CS2001- Data Structures
Week 03

Muhammad Rafi
September 21, 2021

Agenda

- Function vs. Recursion
- Recursion Basic
- Recursive Function – Compiler Support
- Classification of Recursion
  - Tail vs. Non-tail
  - Direct vs. Indirect
  - Excessive vs. simple
- Fibonacci and Comparison
- Conclusion

Agenda

- Fibonacci workload vs. Memorization
- N-Queen Problem
- Backtracking <-> Recursion
- Conclusion

Function Vs. Recursion

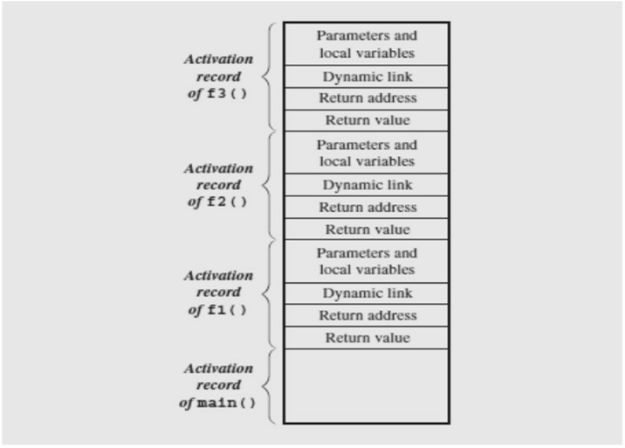
- An activation record usually contains the following information:
  - Values for all parameters to the function, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items.
  - Local variables that can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations where they are stored.
  - The return address to resume control by the caller, the address of the caller's instruction immediately following the call.

Function Vs. Recursion

- An activation record usually contains the following information:
  - A dynamic link, which is a pointer to the caller's activation record.
  - The returned value for a function not declared as void. Because the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller

Function Vs. Recursion

Contents of the run-time stack when `main()` calls function `f1()`, `f1()` calls `f2()`, and `f2()` calls `f3()`.



Major Requirements for Recursion

- A function that call itself before the completion is called recursive function.
- A Base Case
  - Terminate all recursive calls
  - Clearly identifiable
- A Recurrence
  - Progression on input
  - Progression to base case

Recursive Function- Compiler Supports

- Activation Record for every recursive call
  - Return Address and Return Value
  - Local Variables
  - Dynamic Link Pointer
- Function Call Record is different than AR for a recursive function.

Recursive Function

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

- A power function is defined in term of recursive definition.

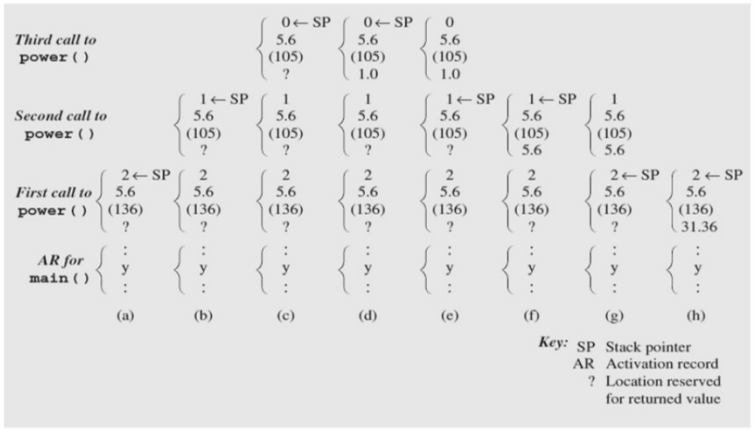
```
/* 102 */ double power (double x, unsigned int n) {
/* 103 */      if (n == 0)
/* 104 */           return 1.0;
          // else
/* 105 */           return x * power(x,n-1);
          }
```

- Non-recursive

```
double nonRecPower(double x, unsigned int n) {
  double result = 1;
  for (result = x; n > 1; --n)
       result *= x;
  return result;
}
```

Recursive Function

FIGURE 5.2 Changes to the run-time stack during execution of `power(5.6,2)`.



Advantages / Disadvantages

- A lot of algorithms are naturally defined as recursive steps. Using recursion will make these program more clear to understand and elegant to review.
- Recursion is memory hungry and a bit slow, you need to have a good reason to use recursion.
- In computer science- recursion is generally supported for readability, understandability and clean code.

