PH502: Scientific Programming Concepts

Irish Centre for High End Computing (ICHEC)

March 1, 2018

Overview

ICHEC
Irish Centre for High-End Computing

- Recap
- Introduction to Computer Algorithms
- The Standard Library: Input/Output and Much More

ICHEC
Irish Centre for High-End Computing

RECAP

Basic Program Layout

```
Main

   declare variables

   initialise variables or load initial state

   command to modify store /* O_1 */

   command to modify store /* O_2 */

   ...

   command to modify store /* O_n */

   save final state to disk as required
end
```

Variable Declaration

- Declaration of Variables:

```c
// C Code
data_type var_name;
```

```fortran
! Fortran Code
data_type :: var_name
```

- Variable Types: integer, real, character, logical.
- Examples:

```c
int i,j,k;
float x,y,z;
double xx;
char abc, ABC;
char s[5];
```

```fortran
integer (kind=4) :: i,j,k
real (kind=4) :: x,y,z
real (kind=8) :: xx
character (len=1) :: abc
character (len=5) :: s
logical (kind=4) :: truefalse
```

- Changing one variable into any other variable type, In C: (int)x, (double)x or (char)x. In Fortran: int(x), real(x) and dble(x).

C Operators

Ascending Precedence Order

Operator	Description
()	brackets
!	logical not
++, −−	add/remove 1
*, /	multiply/divide two numbers
%	integer remainder
+, -	add/subtract two numbers
<,<=	less than/or equal
>, >=	greater than/or equal
==, !=	equal to/not equal
&&, ||	logical And/Or
=	assignment

Fortran Operators

Ascending Precedence Order

Operator	Description
()	brackets
**	to the power of
*, /	multiply/divide two numbers
+, -	add/subtract two numbers
.LT., .LE.	less than/or equal
.GT., .GE.	greater than/or equal
.EQ., .NE.	equal to/not equal
.NOT.	logical Not
.AND.	logical And
.OR.	logical Or
=	assignment

Intrinsic Functions

Treat the functions within the maths library in C.

Function	Description
abs(x)	absolute value (FORTRAN)
fabs(x)	absolute value for (C)
cos(x), acos(x)	cosine and arccosine
exp(x)	e^x
log(x), log10(x)	natural, base 10 log
sin(x), asin(x)	sine and arcsine
sqrt(x)	square root
tan(x), atan(x)	tan and arctan

Loops

- **for** loops are designed to execute the same set of instructions a fixed number of times.

```
for(init; condition; update)
{
  code to be executed;
}
```

```
do init, condition, update
  code to be executed
end do
```

- **while** loops repeat until a condition is meet. The danger here is that the loop never terminates.

```
while(condition)
{
  code to be executed;
}
```

```
do while(condition)
  code to be executed
end do
```

Conditional Statements

- **if-else**: allows us to check whether a statement is true or not.

```
if (statement) {
   commands to be executed
}
else {
   commands to be executed
}
```

```
if (statement) then
  commands to be executed
else
  commands to be executed
endif
```

- **switch-case**: permits us to execute different statements based on the different values of a parameter.

```
switch(expression){
    case constant1:
       statement(s);
       break;
    case constant2:
       statement(s);
       break;
    default :
       statement(s);
}
```

```
select case (expression)
  case (constant1)
       statement(s)
  case (constant2)
       statement(s)
  case default
       statement(s)
end select
```

Arrays

ICHEC
Irish Centre for High-End Computing

One dimensional Arrays:

- Declaration:

```
data_type array_name[array_length];
```

```
data_type :: array_name(array_length)
```

- Elements of the array can be accessed using an index.
  1. In C: the elements are indexed from $0 \rightarrow n-1$
  2. In Fortran: the elements are indexed from $1 \rightarrow n$
- Matrix: Two dimensional array. Elements are accessed using nested loops.

Functions in C

- Function Declaration:

```
return_type function_name(argument-list);
```

- Definition:

```
return_type function_name(argument-list)
{
   body of the function
}
```

- Calling the function:

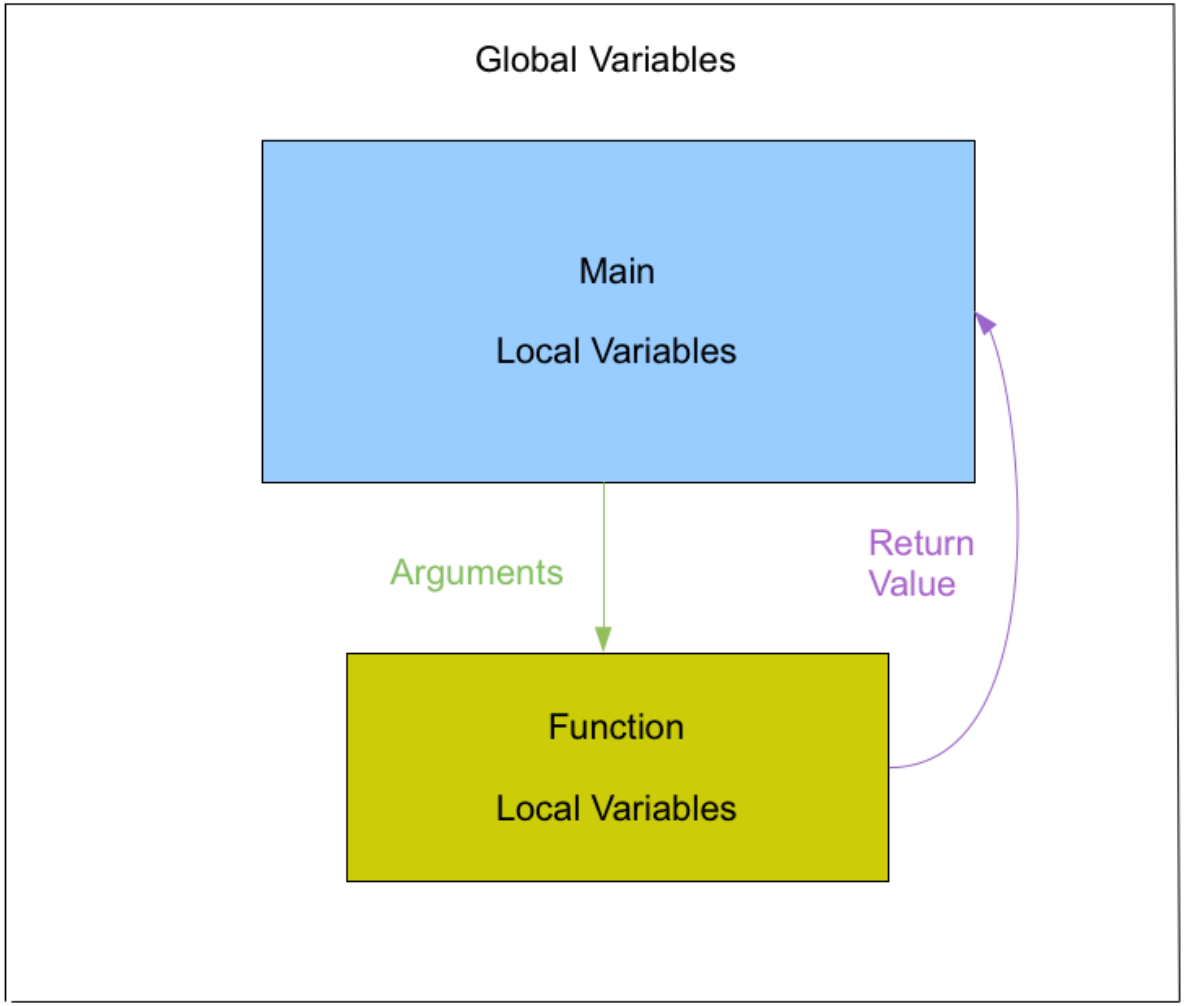
```
int main(void)
{
   //Declarations

   output=function_name(input);

   return 0;
}
```

Functions/Subroutines in Fortran

ICHEC
Irish Centre for High-End Computing

```
program program_name
    use mymods
    !Declarations

    output=function_name(input)
    call subroutine_name(input1, input2)
end program program_name
module mymods
    contains
    return_type function function_name(argument-list)
        !body of the function
    end function function_name

    Subroutine subroutine_name(argument-list)
        !body of the subroutine
    End Subroutine subroutine_name
end module mymods
```

Schematic of Scope Rules



Global Variables-Example

```c
// C Code
#define nrows 5
#define ncols 4
double a[nrows][ncols];

int main(){
   //...
}
```

```fortran
! Fortran Code
module module_name
   integer (kind=4), parameter :: &
                        nrows=5, ncols=4
end module module_name

program program_name
   use module_name
   real (kind=8) :: a(nrows, ncols)
   !...
end program program_name
```

Pointers - Example

- Declaration:

```
int var=1;
int *ptr;
```

- Assign the address of a variable to a pointer:

```
ptr=&var;

printf("Address of var: %x\n", &var);
printf("Address stored in ptr: %x\n", ptr);
```

- Dereferencing:

```
printf("The value: %d\n", *ptr);
```

- Pointer to Pointer:

```
int **ptr2ptr;
ptr2ptr=&ptr;

printf("Address stored in ptrptr: %x\n", ptr2ptr);
printf("The value: %d\n", **ptr2ptr);
```

