠT

KHÔNG ĐẠT

Học lại từ đầu

THI KẾT THÚC HỌC PHẦN

KHÔNG ĐẠT

Thi lại

KHÔNG ĐẠT

ĐẠT

ĐẠT

ĐẠT

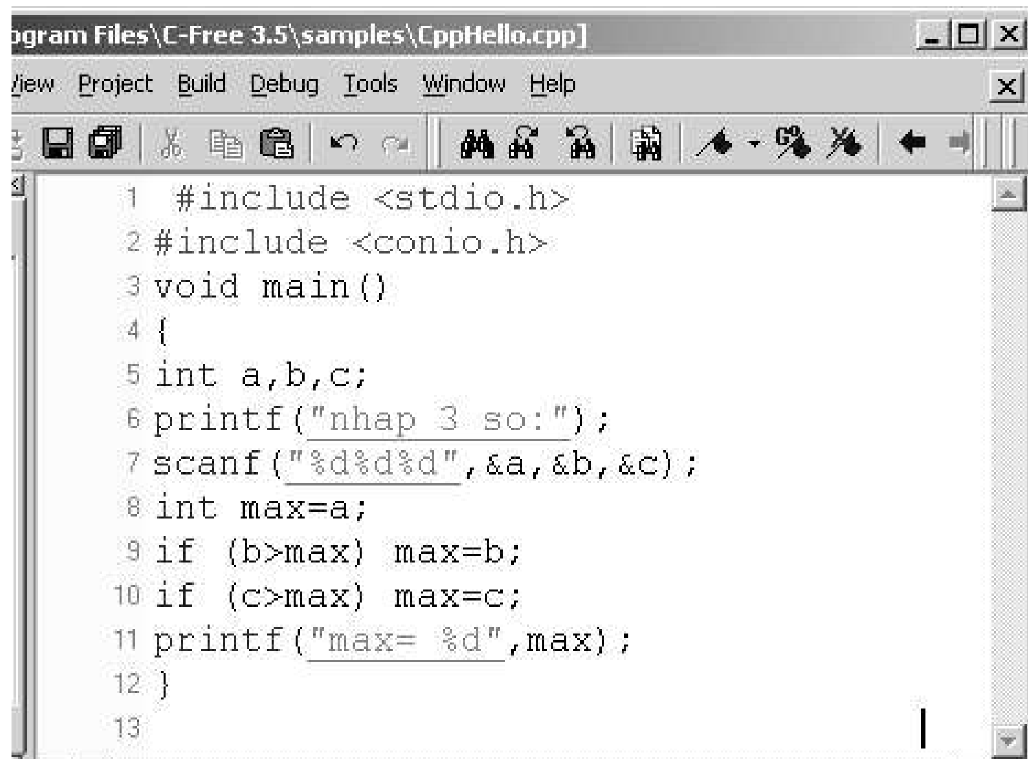
Software: C/C++ edittor

- **BC++, TC++**
- **C-Free** is a professional C/C++ integrated development environment (IDE) that support multi-compilers. Use of this software, user can edit, build, run and debug programs freely.

  With C/C++ source parser included
- Lightweight C/C++ development tool.
- http://www.programarts.com/cfree_en/

C/C++ edittor: demo

- Find max of 3 numbers: a,b,c
  - Using scanf, prinf (C standard)
  - Using cin, cout (Cpp)



CHAPTER 0: INTRODUTION

- ***What is Data Structures?***
  - ***A data structure is defined by***
    - ***(1) the logical arrangement of data elements, combined with***
    - ***(2) the set of operations we need to access the elements.***

Atomic Variables

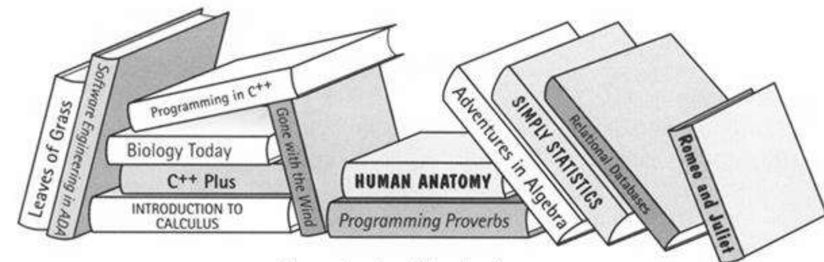
- Atomic variables can only store one value at a time.

```
int num;
float s;
```

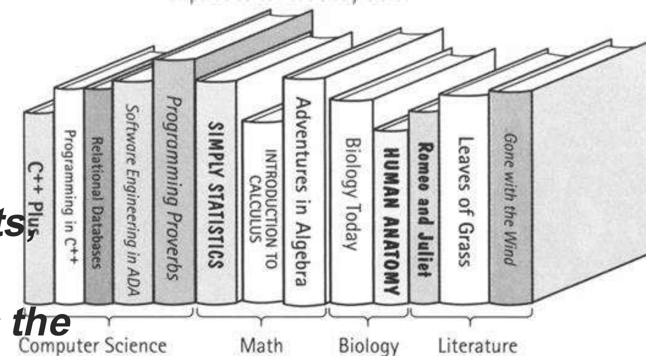
- A value stored in an atomic variable cannot be subdivided.

What is Data Structures?

- Example:library
  - is composed of elements (books)
  - Accessing a particular book requires knowledge of the arrangement of the books
  - Users access books only through the librarian



the logical arrangement of data elements, combined with
the set of operations we need to access the elements.

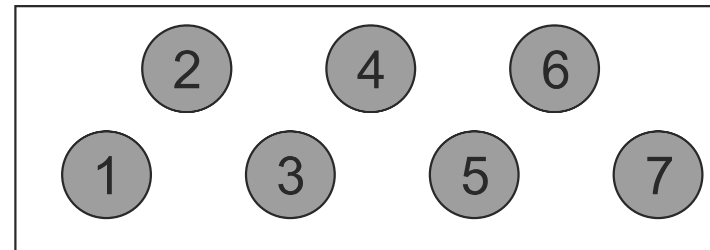


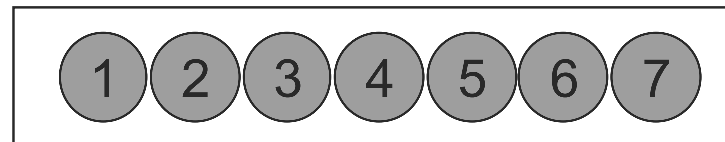
Basic Data Structures

- Structures include
  - linked lists
  - Stack, Queue
  - binary trees
  - …and others

What is Algorithm?

- **Algorithm:**
  - A computable set of steps to achieve a desired result
  - Ralationship to Data Structure
    - Example: Find an element





Sumary

Algorithms + Data Structures = Programs

Algorithms $\longleftrightarrow$ Data Structures

Chapter 0: C LANGUAGE

1. ***ADDRESS***
2. ***POINTERS***
3. ***ARRAYS***
4. ***ADDRESS OF EACH ELEMENT IN AN ARRAY***
5. ***ACCESSING & MANIPULATING AN ARRAY USING POINTERS***
6. ***ANOTHER CASE OF MANIPULATING AN ARRAY USING POINTERS***
7. ***TWO-DIMENSIONAL ARRAY***
8. ***POINTER ARRAYS***
9. ***STRUCTURES***
10. ***STRUCTURE POINTERS***

Chapter 0: C LANGUAGE

1. *ADDRESS*

For every variable there are two attributes:
<u>address</u> and <u>value</u>

1	4096
2	"Dave"
3	45
4	"Matt"
5	95.5
6	"wbru"
7	0
8	"zero"

In memory with address 3: value: 45.
In memory with address 2: value "Dave"

```
cout << "Value of 'y' is: " << y << "\n";
cout << "Address of 'y' is: " << &y << "\n\n";
```

Chapter 0: C LANGUAGE

2. POINTERS

1. ***is a variable whose value is also an address.***
2. ***A pointer to an integer is a variable that can store the address of that integer***

ia: value of variable

&ia: address of ia

***ia means you are printing the value at the location specified by ia**

1000, i	10
4000, ia	—, 1000

Chapter 0: C LANGUAGE

```
int i;          //A
  int * ia;      //B
   cout<<"The address of i "<< &i << " value="<<i <<endl;
   cout<<"The address of ia " << &ia << " value = " << ia<< endl;

  i = 10;        //C
  ia = &i;   //D

          cout<<"after assigning value:"<<endl;
   cout<<"The address of i "<< &i << " value="<<i <<endl;
   cout<<"The address of ia " << &ia << " value = " << ia<< " point to: "<< *ia;
```

Chapter 0: C LANGUAGE

Points to Remember

- Pointers give a facility to access the value of a variable indirectly.
- You can define a pointer by including a * before the name of the variable.
- You can get the address where a variable is stored by using &.

Chapter 0: C LANGUAGE

3. ARRAYS

1. ***An array is a data structure***
2. ***used to process multiple elements with the same data type when a number of such elements are known.***
3. ***An array is a composite data structure; that means it had to be constructed from basic data types such as array integers.***

    1. ***int a[5];***
    2. ***for(int i = 0;i<5;i++)***
        1. ***{a[i]=i; }***

Chapter 0: C LANGUAGE

4. ADDRESS OF EACH ELEMENT IN AN ARRAY

Each element of the array has a memory address.

```
void printdetail(int a[])
{
    for(int i = 0;i<5;i++)
    {
        cout<< "value in array “<< a[i] <<“ at address: “ << &a[i]);
    }
```

Chapter 0: C LANGUAGE

5. ACCESSING & MANIPULATING AN ARRAY USING POINTERS

- ***You can access an array element by using a pointer.***
- ***If an array stores integers->use a pointer to integer to access array elements.***

```
void printarr_usingpointer(int a[])
{
    int *pi;
    pi=a;
     for(int i = 0;i<5;i++)
      {
         cout<<"value array:" << *pi << " address: "<< pi<<endl;
       pi++;
      }
}
```

Chapter 0: C LANGUAGE

6. ANOTHER CASE OF MANIPULATING AN ARRAY USING POINTERS

The array limit is a pointer constant : cannot change its value in the program.

int a[5]; int *b;

a=b; //error

b=a; //OK

```
void print_usingptr_a(int a[])
{
    for(int i = 0;i<5;i++)
      {cout<<"a[" << i << "]=" <<*a<<endl;
        a++;}
}
```

It works correctly even using a++ ???

Chapter 0: C LANGUAGE

7. TWO-DIMENSIONAL ARRAY

int a[3][2];

3	1
5	2
8	7

```
int a[3][2];
for(int i = 0;i<3;i++)
    for(int j=0;j<2 ;j++)
    {
            {
                    a[i][j]=i+j+i*j;
            }
    }
```

```
void print_usingptr(int a[][2])
{
    int *b;
    b=a[0];
    for(int i = 0;i<6;i++)
    {
        cout<<"value :" << *b<<" in address: "<<b<<endl;
        b++; }
}
```

Chapter 0: C LANGUAGE

8. POINTER ARRAYS

- ***You can define a pointer array (similarly to an array of integers).***
- ***In the pointer array, the array elements store the pointer that points to integer values.***

```
int *a[5];
main()
{   int i1=4,i2=3,i3=2,i4=1,i5=0;
    a[0]=&i1;
    a[1]=&i2;
    a[2]=&i3;
    a[3]=&i4;
    a[4]=&i5;

    printarr(a);
    printarr_usingptr(a);
}
```

```
void printarr(int *a[])
{
    printf("Address1\tAddress2\tValue\n");
    for(int  j=0;j<5;j++)
    {
        cout<<*a[j]<<"\t"<<a[j]<<"\t"<<&a[j]<<endl;
    }
}
```

Chapter 0: C LANGUAGE

9. STRUCTURES

- ***Structures are used when you want to process data of multiple data types***
- ***But you still want to refer to the data as a single entity***
- ***Access data: structurename.membername***

```
struct student
{
    char name[30];
    float marks;
};

main ( )
{
    student student1;
    char s1[30];float f;
    cin>>student1.name;
    cin>>student1.marks;

    cout<<"Name is:"<<student1.name;
    cout<< "Marks are:" << student1.marks;
}
```

