Data Structures – Week #2

Algorithm Analysis

&

Sparse Vectors/Matrices

&

Recursion

Outline

- Performance of Algorithms
- Performance Prediction (Order of Algorithms)
- Examples
- Exercises
- Sparse Vectors/Matrices
- Recursion
- Recurrences

Algorithm Analysis

Performance of Algorithms

- Algorithm: a *finite sequence of instructions* that the computer follows to solve a problem.
- Algorithms solving the *same problem* may *perform differently*. Depending on *resource requirements* an algorithm may be *feasible* or not. To find out whether or not an algorithm is usable or relatively better than another one solving the same problem, its resource requirements should be determined.
- The *process of determining the resources of an algorithm* is called algorithm analysis.
- Two essential resources, hence, *performance criteria* of algorithms are
  - *execution or running time*
  - *memory space used.*

Performance Assessment - 1

- **Execution time** of an algorithm is hard to assess unless one knows
  - the *intimate details of the computer architecture*,
  - the operating system,
  - the compiler,
  - the quality of the program,
  - the current load of the system and
  - other factors.

Performance Assessment - 2

- Two ways to assess performance of an algorithm
  - Execution time may be compared for a given algorithm using some special performance programs called *benchmarks* and evaluated as such.
  - *Growth rate* of *execution time* (or *memory space*) of an algorithm with the growing input size may be found.

Performance Assessment - 3

- Here, we define the *execution time* or the *memory space* used as a *function of the input size*.

- By "*input size*" we mean
  - the number of elements to store in a data structure,
  - the number of records in a file etc…
  - the nodes in a LL or a tree or
  - the nodes as well as connections of a graph

Assessment Tools

- We can use the concept the "*growth rate or order of* an algorithm" to assess both criteria. However, our main concern will be the execution time.
- We use *asymptotic notations* to symbolize the *asymptotic running time of an algorithm* in terms of the input size.

Asymptotic Notations

- We use *asymptotic notations* to symbolize the *asymptotic running time of an algorithm* in terms of the input size.
- The following notations are frequently used in algorithm analysis:
  - ***O*** *(Big Oh)* Notation (*asymptotic upper bound*)
  - ***Ω*** *(Omega)* Notation (*asymptotic lower bound*)
  - ***Θ*** *(Theta)* Notation (*asymptotic tight bound*)
  - ***o*** *(little Oh)* Notation (*upper bound that is **not** asymptotically tight*)
  - ***ω*** *(omega)* Notation (*lower bound that is **not** asymptotically tight*)
- Goal: *To find a function that asymptotically limits the execution time or the memory space of an algorithm.*

O-Notation ("Big Oh")

Asymptotic Upper Bound

- Mathematically expressed, the "***Big Oh***" (*O()*) concept is as follows:
- Let g: $\boldsymbol{N} \to \boldsymbol{R^*}$ be an arbitrary function.
- $O(g(n)) = \{f: \boldsymbol{N} \to \boldsymbol{R^*} \mid (\exists c \in \boldsymbol{R^+})(\exists n_0 \in \boldsymbol{N})(\forall n \geq n_0)\ [f(n) \leq cg(n)]\}$,
    - where $\boldsymbol{R^*}$ is the set of nonnegative real numbers and $\boldsymbol{R^+}$ is the set of strictly positive real numbers (excluding 0).

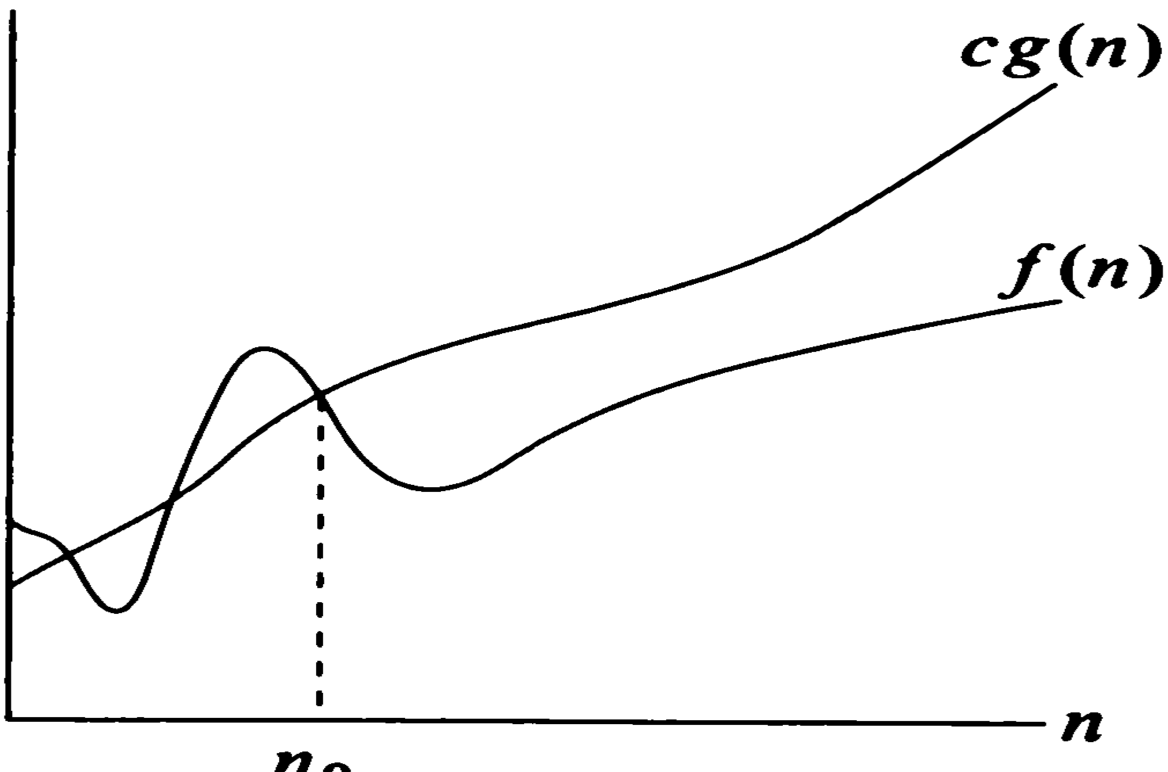
O-Notation by words

- ***Expressed by words***; $O(g(n))$ is the set of all functions $f(n)$ mapping ($\rightarrow$) integers (***N***) to nonnegative real numbers (***R****) such that (|) there exists a positive real constant c ($\exists c \in \mathbf{R}^{+}$) *and* there exists an integer constant n_0 ($\exists n_0 \in \mathbf{N}$) such that *for all* values of n greater than or equal to the constant ($\forall n \geq n_0$), the function values of $f(n)$ are less than or equal to the function values of $g(n)$ multiplied by the constant c ($f(n) \leq cg(n)$).