Arrays and Marices - Dynamic Allocation

ICHEC
Irish Centre for High-End Computing

```
int n=10, nrows=5, ncols=4;
int *a=(int *)malloc(n*sizeof(int));
int **M=(int **)malloc(nrows*sizeof(int *));
  for(int i=0; i<nrows; i++){
    M[i]=(int *)malloc(ncols*sizeof(int));
  }
int *N=(int *)malloc(nrows*ncols*sizeof(int));

//Vector and Matrix Operations
//Access the elements of M by M[i][j]
//Access the elements of N by N[i*ncols+j]

free(a);
for(i=0;i<nrows;i++){
  free(M[i]);
}
free(M);
free(N);
```

Arrays and Marices - Dynamic Allocation

ICHEC
Irish Centre for High-End Computing

```
integer(kind=4), allocatable :: a(:), M(:,:)
integer(kind=4) :: n=5, nrows=5, ncols=4

allocate(a(n))
allocate(M(nrows,ncols))

!Vector and Matrix Operations

deallocate(a)
deallocate(M)
```

Structs and Types

- User defined data type that allows you to combine data items of different types.
- Definition:

```c
struct PMclass {
  float matrix[4][4]
  int matdim;
  float determinant;
}
```

```
type PMclass
  real (kind=4) :: matrix(4,4)
  integer (kind=4) :: matdim
  real (kind=4) :: determinant
end type
```

- Accessing:

```c
struct PMclass Mat1;

Mat1.matrix[0][1]=3.0;
Mat1.matdim=4;
Mat1.determinant=0.0;
```

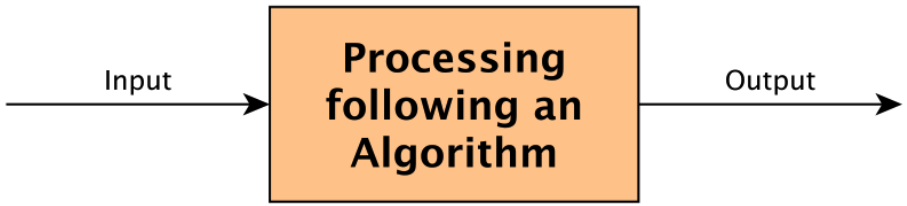
```
type (PMclass) :: Mat1

Mat1%matrix(1,2)=3.0
Mat1%matdim=4
Mat1%determinant=0.0
```

ICHEC
Irish Centre for High-End Computing

Introduction to Computer Algorithms

What is an Algorithm?

- A computational problem is a specification of input-output relation.



- A set of well-defined instructions for performing a computation to solve problems.
- The description must be precise so that it is clear how to perform the computation.
- The instructions may contain mathematical or logical operations, repetition, procession to another instruction, or a set of another instructions.

Examples

- Google's Page Rank Algorithm
- Traveling Sales Man Problem
- Linear Assigment Problems
- Stable Marriage Problem
- The Game of Life
- Sudoku Solving Algorithms

Simple White Cake

Ingredients:

- 1 cup white sugar
- 1/2 cup butter
- 2 eggs
- 2 teaspoons vanilla extract
- 1 1/2 cups all-purpose flour
- 1 3/4 teaspoons baking powder
- 1/2 cup milk

Directions:

1. Preheat oven to 175 degrees C. Grease and flour a 9x9 inch pan.
2. In a medium bowl, cream together the sugar and butter. Beat in the eggs, one at a time. Then stir in the vanilla. Combine flour and baking powder. Add to the creamed mixture and mix well. Finally stir in the milk until batter is smooth.
3. Pour or spoon batter into the prepared pan.
4. Bake for 30 to 40 minutes in the preheated oven.
5. Cake is done when it springs back to the touch.

Human vs Computer Algorithms

Human Algorithms	**Computer Algorithms**
Written in natural language	Written in programming language
Informal steps	Formal, effective steps
Leave obvious steps implicit	All steps are well defined
Regarded as well-understood	Operations are unambiguous

Euclid's Algorithm

- Euclid's Algorithm describes how to find the greatest common divisor of two numbers.
- Given two positive integers a and b, their greatest common divisor denoted by $gcd(a, b)$ is the largest positive integer that divides them both. Ex: gcd(10, 8)=2.
- One of the oldest numerical algorithms (300 BC) still in use.
- Used in various fields: integer factorisation and cryptography, important for electronic commerce,...
- Algorithm:
  1. Divide a by b and let r be the remainder.
  2. If r=0, then b is the answer. Stop.
  3. Else, Set a=b and b=r and Go back to Step 1.

Expressing Algorithms: Natural Language

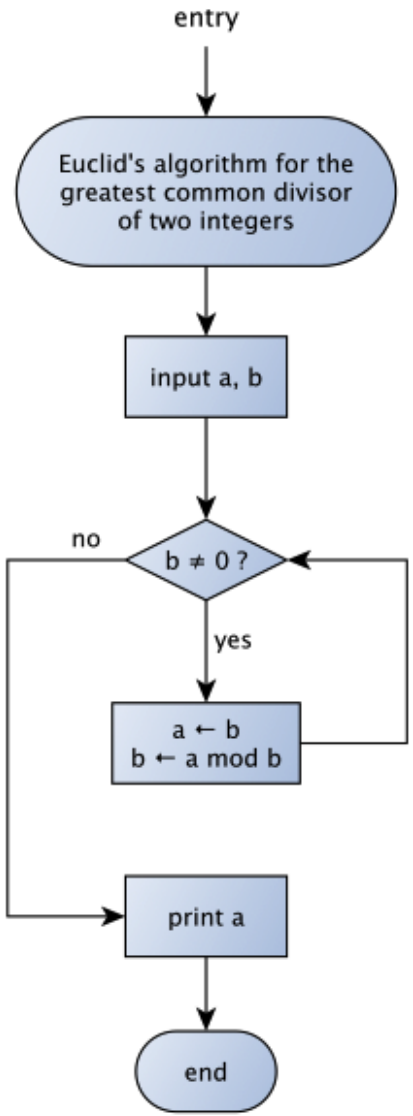
ICHEC
Irish Centre for High-End Computing

Euclid's Algorithm:

Given a and b integers, where a is larger than or equal to b, the procedure takes the remainder of a and b. If it is zero, a and b's GCD is b itself; if not, repeat this procedure for both b and the remainder of a and b.

- Ambiguous and non-structured steps
- Difficult to understand as the steps are overlapped

Expressing Algorithms: Flowchart



- Clear method for small algorithms; becomes confusing for large ones
- Still redundant and visually polluted

Expressing Algorithms: Pseudocode

- Similar to programming languages (but simplified)
- Clear and concise
- Not concerned with issues of software engineering, i.e., Data abstraction, modularity, error handling (ignored for simplicity)

GCD(a, b)
while $b \neq 0$ **do**
  temp=b
  b=a mod b
  a=temp
end while
return a

Expressing Algorithms: Programming Language

- Can be understood and executed by computers
- Representation closer to the machine
- Syntax and semantics not appropriate for easy handling

```
int gcd(int a, int b) {
    while (b != 0) {
      int temp = b;
      int b = a % b;
      int a = temp;
    }
    return a;
}
```

Properties of Algorithms

- Finiteness: Terminates after a finite number of steps. Otherwise it cannot be called as computational method.
- Definiteness: Each step is unambiguous. (Each step must be precisely defined.)
- Effectiveness: Consist of basic instructions that are realisable/computable. Its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone usilng pencil and paper.
- Set of Inputs: Quantities that are given to the algorithm initially before it begins, or dynamically as it runs, i.e., a, b,
- Set of Outputs: Quantities that have a specified relation to the inputs, i.e., gcd(a, b).

Properties of Algorithms

- Adaptability: Can it be modified or evolved over time to meet changes?
- Reusability: Can it be used as a component in different systems or in different application domains?
- Correctness: Does it solve the problem for all possible inputs?
- Efficiency:
  - Space Complexity: How much memory taken? - minimal use of computational resources
  - Time Complexity: How long it takes to solve? - fast runtime

There are many algorithms, each with different features, to solve the same problem. As we shall see later many of the better ones are freely available.

Why study Algorithms?

"Having a solid base of algorithmic knowledge and technique is one characteristic that separates the truly skilled programmers from the novices. With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more."

Introduction to Algorithms (3rd Ed.) MIT Press

Algorithm Analysis

- Theoretical study of computer program performance and resource usage.
  - Memory, disk, communication bandwidth, energy spent, but mostly computational time
- Computational time:
  - Depends on input data (sorted or not sorted)
  - Depends on input size (10 versus 10×10^6)
  - Maximum time given any input of size n (Worst case).
- Which algorithm is the best one to use? - *Depends on the computer*
  - Relative speed (on the same computer)
  - Absolute speed (on different computers)

Asymptotic Analysis

- No matter what the computer is, we compare the growth rate!
- Which algorithm is more efficient?

