Recursion

Definition

Recursion: defining the solution to a problem in terms of a simpler version of the same problem. It must eventually end in a version which does not contain any recursion at all and then immediately returns an answer through the preceding levels of recursive calls.

Example

The factorial function $f(n) = n!$ can be defined recursively as

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

C++ Implementation of the Factorial Function.

```
long factorial(int n) {
   if (n == 0) return 1;
   else return n*factorial(n-1);
}
```

The essence of the recursion is to recognize that the solution to a problem is stated in terms of the solution to the same problem at a simpler (or previous) level. Here `factorial(n-1)` is simpler than `factorial(n)` and `factorial(0)` is at the simplest (base) level where recursion is not used (because the result is 1).

How does the Recursion Work?

If a function has formal parameters, they have to be initialized to the values passed as actual parameters (arguments). The operating system has to know where to resume execution of the program after the function has finished. The function can be called by other functions or by the main program (`main()`). The information indicating where the function has been called from has to be remembered by the system. For a function call, more information has to be stored than just a return address. Therefore, this information is stored dynamically using the **run-time stack**).

A Stack Frame

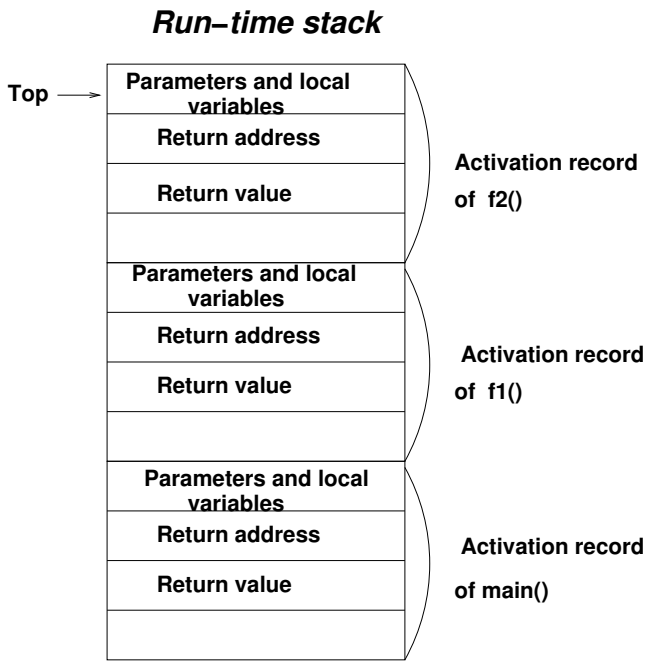
At least the following information is stored when a function is called (this is system-dependent):

1. values for all formal parameters of the function
2. local variables (which can be stored elsewhere, but then pointers to them must be stored)
3. the return address indicating where to restart its caller—the address of the caller's instruction immediately following the call
4. a returned value for a function (when not declared as `void`)

This set of information is called an *activation record* (or *stack frame*) and is allocated on the run-time stack. An activation record exists for as long as a function owning it has not completed its execution. Then it is removed from the stack by the system.

Function Calls and Run-Time Stack (cont.)

Creating an activation record whenever a function is called allows the system to handle recursion properly. Recursion means calling a function of the same name as the caller with the same formal arguments. Recursive callings are represented internally by different activation records and are thus differentiated by the system.




```
f2() {
  ...
}
f1() {
  ...
  f2();
...
}
```

```
main()
{
  ...
  f1();
  ...
}
```

Illustration of Recursive Calls

We want to compute `factorial(4)`. Below are the functions calls and the return values:

```
factorial(4)
  factorial(3)   <-- is called by factorial(4)
    factorial(2)   <-- is called by factorial(3)
      factorial(1)  <-- is called by factorial(2
        factorial(0)  <-- is called by factorial
        1               <-- returned by factorial(
      1*1 = 1         <-- returned by factorial(1)
    2*1 = 2         <-- returned by factorial(2)
  3*2 = 6         <-- returned by factorial(3)
4*6 = 24        <-- returned by factorial(4)
```

The Iterative Factorial()

The function `factorial(n)` can be implemented without using any recursion:

```
long factorialNonRecur(int n) {
   int result = 1;
   for (int k = n; k > 0; k--)
      result *= k;
   return result;
}
```

Is any gain by using recursion instead of a loop? The recursive version seems to be more intuitive since it is similar to the original definition of the factorial function. The recursive version increases program readability, improves self-documentation, and simplifies coding. In the above example the non-recursive version is not larger than the recursive version, but for most recursive implementations code is shorter than it is in the non-recursive version.

