Data Structures

Recursive Programming

Repeating a Code Segment

- There are two basic methods for repeating a code segment:
  - Loops (do-while, while, for, ...)
  - Recursion

- Recursion:
  A function calling itself
  - Simplifies code writing for the solutions of some types of problems.
  - Is an important advanced programmed technique and shortens the code when used appropriately.
  - Downside: The program may take longer to run.

Recursive Programming

- The basic approach is to define the problem to be solved in terms of simpler subproblems.
- The recursive function can solve the base case and returns the result if there is one.
- If the input problem is not the base case, the problem can be divided into simple parts, and the function calls itself for each part.
- Since recursive program writing is different from the usual style, it takes more practice to learn.

Example: Computing the Factorial

Definition of the factorial function:

$$n! = \begin{cases} 1 & if \quad n = 0 \\ n.(n-1).(n-2)....3.2.1 & if \quad n \geq 1 \end{cases}$$

n! = n · (n-1)· ··· · 3· 2· 1
5! = 5 · 4· 3· 2· 1

We could also define the function in terms of itself:
Example: fact(5) = 5·(4·3·2·1) = 5 · fact(4)

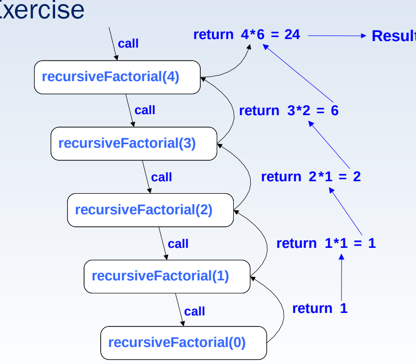
$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

Factorial Function

```
int recursiveFactorial(int n){
  int m;
  if (n == 0) return 1;   // base case
  else{                   // recursion step
     m = recursiveFactorial(n-1);
     return n * m;
     return n * recursiveFactorial(n-1);
  }
}
```

Note!
The base case is, at the same time, the termination condition of the function and should not be left out. Otherwise, there will be "infinite" recursion (analogous to the problem of an infinite loop in an iterative solution).

Exercise



Is Recursion Always Useful?

- Let us write the same function iteratively:

```
int iterativeFactorial(int n){
  int i,p;
  p=1;
  for(i=2; i<=n; i++)
    p*=i;
  return p;
}
```

- This solution wastes less space in memory and works faster.

Recursion Types

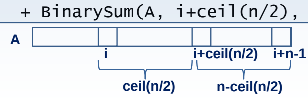
- Linear
  - Each time a function (method) is called, it makes at most one recursive call.
- Tail
  - If there is linear recursion and the last command of the function is a recursive call, tail recursion occurs.
  - These types of methods can be converted from the recursive state to the iterative state.
- Binary
  - If the function calls itself twice at a time, there is binary recursion.

Binary Recursion Example

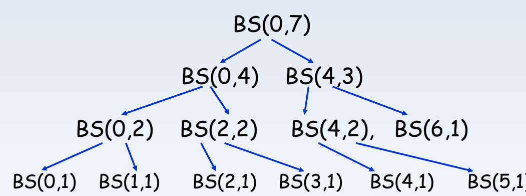
Let us write the function that sums elements of an array.

Algorithm:

```
BinarySum(A,i,n)
   Input:   array A, integers i and n
   Output:  sum of n numbers in array A starting with
            index i
if n = 1 then
   return A[i]
else
   return BinarySum(A, i, ceil(n/2))
       + BinarySum(A, i+ceil(n/2), n-ceil(n/2))
```



BinarySum() Call Sequence



- Binary recursion is generally used in algorithms designed with the "**divide and conquer**" method.
- In these types of algorithms, recursion may prove more effective than an iterative solution.

Calculating the Square of an Integer

- The square of the number x will be calculated using only addition, subtraction, and left shift (multiplication by 2) operations.
- In integer multiplication operations, instead of the much slower direct low-level multiplication command, compilers use addition/subtraction and right/left shift low-level commands.
- Solution:

$x^2 = (x - 1 + 1)^2$

$= (x - 1)^2 + 2(x - 1) + 1^2$

$= (x - 1)^2 + 2x - 1$

Squaring an Integer

```
unsigned int square(unsigned int x){
   if (x <= 1)
      return 1;
   else
      return square(x - 1) + (x << 1) - 1;
}
```

Permutation

- Example: permutations of the set {1,2,3} :

```
1 2 3      1 3 2
2 1 3      2 3 1
3 1 2      3 2 1
```

- To generate these permutations, we could use the code below:

```
for (i1 = 1; i1 <= 3; i1++)
  for (i2 = 1; i2 <= 3; i2++)
    if (i2 != i1)
       for (i3 = 1; i3 <= 3; i3++)
          if ((i3 != i1) && (i3 != i2))
             cout << i1 << “ ”<< i2 << “ ”<< i3 << endl;
```

- However, this code can only generate ternary (3-ary) permutations. How do we generate n-ary permutations?