Algorithm	Input 1	Input 2	Input 3
Algorithm 1	30s	35s	40s
Algorithm 2	1s	4s	30s

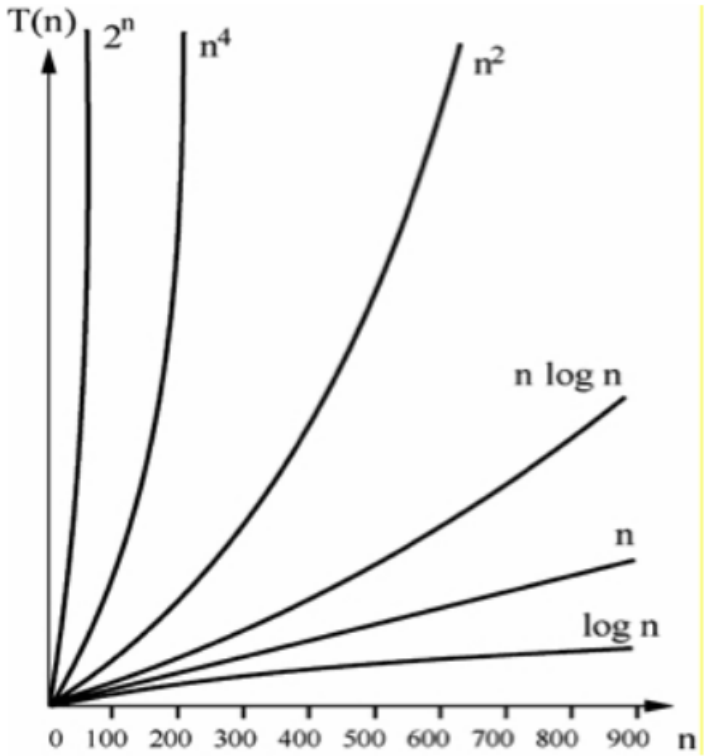
- The efficiency of an algorithm is not tied to the consumption of a resource for a specific input, but to how it responds to changes in input size: Growth Rate.
- How well does the algorithm perform as the input size grows;

$$n \to \infty$$

Time Complexity

When the input size increases, the runtime may...

- Remain constant: $\Theta(1)$
- Increase logarithmically: $\Theta(logn)$
- Increase linearly: $\Theta(n)$
- Increase quadratically: $\Theta(n^2)$
- Increase cubically etc: $\Theta(n^3)$ or $\Theta(n^4)$ or ... (*Polynomial complexities.*)
- Increase exponentially: $\Theta(2^n)$



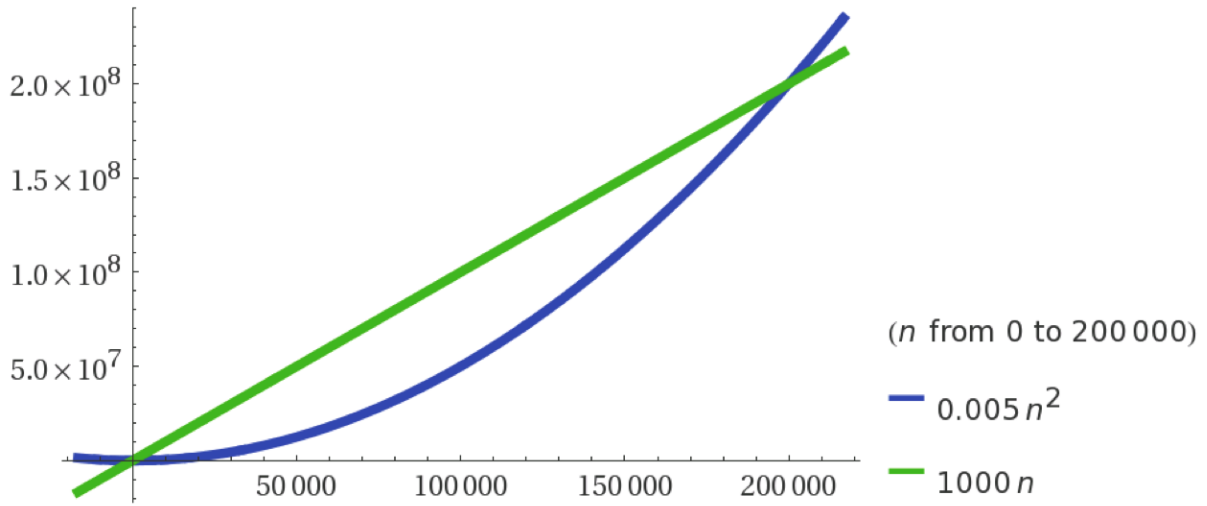
from lecture notes by Georgy Gimel'farb

Big-Theta Notation: how it works?

- Complexity is expressed using Θ notation and is written as $\Theta(t(n))$ where n is the input size.
- $t(n)$: the number of elementary operations performed by the algorithm.
- The part of $t(n)$ that increases the fastest as the value of n increases.
- Drop lower terms, Ignore leading constants.
  - $t(n) = 100n \Rightarrow \Theta(n)$
  - $t(n) = 2n^2 + 55n + 10 \Rightarrow \Theta(n^2)$
  - $t(n) = 10 \Rightarrow \Theta(1)$

Big-Theta Notation: how it works?

- Efficiency relates to the general case for a large enough input set.
  - $t_1(n) = 1000n\,(\Theta(n))$
  - $t_2(n) = 0.005n^2\,(\Theta(n^2))$
- For small input sizes t_2 is faster than t_1, but for values above a certain threshold this is not true.



- $\Theta(n)$ always beats $\Theta(n^2)$.

Big-O, Big-Omega and Big-Theta

Big-O Notation

- Let f and g be positive functions of a single positive integer argument n. Then $f(n)$ is said to be $O(g(n))$ if and only if there are positive integers c and n_0 such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n)$$

- f is (asymptotically) less than or equal to g.
- For example: $t(n) = 4n^2 + 10n + 78$ is $O(n^3)$.
- As $10n^2 > t(n)$ for $n \geq 5$, $t(n)$ is also $O(n^2)$.
- Big-O notation is an upper bound, expressing the worst-case time required to run an algorithm on various inputs.

Big-O, Big-Omega and Big-Theta

Big-Omega Notation

- Let f and g be defined as before. Then $f(n)$ is said to be $\Omega(g(n))$ if and only if there are positive integers c and n_0 such that for every integer $n \geq n_0$,

$$f(n) \geq cg(n)$$

- f is (asymptotically) greater than or equal to g
- For example: $t(n) = 400n + 23$ is $\Omega(1)$
- Big-Omega is a lower bound, expressing the best-case time.

Big-O, Big-Omega and Big-Theta

Big-Theta Notation

- Let f and g be defined as before. Then $f(n)$ is said to be $\Theta(g(n))$ if and only if there are positive integers c_1, c_2 and n_0 such that for every integer $n \geq n_0$,

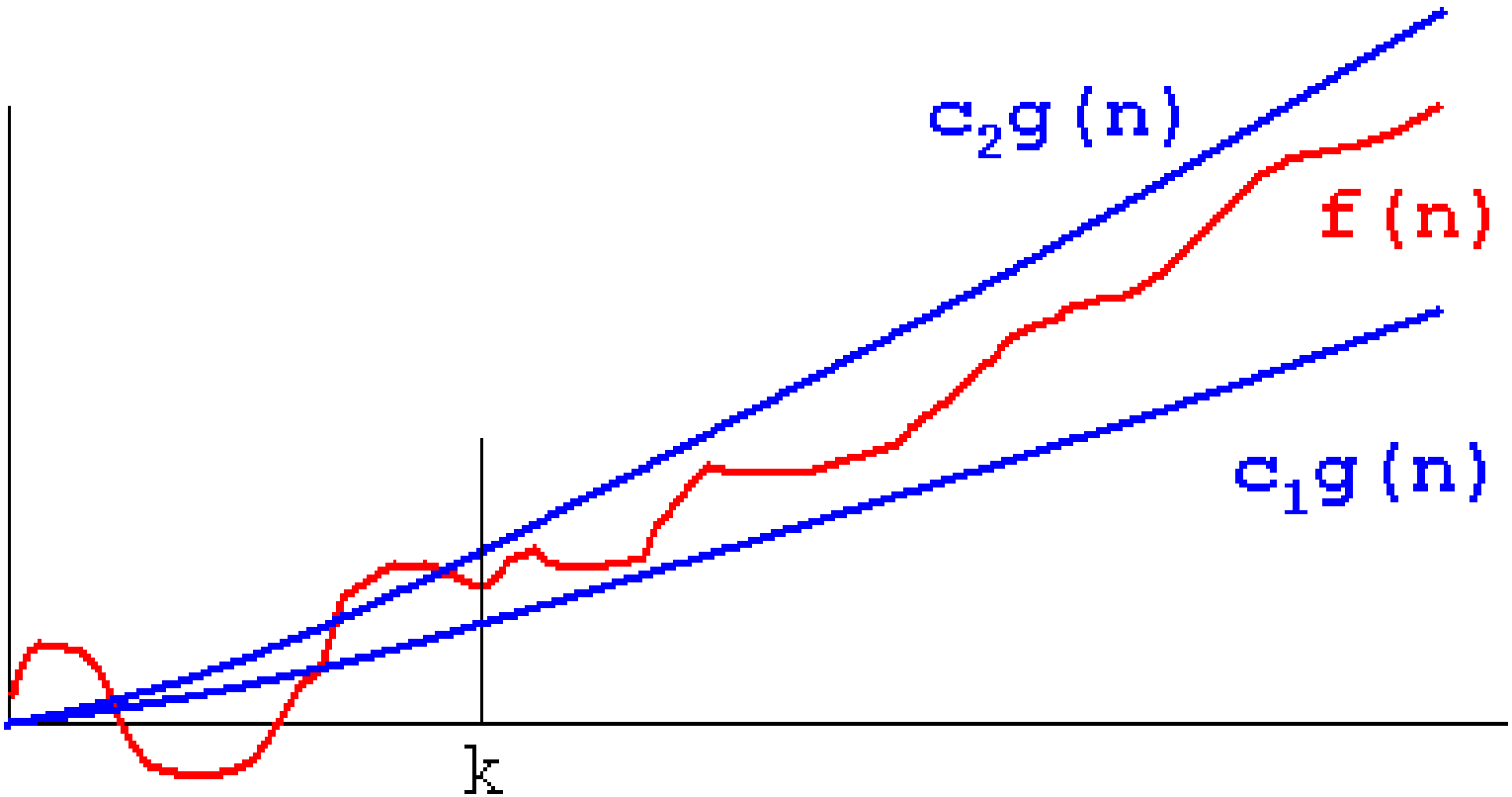
$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

- f is bounded above and below by g.
- Big-Theta combines both upper and lower bounds; gives an asymptotic equivalence.