- In other words, $O(g(n))$ is the set of all functions $f(n)$ bounded above by a positive real multiple of $g(n)$, provided n is sufficiently large (greater than n_0). $g(n)$ denotes the *asymptotic upper bound* for the running time $f(n)$ of an algorithm.

O-Notation ("Big Oh")

Asymptotic Upper Bound



$$f(n) = O(g(n))$$

Θ-Notation ("Theta")

Asymptotic Tight Bound

- Mathematically expressed, the "***Theta***" ($\Theta()$) concept is as follows:
- Let $g: \boldsymbol{N} \rightarrow \boldsymbol{R^*}$ be an arbitrary function.
- $\Theta(g(n)) = \{f: \boldsymbol{N} \longrightarrow \boldsymbol{R^*} \mid (\exists c_1, c_2 \in \boldsymbol{R^+})(\exists n_0 \in \boldsymbol{N})(\forall n \geq n_0)$
  $[0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)]\}$,
  - where $\boldsymbol{R^*}$ is the set of nonnegative real numbers and $\boldsymbol{R^+}$ is the set of strictly positive real numbers (excluding 0).

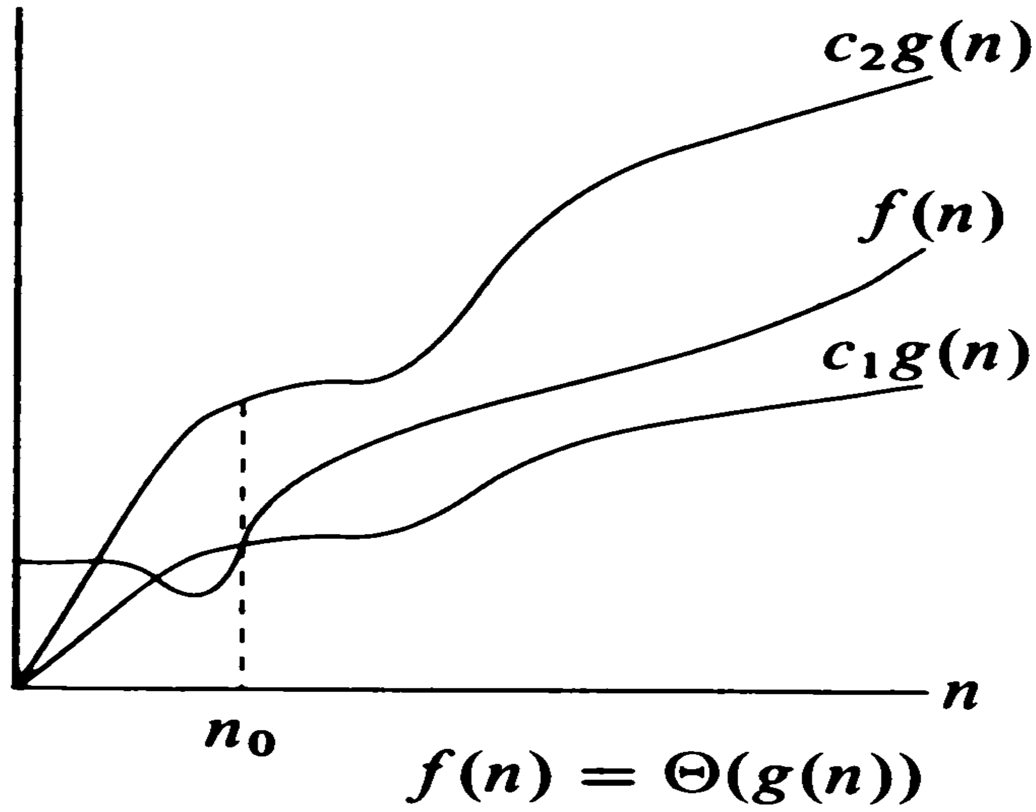
Θ-Notation by words

- ***Expressed by words***; A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive real constants c_1 and c_2 $(\exists c_1, c_2 \in \mathbf{R}^+)$ such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$ $([0 \le c_1 gn) \le f(n) \le c_2 g(n)])$, for sufficiently large n $(\forall n \ge n_0)$.

- In other words, $\Theta(g(n))$ is the set of all functions $f(n)$ tightly bounded below and above by a pair of positive real multiples of $g(n)$, provided n is sufficiently large (greater than n_0). $g(n)$ denotes the *asymptotic tight bound* for the running time $f(n)$ of an algorithm.

Θ-Notation ("Theta")

Asymptotic Tight Bound



Ω-Notation ("Big-Omega")

Asymptotic Lower Bound

- Mathematically expressed, the "Omega" ($\Omega()$) concept is as follows:
- Let g: $\boldsymbol{N} \rightarrow \boldsymbol{R^*}$ be an arbitrary function.
- $\Omega(g(n)) = \{f: \boldsymbol{N} \rightarrow \boldsymbol{R^*} \mid (\exists c \in \boldsymbol{R}^+)(\exists n_0 \in \boldsymbol{N})(\forall n \geq n_0)$

  $[0 \leq cg(n) \leq f(n)]\}$,
  - where $\boldsymbol{R^*}$ is the set of nonnegative real numbers and $\boldsymbol{R}^+$ is the set of strictly positive real numbers (excluding 0).