Chapter 1: C LANGUAGE

10. STRUCTURE POINTERS

Process the structure using a structure pointer

```
struct student
{   char name[30];
    float marks;   };
main ( )
{
     struct student *student1;
     struct student student2;
     student1 = &student2;
     cin>>student1->name; // cin>>(*student1).name;
     cin>>student2.marks;
     cout<<(*student1).name<<" : " << student2.marks;
}
```

CHAPTER 2: **FUNCTION & RECURSION**

- 1. FUNCTION
- 2. THE CONCEPT OF STACK
- 3. THE SEQUENCE OF EXECUTION DURING A FUNCTION CALL
- 4. PARAMETER PASSING
- 5. CALL BY REFERENCE
- 6. RESOLVING VARIABLE REFERENCES
- 7. RECURSION
- 8. STACK OVERHEADS IN RECURSION
- 9. WRITING A RECURSIVE FUNCTION
- 10. TYPES OF RECURSION

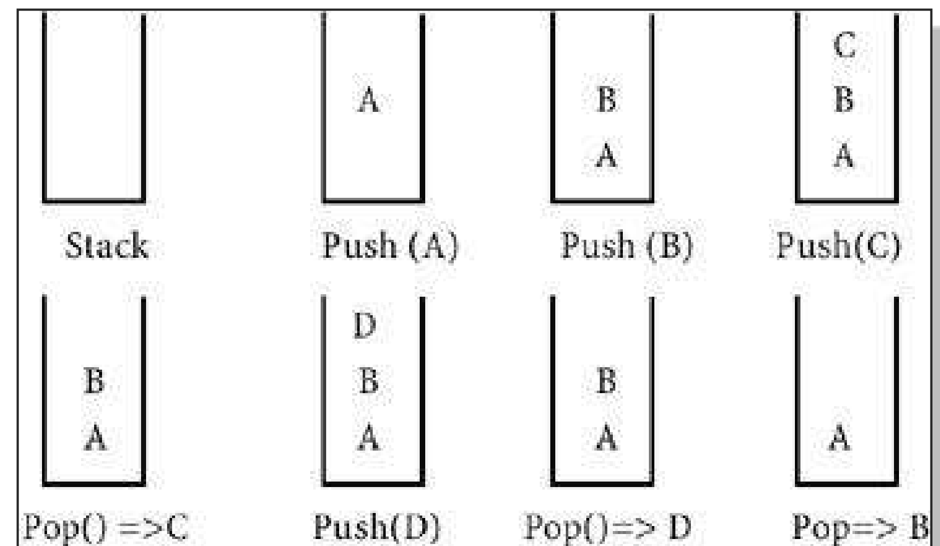
CHAPTER 2: **FUNCTION & RECURSION**

- 1. FUNCTION
  - provide modularity to the software
  - divide complex tasks into small manageable tasks
  - avoid duplication of work

```
int add (int x, int y)
{
    int z;
    z = x + y;
    return z;
}
```

CHAPTER 2: **FUNCTION & RECURSION**

- 2. THE CONCEPT OF STACK
  - A *stack* is memory in which values are stored and retrieved in "last in first out" manner by using operations called *push* and *pop*.



CHAPTER 2: **FUNCTION & RECURSION**

- 3. THE SEQUENCE OF EXECUTION DURING A FUNCTION CALL
  - When the function is called, the current execution is temporarily stopped and the control goes to the called function. After the call, the execution resumes from the point at which the execution is stopped.
  - To get the exact point at which execution is resumed, the address of the next instruction is stored in the stack. When the function call completes, the address at the top of the stack is taken.

- 3. THE SEQUENCE OF EXECUTION DURING A FUNCTION CALL
  - Functions or sub-programs are implemented using a stack.
  - When a function is called, the address of the next instruction is pushed into the stack.
  - When the function is finished, the address for execution is taken by using the pop operation.

CHAPTER 2: **FUNCTION & RECURSION**

- 3. THE SEQUENCE OF EXECUTION DURING A FUNCTION CALL
- Result:?

```
main ( )
{
    printf ("1 \n");
    printf ("2 \n");
    f1();
    printf ("3 \n");
    printf ("4 \n");
}
void f1()
{
    printf ("f1-5 \n");
    printf ("f1-6 \n");
    f2 ( );
    printf ("f1-7 \n");
    printf ("f1-8 \n");
}
void f2 ()
{
    printf ("f2-9 \n");
    printf ("f2-10 \n");
}
```

CHAPTER 2: **FUNCTION & RECURSION**

- 4. PARAMETER * REFERENCE PASSING
  - *passing by value*
    - the value before and after the call remains the same
  - *passing by reference*
    - *changed value after the function completes*

```
void (int *k)
{
    *k = *k + 10;
}
```

```
void (int &k)
{
    k = k + 10;
}
```

CHAPTER 2: **FUNCTION & RECURSION**

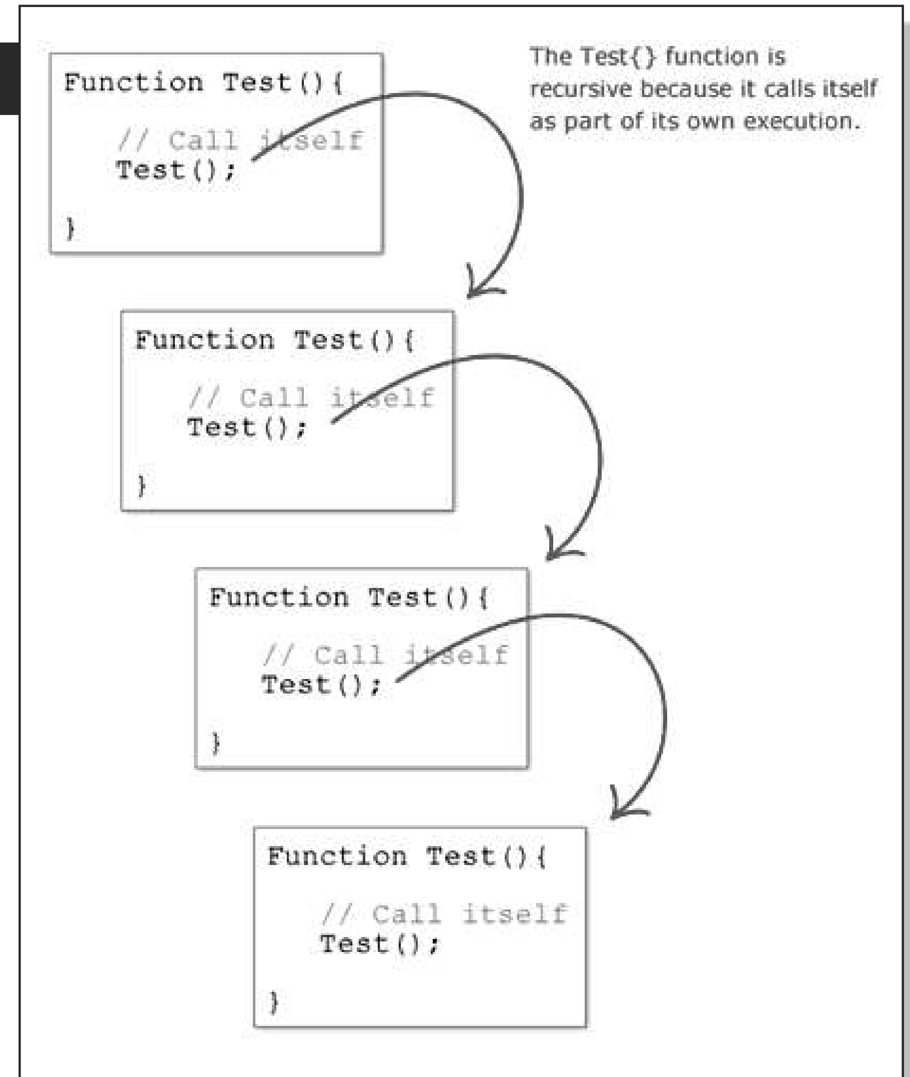
- 6. RESOLVING VARIABLE REFERENCES

When a variable can be resolved by using multiple references, the local definition is given more preference

```
int i =0;  //Global variable
main()
{
    int i ;   // local variable for main
    void f1(void) ;
    i =0;
    cout<<"value of i in main: "<< i <<endl;
    f1();
    cout<<"value of i after call:" << i<<endl;
}
void f1()
{
    int i=0;   //local variable for f1
    i = 50;
}
```

CHAPTER 2: **FUNCTION & RECURSION**

- 7. RECURSION
  - A method of programming whereby a function directly or indirectly calls itself
  - Problems: stop recursion?



CHAPTER 2: **FUNCTION & RECURSION**

- 7. RECURSION

Example: Factorial

- The factorial is defined in this way:

$$n! = \begin{cases} 1 & n = 0 \\ 1 \times 2 \times \cdots \times n & n > 0 \end{cases}$$

This can be computed by a loop.

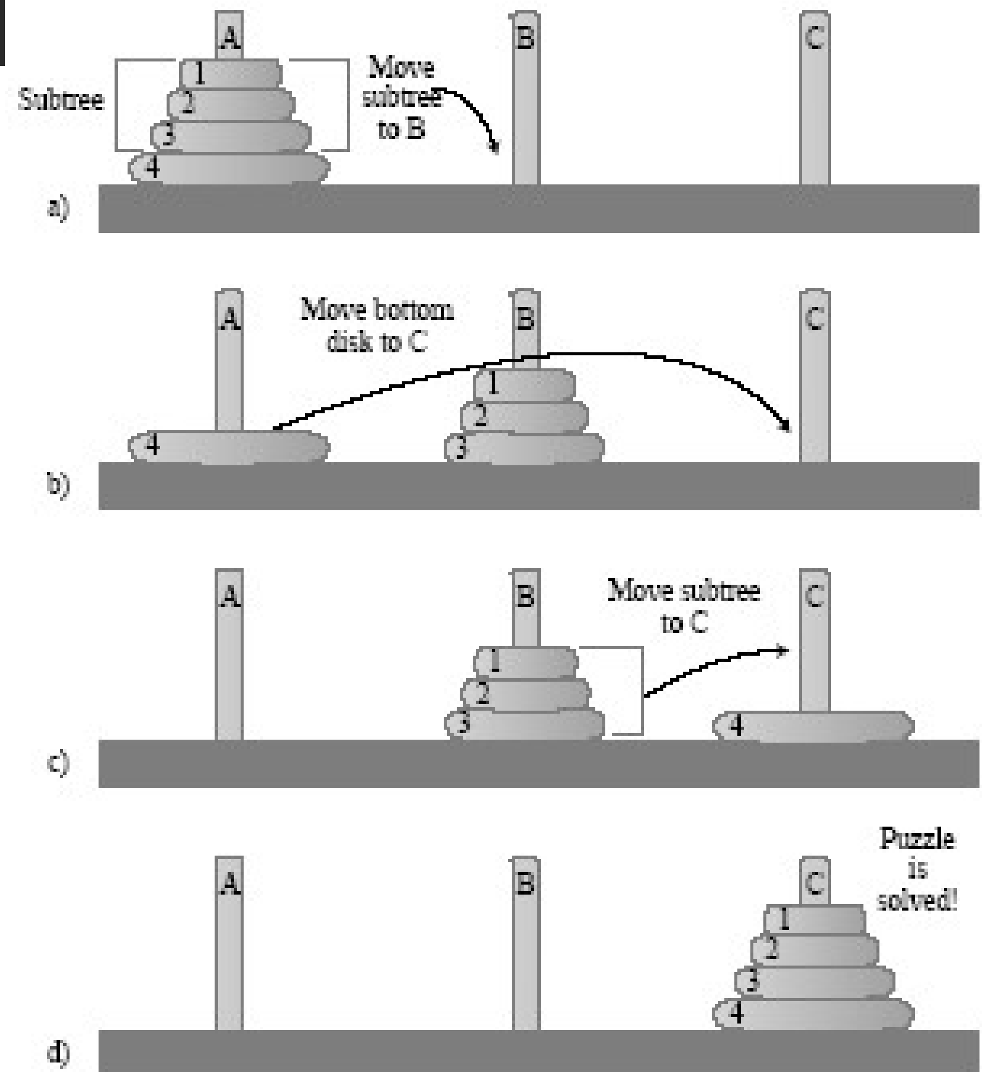
- We can also define the factorial as:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

So, if we know how to compute the factorial of $n - 1$, we know how to compute the factorial of n.