Tail Recursion

Tail recursion is characterized by the use of only one recursive call at the end of a function. When the call is made, there are no statements left to be executed by the function; the recursive call is the last statement,

```
void tail(int n)
{
 if (n > 0) {
 cout << n << " ";
 tail(n - 1);
} else
 cout << endl;
}
```

Calling `tail(15)` in the main function gives the following output sequence:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

The Iterative Version (cont.)

The tail function can be rewritten using the non-recursive approach:

```
    void tailNonRecur(int n)

{
 for (int k = n; k > 0; k--)
cout << k << " ";
 cout << endl;
}
```

In most languages (such as C++ or Java) there is no any advantage in using tail recursion over iteration, and in such languages tail recursion is not a recommendable feature. In some other languages (Prolog, Lisp), which have no explicit loop constructs, tail recursion is important.

Reversing an Array

Example

The tail recursion

```
Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at inde
          and ending at j
if i < j then
   Swap A[i] and A[ j]
ReverseArray(A,i+1,j-1)
return
```

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal function as ReverseArray(A, i, j), not ReverseArray(A).

A Good Fibonacci Algorithm

```
Algorithm: LinearFibonacci(k)
Input: A nonnegative integer k
Output: A pair of Fibonacci numbers (F(k),F(k-1))
if k=1 then
     return (k,0)
else (i,j)= LinearFibonacci(k-1)
     return (i+j,i)
```

Computing Power Function

The power function, $p(x, n) = x^n$, can be defined recursively:

$$p(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{if } n > 0 \end{cases}$$

This leads to an power function that runs in linear, $O(n)$ time (for we make n recursive calls).

Recursive Squaring

We can design a more efficient linearly recursive algorithm by using repeated squaring for $n = 1$ return 1 $n > 0$

$$p(x,n) = \begin{cases} x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0,\ n \text{ odd} \\ p(x,n/2)^2 & \text{if } x > 0,\ n \text{ even} \\ 1 & \text{if } x > 0,\ n = 0 \end{cases}$$

For example,
$p(x,4) = x^4 = x^{(4/2)\cdot 2} = (x^{4/2})^2$
$p(y,5) = y^5 = y^{1+(4/2)\cdot 2} = y(y^2)^2$

Recursive Squaring Method

```
Algorithm Power(x,n):
Input: A number x and integer n=0
Output: The value x^n
if n=0 then
   return 1
if n is odd then
   y=Power(x,(n-1)/2)
   return x*y*y
else
   y=Power(x,n/2)
   return y*y
```

- Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls.
- That is, this method runs in O(log n) time.
- It is important that we use a variable twice here rather than calling the method twice.

Non-Tail Recursion

```
void nonTailRecur(int n)
{
if (n > 0) {
 nonTailRecur(n-1);
cout << n << " ";
 nonTailRecur(n-1);
 } else
 cout << endl;
}
```

NP(n) = NP(n-1)+NP(n-1)+1

NP(n) = 2*NP(n-1)+1

NP(0) = 1

The output of `nonTailRecur(4)`:

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 (each number is displayed in a column)

Non-Tail Recursion (cont.)

```
static char hexDigits[] = {'0', '1', '2', '3', '4', '5',
 '6', '7', '8', '9', 'A',
 'B', 'C', 'D', 'E', 'F'};
void printHexadecimal(int n)
{ if (n >= 16)
 printHexadecimal(n/16);
 cout << hexDigits[n % 16];
}
```

It prints out hexadecimal digits of a nonnegative number (given as an argument). This code is easy to write using recursion. But to write a non-recursive version of this function is more difficult. The hexadecimal (base 16) numeral system is usually written using the symbols from 0 to 9 and additionally A to F which represent digits 10 to 15. EXAMPLE: the decimal number 1234 is represented as 4D2 in hexadecimal. In C++, the prefix 0x is used to represent hexadecimal numbers.

Other Types of Recursion

Indirect recursion is when a function $f()$ calls a function $g()$ and the function $g()$ calls the function $f()$.
Nested recursion is when a function not only calls itself but the function call is also used as one of the parameters. As an example we can provide the *Ackermann function*:

$$A(i,j) = \begin{cases} j+1 & \text{if } i = 1, j \geq 1 \\ A(i-1,2) & \text{if } i \geq 2, j = 1 \\ A(i-1, A(i, j-1)) & \text{if } i, j \geq 2 \end{cases}$$

We are not going to discuss further these kinds of recursion.