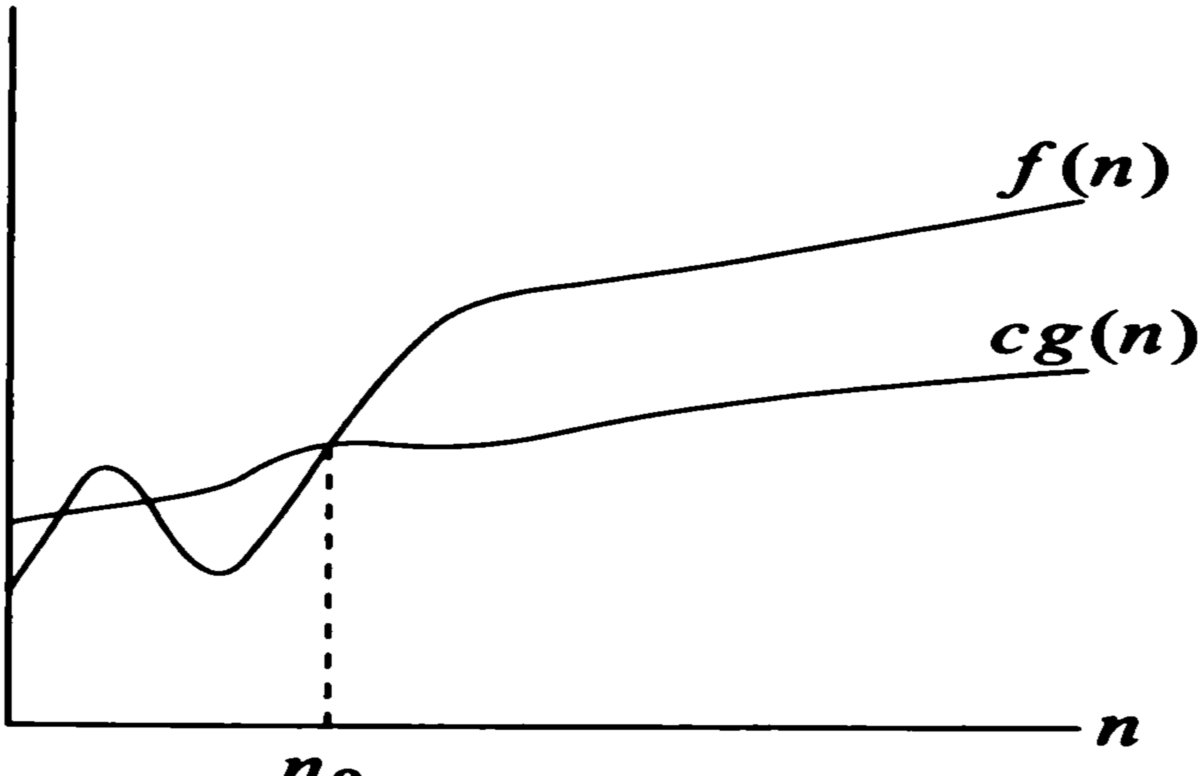
Ω-Notation by words

- ***Expressed by words***; A function *f(n)* belongs to the set *Ω(g(n))* if there exists a positive real constant *c* $(\exists c \in \mathbf{R}^+)$ such that *f(n)* is less than or equal to *cg(n)* $([0 \le cg(n) \le f(n)])$, for sufficiently large *n* $(\forall n \ge n_0)$.

- In other words, *Ω(g(n))* is the set of all functions *t(n)* bounded below by a positive real multiple of *g(n)*, provided *n* is sufficiently large (greater than n_0). *g(n)* denotes the *asymptotic lower bound* for the running time *f(n)* of an algorithm.

Ω-Notation (“Big-Omega”)

Asymptotic Lower Bound



$$f(n) = \Omega(g(n))$$

o-Notation ("Little Oh")

Upper bound NOT Asymptotically Tight

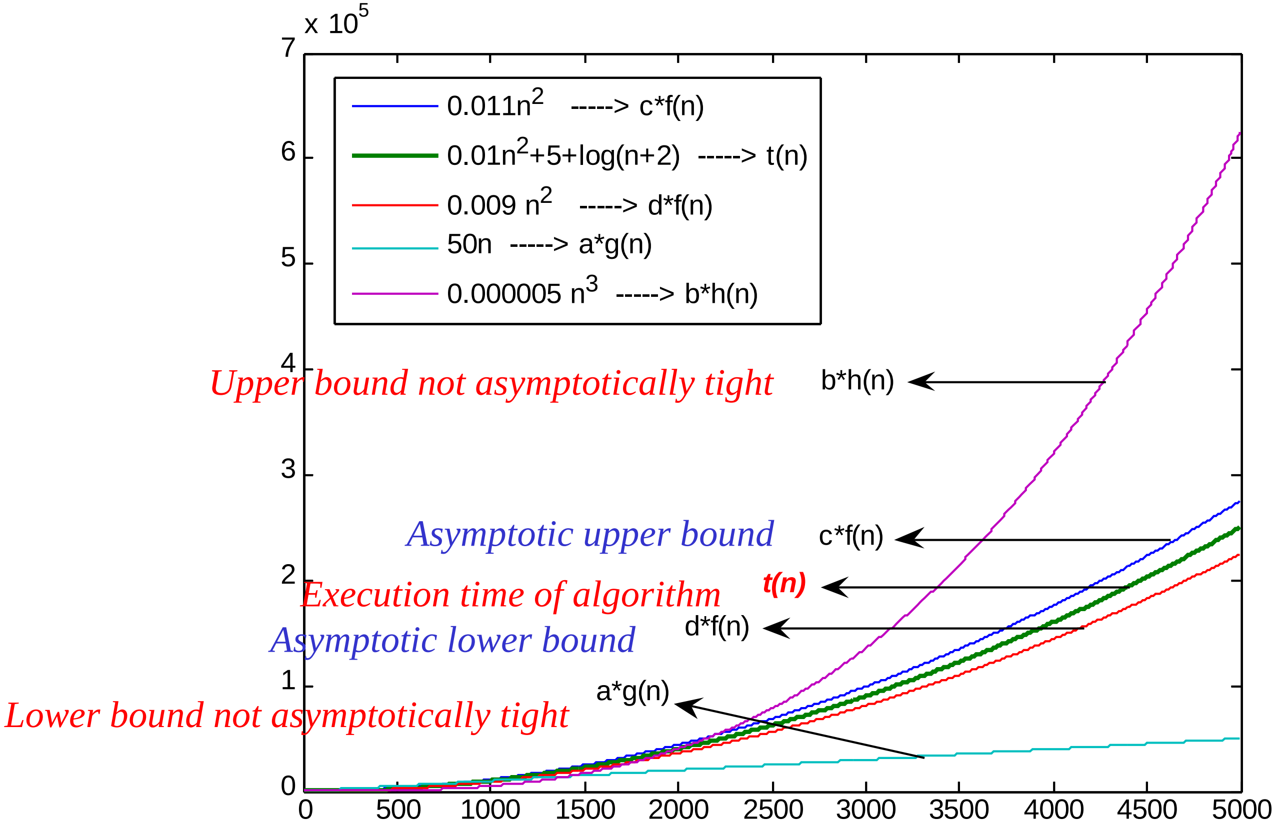
- "*o*" notation does not reveal whether the function *f(n)* is a *tight asymptotic upper bound* for t(n) ($t(n) \leq cf(n)$).
- "Little Oh" or ***o*** notation provides an *upper bound that strictly is NOT asymptotically tight*.
- Mathematically expressed;
- Let f: $\boldsymbol{N} \rightarrow \boldsymbol{R^*}$ be an arbitrary function.
- $o(f(n)) = \{t: \boldsymbol{N} \rightarrow \boldsymbol{R^*} \mid (\exists c \in \boldsymbol{R}^+)(\exists n_0 \in \boldsymbol{N})(\forall n \geq n_0)\ [t(n) < cf(n)]\}$,
  - where $\boldsymbol{R^*}$ is the set of nonnegative real numbers and $\boldsymbol{R}^+$ is the set of strictly positive real numbers (excluding 0).