CHAPTER 2: **FUNCTION & RECURSION**

- 7. RECURSION: Hanoi tower



CHAPTER 2: **FUNCTION & RECURSION**

- 7. RECURSION

Example: Factorial

```
int factorial(int n)  // assumes n >= 0
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

To see how the computation is done, trace `factorial(3)`:

```
factorial(3) = 3 * factorial(2)
             = 3 * (2 * factorial(1))
             = 3 * (2 * (1 * factorial(0)))
             = 3 * (2 * (1 * 1))
```

- 8. STACK OVERHEADS IN RECURSION
  - two important results: the depth of recursion and the stack overheads in recursion

CHAPTER 2: **FUNCTION & RECURSION**

- 9. WRITING A RECURSIVE FUNCTION
  - Recursion enables us to write a program in a natural way. The speed of a recursive program is slower because of stack overheads.
  - In a recursive program you have to specify recursive conditions, terminating conditions, and recursive expressions.

CHAPTER 2: **FUNCTION & RECURSION**

- 10. TYPES OF RECURSION
  - **LINEAR RECURSION (tuyến tính)**
  - **TAIL RECURSION (đuôi)**
  - **BINARY RECURSION (nhị phân)**
  - **EXPONENTIAL RECURSION (đa tuyến)**
  - **NESTED RECURSION (lồng)**
  - **MUTUAL RECURSION (tương hỗ)**

CHAPTER 2: **FUNCTION & RECURSION**

- 10. TYPES OF RECURSION
  - **LINEAR RECURSION**
    - **only makes a single call to itself each time the function runs**
    -

```
int factorial (int n)
{
 if ( n == 0 )
   return 1;
 return n * factorial(n-1);
}
```

CHAPTER 2: **FUNCTION & RECURSION**

- 10. TYPES OF RECURSION
  - **TAIL RECURSION**
    - **Tail recursion is a form of linear recursion.**
    - **In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned.**

```
int gcd(int m, int n)
{
        int r;
        if (m < n) return gcd(n,m);
        r = m%n;
        if (r == 0) return(n);
        else return(gcd(n,r));
}
```

CHAPTER 2: **FUNCTION & RECURSION**

- 10. TYPES OF RECURSION
  - **BINARY RECURSION**
    - **Some recursive functions don't just have one call to themself, they have two (or more).**

```
int choose(int n, int k)
{
        if (k == 0 || n == k) return(1);
        else return(choose(n-1,k) + choose(n-1,k-1));
}
```

CHAPTER 2: **FUNCTION & RECURSION**

- 10. TYPES OF RECURSION
  - **EXPONENTIAL RECURSION**
    - **An exponential recursive function is one that, if you were to draw out a representation of all the function calls, would have an exponential number of calls in relation to the size of the data set**
    - **(exponential meaning if there were *n* elements, there would be $O(a^n)$ function calls where a is a positive number)**

CHAPTER 2: **FUNCTION & RECURSION**

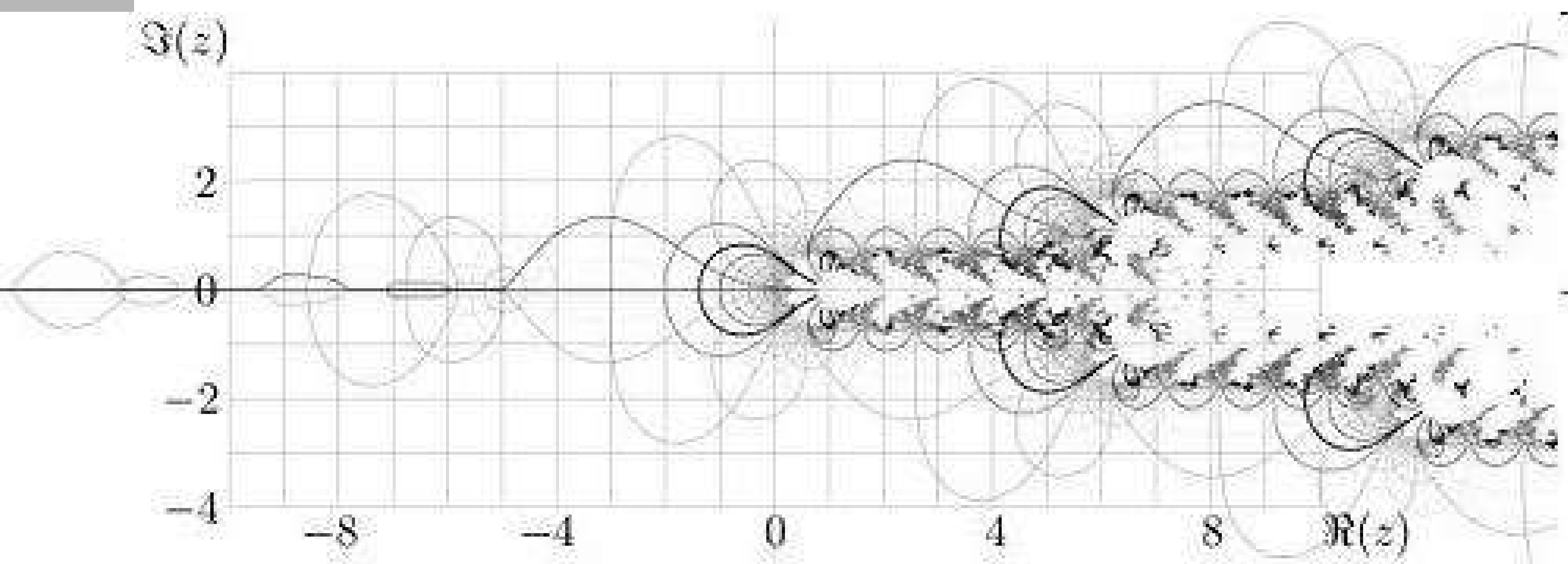
- 10. TYPES OF RECURSION
  - **EXPONENTIAL RECURSION**

```
void print_array(int arr[], int n)
{
        int i;
        for(i=0; i<n; i++) printf("%d ", arr[i]);
        printf("\n");
}

void print_permutations(int arr[], int n, int i)
{
        int j, swap;
        print_array(arr, n);
        for(j=i+1; j<n; j++) {
                swap = arr[i]; arr[i] = arr[j]; arr[j] = swap;
                print_permutations(arr, n, i+1);
                swap = arr[i]; arr[i] = arr[j]; arr[j] = swap;
        }
}
```

CHAPTER 2: **FUNCTION & RECURSION**

- 10. TYPES OF RECURSION
  - **NESTED RECURSION**
    - **In nested recursion, one of the arguments to the recursive function is the recursive function itself**
    - **These functions tend to grow extremely fast.**

```
int ackerman(int m, int n)
{
        if (m == 0) return(n+1);
        else if (n == 0) return(ackerman(m-1,1));
        else return(ackerman(m-1,ackerman(m,n-1)));
}
```



ℑ(z)
2
0
−2
−4
−8
−4
0
4
8
ℜ(z)

m\n	0	1	2	3	4	n
0	0+1	1+1	2+1	3+1	4+1	$n+1$
1	A(0,1)	A(0,A(1,0))	A(0,A(1,1))	A(0,A(1,2))	A(0,A(1,3))	$n+2=2+(n+3)-3$
2	A(1,1)	A(1,A(2,0))	A(1,A(2,1))	A(1,A(2,2))	A(1,A(2,3))	$2n+3=2*(n+3)-3$
3	A(2,1)	A(2,A(3,0))	A(2,A(3,1))	A(2,A(3,2))	A(2,A(3,3))	$2^{(n+3)}-3$
4	A(3,1)	A(3,A(4,0))	A(3,A(4,1))	A(3,A(4,2))	A(3,A(4,3))	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3\text{ twos}}-3$
5	A(4,1)	A(4,A(5,0))	A(4,A(5,1))	A(4,A(5,2))	A(4,A(5,3))	A(4, A(5, n-1))
6	A(5,1)	A(5,A(6,0))	A(5,A(6,1))	A(5,A(6,2))	A(5,A(6,3))	A(5, A(6, n-1))

CHAPTER 2: **FUNCTION & RECURSION**

- 10. TYPES OF RECURSION
  - **MUTUAL RECURSION**
    - A recursive function doesn't necessarily need to call itself.
    - Some recursive functions work in pairs or even larger groups. For example, function A calls function B which calls function C which in turn calls function A.

CHAPTER 2: **FUNCTION & RECURSION**

- 10. TYPES OF RECURSION
  - **MUTUAL RECURSION**

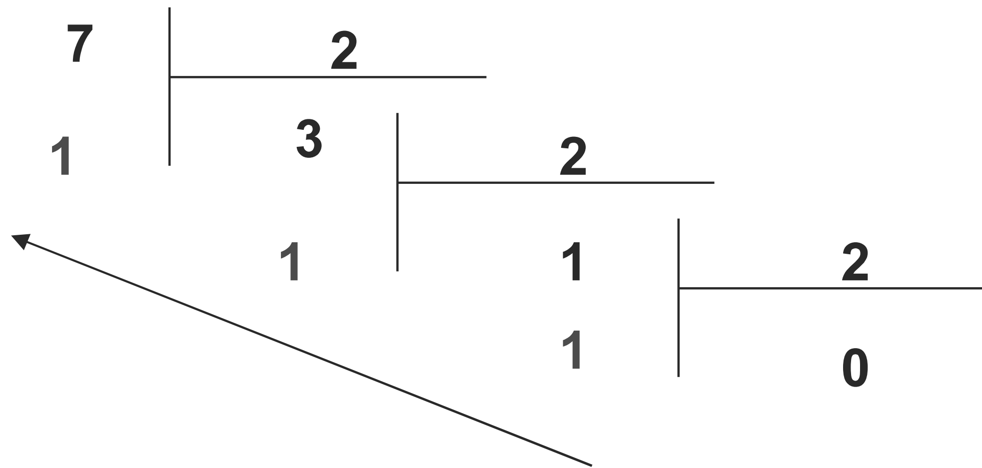
```
int is_even(unsigned int n)
{
        if (n==0) return 1;
        else return(is_odd(n-1));
}

int is_odd(unsigned int n)
{
        return (!is_even(n));
}
```

Exercises 1: Recursion