Big-O, Big-Omega and Big-Theta

Big-Theta Example

- After $n \geq k$ the red curve falls between the blue lines.
- However the curves need to converge assymptotically to define Θ.



Analysing per Case

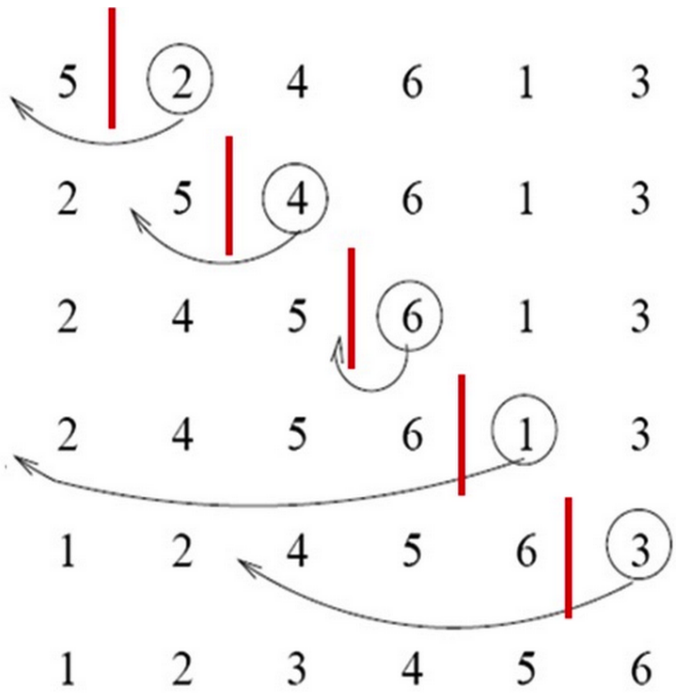
- Very commonly an algorithm exhibits different resource consumption complexity depending on the scenario:
  - Worst-Case scenario: Refers to a case where the algorithm performs especially poorly for a particular data set. The solution is found with difficulty and many steps are executed. The algorithm consumes the most resources.
  - Best-Case scenario: A different data set for the exact same algorithm might have extraordinarily good performance. The solution is easily achieved and the algorithm executes very few of its steps. The algorithm consumes the least resources.
  - Average-Case scenario: The algorithm performs somewhere in between these two extremes. The solution is achieved after an expected number of steps. The algorithm consumes average resources.

Example: Insertion Sort

Input: Sequence of numbers: $A = \{a_1, a_2, ..., a_n\}$

Output: Permutation of this numbers: $a_1\prime, a_2\prime, ..., a_n\prime$ such that $a_1\prime \leq a_2\prime \leq ... \leq a_n\prime$

```
for j ← 2 to n do
    key ← A[j]
    i ← j − 1
    while i > 0 and A[i] > key do
        swap A[i + 1] and A[i]
        i ← i − 1
    end while
    A[i + 1] ← key
end for
```



Analysis of Insertion Sort

- Worst-Case: Reverse Sorted

$$\sum_{j=2}^{n} \Theta(n) = \Theta(n^2)$$

- Best-Case: Already Sorted list

$$\sum_{j=2}^{n} \Theta(1) = \Theta(n)$$

For small n, moderately fast. Not at all for large n. Use *Heap Sort* or *Merge Sort* for larger n.
(`http://en.wikipedia.org/wiki/Sorting_algorithm`)

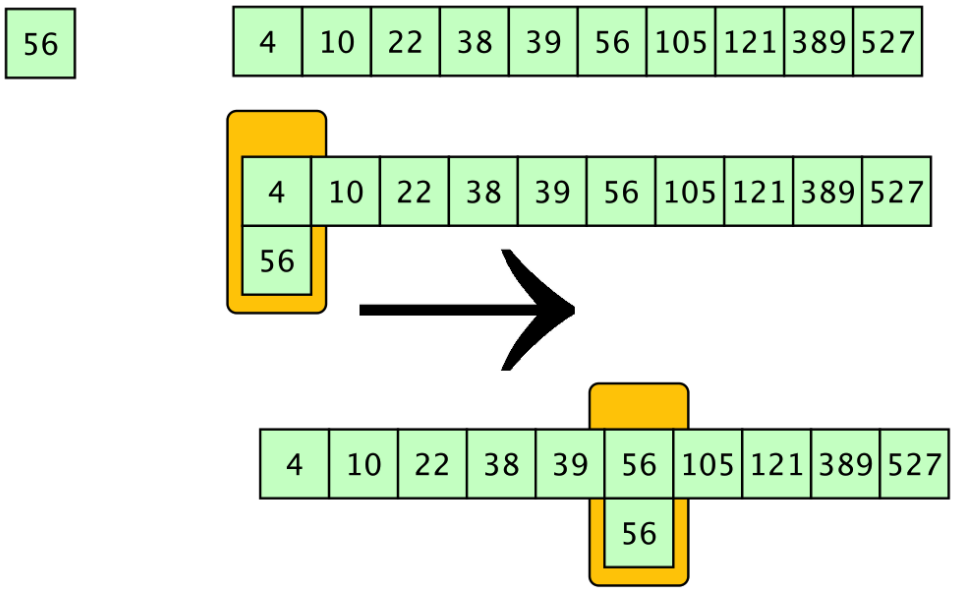
Example: Linear Search

Input: A list of unsorted integers $A = \{a_1, a_2, ..., a_n\}$, A single integer to search for: k

Output: True is the integer is found; False otherwise.

Procedure: Scan the list from beginning to end, comparing the integer to find with every single element.

```
for i ← 1 to n do
    if k=A[i] then
        return TRUE
    end if
end for
return FALSE
```



Analysis of Linear Search

- Worst-Case: Element searched in the last one

$$\Theta(n)$$

- Best-Case: element searched is the first

$$\Theta(1)$$

Binary search is faster (Worst case: $\Theta(logn)$) however it requires the list to be sorted.
(`http://en.wikipedia.org/wiki/Binary_search_algorithm`)

Three Classes

- Three classes of problems:
  1. **P:** solve the problem in polynomial-time. Ex.: Most searching and sorting algorithms
  2. **NP:** solving problem is slower than P but solution is verifiable in polynomial-time. Ex. traveling salesman problem.
  3. **NPC:** (complete) in solving one, of a set of NP problems, the solution is transferrable in P.

Algorithm Development

- Steps in development Algorithms:
  1. Problem Definition
  2. Development of a model
  3. Specification of Algorithm
  4. Designing an Algorithm
  5. Checking the correctness of Algorithm
  6. Analysis of Algorithm
  7. Implementation of Algorithm
  8. Program testing
  9. Document Preparation

Classification of Algorithms

- Iterative vs Recursive
  - Iterative: algorithms which use of loops
  - Recursive: algorithms which invoke itself repeatedly until a condition matches
- Serial vs Parallel vs Distributed
  - Serial: steps are executed in sequence, one at a time
  - Parallel: several steps are executed at the same time by the same processor
  - Distributed: steps are executed by different processors

Classification of Algorithms

- Deterministic vs Nondeterministic
  - Deterministic: every step in the algorithm is predictable
  - Nondeterministic: the steps vary from execution to execution
- Exact vs Approximate
  - Exact: algorithm reaches the solution
  - Approximate: algorithm is not guaranteed to reach the solution, but seeks an approximation to the solution

Example: Factorial

Problem: Find $n!$ for $n \geq 1$.

- The factorial of an integer n is the product of all integers in range $[1, n]$.

$$n! = 1 \times 2 \times 3 \times 4 \times \cdots \times (n-1) \times n$$

Factorial - Iterative Algorithm

- For each integer in range $[1, n]$, aggregate the product and output the final result.

$$n! = \prod_{i=1}^{n} i$$

IterativeFactorial(n)
result=1
for i = 2 to n **do**
    result = result $\times i$
end for
return result

- $\Theta(n)$

Factorial - Recursive Algorithm

- Factorial of n is n times the factorial of $(n-1)$, unless n is 0, for which the result is 1.