Classification of Recursion

- Tail vs. Non-Tail
- Direct vs. Indirect vs. Nested
- Simple vs. Excessive
- Nested Recursion

Tail Recursion

- Tail Recursion is characterized by the use of only one recursive call at the very end of a function implementation.
- This means when a recursive call is invoked there is no work left for the previous call.
- Moreover, there is no pending recursive calls, either direct or indirect.

```
void tail(int i) {
  if (i > 0) {
    cout << i << '';
    tail(i-1);
  }
}
```

```
void nonTail(int i) {
  if (i > 0) {
    nonTail(i-1);
    cout << i << '';
    nonTail(i-1);
  }
}
```

Non-Tail Recursion

- Non-Tail recursion is characterized by the fact that there is still some work left for the first call and a recursive self-call is invoked.

```
/* 200 */ void reverse() {
              char ch;
/* 201 */     cin.get(ch);
/* 202 */     if (ch != '\n') {
/* 203 */         reverse();
/* 204 */         cout.put(ch);
              }
          }
```

```
void simpleIterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.getline(stack,80);
  for (top = strlen(stack) - 1; top >= 0; cout.put(stack[top--]));
}
```

Indirect Recursion

- Indirect Recursion is characterized by two of more distinct functions repeatedly call each other. Hence a recursive phenomena is there.

```
int Fun1(int & n)
{
    if (n <= 20) // do not invoke Fun 2 once n reaches 20
    {
     cout<< n << endl;
     n++;
     Fun2(n);
    }
    else return 0;
}
```

```
int Fun2(int & n)
{
    if (n <= 20) // do not invoke Fun1 once n reaches 20
    {
     cout<< n << endl;
     n++;
     Fun1(n);
    }
    else return 0;
}
```

Excessive Recursion

- Generally while solving a maze, the allowable movements are left, right, up, down (diagonal also allowed), 4 (8) in number.

```
with Start row, col
        FindPath(maze,row, col+1);
        FindPath(maze,row, col-1);
        FindPath(maze,row-1, col);
        FindPath(maze,row+1, col);
```

Nested Recursion

- When a recursive function call is embedded into a recursive function, it is called “Nested Recursion”
- Example of nested recursion is Ackerman function:

A(m,n) = n + 1	**if m == 0**
A(m,n) = A(m-1, 1)	**if m > 0 && n == 0**
A(m,n) = A(m-1, A(m, n-1))	**if m > 0 && n > 0**

  - For example, A(4, 2) is an integer of 19,729 decimal digits

Example

```
/*****************************************************************************
* Author: Muhammad Rafi                                                      *
* Purpose: Recursion Factorial (Examples)                                    *
* Dated: September 22, 2007                                                  *
* Version: 1.0                                                               *
* Last modified: October 10, 2007                                            *
*****************************************************************************/

#include <iostream>

using namespace std;

  int factorial(int n)
  {
    int return_value;
    cout << "Factorial(" << n << ")\n";
    if (n == 0) {
      return_value = 1;
    } else {
      return_value = n * factorial(n-1);
    }
    cout << "Factorial(" << n << ") is returning " << return_value << endl;
    return return_value;
  }

int main()
  {
    int n;

    cout << "Enter a non-negative integer: ";
    cin >> n;
    cout << "Factorial of " << n << " is " << factorial(n) << endl;
    return 0;
  }
```

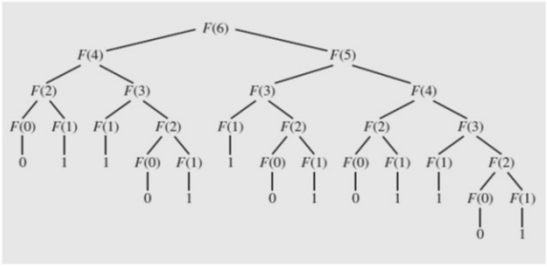
Example

```
#include<iostream>
#include <time.h>       /* clock_t, clock, CLOCKS_PER_SEC */
#include <math.h>       /* sqrt */

using namespace std;

int fib(int n, int next, int result)
{
    cout << "Value for n:  "  << n <<" Value for next :   "  <<  next << " Value for result :  " << result  << endl;
    if (n==0) return result;
    return (fib(n-1, next+result, next));
}

 int main(){

    int n=3;
    int first=1,second=0;

    clock_t t;
    t = clock();
    int result = fib(6,first,second);
    t = clock() - t;
    printf ("It took me %d clicks (%f seconds).\n",t,((float)t)/CLOCKS_PER_SEC);
    cout<< result << endl;

    return 0;
 }
```

Fibonacci Workload

```
int main(){

   int n=3;
   int first=1,second=0;

   clock_t t;
   t = clock();
   int result = fib(6,first,second);
   t = clock() - t;
   printf ("It took me %d clicks (%f seconds).\n",t,((float)t)/CLOCKS_PER_SEC);
   cout<< result << endl;

   return 0;
}
```

The tree of calls for `Fib(6)`.



Comparison

Number of addition operations and number of recursive calls to calculate Fibonacci numbers.

n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

Comparison

Comparison of iterative and recursive algorithms for calculating Fibonacci numbers.