```
*
**
***
****
*****
******
*******
********
```

Exercises 2: Recursion

- Convert number from H10->H2



Week3: Recursion Excercises (1)

- E1. (44/174) Write a program to compute: S = 1 + 2 + 3 + …n using recursion.

Week3: Recursion Excercises (2-3)

- E3(a). Write a program to print a revert number Example: input n=12345. Print out: 54321.

- E3(b). Write a program to print this number Example: input n=12345. Print out: 12345.

Week3: Recursion Excercises (4)

- E4. Write a recursion function to find the sum of every number in a int number. Example: n=1980 => Sum=1+9+8+0=18.

Week3: Recursion Excercises (5)

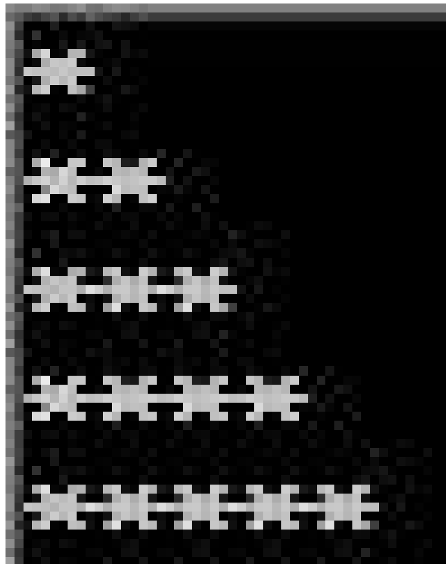
- E4. Write a recursion function to calculate:
  - S=a[0]+a[1]+…a[n-1]
    - A: array of integer numbers

Week3: Recursion Excercises (6)

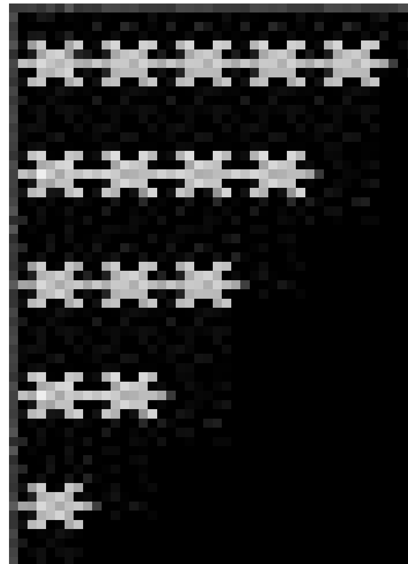
- E4. Write a recursion function to find an element in an array (using linear algorithm)

Week3: Recursion Excercises (7)

- Print triangle

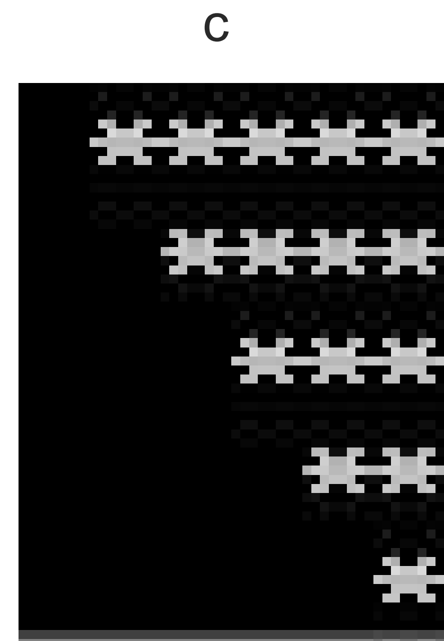


a

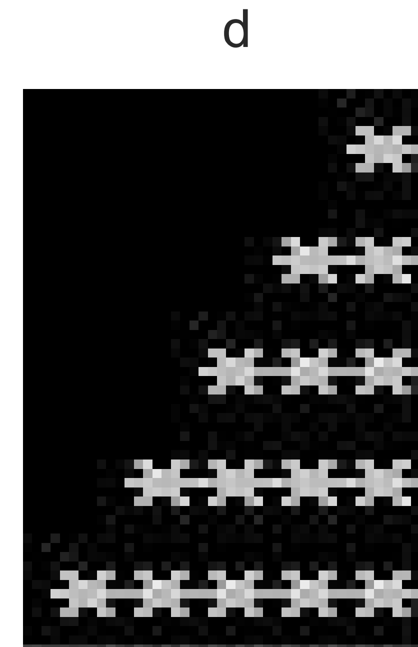


b

c

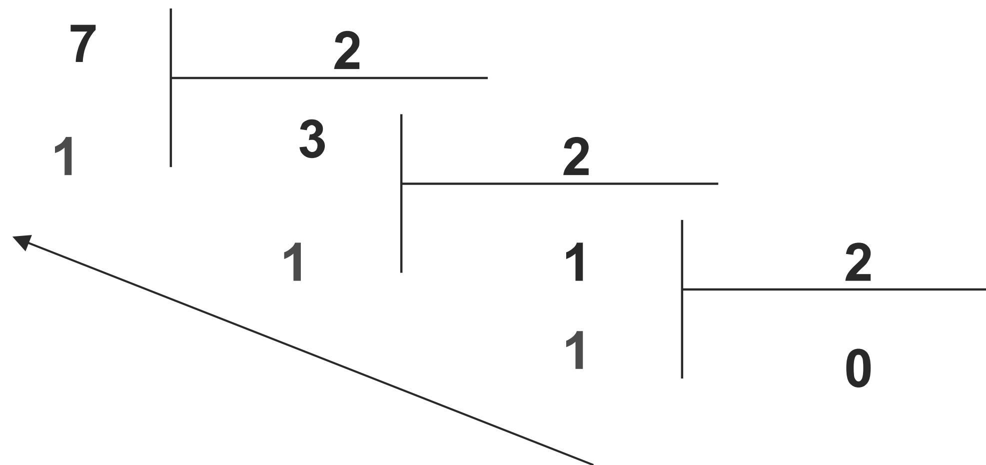


d



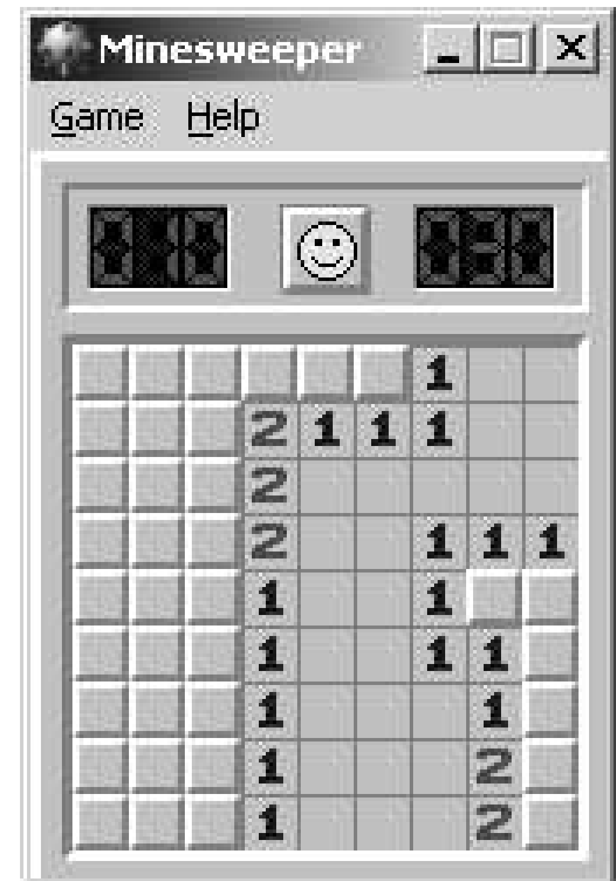
Week3: Recursion Excercises (8)

- Convert number from H_{10}->H_2



Week3: Recursion Excercises (9)

- Minesweeper



Week 4

CHAPTER 3: SEARCHING TECHNIQUES

1. ***LINEAR (SEQUENTIAL) SEARCH***
2. ***BINARY SEARCH***
3. ***COMPLEXITY OF ALGORITHMS***

SEARCHING TECHNIQUES

- To finding out whether a particular element is present in the list.
- 2 methods: linear search, binary search
- The method we use depends on how the elements of the list are organized
  - unordered list:
    - linear search: simple, slow
  - an ordered list
    - binary search or linear search: complex, faster

1. *LINEAR (SEQUENTIAL) SEARCH*

- How?
  - Proceeds by sequentially comparing the key with elements in the list
  - Continues until either we find a match or the end of the list is encountered.
  - If we find a match, the search terminates successfully by returning the index of the element
  - If the end of the list is encountered without a match, the search terminates unsuccessfully.

1. *LINEAR (SEQUENTIAL) SEARCH*

```
void lsearch(int list[],int n,int element)
{   int i, flag = 0;
  for(i=0;i<n;i++)
  if( list[i] == element)
      {  cout<<"found at position"<<i;
         flag =1;
         break;   }
  if( flag == 0)
      cout<<" not found";
}
```

flag: what for???

1. *LINEAR (SEQUENTIAL) SEARCH*

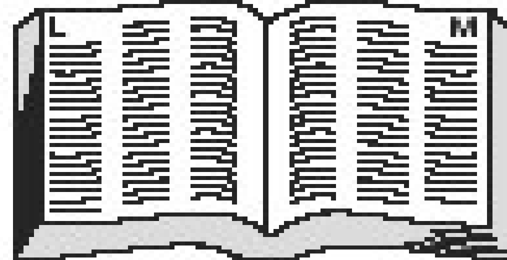
```
int lsearch(int list[],int n,int element)
{   int i, find= -1;
   for(i=0;i<n;i++)
          if( list[i] == element)
                  {find =i;
                  break;}
    return find;
}
```

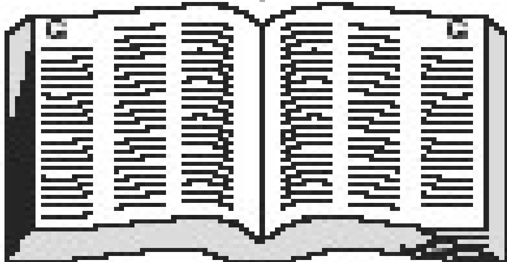
Another way using flag

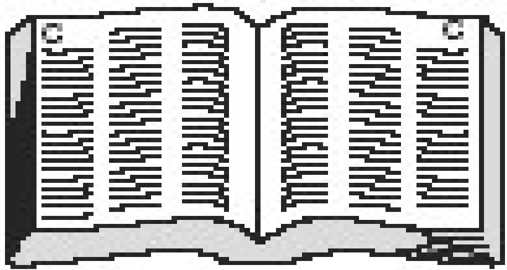
average time: O(*n*)

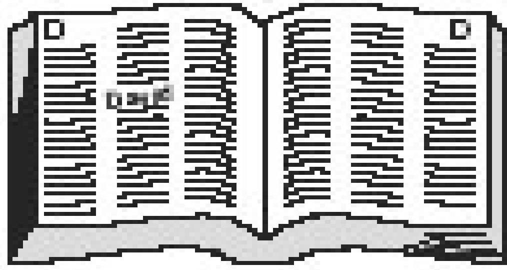
2. BINARY SEARCH

- List must be a sorted one
- We compare the element with the element placed approximately in the middle of the list
- If a match is found, the search terminates successfully.
- Otherwise, we continue the search for the key in a similar manner either in the upper half or the lower half.

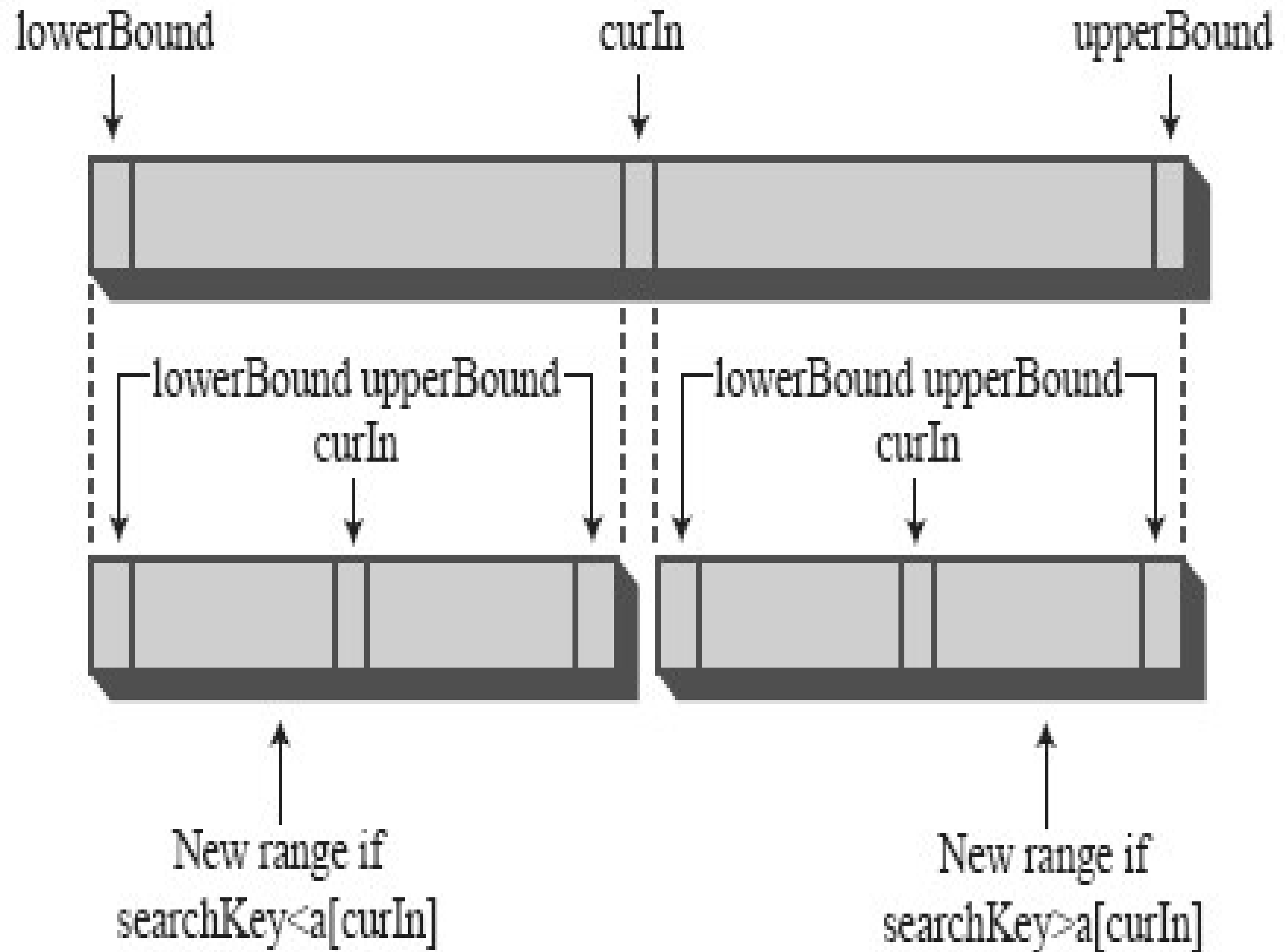








A binary search of the phone book


lowerBound
curIn
upperBound
lowerBound upperBound
curIn
lowerBound upperBound
curIn
New range if
searchKey<a[curIn]
New range if
searchKey>a[curIn]