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

RecursiveFactorial(n)
if $n = 0$ **then**
  return 1
else
  return $n\times$ RecursiveFactorial($n-1$)
end if

- $\Theta(n)$

Recursice Functions

- C has no distinction between recursive and non-recursive functions.
- In Fortran you must explicitly state that the function is recursive.

```
program rf
  implicit none
  interface
     recursive function f(n) result(answer)
        integer (kind=4) :: n,answer
     end function
  end interface
  integer (kind=4) :: x

  x = f(10)
end program

recursive function f(n) result(answer)
  integer (kind=4) :: n,answer
  if (n .gt. 0) answer = n + f(n-1)
  return
end function
```

Designing Algorithms

- Insertion sort uses an *Incremental approach*: Sort array $A[1, ..., j-1]$, insert $A[j]$, get the sorted array $A[1, ..., j]$.
- Design Techniques:
  - Brute-Force
  - Divide and Conquer
  - Greedy Method
  - Backtracking
  - Dynamic Programming

Brute-Force

ICHEC
Irish Centre for High-End Computing

- Trivial naive method that tries every possible solution and check which is the best one
- Straighforward and esiest approach to apply
- Useful for solving small size problems
- Not always efficient
- Very useful when writing a test routine to check the correctness of more efficient algorithms
- Examples:
  - Iterative Algorithm for Factorial
  - Linear search
  - Selection sort
  - Bubble sort

Divide and Conquer

ICHEC
Irish Centre for High-End Computing

- Break up problem into smaller parts of the same problem until each is small enough to be easily solved.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.
- It is important that the subproblems are independent.
- Examples:
  - Recursive Algorithm for Factorial
  - Binary Search
  - Merge Sort

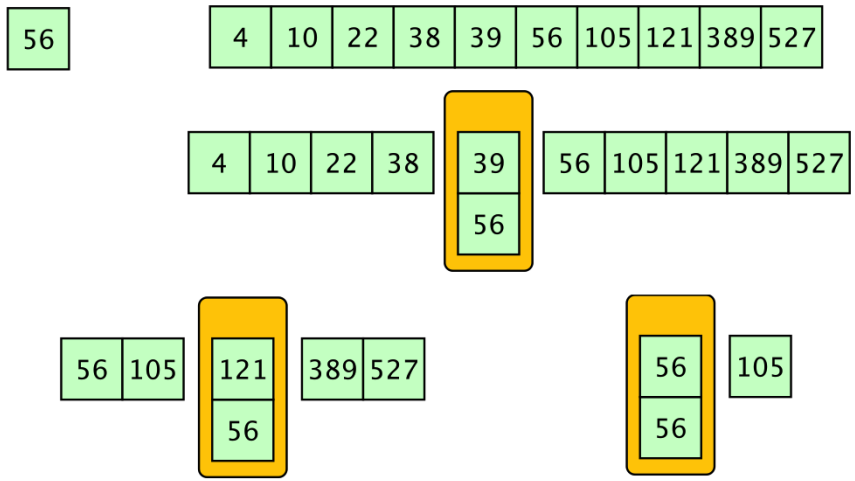
Example: Binary Search

Input: A list of integers sorted in ascending order $A = \{a_1, a_2, ..., a_n\}$, k

Output: True is the integer is found; False otherwise.

Procedure: Select the middle element on the list and compare; if not found, discard half of the list where the element is definitely not placed.

```
begin=1, end=n
while begin ≤ end do
    mid=⌊((begin+end)/2)⌋
    if k=A[mid] then
        return TRUE
    else if k<A[mid] then
        end=mid-1
    else
        end=mid+1
    end if
end while
return FALSE
```



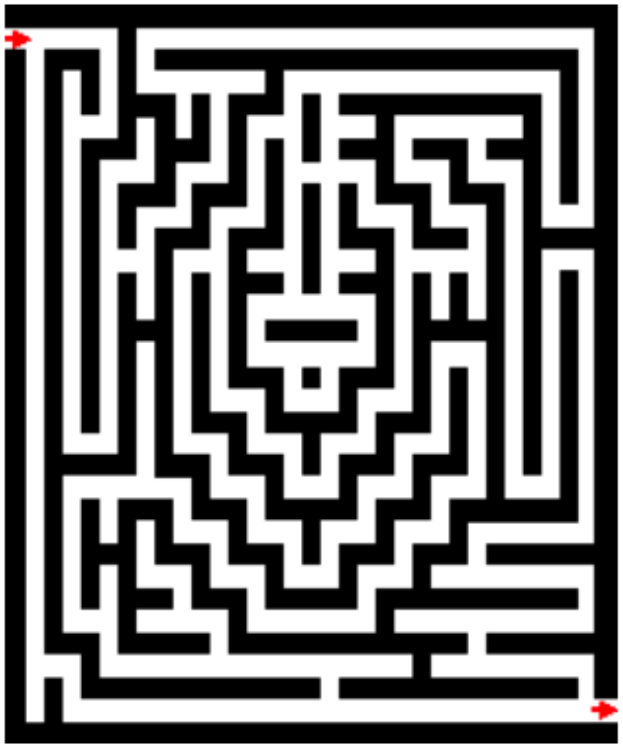
Greedy Method

- The solution is constructed through a sequence of steps. At each step we choose the locally optimal solution.
- Always makes the choice that looks best at the moment and adds it to the current subsolution.
- Simple and straightforward, Easy to invent, easy to implement and most of the time quite efficient
- Many problems cannot be solved correctly by greedy approach
- Mainly used to solve optimization problems
- Examples:
  - Job scheduling problems
  - The Knapsack problem: A thief robbing a store and can carry a maximal weight of w into their knapsack. There are n items and i^{th} item weigh w_i and is worth v_i dollars. What items should thief take?
  - The coin exchange problem: Pay money back to customer using fewest number of coins

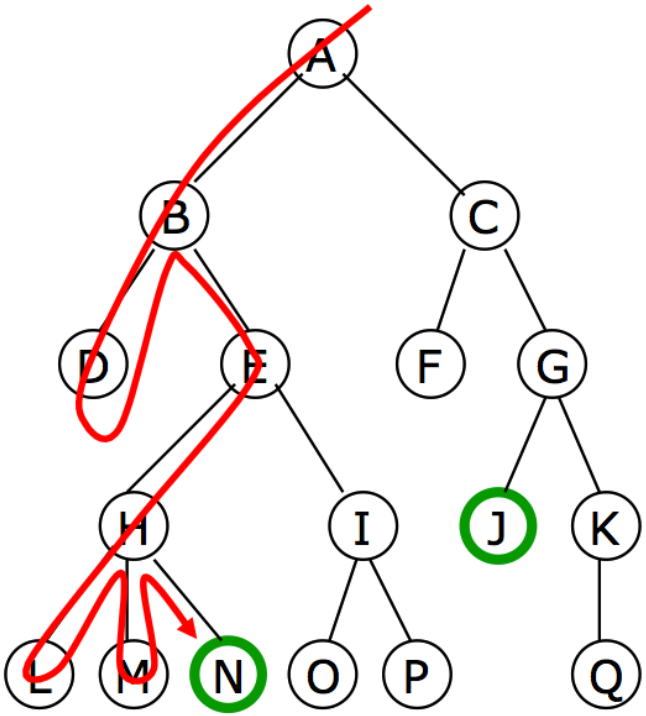
Backtracking

- A choice taken may be wrong, in which case the prospective solution is backtracked to undo the mistake
- A methodical way of trying out various sequences of decisions, until you find one that works
- Example: Maze Path Finder

while not at destination **do**
  Choose a path X
  if You have been in X **then**
    Choose another path Y
    X$\leftarrow$Y
  end if
  Move to X
end while



Example: Depth-First Searching



- It is a combination of brute-force and backtracking.
- It starts at the root and explores nodes from there, looking for a goal node
- It explores a path all the way to a leaf before backtracking and exploring another path
- Node are explored in the order A B D E H L M **N** I O P C F G **J**

Dynamic Programming

- An inverse divide-and-conquer approach
- The smallest sub-problems are firstly solved and their solutions cached;
- The final solution arises from the solutions to each sub-problem
- A bottom-up approach
- Difference from the classical Divide and Conquer is subproblems are solved only once and solutions are stored for reuse.
- Applications: Bioinformatics, Control theory, Information theory, Operations research, ...
- Examples:
  - Viterbi Algorithm for Markov Models
  - Bellman-Ford for finding shortest path in networks
  - The Manhattan Tourist Problem

Example: Fibonacci Sequence

- **Definition:** The first two elements in the sequence are respectively 0 and 1. Every single element is the sum of the previous two elements in the sequence.

$$F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$$

Dynamic Programming:

```
DynFib(n)
Table[0]=0
Table[1]=1
for i=2 to n do
   Table[i]=Table[i-1]+Table[i-2]
end for
return Table[n]
```

Divide and Conquer:

```
DivConFib(n)
if n >= 2 then
   return DivConFib(n-1)+DivConFib(n-2)
end if
return
```

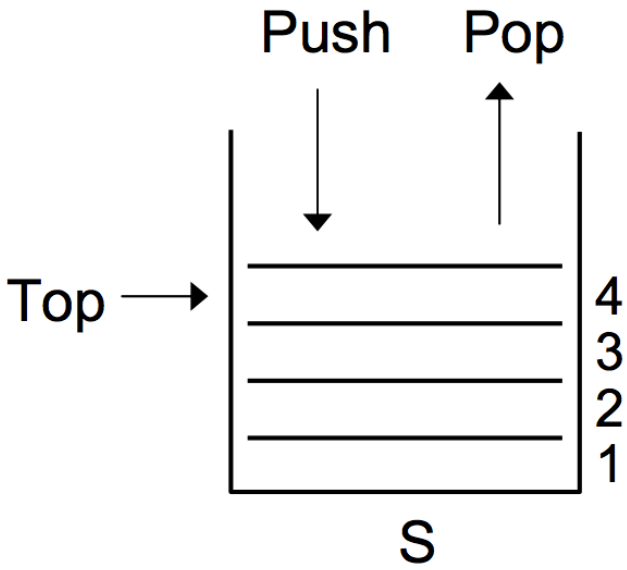
Elementary Data Structures-Stack

Usually, efficient data structures are key to designing efficient algorithms

- Implements the LIFO (last-in, first-out) policy
- S=S[1 ... top(S)]