ω-Notation (“Little-Omega”)

Lower Bound NOT Asymptotically Tight

- ω concept relates to Ω concept in analogy to the relation of “little-Oh” concept to “big-Oh” concept.
- “Little Omega” or ω notation provides a *lower bound that strictly is NOT asymptotically tight*.
- Mathematically expressed, the “Little Omega” ($\omega()$) concept is as follows:
- Let f: $\boldsymbol{N} \rightarrow \boldsymbol{R^*}$ be an arbitrary function.
- $\omega(f(n)) = \{t: \boldsymbol{N} \rightarrow \boldsymbol{R^*} \mid (\exists c \in \boldsymbol{R}^+)(\exists n_0 \in \boldsymbol{N})(\forall n \geq n_0)\ [cf(n) < t(n)]\}$,
  - where $\boldsymbol{R^*}$ is the set of nonnegative real numbers and $\boldsymbol{R}^+$ is the set of strictly positive real numbers (excluding 0).

Asymptotic Notations
Examples



$t(n) \in O(f(n))$
$t(n) \in O(h(n))$

$t(n) \in \Theta(f(n))$
$t(n) \notin \Theta(h(n))$
$t(n) \notin \Theta(g(n))$

$t(n) \in \Omega(f(n))$
$t(n) \in \Omega(g(n))$

$t(n) \in o(h(n))$
$t(n) \notin o(f(n))$

$t(n) \in \omega(g(n))$
$t(n) \notin \omega(f(n))$

Execution time of various structures

- Simple Statement

  $O(1)$, executed within a constant amount of time irresponsive to any change in input size.

- Decision (if) structure

  if (condition) f(n) else g(n)

  O(if structure)$=max(O(f(n)),O(g(n))$

- Sequence of Simple Statements

  $O(1)$, since $O(f_1(n)+\ldots+f_s(n))=O(max(f_1(n),\ldots,f_s(n)))$

Execution time of various structures

- $O(f_1(n)+\ldots+f_s(n))=O(max(f_1(n),\ldots,f_s(n)))$???
- Proof:

$t(n)\in O(f_1(n)+\ldots+f_s(n)) \Rightarrow t(n)\leq c[f_1(n)+\ldots+f_s(n)]$

$\leq \quad sc*max\ [f_1(n),\ldots,\ f_s(n)], sc$ another constant.

$\Rightarrow t(n)\in O(max(f_1(n),\ldots,f_s(n)))$

Hence, hypothesis follows.

Execution Time of Loop Structures

- Loop structures' execution time depends upon whether or not their index bounds are related to the input size.
- Assume *n* is the number of input records
- for (i=0; i<=n; i++) {statement block}, O(?)
- for (i=0; i<=m; i++) {statement block}, O(?)

Examples

Find the execution time t(n) in terms of n!

```
for (i=0; i<=n; i++)
  for (j=0; j<=n; j++)
    statement block;

for (i=0; i<=n; i++)
  for (j=0; j<=i; j++)
    statement block;

for (i=0; i<=n; i++)
  for (j=1; j<=n; j*=2)
    statement block;
```

Exercises

Find the number of times the statement block is executed!

```
for (i=0; i<=n; i++)
   for (j=1; j<=i; j*=2)
     statement block;

for (i=1; i<=n; i*=3)
   for (j=1; j<=n; j*=2)
     statement block;
```

Sparse Vectors and Matrices

Motivation

- In numerous applications, we may have to process vectors/matrices which mostly contain trivial information (i.e., most of their entries are zero!). This type of vectors/matrices are defined to be *sparse*.
- Storing *sparse* vectors/matrices as usual (e.g., matrices in a 2D array or a vector a regular 1D array) causes wasting memory space for storing trivial information.
- Example: *What is the **space requirement** for a matrix* m_{nxn} *with only **non-trivial information in its diagonal** if*
  - *it is stored in a 2D array;*
  - *in some other way? Your suggestions?*

Sparse Vectors and Matrices

- This fact brings up the question:

May the vector/matrix be stored in MM avoiding waste of memory space?