Writing and Verifying Recursive Algorithms

When coding the solution as a recursive function the following conditions must be satisfied:

1. There must be at least one simple (base) case of the problem being solved that does not require recursion. Such cases will normally lead directly to a completion of the function and a return to the point at which it was invoked.
   EXAMPLE: in the factorial function the base case is $n = 0$.
2. Any recursive use of the function must move the problem closer to one of the base cases. This step make have a test that decides which of several possible recursive calls to make and ultimately make just one of these calls.
   EXAMPLE: in the factorial function the recursive call is `factorial(n-1)` which is one step closer than the original call `factorial(n)`

Recursive Algorithms (cont.)

Verifying the correctness of a recursive function is equivalent to verifying the following conditions:

1. The function contains (non-recursive) base cases.
2. All recursive calls of the function involve a case of the problem that is closer to one of the base cases.
3. The recursion properly implements the problem to be solved.

Binary Recursion

Binary recursion occurs whenever there are two recursive calls for each non-base case.

Example

Computing Fibonacci Numbers

Fibonacci numbers are defined recursively:
$F_0 = 0$ $F_1 = 1$ $F_i = F_{i-1} + F_{i-2}$ for i>1.
Recursive algorithm (first attempt):

```
Algorithm BinaryFib(k):
Input: Nonnegative integer k
Output: The kth Fibonacci number Fk
if k = 1 then
  return k
else
  return BinaryFib(k - 1) + BinaryFib(k - 2)
```

Analysis

The recurrence formula for the Fibonacci algorithm:

$$T(n) = T(n-1) + T(n-2) + 1$$

$T(0) = T(1) = 1$

Notice that: $T(n-1) = T(n-2) + T(n-3) + 1$

$$T(n) = 2T(n-2) + T(n-3) + 2$$

Not that: $T(n-2) = T(n-3) + T(n-4) + 1$

$$T(n) = 3T(n-3) + 2T(n-4) + 4$$

Not that: $T(n-3) = T(n-5) + T(n-4) + 1$

$$T(n) = 5T(n-4) + 3T(n-5) + 7$$

Not that: $F_4 = 5$, $F_3 = 3$, and $F_5 - 1$=7. Using this observation we can write:

$$T(n) = F_{n-1}T(1) + F_{n-2}T(0) + F_n - 1 = 2F_n - 1$$

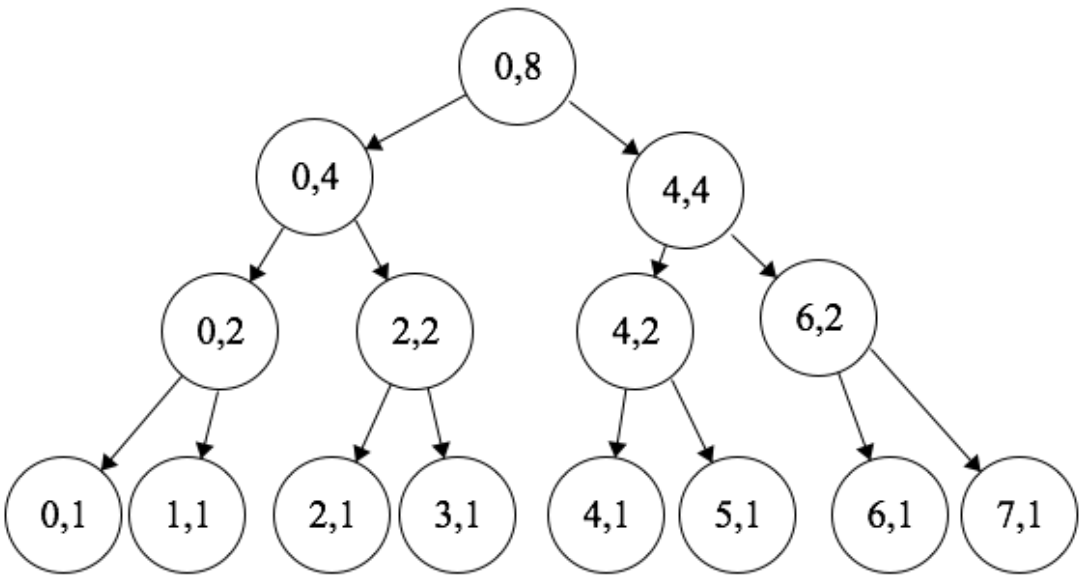
A closed form is given by:

$$T(n) = 2\frac{\Phi^n - \hat{\Phi}^n}{\sqrt{5}} - 1 = \frac{\Phi^n}{\sqrt{5}}, \text{ when } \Phi = \frac{1+\sqrt{5}}{2} = 1.61$$