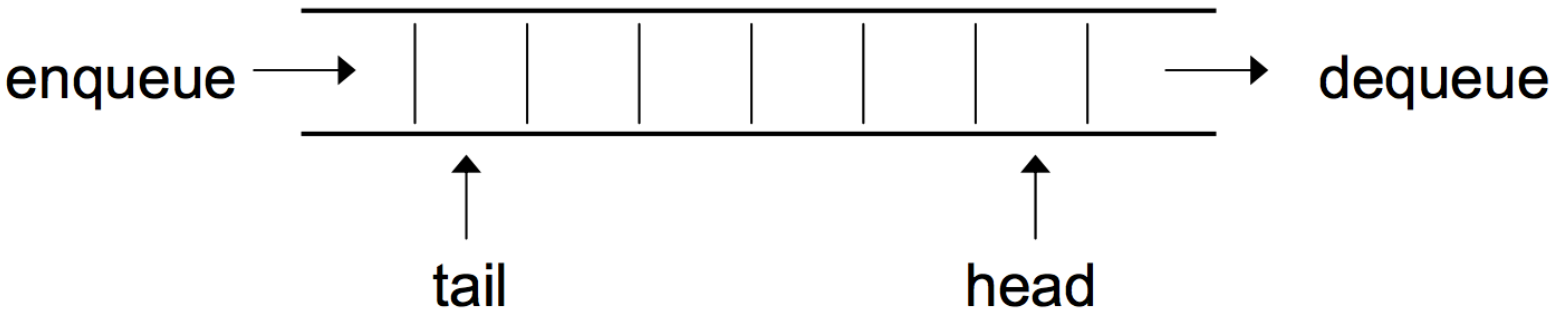
```
Push(S, x)
if Stack_Full(S) then
    Error: Overflow
else
    top[S]=top[S]+1
    S[top[S]]=x
end if
```

```
Pop(S)
if Stack_Empty(S) then
    Error: Underflow
else
    top[S]=top[S]-1
    return S[top[S]+1]
end if
```



Elementary Data Structures-Queues

- Implements the FIFO (first-in, first-out) policy
- Q=Q[head[Q], head[Q]+1, ..., tail[Q]]



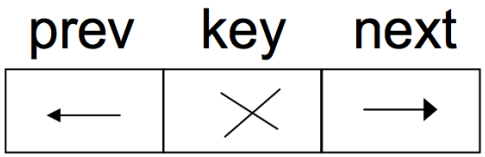

```
Enqueue(Q, x)
if Not Queue_Full(Q) then
   Q[tail[Q]]=x
   if tail[Q]=length[Q] then
      tail[Q]=1
   else
      tail[Q]=tail[Q]+1
   end if
end if
```

```
Dequeue(Q)
if Not Queue_Empty(Q) then
   x=Q[head[Q]]
   if head[Q]=length[Q] then
      head[Q]=1
   else
      head[Q]=head[Q]+1
   end if
   return x
end if
```

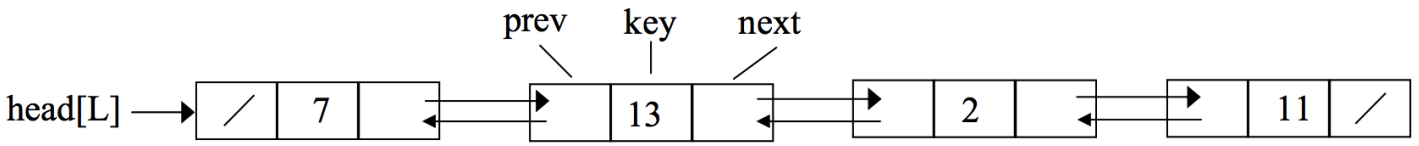
Elementary Data Structures-Linked List

- A concrete data structure consisting of a sequence of nodes. Each node has
  - a pointer to the previous node (except the first one)
  - a pointer to the next one (except the last one)
  - a field that contains a key

One element x :
(consist of 3 fields)

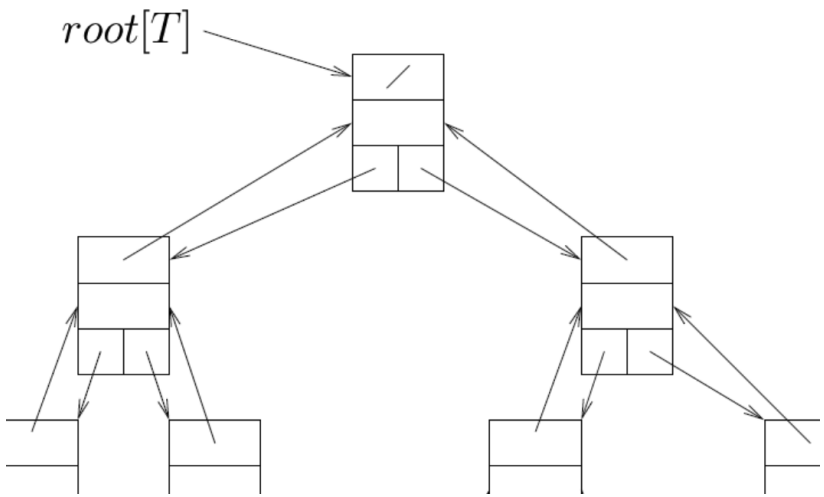


- Singly or Doubly Linked, Sorted or Unsorted, Circular or not
- Insert, Delete and Search Operations



Elementary Data Structures-Trees

- Abstract model of a hierarchical structure.
- A binary tree consists of nodes with
  - A key field
  - Pointer to the parent node
  - Pointer to the left child
  - Pointer to the right child
- Examples:
  - P-nary trees, Morse trees, Heaps, Search Trees, Red-Black Trees
- Insertion, Deletion, Traversal and IsRoot



Some References

1. Introduction To Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, MIT Pressi 3rd Edition, 2009.
2. The art of computer programming, Donald Ervin Knuth, Volume 1-3, 1997-1998.
3. Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms (3rd Edition), Robert Sedgewick, 2001.
4. An Introduction to the Analysis of Algorithms, Robert Sedgewick, Philippe Flajolet, 2nd Edition, 2013.
5. An Introduction to the Analysis of Algorithms, Michael Soltys, World Scientific Publishing Co. Pte. Ltd., 2010.

ICHEC
Irish Centre for High-End Computing

Libraries

The Standard Library

- The Standard Library is a set of functions defined as part of the C ANSI standard.
- They will always be available for you to use.
- We will have a quick overview of some of the most useful bits you may need and also cover the Fortran equivalents.
- By the way, you already used the standard library: think `printf()`, `scanf()`, `tan()`, `malloc()`...

Library and Header Files

- Header Files: These are the files that are included at the top of any program. If we use any function inside a program, then the header file containing declaration or definition of that function ,has to be included. Like printf() is specified in stdio.h.
- Library Files: These are the files which the compiler uses in order to define the functions which have been used in the program and had been declared inside the header file. Like, printf() has its complete definition, like how it will work etc.

- A header file is included during the preprocessing stage. A library file is linked in by the linker.

Header Files

- Each set of functions is prototyped in a *header* file, with the extension `.h`. We will cover the ones in bold:

assert.h	Diagnostics
ctype.h	Character Class Tests
errno.h	Error Codes Reported by (Some) Library Functions
float.h	Implementation-defined Floating-Point Limits
limits.h	Implementation-defined Limits
locale.h	Locale-specific Information
math.h	Mathematical Functions
setjmp.h	Non-local Jumps
signal.h	Signals
stdarg.h	Variable Argument Lists
stddef.h	Definitions of General Use
stdio.h	Input and Output
stdlib.h	Utility functions
string.h	String functions
time.h	Time and Date functions

Using a set of function

- In order to use a function from the standard library, you need to include the header file where it is prototyped.
- This is done through the `#include` statement at the beginning of your code.

```
#include <stdio.h>
#include <stdlib.h>
```

stdio.h : Input/Output

- `stdio.h` gives you access to a wide set of functions allowing the exchange of data between the program and its environment.
- The environment you're familiar with consists of the user's keyboard (for input) and the screen (for output).
- There's much more that that: in particular, you'll want to read/write from/into files.

int printf(const char* format, ...)

- Your basic formatted output function.
- The first argument is a string containing both static content and special markers marking the formatted output of an evaluated expression.
- The next arguments are a number of expressions that will be evaluated and placed within the static text content of the first argument.

```
int i = 4;
float e = 2.7182818;
double pi = 3.14159265358979;
char  c = 'z', s[]="hello world";
printf("i = %d, e = %f, pi = %lf, c = '%c', s = \"%s\"\n",
        i, e, pi, c, s);
```

printf : format codes

d	int	decimal (base ten) number
o	int	octal number
x or X	int	hexadecimal number
ld	long-sized int	decimal number
u	unsigned	decimal number
lu	unsigned long	decimal number
c	char	single character
s	char pointer	string
f	float	number with 7 digits of precision
g	float	number with up to 7 digits of precision
e	float	number with up to 7 digits of precision, sci.not.
lf	double	number with 16 digits of precision
lg	double	number with up to 16 digits of precision
le	double	number with up to 16 digits of precision, sci.not.

Fortran Write

- In FORTRAN output is written to a logical unit. The screen normally has the logical unit number 6.
- A "write" statement is constructed with a set of qualifiers followed by the list of variables. The $r - value$ of each variable is output.
- For formatted output:

```
write(UNIT=n, FMT=format, ADVANCE=s, &
      IOSTAT=int, ERR=label) var1,var2,etc
```

Qualifier	Description
UNIT	logical unit number
FMT	string containing formatting
ADVANCE	'YES' newline (default) or 'NO' no newline
IOSTAT	integer variable, if zero no error
ERR	the program skips to the labeled line, if there is an error

Format String

Descriptor	Description
rIw.m	Integer decimal
rBw.m	binary
rOw.m	octal
rZw.m	hexadecimal
	'r' is the number of times repeated
	'w' width of number (including blanks)
	'm' optional minimum of digits (including leading zeros)
rFw.d	real number
rEw.dEe	real number in scientific notation
	'd' is the number of decimals in mantissa
	'e' number characters for exponent
A	character string
rX	space
/	newline

Example

- Example

```
   i = 10; j = 1000000;
   x = 123.456

  write(6,fmt='(4x,a,i6,i6,/,5x,a,e10.2e3)',err=10) &
     ' Test ints ',i,j,' and real ',x

10 continue
```

- The result is (^ indicates a space);

```
^^^^ Test ints    10******
^^^^^ and real 0.12E+03'
```

- The "******" field is because the integer "j" is too big for the 6 characters provided for it.

int scanf(const char* format, ...)