```
void  bsearch(int list[],int n,int element)
{
   int l,u,m, flag = 0;
   l = 0;    u = n-1;
   while(l <= u)
   {  m = (l+u)/2;
      if( list[m] == element)
              {cout<<"found:"<<m;
               flag =1;
               break;}
      else
          if(list[m] < element)
               l = m+1;
          else
               u = m-1;
   }
   if( flag == 0)
     cout<<"not found";
}
```

average time: O($\log_2 n$)

BINARY SEARCH: Recursion

```
int Search (int list[], int key, int left, int right)
  {
  if (left <= right) {
    int middle = (left + right)/2;
    if (key == list[middle])
         return middle;
    else if (key < list[middle])
         return Search(list,key,left,middle-1);
    else     return Search(list,key,middle+1,right);
  }
  return -1;
 }
```

3. COMPLEXITY OF ALGORITHMS

- In Computer Science, it is important to measure the quality of algorithms, especially the specific amount of a certain resource an algorithm needs
- Resources: time or memory storage (PDA?)
- Different algorithms do same task with a different set of instructions in less or more time, space or effort than other.
- The analysis has a strong mathematical background.
- The most common way of qualifying an algorithm is the Asymptotic Notation, also called Big O.

3. COMPLEXITY OF ALGORITHMS

- It is generally written as $O(f(x_1, x_2, \ldots, x_n))$
- Polynomial time algorithms,
  - $O(1)$ --- Constant time --- the time does not change in response to the size of the problem.
  - $O(n)$ --- Linear time --- the time grows linearly with the size (n) of the problem.
  - $O(n^2)$ --- Quadratic time --- the time grows quadratically with the size (n) of the problem. In big O notation, all polynomials with the same degree are equivalent, so $O(3n^2 + 3n + 7) = O(n^2)$
- Sub-linear time algorithms
  - $O(\log n)$ -- Logarithmic time
- Super-polynomial time algorithms
  - $O(n!)$
  - $O(2^n)$

3. COMPLEXITY OF ALGORITHMS

- Example1: complexity of an algorithm

```
void f ( int a[], int n )
{
    int i;
    cout<< "N = “<< n;
    for ( i = 0; i < n; i++ )
        cout<<a[i];
    printf ( "n" );
}
```

2 * O(1) + O(N)

O(N)

3. COMPLEXITY OF ALGORITHMS

- Example2: complexity of an algorithm

```
void f ( int a[], int n )
{    int i;
    cout<< "N = “<< n;
    for ( i = 0; i < n; i++ )
            for (int j=0;j<n;j++)
                    cout<<a[i]<<a[j];