Sparse Vectors and Matrices

- Assuming that the vector/matrix is *static* (i.e., it is not going to change throughout the execution of the program), we should study *two cases*:

    1. Non-trivial information is placed in the vector/matrix *following a specific order*;

    1. Non-trivial information is *randomly* placed in the vector/matrix.

Case 1: Info. follows an order

- Example structures:
  - Triangular matrices (upper or lower triangular matrices)
  - Symmetric matrices
  - Band matrices
  - Any other types ...?

Triangular Matrices

$$m=\begin{bmatrix} m_{11} & m_{12} & m_{13} & \cdots & m_{1\mathrm{n}} \\ 0 & m_{22} & m_{23} & \cdots & m_{2\mathrm{n}} \\ 0 & 0 & m_{33} & \cdots & m_{3\mathrm{n}} \\ 0 & 0 & 0 & \vdots & \vdots \\ 0 & 0 & 0 & 0 & m_{nn} \end{bmatrix}$$

Upper Triangular Matrix

$$m=\begin{bmatrix} m_{11} & 0 & 0 & \cdots & 0 \\ m_{21} & m_{22} & 0 & \cdots & 0 \\ m_{31} & m_{32} & m_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \cdots & m_{nn} \end{bmatrix}$$

Lower Triangular Matrix

Symmetric and Band Matrices

$$m=\begin{bmatrix} m_{11} & m_{12} & m_{13} & \cdots & m_{1\mathrm{n}} \\ m_{12} & m_{22} & m_{23} & \cdots & m_{2\mathrm{n}} \\ m_{13} & m_{23} & m_{33} & \cdots & m_{3\mathrm{n}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m_{1\mathrm{n}} & m_{2\mathrm{n}} & m_{3\mathrm{n}} & \cdots & m_{nn} \end{bmatrix}$$

Symmetric Matrix

$$m=\begin{bmatrix} m_{11} & m_{12} & 0 & \cdots & 0 \\ m_{21} & m_{22} & m_{23} & \cdots & 0 \\ 0 & m_{32} & m_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & m_{n-1,n} \\ 0 & 0 & \cdots & m_{n,n-1} & m_{nn} \end{bmatrix}$$

Band Matrix

Case 1:How to Efficiently Store...

- Store only the non-trivial information in a *1-dim* array *a*;
- Find a function *f* mapping the indices of the *2-dim* matrix (i.e., *i* and *j*) to the index *k* of *1-dim* array *a*, or

$$f : N_0^2 \rightarrow N_0$$

such that

$$k=f(i,j)$$

Case 1: Example for Lower Triangular Matrices

$$m=\begin{bmatrix} m_{11} & 0 & 0 & \cdots & 0 \\ m_{21} & m_{22} & 0 & \cdots & 0 \\ m_{31} & m_{32} & m_{33} & \cdots & 0 \\ \vdots & \vdots & m_{ij} & \vdots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \cdots & m_{nn} \end{bmatrix} \Rightarrow$$

$k \rightarrow$ 0	1	2	3	4	5	....	n(n-1)/2			....	
m_{11}	m_{21}	m_{22}	m_{31}	m_{32}	m_{33}	.....	m_{n1}	m_{n2}	m_{n3}	......	m_{nn}

$$k=f(i,j)=i(i-1)/2+j-1$$

$$\Rightarrow$$

$$m_{ij} = a[i(i-1)/2+j-1]$$

Case 2: Non-trivial Info. Randomly Located

Example:

$$m = \begin{bmatrix} a & 0 & 0 & \cdots & 0 \\ 0 & b & 0 & \cdots & f \\ 0 & c & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & g & \vdots \\ e & 0 & d & \cdots & 0 \end{bmatrix}$$

Case 2:How to Efficiently Store...

- Store only the non-trivial information in a *1-dim* array *a* along with the entry coordinates.
- Example:

a							
	a;0,0	*b;1,1*	*f;1,n-1*	*c;2,1*	*g;i,j*	*e;n-1,0*	*d;n-1,2*

Recursion

Recursion

Definition:

Recursion is a mathematical concept referring to programs or functions calling or using itself.

A *recursive function* is a functional piece of code that invokes or calls itself.

Recursion

Concept:

- A recursive function divides the problem into two conceptual pieces:
  - a piece that the function knows how to solve (**base case**),
  - a piece that is very similar to, but *a little simpler than*, the original problem, hence still unknown how to solve by the function (**call(s) of the function to itself**).

Recursion… cont’d

- ***Base case*****:** the simplest version of the problem that is *not further reducible*. The function actually knows how to solve this version of the problem.

- To make the recursion feasible, the latter piece must be slightly simpler.