Recursive Solution for Permutation

```
void perm(int *list, int k, int m){
  int i;
  if (k == m){
     for (i = 0; i <= m; i++)
        cout << list[i] << " ";
     cout << endl;
  }
  else
     for (i = k; i <= m; i++){
        swap(&list[k], &list[i]);
        perm(list, k+1, m);
        swap(&list[k], &list[i]);
     }
}
```

list[0..x] -> an array holding the data that can be printed to the screen
k: starting index
m: ending index
First call: perm(list,0,x)

```
void swap(int *a, int *b){
      int temp = *a;
      *a = *b;
      *b = temp;
}
```


Recursive Solution for Permutation

- At each step, a position is held constant and the others are permuted. For example, when writing the permutation of (1 2 3 4), these numbers are placed into the first position in order and the remaining three positions are written as a permutation.
  1 … permutations that start with 1
  2…. permutations that start with 2
  3…
  4…

Thus, the problem is divided into subproblems.

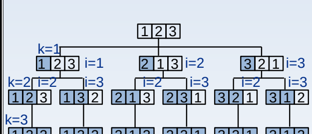
Recursive Solution for Permutation

```
for (i = k; i <= m; i++){
      swap(list[k], list[i]);
      perm(list, k+1, m);
      swap(list[k], list[i]);
}
```

k=1, m=3
1 2 3

First, hold the 1st position constant. Write the permutation of the rest (We do not worry about how to do this for now) next to it.
>> 1 2 3
>> 1 3 2
Then, swap the number in the 1st position with the number in the 2nd position.
2 1 3
Now, again holding the first position constant, write the permutation of the rest next to it.
>> 2 1 3
>> 2 3 1
Reverse the swap operation
1 2 3

Recursive Solution for Permutation



At each stage of permutation:
- Subset of values that have already been processed
- Subset of values that still needs to be processed

Exchange in kth step, the kth value with the value being chosen at that stage. This approach leaves the first subset in the first k locations of the outcome.

k = 1, i = 1:
swap(list[1],list[1])
perm(list,2,3)
swap(list[1],list[1])

k = 2, i = 2:
swap(list[2],list[2])
perm(list,3,3)
swap(list[2],list[2])

k = 2, i = 3:
swap(list[2],list[3])
perm(list,3,3)
swap(list[2],list[3])

k = 1, i = 2:
swap(list[1],list[2])
perm(list,2,3)
swap(list[1],list[2])

k = 2, i = 2:
swap(list[2],list[2])
perm(list,3,3)
swap(list[2],list[2])

k = 2, i = 3:
swap(list[2],list[3])
perm(list,3,3)
swap(list[2],list[3])

k = 1, i = 3:
swap(list[1],list[3])
perm(list,2,3)
swap(list[1],list[3])

k = 2, i = 2:
swap(list[2],list[2])
perm(list,3,3)
swap(list[2],list[2])

k = 2, i = 3:
swap(list[2],list[3])
perm(list,3,3)
swap(list[2],list[3])

Eight Queens Problem

- A classical problem
- Goal: Place 8 queens on a 8X8 chessboard such that they cannot attack each other
- Two queens cannot be located on the
  - Same row
  - Same column
  - Same diagonal
- A solution for a 5X5 board is shown on the right.
  - At the top of every column, number of the row where queen is located is shown.

5	3	1	4	2
		q		
				q
	q			
			q	
q				

Solution to the Eight Queens Problem

- The problem has more than one solution.
- Since only one queen may be located on a row, the potential solutions may be found by generating permutations of the row numbers shown on the example board.
  - All permutations may not be solutions.
  - However, all solutions are among these permutations.

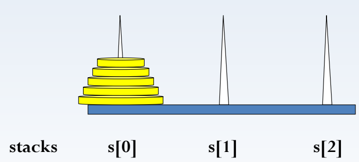
4	1	3	5	2
	q			
				q
		q		
q				
			q	

Solution to the Eight Queens Problem

- A permutation generator with the addition of "diagonal attack" checks could be used to solve the n queens problem.