    for ( i = 0; i < n; i++ )
            cout<<a[i];
    printf ( "n" );
}
```

$2 * O(1) + O(N)+O(N^2)$

O(N2)

3. COMPLEXITY OF ALGORITHMS

- Linear Search
  - O(n).
- **Binary Search**
  - **O(log2 N)**

n	$\log_2 n$
10	3
100	6
1,000	9
10,000	13
100,000	16

Week 4

CHAPTER 4: SORTING TECHNIQUES

1. ***BUBBLE SORT***
2. ***INSERTION SORT***
3. ***SELECTION SORT***
4. ***QUICK SORT***

SORTING TECHNIQUES

SORTING TECHNIQUES

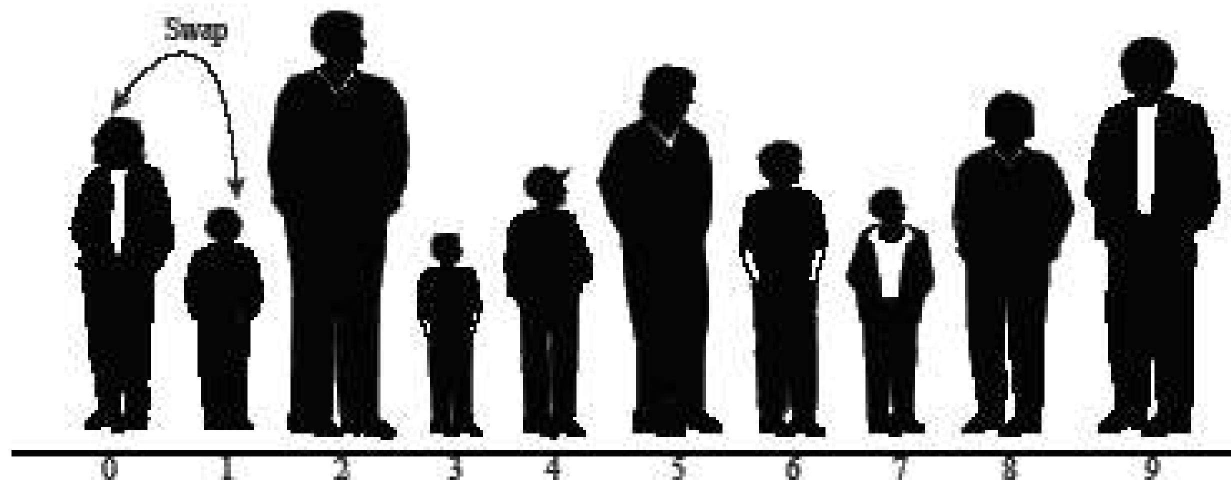
We need to do sorting for the following reasons :
a) By keeping a data file sorted, we can do binary search on it.
b) Doing certain operations, like matching data in two different files, become much faster.
There are various methods for sorting: Bubble sort, Insertion sort, Selection sort, Quick sort, Heap sort, Merge sort…. They having different average and worst case behaviours:

1. *BUBBLE SORT*

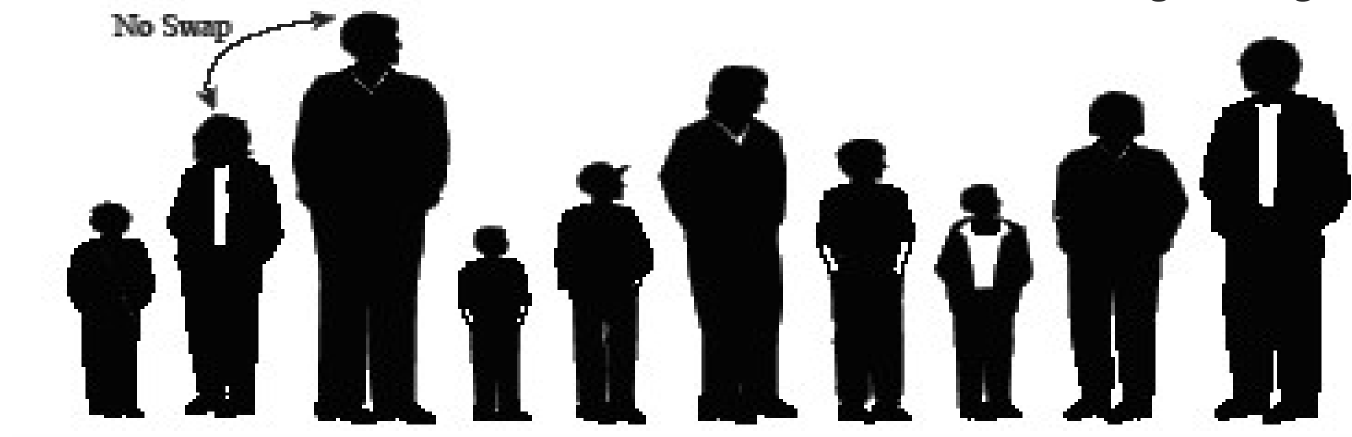
Introduction:

Bubble sorting is a simple sorting technique in which we arrange the elements of the list by forming pairs of adjacent elements. That means we form the pair of the ith and (i+1)th element. If the order is ascending, we interchange the elements of the pair if the first element of the pair is greater than the second element.

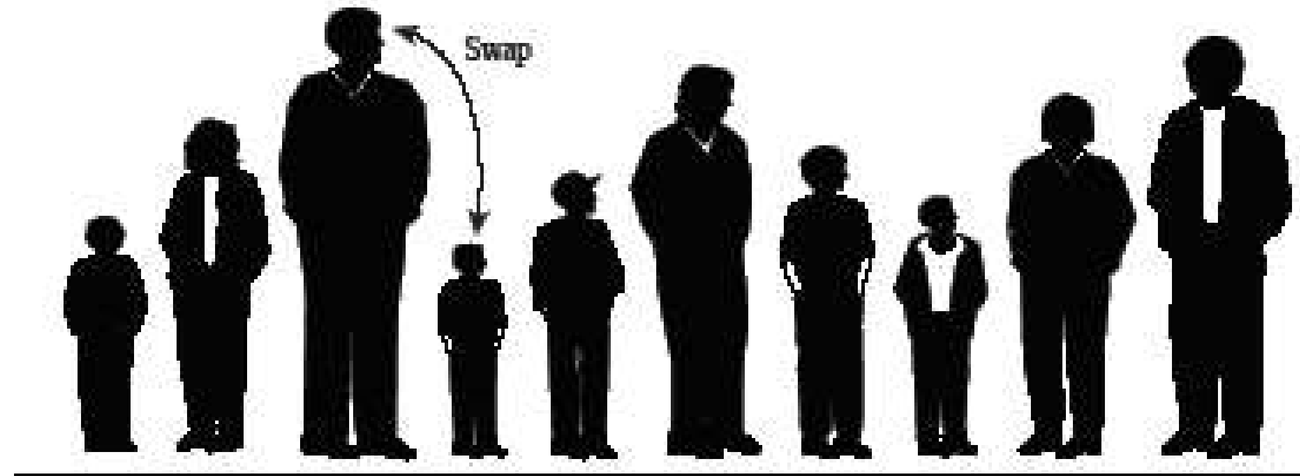
1. *BUBBLE SORT*

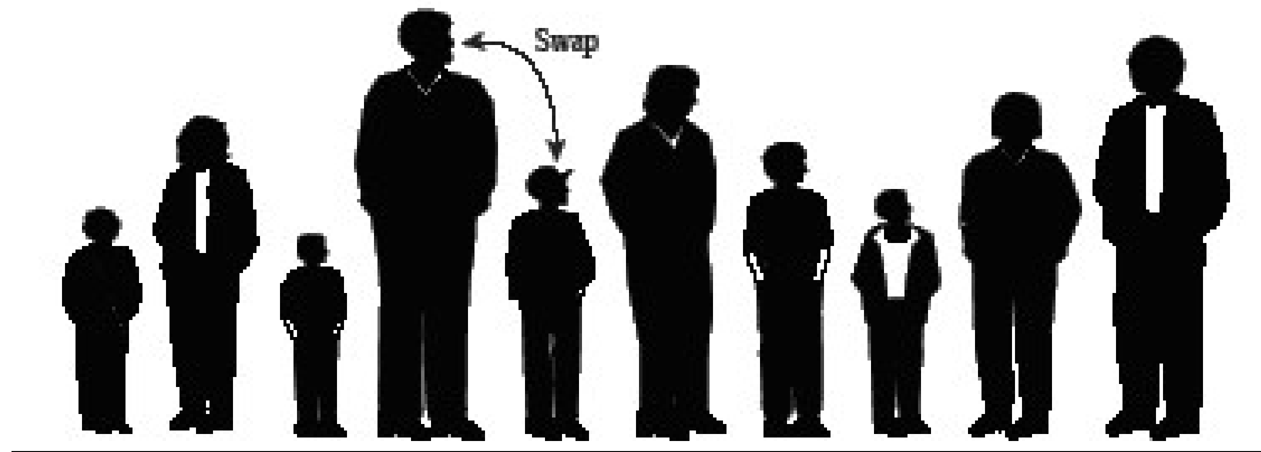


Bubble sort: beginning of first pass

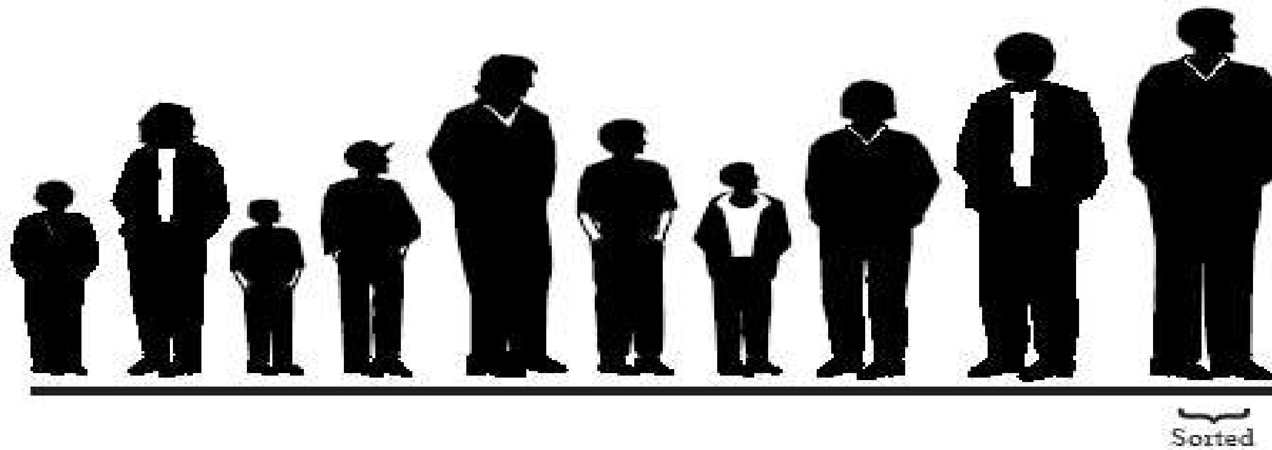


1. *BUBBLE SORT*





1. *BUBBLE SORT*



Bubble sort: end of First pass

1. *BUBBLE SORT*

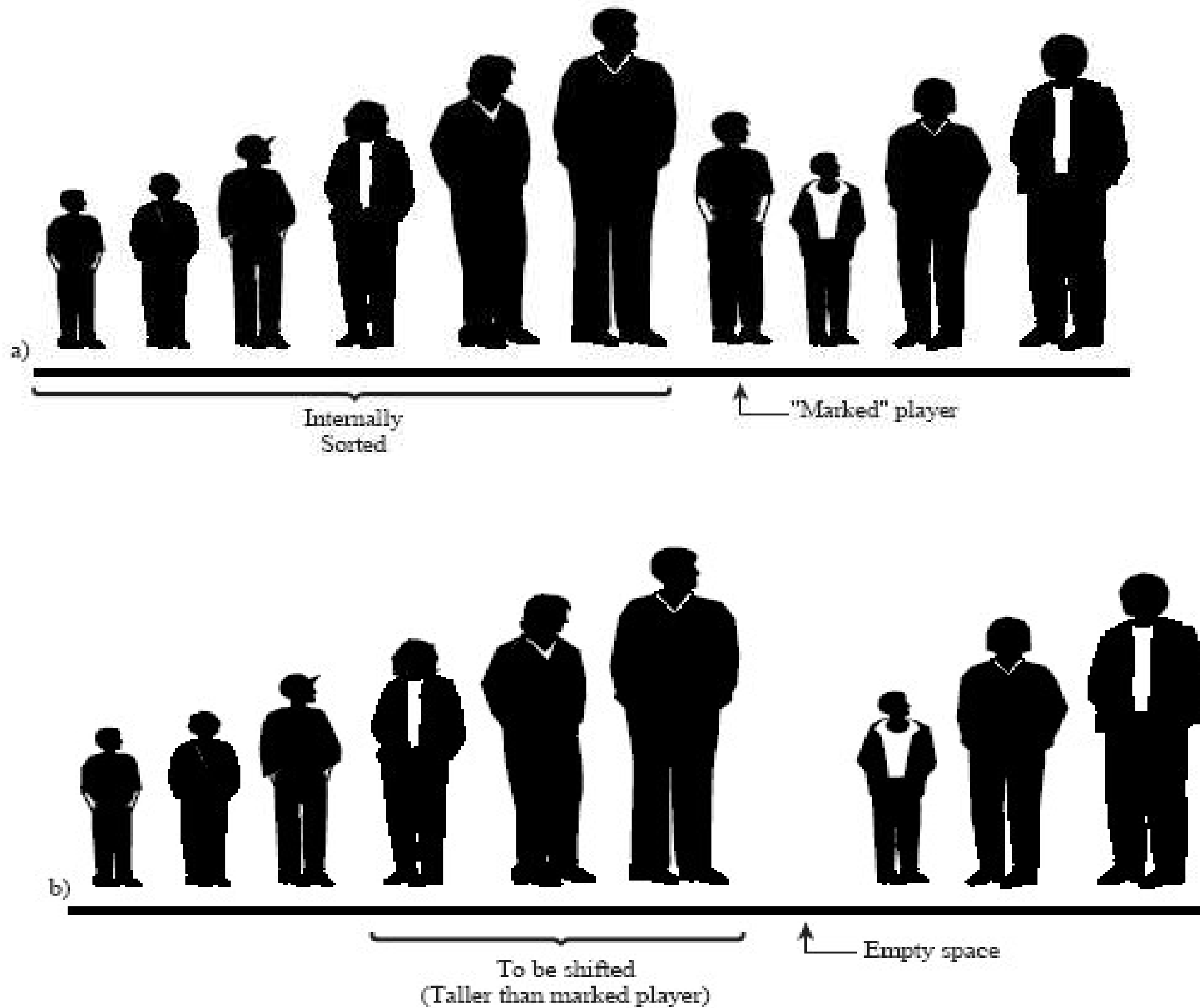
```
void bsort(int list[], int n)
{
 int i,j;
   for(i=0;i<(n-1);i++)
   for(j=0;j<(n-(i+1));j++)
        if(list[j] > list[j+1])
        swap(&list[j],&list[j+1]);
}
```

2. INSERTION SORT

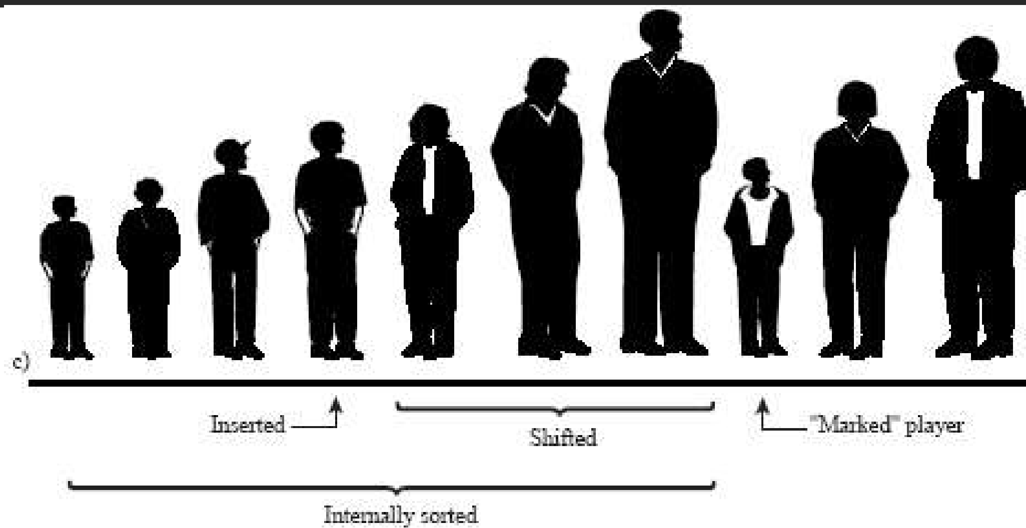
Introduction:

Basic Idea: Insert a record R into a sequence of ordered records: R1,R2,.., Ri with keys K1 $\le$ K2 $\le$ ... $\le$ Ki , such that, the resulting sequence of size i+1 is also ordered with respect to key values.

2. INSERTION SORT



2. INSERTION SORT



2. INSERTION SORT