- Your basic formatted input function. Uses almost the same format as `printf()`.
- First argument describes how the input is going to be parsed. If static content is present, it is expected to be present in the input read.
- The next arguments are pointers to memory blocks of the correct type/size. Typically, to store some input into an already declared variable, one uses the & operator to get its address.

```
int i,j,k;
long l;
float f;
double g;
char z;
scanf("%d", &i); // reads an integer
scanf("(%d,%d)", &j, &k); // reads two integers between
    // brackets with a coma separating them
scanf("%c", &z); // reads a single character
scanf("%ld,%f,%lf", &l, &f, &g); // reads a long,
    // a float, a double, separated by comas
```

- Format codes are the same as shown earlier for `printf()` (almost).

Fortran Read

- Like writing the "read" statement is also associated with a logical unit. You can read and write to the same unit.
- Reading user input is done on unit 5.
- The "read" statement is similar to write, format follows the same notation.

```
read(UNIT=n,FMT=format,ADVANCE=s,IOSTAT=ivar,ERR=label1,&
END=label2) var1,var2,etc
```

FILE* fopen(const char* filename, const char* mode)

- Opens the system file called *filename* and returns a *stream pointer* to it, if successful. Returns NULL otherwise.
- The *mode* parameter defines which permissions are required on the file. The most common you'll need are "r" (read), "w" (write) and "a" (append).
- When done with a file, close it with `fclose()`.

```
FILE *f;
f = fopen("/home/bruno/project/readme.txt", "w");
// [...]
// write some content into the file
// [...]
fclose(f);
```

Read/Writing to File

FORTRAN, OPEN

- Read/writing to the screen and to file are very similar.
- However, first the file must be opened, on a unit other than 5 or 6. The unit number must be a positive integer less than $2^{31} - 1$.
- The open statement has a set of qualifiers, some are the same as the write statement.

```
open(UNIT=n,FILE=name,ACCESS=s1,ACTION=s2,FORM=s3,&
     STATUS=s4,IOSTAT=ivar,ERR=label)
```

Qualifier	Description
FILE	name of file
ACCESS	'sequential' or 'direct'
ACTION	'write', 'read' or 'readwrite'
FORM	'formatted' or 'unformatted'
STATUS	'old', 'new', 'scratch'
	'replace' or 'unknown'

Qualifiers

- In FORTRAN, human or machine readable files can be read or written. We shall concern ourselves with human readable files.
- Thus "ACCESS='sequential'" and "FORM='formatted'".
- "STATUS" relates to the existence of the file. For example "STATUS='old'" means that the file should already exist, if not then an error is generated. Use "unknown" if the status of the file is variable.
- Below is an example of opening and closing a human readable file.

```
   open(unit=1,file='myfile.dat',status='old',&
        form='formatted',access='sequential',err=10)

   read(1,*) a,b,c
10 continue
   close(unit=1,status='keep')
```

int fprintf(FILE* stream, const char* format, ...)

ICHEC
Irish Centre for High-End Computing

- Generic version of `printf()` that writes to any given *stream* instead of standard output.

```
FILE *f;
int value=42;

f = fopen("readme.txt", "w");
fprintf(f, "%d", value);
fclose(f);
```

fscanf(FILE* stream, const char* format, ...)

- Generic version of `scanf()` that reads from any given *stream* instead of standard input.
- Returns the number of items read or the `EOF` constant if end of file has been reached.

```
FILE *f;
int value;

f = fopen("readme.txt", "r");
if (f == NULL){
    printf("!! can't open file\n");
    exit(1);
}
fscanf(f, "%d", &value);
printf("The value in the file is %d.\n", value);

fclose(f);
```

- To get the current position within a file at any time with ftell(), and reset the current position to arbitrary values with fseek().

Example: copying a file

```
int myfilecopy(char *sourcefile, char *destfile){
    char buff[1024];
    FILE *s, *d;
    int endoffile=1;
    s = fopen(sourcefile, "r");
    d = fopen(destfile, "w");
    while ( endoffile != EOF) {
      endoffile = fscanf(s, "%s\n", &buff);
      fflush(s);   /* Flush input stream */
      fprintf(d, "%s\n", buff);
    }
    fclose(s);
    fclose(d);
}
```

Read/Write to File

FORTRAN, READ&WRITE

- In FORTRAN there are no special functions to read and write to file.
- Use the “read” and “write” statements in the same way as you would to screen. The only difference being the unit number. The file must be opened before you access it.
- For human readable files, each line in the file is a separate record. By default after each read or write statement we advance to the next record.

Other useful I/O stuff

C

- sprintf(): Write formatted data to string
- sscanf(): Read formatted data from string
- putchar()/fputc(): Writes a single character to the standard output/file
- getchar()/fgetc(): Reads a single character from the standard input/file
- puts()/fputs(): Writes a full line to the standard output/file
- gets()/fgets(): Reads a full line from the standard input/file
- fflush(): flushes stream buffer but not stdin
- fpurge(): flushes stream buffer, can be used for stdin

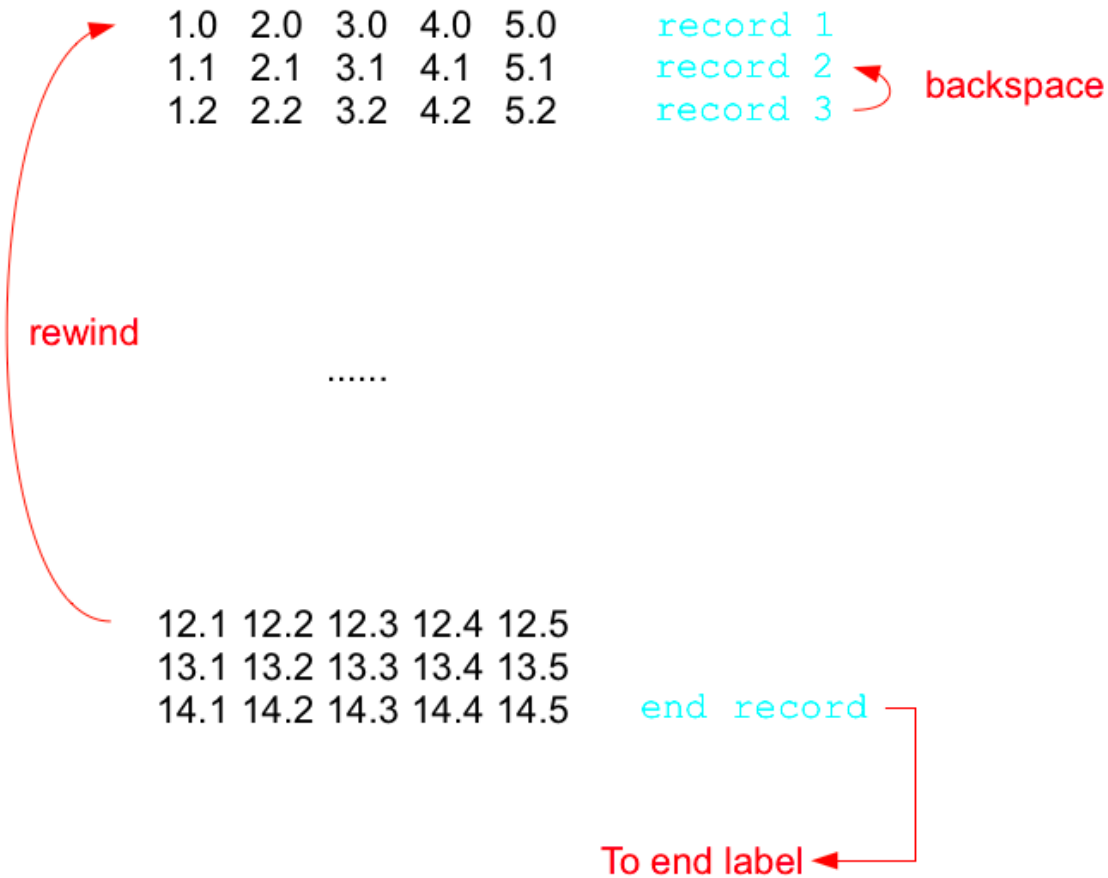
Other useful I/O Stuff

FORTRAN

- Below are some useful operations, the unit number must be given.