Recursion Examples

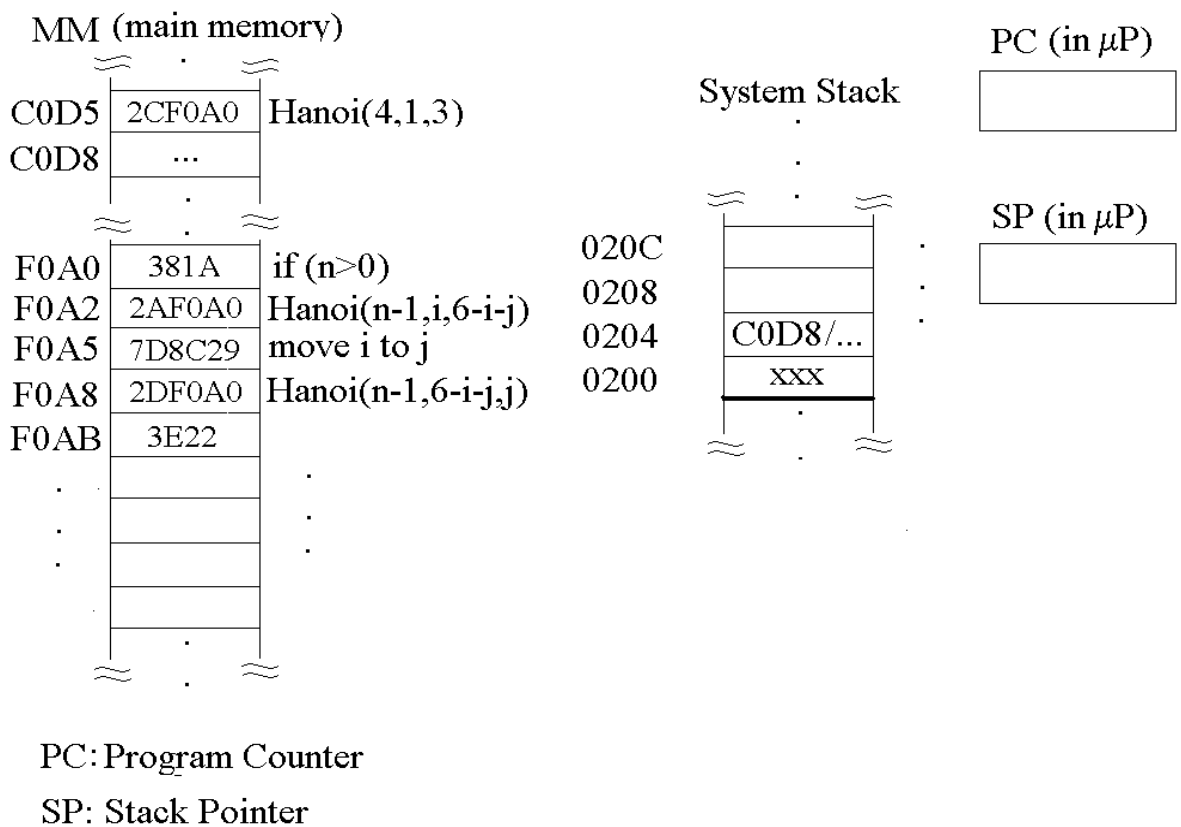
- **Towers of Hanoi**
- Story: According to the legend, the life on the world will end when Buddhist monks in a Far-Eastern temple move 64 disks stacked on a peg in a decreasing order in size to another peg. They are allowed to move one disk at a time and a larger disk can never be placed over a smaller one.

Towers of Hanoi… cont’d

Algorithm:

```
        Hanoi(n,i,j)
        // moves n smallest rings from rod i to rod j
F0A0    if (n > 0) {
            //moves top n-1 rings to intermediary rod (6-i-j)
F0A2        Hanoi(n-1,i,6-i-j);
            //moves the bottom (nth largest) ring to rod j
F0A5        move i to j
            // moves n-1 rings at rod 6-i-j to destination rod j
F0A8        Hanoi(n-1,6-i-j,j);
F0AB    }
```

Function Invocation in MM



Function Invocation (Call) in MM

- Code and data are both in MM.
- Hanoi function is called by the instruction at MM cell C0D5 with arguments (4,1,3).
- *Program counter* is a register in μP that holds MM address of next instruction to execute.
- If current instruction is a function call, the serial flow of execution is interrupted.

Function Call in MM… cont’d

- Following problems arise:
  - how to keep the return address from the function called (Hanoi) back to the caller function (C0D8 at main and both F0A5 and F0AB at Hanoi);
  - how to store the values of variables local to caller function.
- Both problems are solved by keeping the return address and local variables’ values in a portion of the main memory called *system stack*.
- Another register called ***S****tack* ***P****ointer* points to the address pushed most recently to *system stack*. Return addresses are retrieved from *system stack* in a last-in-first-out (LIFO) fashion. We will see stacks later.