Binary Recursion (cont.)

```
Problem: add all the numbers in an integer array A:
Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
  if n = 1 then
    return A[i]
return BinarySum(A,i,n/2)+BinarySum(A,i+n/2,n/2)
```

A trace Tree



Quick Sort

Quick sort is a sorting algorithm based on recurrence. To describe it, assume that we sort a subarray $A[p..r]$. The notation $A[p..r]$ stands for an array with elements $A[p], A[p+1], \ldots, A[r]$. These are two basic steps:

- Divide the array $A[p..r]$ into two nonempty subarrays $A[p..q]$ and $A[(q+1)..r]$ such that each element of $A[p..q]$ is less than or equal to each element of $A[(q+1)..r]$. The element q (called the *pivot*) is computed by the algorithm.
- Recursively call the Quick sort to sort $A[p..q]$ and $A[(q+1)..r]$.
- The base case is when there is one element to be sorted (which is returned as is because it is already sorted).

We show a C++ implementation of this algorithm as the function `quicksort`.

The Quick Sort Algorithm

```
int main(void)
{
   int A[100];
   ... // initialize the array A
   quicksort(A, 100);
   ... // use the sorted array A
}
void quicksort(int A[], int n) {
   quicksort(A, 0, n);
}
void quicksort(int A[], int low, int high)
{
   if (low < high) {
      int middle = qsPartition(A, low, high);
      quicksort(A, low, middle);
      quicksort(A, middle+1, high);
   }
}
```

The Quick Sort Partition

```
int qsPartition(int A[], int low, int high)
{
 int pivot = A[low];
 int i = low-1, j = high+1;
 while (true) {
do { j--;
 } while (pivot < A[j]);
 do { i++;
 } while (A[i] < pivot);
 if (i < j) { //swap A[i] and A[j]
 int temp = A[i];
 A[i] = A[j];
 A[j] = temp;
} else return j;
 }
}
```

Efficiency of Quick Sort

The average run-time efficiency of the quick sort is $O(n \log_2 n)$, where n is the number of elements in an array to be sorted. The best-case efficiency is $O(n \log_2 n)$. The worst-case efficiency is $O(n^2)$. The worst case for the algorithm is when the array is already sorted. The worst-case situation depends on the choice of the pivotal element (`pivot`).

Merge Sort

Merge sort uses recursion to sort an array of numbers. The merge sort algorithm can be described as follows:

- Divide an n-element input sequence to be sorted into two subsequences of $n/2$ elements each.
- Sort the two subsequences recursively using merge sort.
- The base case is when the subsequence contains one or zero elements (which are already sorted).
- Merge the two sorted subsequences to produce the sorted sequence.

The function `mergeSort` is a simple driver that declares a temporary array and calls the recursive function `mergeSort`.

Using Merge Sort

```
int main(void) {
 int A[100];
 ...  // initialize A
 mergeSort(A, 100);
 ...  // use the sorted A
}
void mergeSort(int A[], int n) {
 int *copyA = new int[n];
for (int i = 0; i < n; i++)
 copyA[i] = A[i];
 mergeSort(copyA, A, 0, n-1);
delete [] copyA;

    }
```

The Merge Sort Algorithm

Here `left` $\leq$ `right`, and they denote lower and upper indices of the subarray to be sorted. We need a copy of the array `A` called `copyA`. The sorting itself is done in the function `merge`.

```
void mergeSort(int A[], int copyA[], int left, int
right)
{
 if (left < right) {
 int center = (left + right)/2;
 mergeSort(copyA, A, left, center);
 mergeSort(copyA, A, center+1, right);
 merge(A, copyA, left, center+1, right);
 }
}
```

The Merge Function (cont.)

```
void merge(int A[], int copyA[], int leftPos, int
rightPos,
 int rightEnd)
{
 int leftEnd = rightPos - 1;
 int j = leftPos; // index for copyA
 while (leftPos <= leftEnd && rightPos <= rightEnd)
 if ( A[leftPos] < A[rightPos] )
 copyA[j++] = A[leftPos++];
else
 copyA[j++] = A[rightPos++];
 // copy the rest
 while (leftPos <= leftEnd)
 copyA[j++] = A[leftPos++];
 while (rightPos <= rightEnd)
 copyA[j++] = A[rightPos++];
}
```

Merge Sort Efficiency

The merge sort requires $O(n \log_2 n)$ comparisons, where n is the length of an array to be sorted. There are no best-case or worst-case for the merge sort. Because the merge sort needs a duplicate copy of the array being sorted therefore in many situations it can be impractical.