```
Algorithm Insertion_Sort; (* Assume Ro has Ko = -maxint *)
void InsertionSort( Item &list[])
{ // Insertion_Sort
     Item r;
     int i,j;
     list[0].key = -maxint;
     for (j=2; j<=n; j++)
     {       r=list[j];
             i=j-1;
     while ( r.key < list[i].key )
     {// move greater entries to the right
             list[i+1]:=list[i];
             i:=i-1;
     };
     list[i+1] = r // insert into it's place
     }
}
```

3. SELECTION SORT

Selection sort is a simplicity sorting algorithm. It works as its name as it is. Here are basic steps of selection sort algorithm:

1. Find the minimum element in the list
2. Swap it with the element in the first position of the list
3. Repeat the steps above for all remainder elements of the list starting at the second position.

3. SELECTION SORT

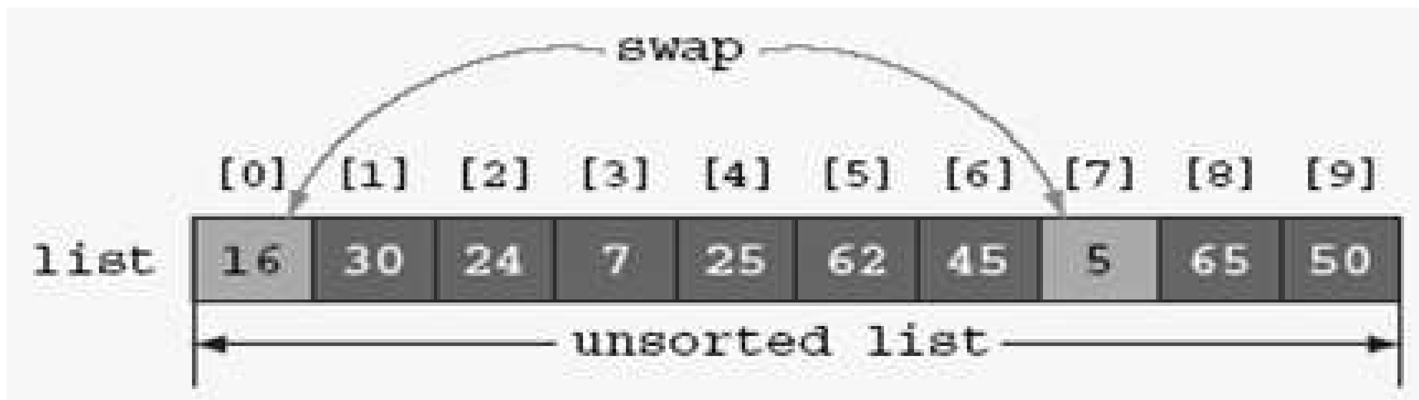
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
list	16	30	24	7	25	62	45	5	65	50

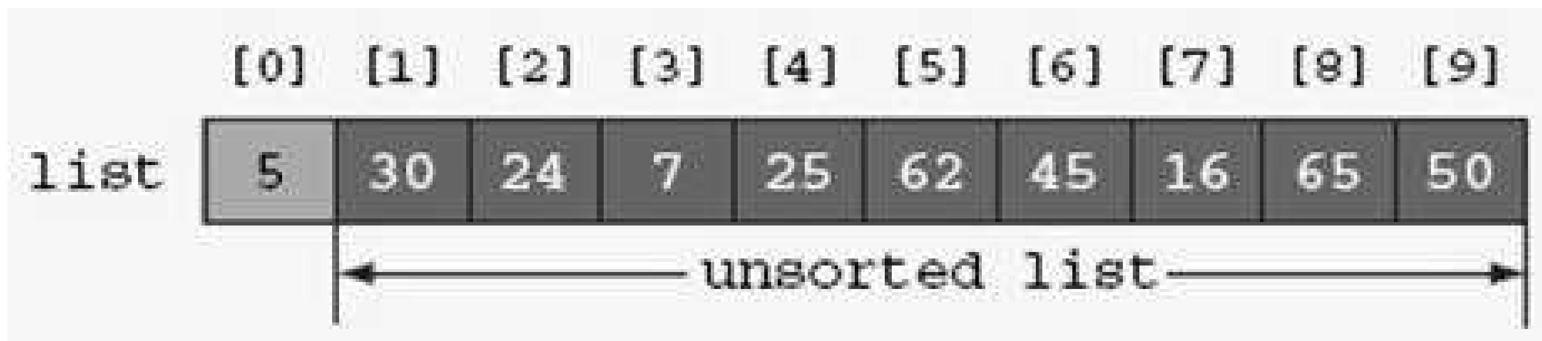
smallest

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
list	16	30	24	7	25	62	45	5	65	50

unsorted list

3. SELECTION SORT

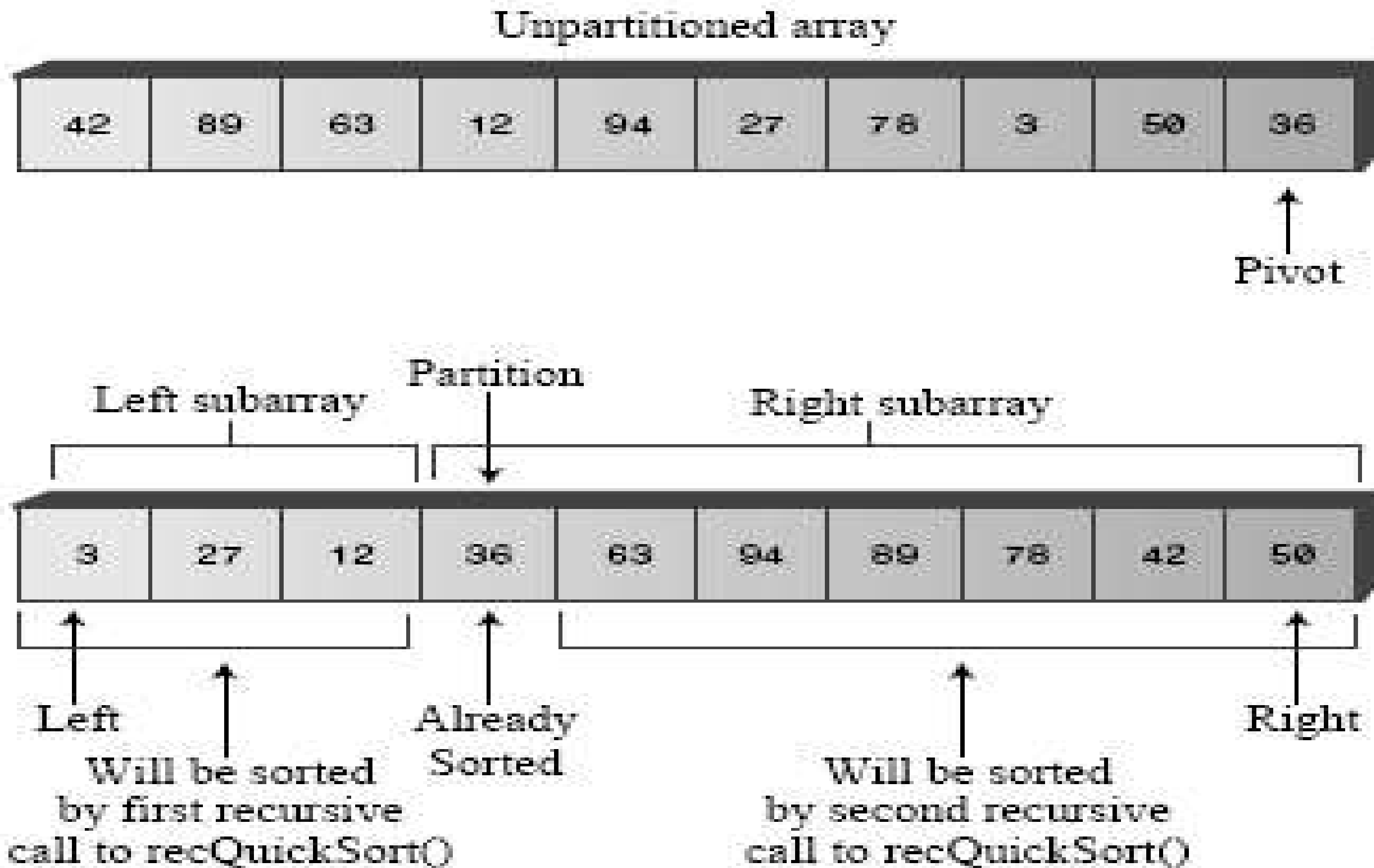




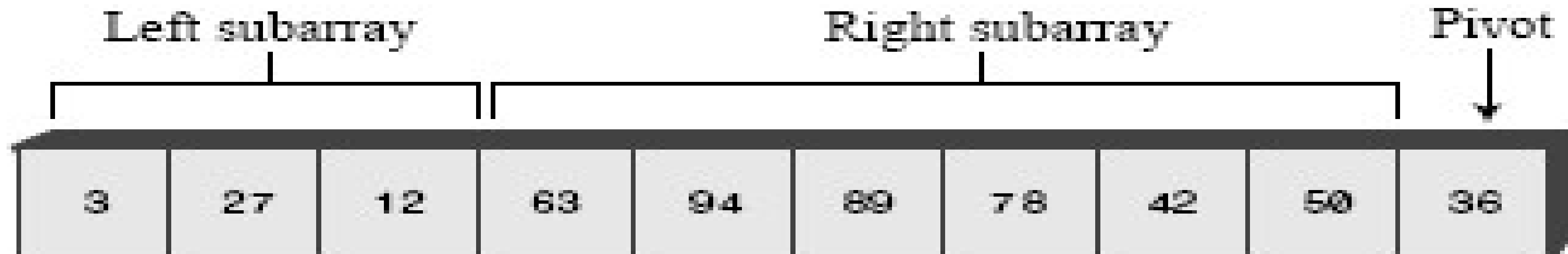
3. SELECTION SORT

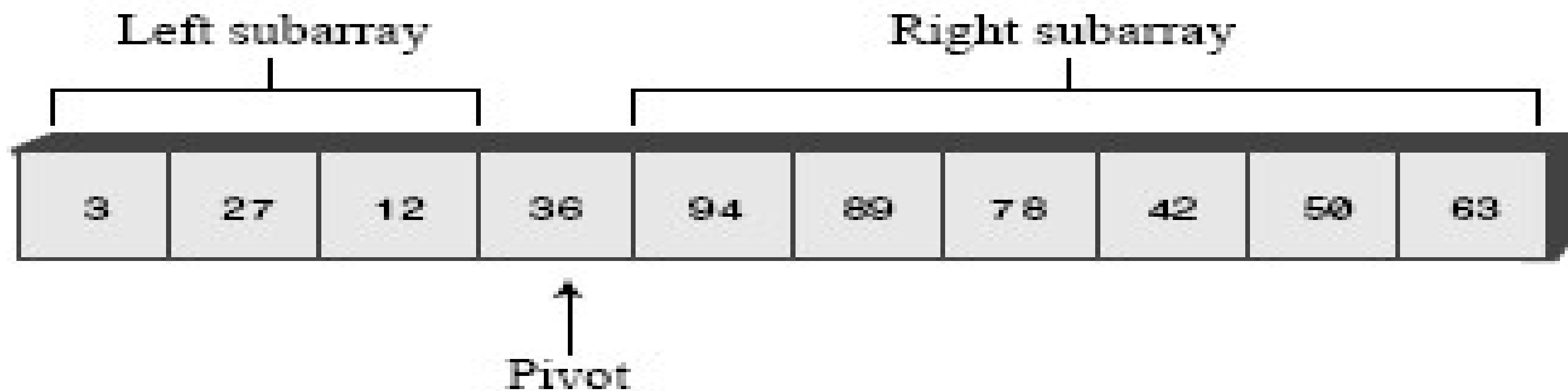
```
void selection_sort(int list[], int n){
        int i, j, min;   for (i = 0; i < n - 1; i++) {
                min = i;
                for (j = i+1; j < n; j++) {
                        if (list[j] < list[min]) {
                                min = j;
                        }
              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