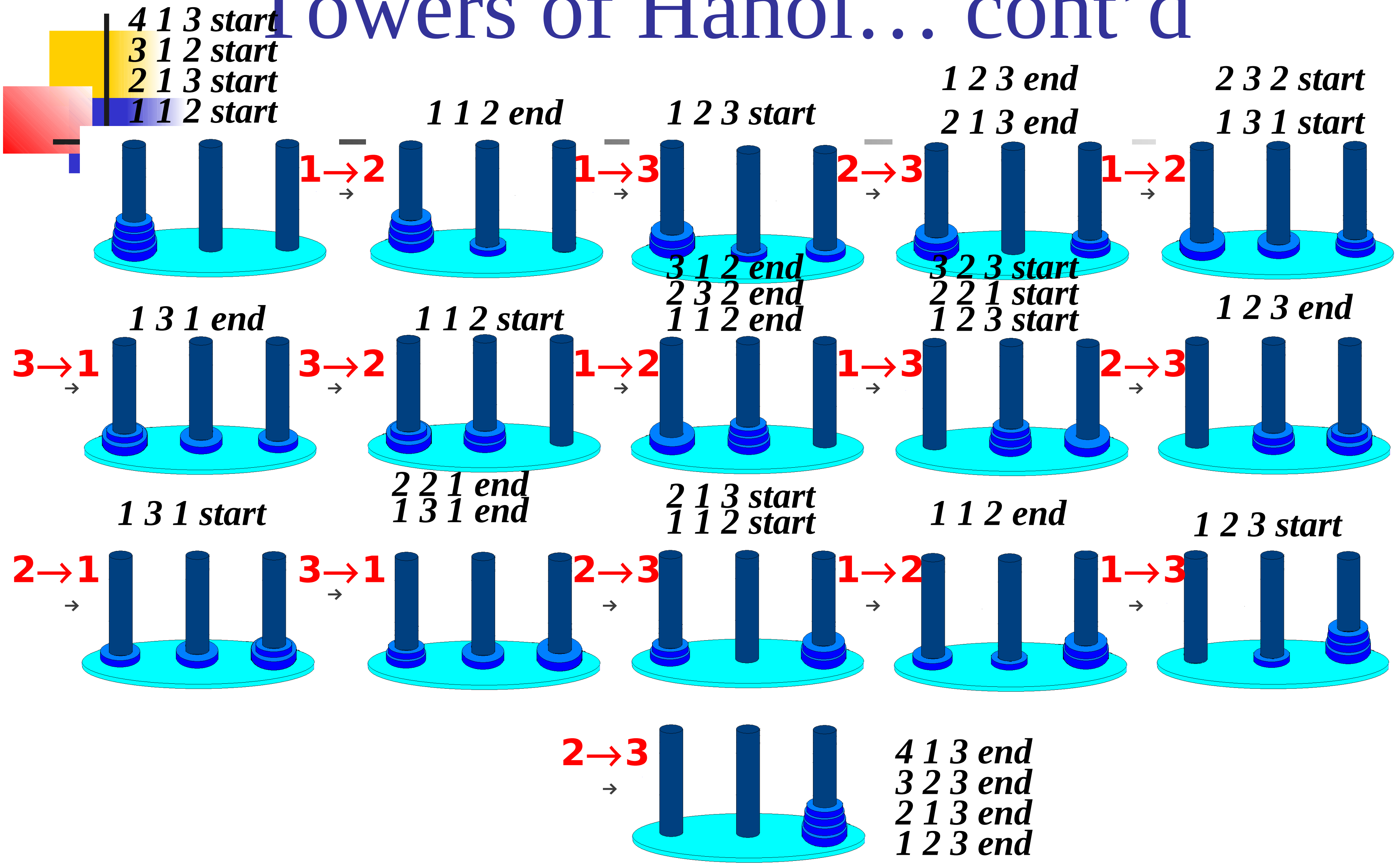
Towers of Hanoi… cont'd

Example: Hanoi(4,i,j)

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

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

Towers of Hanoi… cont'd



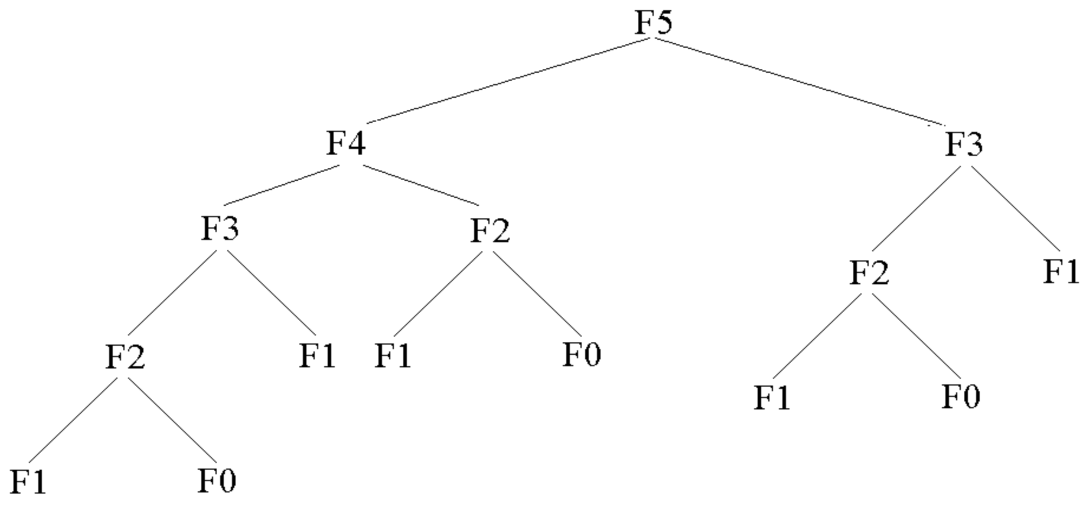
Recursion Examples

- **Fibonacci Series**
  - $t_n = t_{n-1} + t_{n-2}$; $t_0=0$; $t_1=1$
- Algorithm

```
long int fib(n)
{
if (n==0 || n==1)
    return n;
else
    return fib(n-1)+fib(n-2);
}
```

Fibonacci Series… cont'd

- Tree of recursive function calls for fib(5)
- Any problems???



Fibonacci Series… cont'd

- Redundant function calls slow the execution down.
- A **lookup table** used to store the Fibonacci values already computed saves redundant function executions and speeds up the process.
- ***Homework***: Write `fib(n)` with a lookup table!

Recurrences

Recurrences or Difference Equations

- **Homogeneous Recurrences**
- Consider $a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$.
- The recurrence
  - contains t_i values which we are looking for.
  - is a linear recurrence (i.e., t_i values appear alone, no powered values, divisions or products)
  - contains constant coefficients (i.e., a_i).
  - is homogeneous (i.e., RHS of equation is 0).

Homogeneous Recurrences

We are looking for solutions of the form:

$$t_n = x^n$$

Then, we can write the recurrence as

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k} = 0$$

- This k^{th} degree equation is the **characteristic equation (CE)** of the recurrence.

Homogeneous Recurrences

If r_i, $i=1,\ldots, k$, are k distinct roots of $a_0x^k + a_1x^{k-1} + \ldots + a_k = 0$, then

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

If r_i, $i=1,\ldots, k$, is a single root of multiplicity k, then

$$t_n = \sum_{i=1}^{k} c_i n^{i-1} r^n$$

Inhomogeneous Recurrences

Consider

- $a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n)$
- where b is a constant; and $p(n)$ is a polynomial in n of degree d.

Inhomogeneous Recurrences

Generalized Solution for Recurrences

Consider a general equation of the form

$$(a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k}) = b_1^n p_1(n) + b_2^n p_2(n) + \ldots$$

We are looking for solutions of the form:

$$t_n = x^n$$

Then, we can write the recurrence as

$$\left(a_0 x^k + a_1 x^{k-1} + \cdots + a_k\right)\left(x - b_1\right)^{d_1+1}\left(x - b_2\right)^{d_2+1}\cdots = 0$$

where d_i is the polynomial degree of polynomial $p_i(n)$.

This is the ***characteristic equation (CE)*** of the recurrence.

Generalized Solution for Recurrences

If r_i, $i=1,\ldots, k$, are k distinct roots of

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k) = 0$$

$$t_n = \sum_{i=1}^{k} c_i r_i^n + c_{k+1} b_1^n + c_{k+2} n b_1^n + \cdots + c_{k+1+d_1} n^{d_1 - 1} b_1^n +$$

$$+ c_{k+2+d_1} b_2^n + c_{k+3+d_1} n b_2^n + \cdots + c_{k+2+d_1+d_2} n^{d_2 - 1} b_2^n$$

Examples

Homogeneous Recurrences

Example 1.

