Design and Analysis of Algorithms

Lecture 5: Principles of Recursive Program Design

Material is from Chapter 3: Kruse's book

Designing Recursive Algorithms

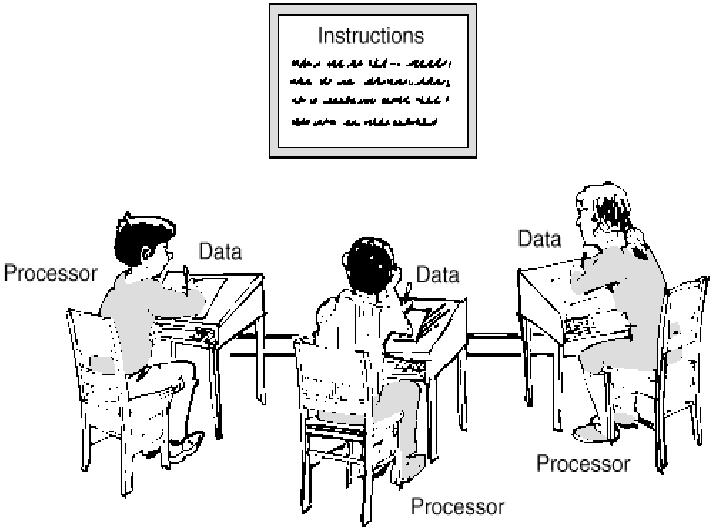
- Find the key step
  - How can a problem be divided into parts?
  - How will be the key step executed?
  - Avoid ending with multitude of special cases.
- Find a stopping rule
  - Stopping rule is the smallest case that is solved without a recursion
- Outline your algorithm
  - Combine the stopping rule and the key step, using if statement to select between them

Designing Recursive Algorithms (cont.)

- Check termination
  - Verify that the recursion will always terminate
  - Be sure that your algorithm will handle extreme cases
- Draw a recursion tree
  - The height of the tree will represent the amount of memory that the program will require
  - The number of nodes in the tree will represent the number of times the key step will be executed

Implementing Recursion in a Program

- Implementation is separate from design
  - Implementation can be done in any language that supports recursion
- Multiprocessor solution
  - Processes that take place simultaneously are called **concurrent**
- Single processor solution
  - Uses multiple storage areas with a single processor
- Re-entrant programs