4. QUICK SORT

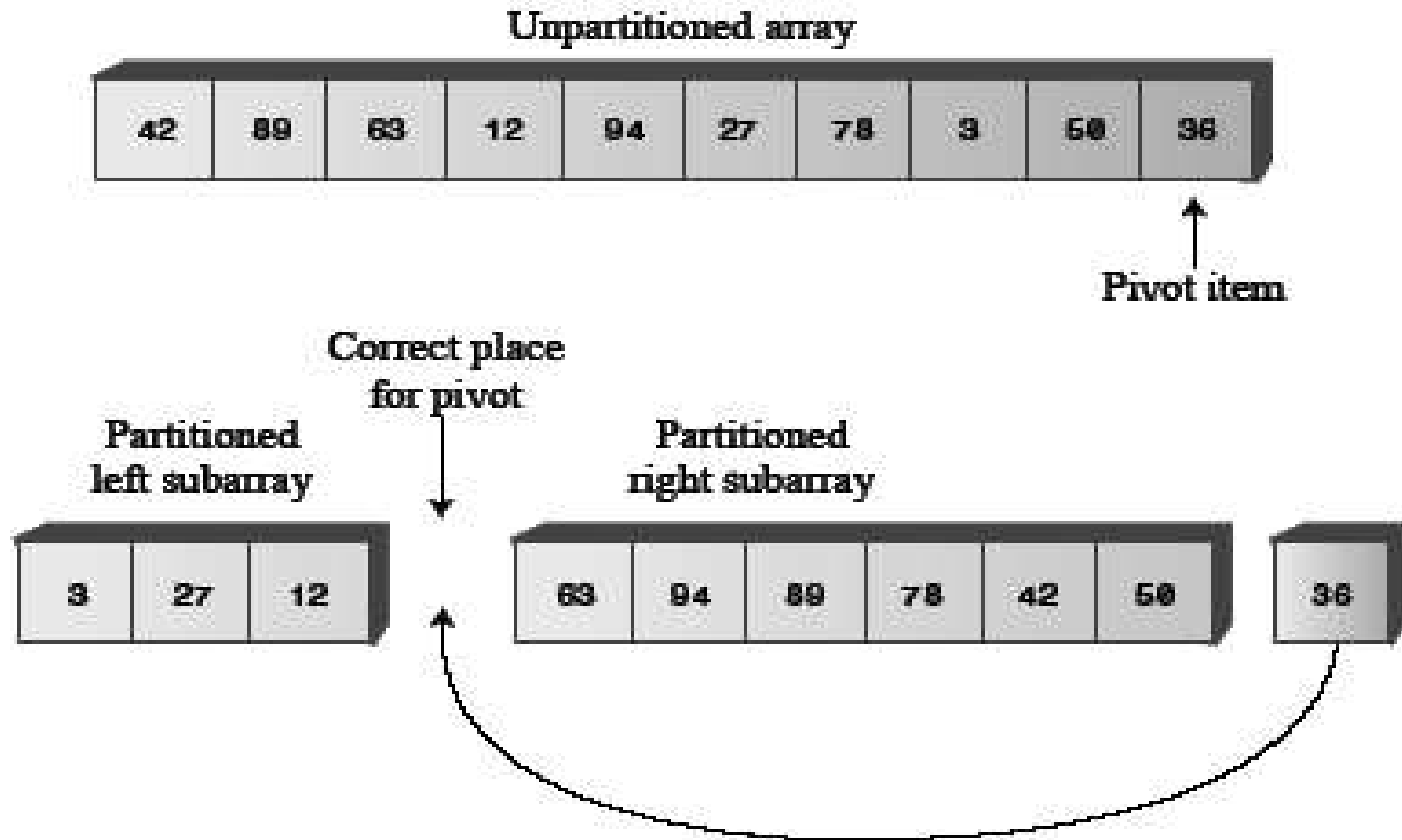


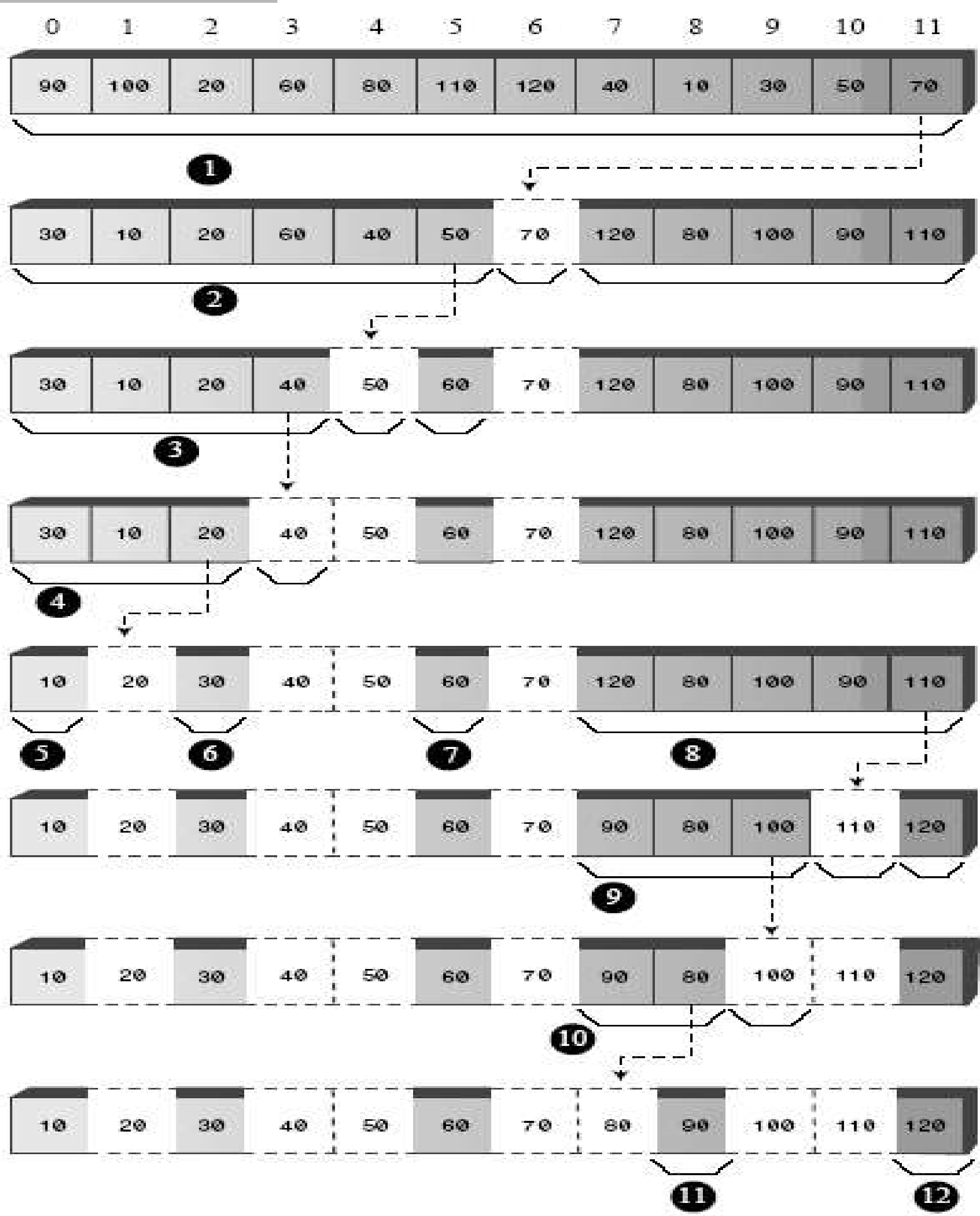
4. QUICK SORT





4. QUICK SORT





0 1 2 3 4 5 6 7 8 9 10 11
90 100 20 60 80 110 120 40 10 30 50 70
1
30 10 20 60 40 50 70 120 80 100 90 110
2
30 10 20 40 50 60 70 120 80 100 90 110
3
30 10 20 40 50 60 70 120 80 100 90 110
4
10 20 30 40 50 60 70 120 80 100 90 110
5
6
7
8
10 20 30 40 50 60 70 90 80 100 110 120
9
10 20 30 40 50 60 70 90 80 100 110 120
10
10 20 30 40 50 60 70 80 90 100 110 120
11
12

4. QUICK SORT

Chọn ngẫu nhiên một phần tử X của dãy (chẳng hạn phần tử đầu tiên) và cố gắng phân chia dãy này thành 3 dãy con liên tiếp nhau:

+ Dãy 1: Gồm những phần tử nhỏ hơn X.
+ Dãy 2: Gồm những phần tử bằng X.
+ Dãy 3: Gồm những phần tử lớn hơn X.

Sau đó áp dụng lại giải thuật này cho dãy con thứ nhất và dãy con thứ ba (dãy con này có số phần tử lớn hơn 1).

4. QUICK SORT

Trước khi xem xét chi tiết của các giải thuật, chúng ta sẽ thực hiện việc sắp xếp một danh sách cụ thể có 7 số như sau:

26 33 35 29 19 12 22

Chúng ta sử dụng ví dụ này cho `Quick_sort`.

Để sử dụng `Quick_sort`, trước tiên chúng ta phải xác định phần tử trụ. Phần tử này có thể là phần tử bất kỳ nào của danh sách, tuy nhiên, để cho thống nhất chúng ta sẽ chọn phần tử đầu tiên. Trong các ứng dụng thực tế thường người ta có những cách xác định phần tử trụ khác tốt hơn.

Theo ví dụ này, phần tử trụ đầu tiên là 26. Do đó hai danh sách con được tạo ra là:

$$19\ 12\ \ 22\ \ \text{và}\ \ 33\ 35\ 29$$

Hai danh sách này lần lượt chứa các số nhỏ hơn và lớn hơn phần tử trụ. Ở đây thứ tự của các phần tử trong hai danh sách con không đổi so với danh sách ban đầu nhưng đây không phải là điều bắt buộc.

Chúng ta tiếp tục sắp xếp các chuỗi con. Với chuỗi con thứ nhất, chúng ta chọn phần tử trụ là 19, do đó được hai danh sách con là 12 và 22. Hai danh sách này

chỉ có một phần tử nên đương nhiên có thứ tự. Cuối cùng, gom hai danh sách con và phần tử trụ lại ta có danh sách đã sắp xếp

12 19 22

Áp dụng phương pháp trên cho phần thứ hai của danh sách, ta được danh sách cuối cùng là

29 33 35

Gom hai danh sách con đã sắp xếp này và phần tử trụ đầu tiên ta được danh sách có thứ tự sau cùng:

12 19 22 26 29 33 35

Các bước của giải thuật được minh hoạ bởi hình sau.

Sort (26, 33, 35, 29, 12, 22)

Partition into (19, 12, 22) and (33, 35, 29); pivot = 26
Sort (19, 12, 22)

Partition into (12) and (22); pivot = 19
Sort (12)
Sort (22)
Combine into (12, 19, 22)

Sort (33, 35, 29)

Partition into (29) and (35); pivot = 33
Sort (29)
Sort (35)
Combine into (29, 33, 35)

Combine into (12, 19, 22, 26, 29, 33 35)

Hình 8.9- Các bước thực thi của `Quick_sort`

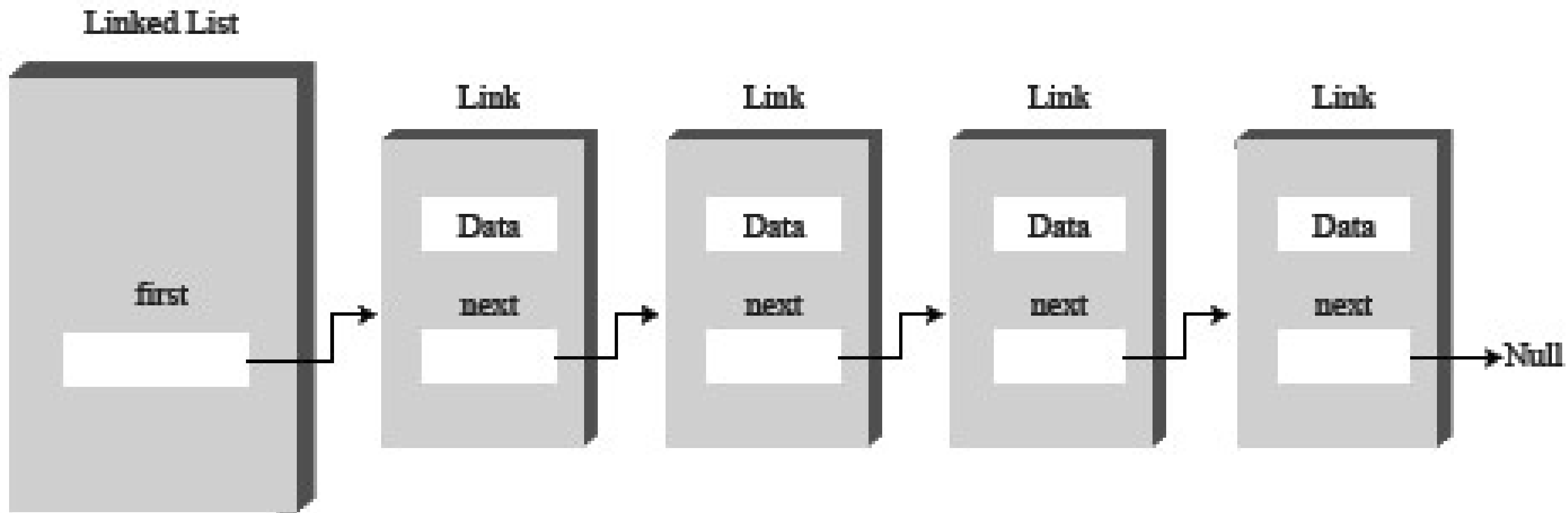
Week 6

CHAPTER 6: LINKED LISTS

Introduction

When dealing with many problems we need a dynamic list, dynamic in the sense that the size requirement need not be known at compile time. Thus, the list may grow or shrink during runtime. A *linked list* is a data structure that is used to model such a dynamic list of data items, so the study of the linked lists as one of the data structures is important.

Concept



Data Structures

```
struct node
{
     int data;
    struct node *link;
};
```

Data Structures

```
struct node *insert(struct node *p, int n)  {
        struct node *temp;
        if(p==NULL){
                 p=new node;
                 if(p==NULL){
                                   printf("Error\n");
                                   exit(0);
                 }
                 p-> data = n;
                 p-> link = p;
        }
        else{
        temp = p;
                 while (temp-> link != p)
                          temp = temp-> link;
                 temp-> link = new node;
                 if(temp -> link == NULL){
                                  printf("Error\n");
                                  exit(0);
                 }
     temp = temp-> link;
     temp-> data = n;
     temp-> link = p;
      }
return (p);
```

```
struct node *insert(struct node *p, int n)  {
        struct node *temp;
        if(p==NULL){
                p=new node;
                if(p==NULL){
                        printf("Error\n");
                        exit(0);
                }
                p-> data = n;
                p-> link = p;
        }
        else{
        temp = p;
                while (temp-> link != p)
                    temp = temp-> link;
                temp-> link = new node;
                if(temp -> link == NULL){
                        printf("Error\n");
                        exit(0);
                }
    temp = temp-> link;
    temp-> data = n;
    temp-> link = p;
    }
return (p);
}
```

INSERTING A NODE BY USING RECURSIVE PROGRAMS

```
struct node *insert(struct node *p, int n){
      struct node *temp;
      if(p==NULL){
         p=(struct node *)malloc(sizeof(struct
node));
         if(p==NULL){
                printf("Error\n");
                exit(0);
           }
      p-> data = n;
      p-> link = NULL;
      }
      else
         p->link = insert(p->link,n);
      return (p);
 }
```