Operation	Description
REWIND	goto the beginning of the file
BACKSPACE	go back one line

Diagram of File Structure



stdlib.h : Utility functions

ICHEC
Irish Centre for High-End Computing

- Various useful utility functions: memory management, interaction with operating system, type conversions, random numbers...
- We already covered the main memory management functions so won't do it again: `malloc()`, `realloc()`, `free()`.

stdlib: Exiting a program

- `void exit(int status)` : exits a program normally with `status` as return value (0 or `EXIT_SUCCESS` means successful execution).

```
if (myfinalcount == expectedcount){
    exit(0); // exit successfully
}
else{
    exit(1); // exit with error code 1
}
```

stdlib: OS interaction

- `int system(const char* s)` : executes a system command.

```
system("ls > /tmp/ls.txt");
```

- `char* getenv(const char* name);` : returns the current value of a system environment variable.

```
char *homedir;
homedir = getenv("HOME");
printf("Home directory: %s\n", homedir);
```

stdlib: Type conversions

- `int atoi(const char* s);` : converts string to integer.

```
int i;
i = atoi("42");
```

- `long atol(const char* s);` : converts string to long int.

```
long l;
l = atol("999999999");
```

- `double atof(const char* s)` : converts string to double.

```
double d;
d = atof("2.41286734537455343");
```

stdlib: Random numbers

- `int rand(void)` : returns a pseudo-random number between 0 and `RAND_MAX`(at least 32767).

```
 //Generates a number between 0 and 1
r1=((float) rand() / (RAND_MAX));
//Generates a number between min and max
r2=((float) rand() % (max+1-min))+min;
```

- `void srand(unsigned int seed)` : sets the random number generation seed. Used either to make generation more random (use current time) or reproducible (use a set value).
  - Prevents random numbers from being the same every time the program is executed.

```
int a;

srand(time(0));
a=rand();
```

  - If seed is set to 1, the generator is reinitialized to its initial value.

Random Numbers

FORTRAN

- In FORTRAN there are two subroutines, one sets the seed and the other gets a REAL vector of pseudo random numbers.
- If the seed is the same between different executions of the program the set of random numbers is the same.

```
real (kind=4) :: rvec(1000)
call random_seed
call random_number(rvec)
```

string.h : Manipulating characters

ICHEC
Irish Centre for High-End Computing

- `string.h` provides functions manipulating what C defines as a string: a sequence of characters terminated by `NUL` (`'\0'`).
- It does *not* provide a string variable type.
- It provides: string copy, concatenation, comparisons...

string.h: String copy

- `char* strcpy(char* s, const char* ct)` : copies a string to another including terminating NUL.

```
char s[255];
strcpy(s, "Hello World");
```

- `char* strncpy(char* s, const char* ct, size_t n)` : copies at most *n* characters from a string to another. Warning: may leave *s* with no NUL termination.

```
char s[6];
strncpy(s, "Hello World", 5);
s[5] = '\0'; // required to make s a 'proper' C string
```

string.h: String concatenation

- `char* strcat(char* s, const char* ct)` : Concatenate ct to s and return s. No memory management is made so it's up to you to make sure s has enough memory allocated to store the result.

```
char a[255] = "Hello ";
strcat(a, "world");
printf("%s\n", a);
```

- `char* strncat(char* s, const char* ct, size_t n)` : Concatenate at most n characters of ct to s. NUL-terminates s and return it. Same memory concerns as above.

```
char a[255] = "Hello ";
strncat(a, "world", 3);
printf("%s\n", a);
```

string.h: String comparisons

- `int strcmp(const char* cs, const char* ct)` : Compares cs with ct, returning negative value if $cs < ct$, zero if $cs == ct$, positive value if $cs > ct$.
- C compares the integer equivalents of each character.
- Comparison is made character by character, left from right, dictionary-style.

```
char a[] = "chuck norris";
char b[] = "john muldoon";
int comp = strcmp(a,b);
if (comp<0){
    printf("%s < %s\n", a, b);
} else if (comp>0){
    printf("%s > %s\n", a, b);
} else printf("%s == %s\n", a, b);
// chuck norris < john muldoon as 'c'=99 'j'=106
```

string.h: String comparisons

- Lowercase characters have a higher value than uppercase characters. Therefore uppercase/lowercase will affect the comparison:

```
char a[] = "abcd";
char b[] = "Abcd";
int comp = strcmp(a,b);
if (comp<0){
    printf("%s < %s\n", a, b);
} else if (comp>0){
    printf("%s > %s\n", a, b);
} else printf("%s == %s\n", a, b);
// abcd > Abcd
```

- Try this to see character values:

```
printf("A=%d    a=%d\n", 'A', 'a');
//'A'=65 'a'=97
```

string.h: String comparisons

- `int strncmp(const char* cs, const char* ct, size_t n)` : Compares at most (the first) n characters of cs and ct, returning negative value if $cs < ct$, zero if $cs == ct$, positive value if $cs > ct$.

```
char a[] = "this is sparta";
char b[] = "this is galway";
int comp = strncmp(a,b,8);
if (comp<0){
    printf("%s < %s\n", a, b);
} else if (comp>0){
    printf("%s > %s\n", a, b);
} else printf("%s == %s\n", a, b);
// this is sparta == this is galway
```

string.h: Other useful functions

- `size_t strlen(const char* cs)` : Returns length of cs. Doesn't count the termination character.

```
char s[] = "hello world";
printf("length: %d\n", strlen(s)); // 11
```

- `char* strstr(const char* cs, const char* ct)` : Returns pointer to first occurrence of ct within cs, or NULL if none is found..

```
char text[] = "this is a search for a pattern";
char pattern[] = "is";
char *s = strstr(text,pattern);
if (s == NULL){
    printf ("!! pattern not found in string.\n");
} else{
    printf("Substring: %s\n", s);
}
```

math.h : Mathematics

- Contains a small number of useful basic mathematical functions.
- Unique among the rest of the standard library, to use functions from `math.h`, in addition to including the header in your code, you also need to tell the linker to link the math library with *-lm*

```
gcc -o myexec -lm mycode.c
```

math.h : Basic functions

double exp(double x)	exponential of x
double log(double x)	natural logarithm of x
double log10(double x)	base-10 logarithm of x
double pow(double x, double y)	x raised to power y
double sqrt(double x)	square root of x
double ceil(double x)	smallest integer not less than x
double floor(double x)	largest integer not greater than x
double fabs(double x)	absolute value of x

math.h : Basic trigonometry

double sin(double x)	sine of x
double cos(double x)	cosine of x
double tan(double x)	tangent of x
double asin(double x)	arc-sine of x
double acos(double x)	arc-cosine of x
double atan(double x)	arc-tangent of x
double atan2(double y, double x)	arc-tangent of y/x
double sinh(double x)	hyperbolic sine of x
double cosh(double x)	hyperbolic cosine of x
double tanh(double x)	hyperbolic tangent of x

math.h : example

```c
double x;
printf("enter a number: ");
scanf( "%lf", &x); //Enter 1.2345
printf("original value: %lf\n", x); //1.234500
printf("ceil: %lf\n", ceil(x)); //2.000000
printf("floor: %lf\n", floor(x)); //1.000000
if( x >= 0 )
    printf("Square root: %lf\n", sqrt(x) ); //1.111081
printf("sin: %lf\n", sin(x)); //0.943983
printf("cos: %lf\n", cos(x)); //0.329993
```

time.h : Manipulating date and time

ICHEC
Irish Centre for High-End Computing

- C represents time in two ways:
  - the number of seconds elapsed since midnight on January 1, 1970. This is stored as a `time_t` which is itself defined as a long integer.
  - the `struct tm` structure breaking down a point in time into its components: year, month, day...
- Here is the definition of the `struct tm` type:

```
struct tm {
    int tm_sec;   /* seconds after the minute - [0,59] */
    int tm_min;   /* minutes after the hour - [0,59]   */
    int tm_hour;  /* hours since midnight - [0,23]     */
    int tm_mday;  /* day of the month - [1,31]         */
    int tm_mon;   /* months since January - [0,11]     */
    int tm_year;  /* years since 1900                  */
    int tm_wday;  /* days since Sunday - [0,6]         */
    int tm_yday;  /* days since January 1 - [0,365]    */
    int tm_isdst; /* daylight savings time flag        */
};
```

time.h: getting current time and displaying

- `time_t time(time_t *timeptr)` returns the current time, and also stores it at the address provided.

```
time_t t;
t = time(NULL); // fine
t = time(0);    // 0 == NULL, so works too
time(&t);       // valid too
printf("Seconds since 01/01/1970: %ld\n", t);
```

- `char *ctime(time_t *ptr)` converts to a human-readable string representation.

```
time_t t;
t = time(0);
printf("%s\n", ctime(&t));
// Sample output:
// Sun Feb 16 15:45:22 2014
```

time.h: converting time formats

- `struct tm *localtime(const time_t* tp)` converts the time in seconds to the time structure.
- `char *asctime(const struct tm* tp)` converts a time structure to human-readable string representation.

```
time_t t;
struct tm *mytime;

t = time(0);
mytime = localtime( &t );
printf("%s\n", asctime(mytime));
// Sun Feb 16 15:45:22 2014
```

time.h: time differences

- `double difftime(time_t time2, time_t time1)` : returns the difference in seconds between *time*2 and *time*1.
- `clock_t clock(void)` : returns the time in internal clock units since the current program began its execution. To convert to seconds divide by the constant `CLOCKS_PER_SEC`.

```
int i;
time_t tst, tend;
clock_t cst, cend;

tst = time(0);
cst = clock();
for(i=0; i<=99999999; i++){
    sqrt(i);
}
tend = time(0);
cend = clock();
printf("Elapsed: %lf\n", difftime(tend, tst));
printf("Elapsed: %ld\n", cend-cst);
```

Time Functions in Fortran

- There are two subroutines in FORTRAN for time.
  1. The first returns the current date, all arguments are optional.

```
character (len=10) :: date,time,zone
integer (kind=4) :: values(8)
call date_and_time(date,time,zone,values)
```

  2. The other subroutine gives number of counts, number of counts per second, maximum number of counts.

```
integer (kind=4) :: cnt1,cnt2,rcnt,maxcnt

call system_clock(cnt1,rcnt,maxcnt)
! Do something
call system_clock(cnt2,rcnt,maxcnt)

write(6,*) ' Time taken ',cnt2-cnt1
```