$t_n + 5t_{n-1} + 4\,t_{n-2} = 0$; sol'ns of the form $t_n = x^n$

$x^n + 5x^{n-1} + 4x^{n-2} = 0$; (CE) n-2 trivial sol'ns (i.e., $x_{1,\ldots,n-2}=0$)

$(x^2+5x+4) = 0$; characteristic equation (simplified CE)

$x_1=-1;\ x_2=-4;$ nontrivial sol'ns

$\Rightarrow \quad t_n = c_1(-1)^n + c_2(-4)^n$; general sol'n

Examples

Homogeneous Recurrence

Example 2.

t_n-6 t_{n-1}+12t_{n-2}-8t_{n-3}=0; $t_n = x^n$

x^n-6x^{n-1}+12x^{n-2}-8x^{n-3}= 0; n-3 trivial sol'ns

CE: $(x^3-6x^2+12x-8) = (x-2)^3 = 0$; by polynomial division

$x_1 = x_2 = x_3 = 2$; roots not distinct!!!

⇒ $t_n = c_1 2^n + c_2 \mathbf{n} 2^n + c_3 \mathbf{n}^2 2^n$; general sol'n

Examples

Homogeneous Recurrence

Example 3.

$t_n = t_{n-1} + t_{n-2}$; Fibonacci Series

$x^n - x^{n-1} - x^{n-2} = 0$; $\Rightarrow$ CE: $x^2 - x - 1 = 0$;

$x_{1,2} = \dfrac{1 \pm \sqrt{5}}{2}$; distinct roots!!!

$$\Rightarrow t_n = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$; general sol'n!!

We find coefficients c_i using initial values t_0 and t_1 of Fibonacci series on the next slide!!!

Examples

Example 3… cont'd

We use as many t_i values

as c_i

$$t_0=0=c_1\left(\frac{1+\sqrt{5}}{2}\right)^0+c_2\left(\frac{1-\sqrt{5}}{2}\right)^0=c_1+c_2=0\Rightarrow c_1=-c_2$$

$$t_1=1=c_1\left(\frac{1+\sqrt{5}}{2}\right)^1+c_2\left(\frac{1-\sqrt{5}}{2}\right)^1=c_1\left(\frac{1+\sqrt{5}}{2}\right)-c_1\left(\frac{1-\sqrt{5}}{2}\right)\Rightarrow c_1=\frac{1}{\sqrt{5}},\, c_2=-\frac{1}{\sqrt{5}}$$

$$\Rightarrow t_n=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Check it out using t_2!!!

Examples

Example 3… cont'd

What do n and t_n represent?

n is the location and t_n the value of any Fibonacci number in the series.

Examples

Example 4.

$t_n = 2t_{n-1} - 2t_{n-2}$; $n \geq 2$; $t_0 = 0$; $t_1 = 1$;

CE: $x^2-2x+2 = 0$;

Complex roots: $\boldsymbol{x_{1,2}=1\pm i}$

As in differential equations, we represent the complex roots as a vector in polar coordinates by a combination of a real radius r and a complex argument θ:

$$z=r*e\theta i;$$

Here,

$$\mathbf{1+i=\sqrt{2} * e}(\pi/4)i$$
$$\mathbf{1-i=\sqrt{2} * e}(-\pi/4)i$$

Examples

Example 4… cont'd

Solution:

$$t_n = c_1\,(2)^{n/2}\, \boldsymbol{e}^{(n\pi/4)i} + c_2\,(2)^{n/2}\, \boldsymbol{e}^{(-n\pi/4)i};$$

From initial values $t_0 = 0,\ t_1 = 1,$

$$t_n = 2^{n/2}\,\sin(n\pi/4);\quad (\textit{prove that!!!})$$

Hint:

$$e^{i\theta} = \cos\theta + i\sin\theta$$

$$e^{in\theta} = \left(\cos\theta + i\sin\theta\right)^n = \cos n\theta + i\sin n\theta$$

Examples

Inhomogeneous Recurrences

Example 1. (*From Example 3*)

We would like to know how many times `fib(n)` on page 22 is executed in terms of *n*. To find out:

1. choose a barometer in `fib(n)`;
2. devise a formula to count up the number of times the barometer is executed.

Examples

Example 1… cont'd

In fib(n), the only statement is the *if* statement. Hence, *if* condition is chosen as the barometer. Suppose fib(n) takes t_n time units to execute, where the barometer takes one time unit and the function calls fib(n-1) and fib(n-2), t_{n-1} and t_{n-2}, respectively. Hence, the recurrence to solve is

$$t_n = t_{n-1} + t_{n-2} + 1$$

Examples

Example 1… cont'd

$t_n - t_{n-1} - t_{n-2} = 1$; inhomogeneous recurrence

The homogeneous part comes directly from Fibonacci Series example on page 52.

RHS of recurrence is 1 which can be expressed as $1^n x^0$. Then, from the equation on page 48,

CE: $(x^2-x-1)(x-1) = 0$; from page 49,

$$t_n = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n + c_3 1^n$$

Examples

Example 1… cont'd

$$t_n = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n + c_3$$

Now, we have to find $c_1,\ldots,c_3$.

Initial values: for both $n=0$ and $n=1$, *if* condition is checked once and no recursive calls are done.

For $n=2$, *if* condition is checked once and recursive calls `fib(1)` and `fib(0)` are done.

$\Rightarrow t_0 = t_1 = 1$ and $t_2 = t_0 + t_1 + 1 = 3$.

Examples

Example 1… cont'd

$$t_n = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n + c_3; t_0 = t_1 = 1, t_2 = 3$$

$$c_1 = \frac{\sqrt{5}+1}{\sqrt{5}};\ c_2 = \frac{\sqrt{5}-1}{\sqrt{5}};\ c_3 = -1$$

$$t_n = \left[\frac{\sqrt{5}+1}{\sqrt{5}}\right]\left(\frac{1+\sqrt{5}}{2}\right)^n + \left[\frac{\sqrt{5}-1}{\sqrt{5}}\right]\left(\frac{1-\sqrt{5}}{2}\right)^n - 1;$$

Here, t_n provides the number of times the barometer is executed in terms of *n*. Practically, this number also gives the number of times fib(n) is called.