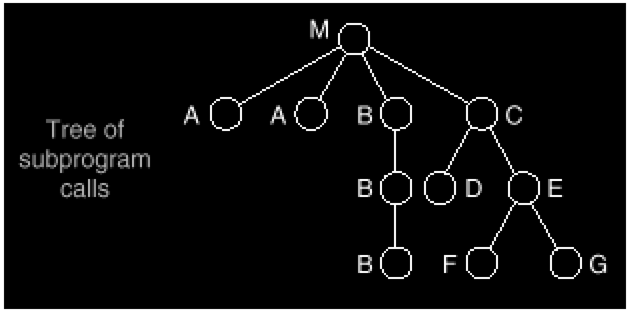


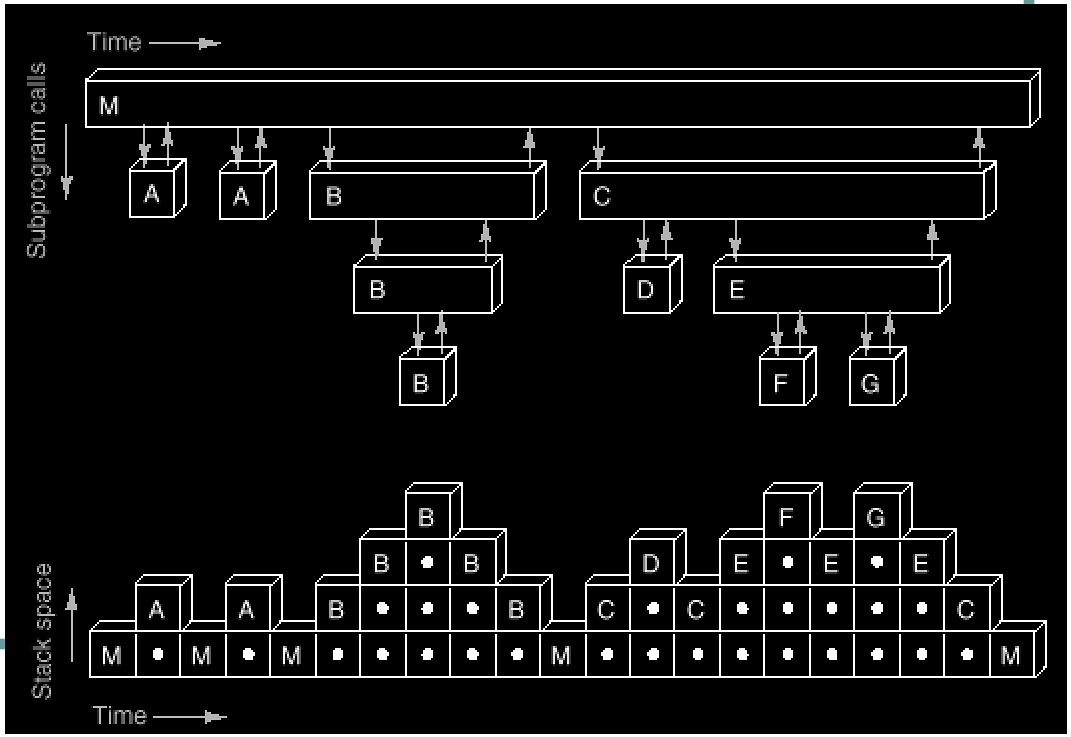
Improving Recursive Programs:

- Tail Recursion

Data Structures for Recursion:

- Stacks and Trees

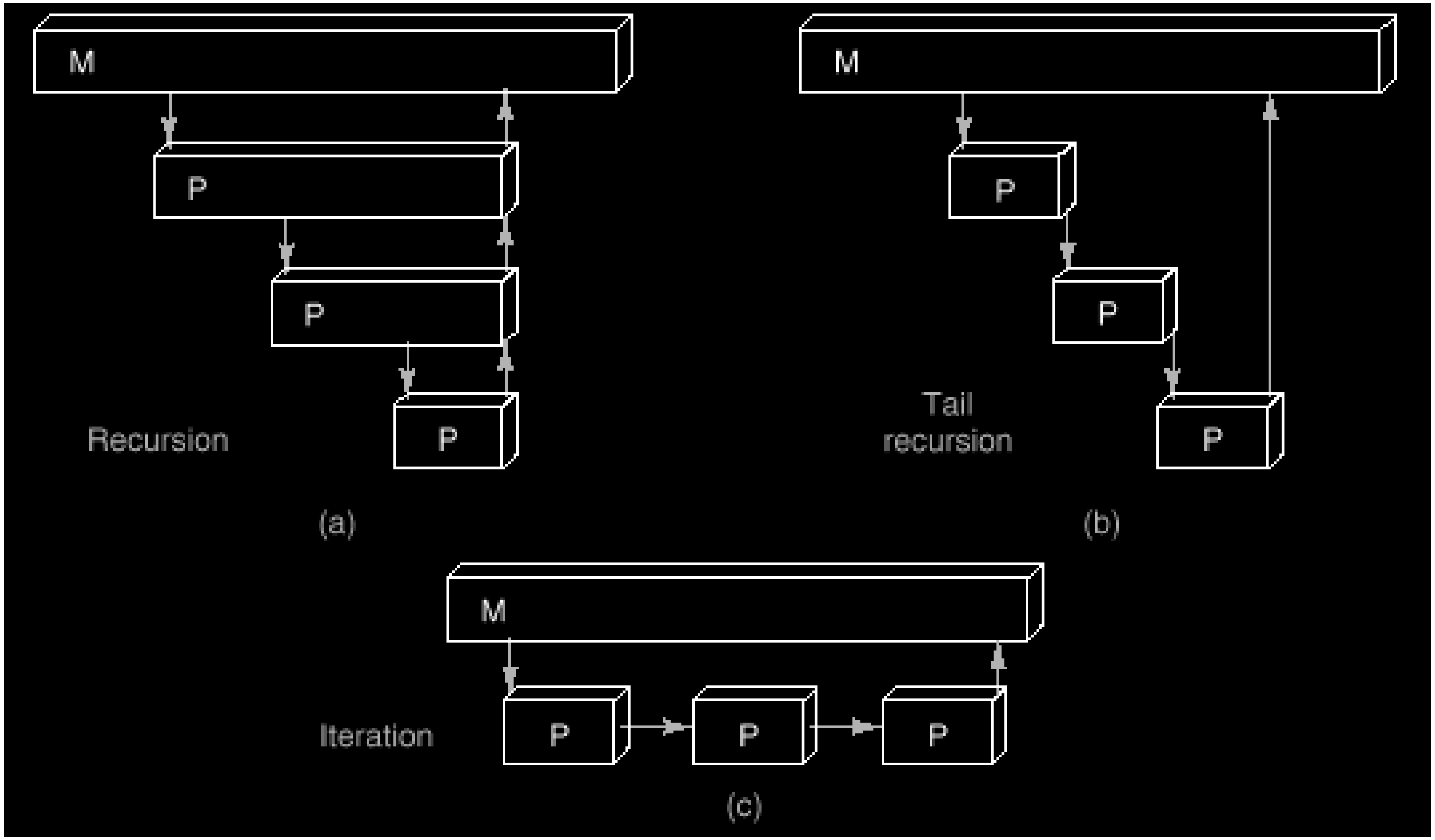




Tail Recursion

- **Def:** Tail recursion is a case when the **last-executed** statement of a function is a recursive call to itself
- In case of tail recursion last call can be eliminated by reassigning the calling parameters to the values specified in the recursive call, and then repeating the whole function

Tail Recursion: Diagram



Can the Tower of Hanoi be simplified?

```
/* Move: moves count disks from start to finish using temp
for temporary storage. */
void Move(int count, int start, int finish, int temp)
{
    if (count > 0)
    {
       Move(count-1, start, temp, finish);
       printf("Move a disk from %d to %d.\n", start, finish);
       Move(count-1, temp, finish, start);
    }
}
```

New Tower Of Hanoi

```
void Move(int count, int start, int finish, int temp)
{
  int swap; /* temporary storage to swap towers */
  while (count > 0)
  {
      Move(count - 1, start, temp, finish);
      printf("Move %d from %d to %d.\n", count, start, finish);
      count--;
      swap = start;
      start = temp;
      temp = swap;
  }
}
```

Use Recursion or Not? Factorial

RECURSION

```
/* Factorial: recursive version.*/
int Factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}
```

Which one will be more efficient?

INTERATION

```
/* Function: iterative version.*/
int Factorial(int n)
{
    int count, product;
     for (product = 1, count = 2; count <= n; count++)
        product *= count;

   return product;
}
```