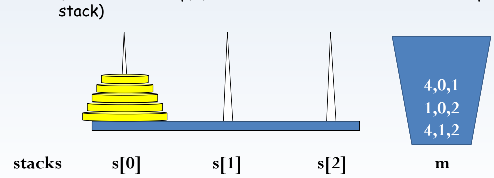
Towers of Hanoi

- The Towers of Hanoi problem can easily be solved using recursion.



Hanoi Iterative Solution (reminder)

- As long as the move stack is full, we pop a move (n,source,destination) from this stack, and push these moves onto the stack:
  - (n-1, temp,destination)
  - (1, source, destination)
  - (n-1, source, temp) (what has to be done first is at the top of the stack)



Hanoi Recursive Solution

```
void Hanoi_recursive(int n, int source, int destination, int
  temp){
  if(n>=1){
        Hanoi_recursive(n-1,source,temp,destination);
        s[destination].push(s[source].pop());
        Hanoi_recursive(n-1,temp,destination,source);
  }
}


Hanoi_recursive(5,0,2,1);
```

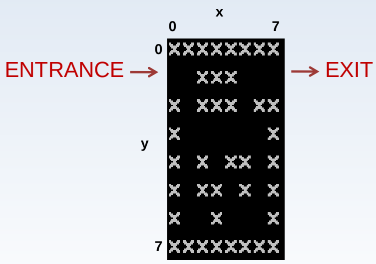

Finding a Path in a Labyrinth

The path finding in a labyrinth problem, which we had previously solved using a stack, could also be solved by writing a recursive function.

```
int main(){
  char lab[8][8]={{'x','x','x','x','x','x','x','x'},
                  {' ',' ','x','x','x',' ',' ',' '},
                  {'x',' ','x','x','x',' ','x','x'},
                  {'x',' ',' ',' ',' ',' ',' ','x'},
                  {'x',' ','x',' ','x','x',' ','x'},
                  {'x',' ','x','x',' ','x',' ','x'},
                  {'x',' ',' ','x',' ',' ',' ','x'},
                  {'x','x','x','x','x','x','x','x'}
  };
  if(find_path(lab,entrance.y,entrance.x, LEFT))
     cout<<"PATH found"<<endl;

  printlab(lab);
  return EXIT_SUCCESS;
}
```

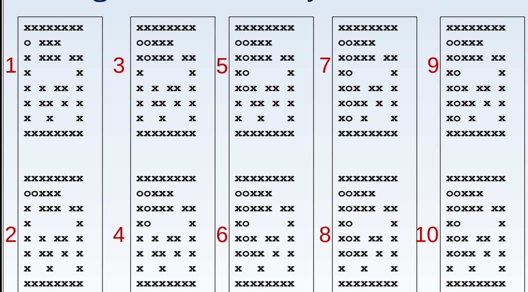
Finding a Path in a Labyrinth



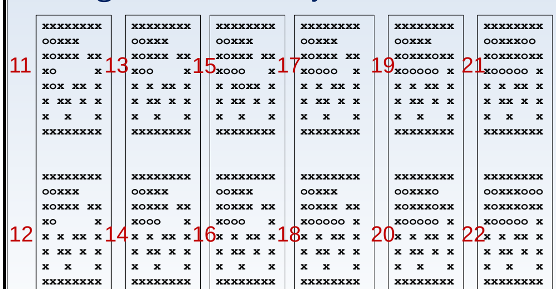
Recursive Path Finding Function

```
bool find_path(char lab[8][8], int y, int x, int camefrom){
   lab[y][x]='o';
   if(x==exit.x && y==exit.y)
         return true;
   printlab(lab);
   if(x>0 && lab[y][x-1]!='x' && camefrom!=LEFT)
         if(find_path(lab,y,x-1,RIGHT)) //left
                return true;
   if(y<7 && lab[y+1][x]!='x' && camefrom!=DOWN)
         if(find_path(lab,y+1,x,UP)) //down
                return true;
   if(y>0 && lab[y-1][x]!='x' && camefrom!=UP)
         if(find_path(lab,y-1,x,DOWN)) //up
                return true;
   if(x<7 && lab[y][x+1]!='x' && camefrom!=RIGHT)
         if (find_path(lab,y,x+1,LEFT)) //right
                return true;
  lab[y][x]=' '; //incorrect paths deleted
   printlab(lab); //the return path is also viewed
   return false;
}
```


Finding a Path in a Labyrinth



Finding a Path in a Labyrinth



In Recitation: To do

- Fibonacci Numbers
- Minmax