		Assignments	
n	Number of Additions	Iterative Algorithm	Recursive Algorithm
6	5	15	25
10	9	27	177
15	14	42	1,973
20	19	57	21,891
25	24	72	242,785
30	29	87	2,692,537

Fibonacci Memorization

FibMemoization.cpp

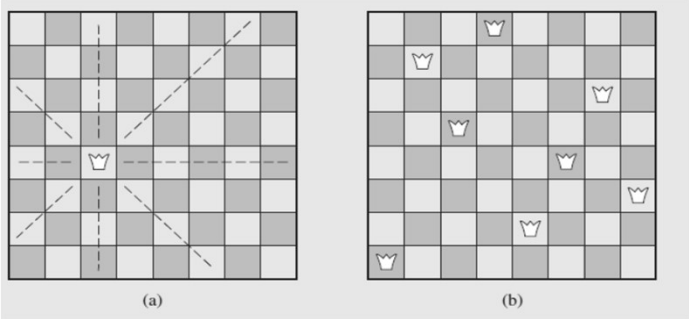
```
/*****************************************************************************
* Author: Muhammad Rafi                                                      *
* Purpose: Recursion Fibonacci with memoization (Examples)                   *
* Dated: October 02, 2007                                                    *
* Version: 1.0                                                               *
* Last modified: October 11, 2007                                            *
*****************************************************************************/


#include <iostream>

using namespace std;

#define MAX 100

long int lookup_table[MAX]={0};

int fib_mem(int n){

 if (lookup_table[n]==0)
   {
      cout << "Running Fib for n="   << n << endl;
       if (n <=1)
       lookup_table[n]=n;
       else
       {
       cout << "Completing Recursive Function for Fib(" << n << ")" <<endl;
       lookup_table[n]= fib_mem(n-2)+ fib_mem(n-1);
       }
   }
   cout << "Retrun Value is "  << lookup_table[n] <<endl;
   return lookup_table[n];
}

```

N-Queen Problem

- N-Queen Problem
  - Place eight queens on the chessboard so that no queen can attack any other queen
  - Legal move – attacking means no 2 queens in same row, col, and diagonal
  - Idea of the size of the problem:
    - There are **4,426,165,368** ways to arrange 8 queens on a chessboard of 64 squares.
    - No queen can reside in a row or a column that contains another queen
    - only **40,320** (8!) arrangements of queens to be checked for attacks along diagonals

N-Queen Problem

- For an N-Queen there are N possible arrangements that are not attacking.
- Finding 8 solutions from 4,426,165,368



Backtracking

- Backtracking
  - A strategy for guessing at a solution and backing up when an impasse is reached
- Recursion and backtracking can be combined to solve problems.

```
putQueen(row)
    for every position col on the same row
      if position col is available
          place the next queen in position col;
          if (row < 8)
            putQueen(row+1);
          else success;
          remove the queen from position col;
```

Backtracking

```
void ChessBoard::initializeBoard() {
    register int i;
    column = new bool[squares];
    positionInRow = new int[squares];
    leftDiagonal  = new bool[squares*2 - 1];
    rightDiagonal = new bool[squares*2 - 1];
    for (i = 0; i < squares; i++)
        positionInRow[i] = -1;
    for (i = 0; i < squares; i++)
        column[i] = available;
    for (i = 0; i < squares*2 - 1; i++)
        leftDiagonal[i] = rightDiagonal[i] = available;
    howMany = 0;
}
```

Backtracking

```
class ChessBoard {
public:
    ChessBoard();    // 8 x 8 chessboard;
    ChessBoard(int); // n x n chessboard;
    void findSolutions();
private:
    const bool available;
    const int squares, norm;
    bool *column, *leftDiagonal, *rightDiagonal;
    int  *positionInRow, howMany;
    void putQueen(int);
    void printBoard(ostream&);
    void initializeBoard();
};

ChessBoard::ChessBoard() : available(true), squares(8), norm(squares-1)
{
    initializeBoard();
}
ChessBoard::ChessBoard(int n) : available(true), squares(n),
norm(squares-1) {
    initializeBoard();
}
```

Backtracking

```
void ChessBoard::putQueen(int row) {
    for (int col = 0; col < squares; col++)
        if (column[col] == available &&
            leftDiagonal [row+col] == available &&
            rightDiagonal[row-col+norm] == available) {
            positionInRow[row] = col;
            column[col] = !available;
            leftDiagonal[row+col] = !available;
            rightDiagonal[row-col+norm] = !available;
            if (row < squares-1)
                 putQueen(row+1);
            else printBoard(cout);
            column[col] = available;
            leftDiagonal[row+col] = available;
            rightDiagonal[row-col+norm] = available;
        }
}
```

Conclusion

- Recursion is a programming tool. Like any other topic in data structures, it should be used with good judgment.
- Recursive solutions are simplified one for understanding, short and precise code, readability and maintainability are increases.
- Sometimes, slow and hard on resources. Specifically available on only compilers that support recursion.
- Sometimes recursion is faster than iterative approach. Running a simple routine implemented recursively and iteratively and comparing the two run times can help to decide if recursion is advisable
- Caution!!!
  - if you do not understand how recursion works, avoid using it!