Fibonacci Numbers

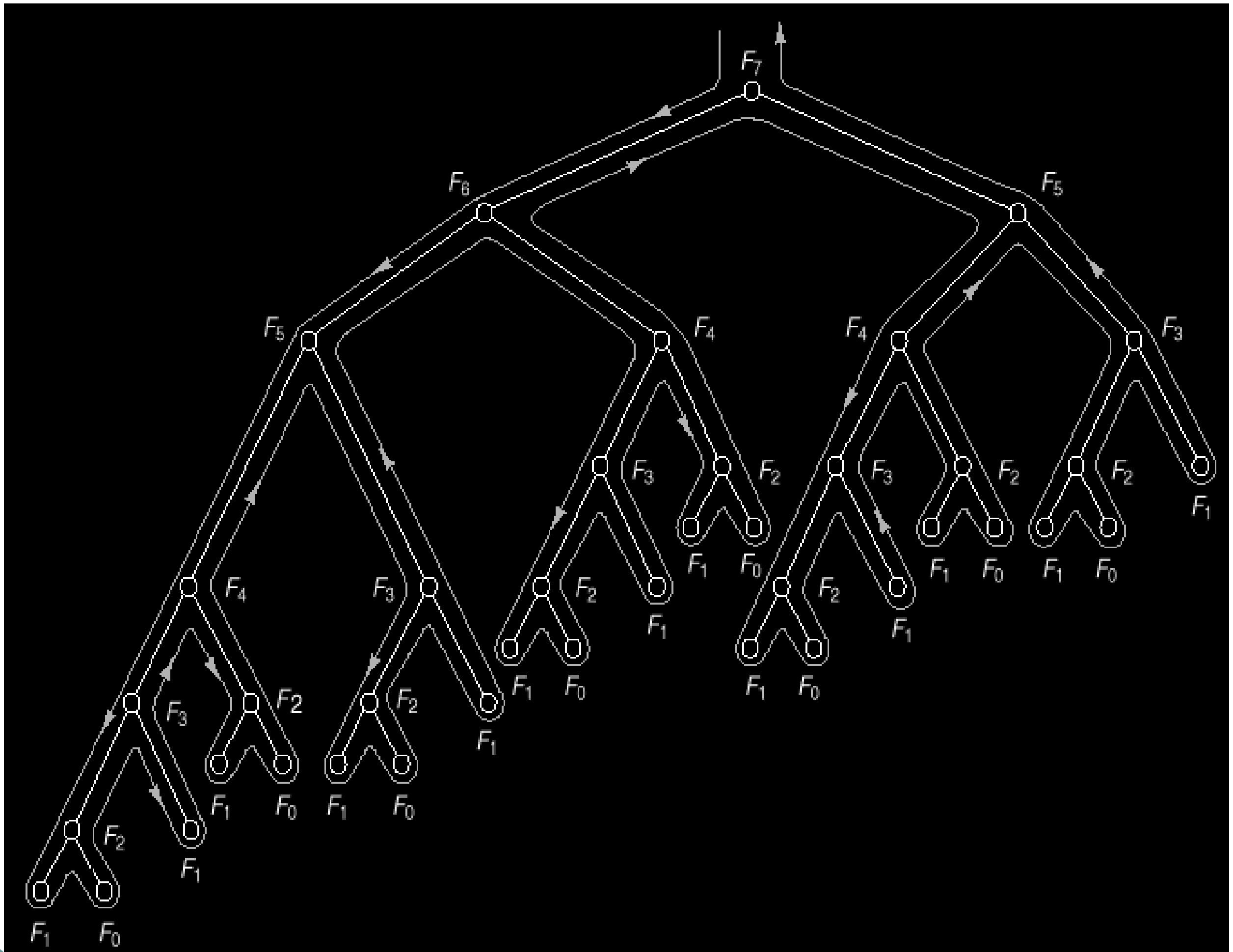
- Fibonacci Number
  - $F_0=0$
  - $F_1=1$
  - $F_n=F_{n-1}+F_{n-1}$ when $n\geq2$

Recursive Solution

```
int Fibonacci(int n)
{
    if (n <= 0)
       return 0;
    else if (n == 1)
       return 1;
       else
          return Fibonacci(n-1) + Fibonacci(n-2);
}
```

How efficient is this solution?

Fibonacci Recursion Tree



What can we say about the running time of this program?

Fibonacci: Interative solution

```
int Fibonacci(int n)
{
        int i;
        int twoback; /* second previous number, F_i-2 */
        int oneback; /* previous number, F_i-1 */
        int current; /* current number, F_i */
        if (n <= 0)
            return 0;
        else
           if (n == 1)
                     return 1;
          else
          {
              twoback = 0;
              oneback = 1;
              for (i = 2; i <= n; i++)
              {
                  current = twoback + oneback;
                  twoback = oneback;
                  oneback = current;
              }
              return current;
          }
}
```

Guidelines for Applying Recursion

- If the recursion tree has a simple form (chain), the iterative version may be better.
- If the recursion tree involves duplicate tasks, then data structures other than stacks might be more appropriate (in general case something like dynamic programming technique would be better).
- If the recursion tree is evenly “bushy”, with little duplicate tasks, then recursion is likely the natural solution.