BM267 - Introduction to Data Structures

5. Recursion

Ankara University

Computer Engineering Department

Objectives

Learn about

- Recursion
- Divide and conquer
- General and binary tree structures
- Implementation of trees
- Mathematical properties of trees.
- Tree operations, tree traversal algorithms

Recursion

- A **recursive definition** is one which uses the word or concept being defined in the definition itself

- Consider the following list of numbers:

24, 88, 40

- Such a list can be defined on paper as

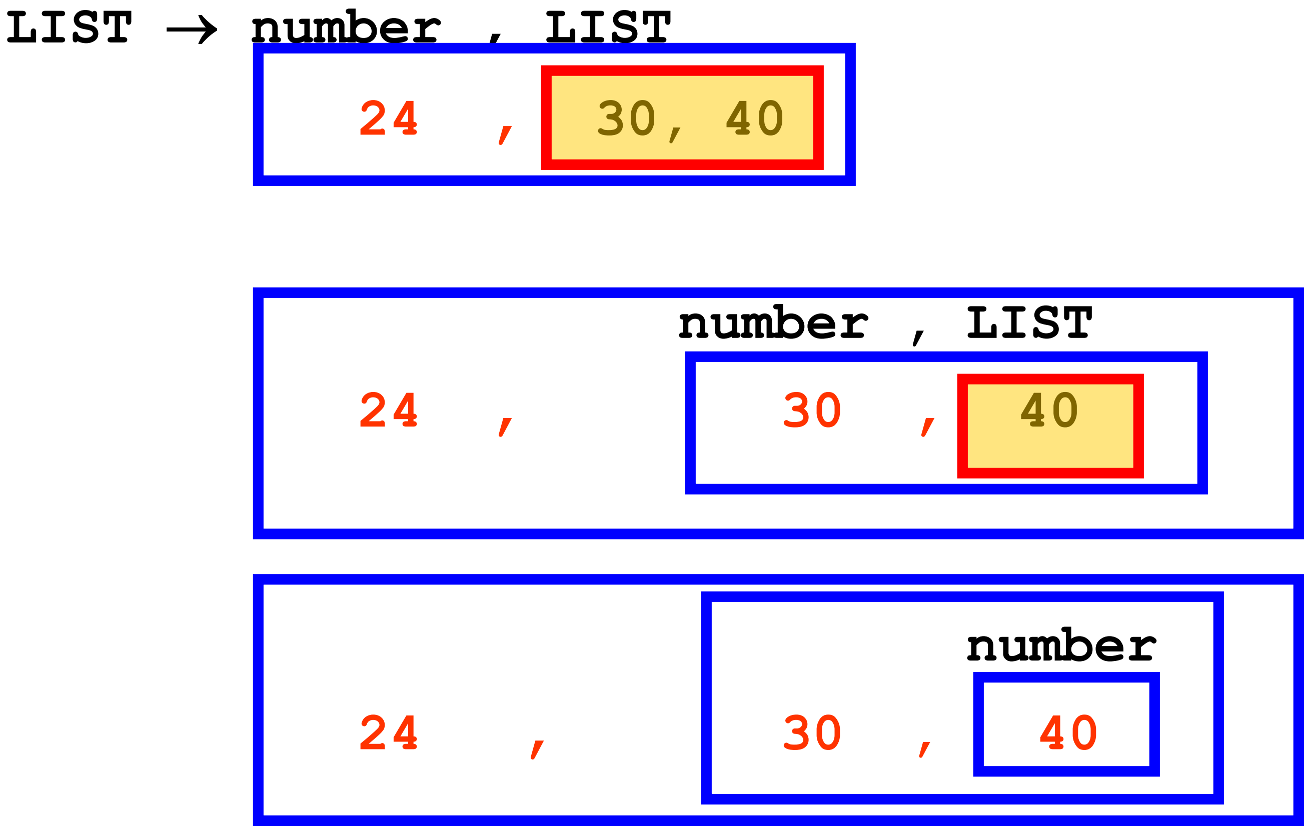
```
A LIST is a:  number
        or a:  number  comma  LIST
```

- That is, a LIST is defined to be a single number,
- Or a number followed by a comma followed by another LIST

- The concept of LIST is used to define itself.

Recursion

- If you apply this definition to the actual list of numbers, the recursive part of the LIST definition is used several times, terminating with the non-recursive part:



Recursion

- All recursive definitions have to have a terminating case

  A LIST is:

either	**number, followed by a comma, followed by another LIST**
or	**number**

- Otherwise, there would be no way to terminate the recursive path

- Such a definition would cause **infinite recursion**

- This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself

- The non-recursive part is often called the *base case*

Recursive functions

- Recursion simply means a function that calls itself.
- The conditions that cause a function to call itself again are called the *recursive case.*
- In order to keep the recursion from going on forever, you must make sure you hit a termination condition called the *base case*.
- The number of nested invocations is called the *depth of recursion.*
- Function may call itself *directly* or *indirectly*. (All of our examples are direct.)

Recursive functions

- Recursive functions must satisfy two basic properties:
  - They must explicitly solve a base case.
  - Each recursive call must involve smaller values of the argument.

Euclid: Greatest common divisor

```
int gcd(int m, int n)
{
  if (n == 0)
    return m;
  return gcd(n, m % n);
}
```

Recursive functions

```
int puzzle(int N)
{
   if (N = = 1)
      return 1;
   if (N % 2 == 0)
      return puzzle(N/2);
   else
      return puzzle(3*N+1);
}
```

Here, we cannot use induction to prove that this program terminates, because not every recursive call uses an argument smaller than the one given.

Recursive functions

Linked list node count:

```
int count(link x)
{
  if (x == NULL)
    return 0;
  return 1 + count(x->next);
}
```

Recursive functions

Factorial function

- N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive
- This definition can be expressed recursively as:
- $0! = 1$
- $1! = 1$
- $N! = N * (N-1)!$
- The concept of the factorial is defined in terms of another factorial
- Eventually, the base case of 1! is reached.

Recursive functions

- Recursion and looping has similar meanings.
- Loop termination condition has the same role as a recursive base case.
- A loop's control variable serves the same role as a general case.

```
sum = 0;
i = 1;
while(i <= 10)
{
   sum += i;
   i++;
}
```

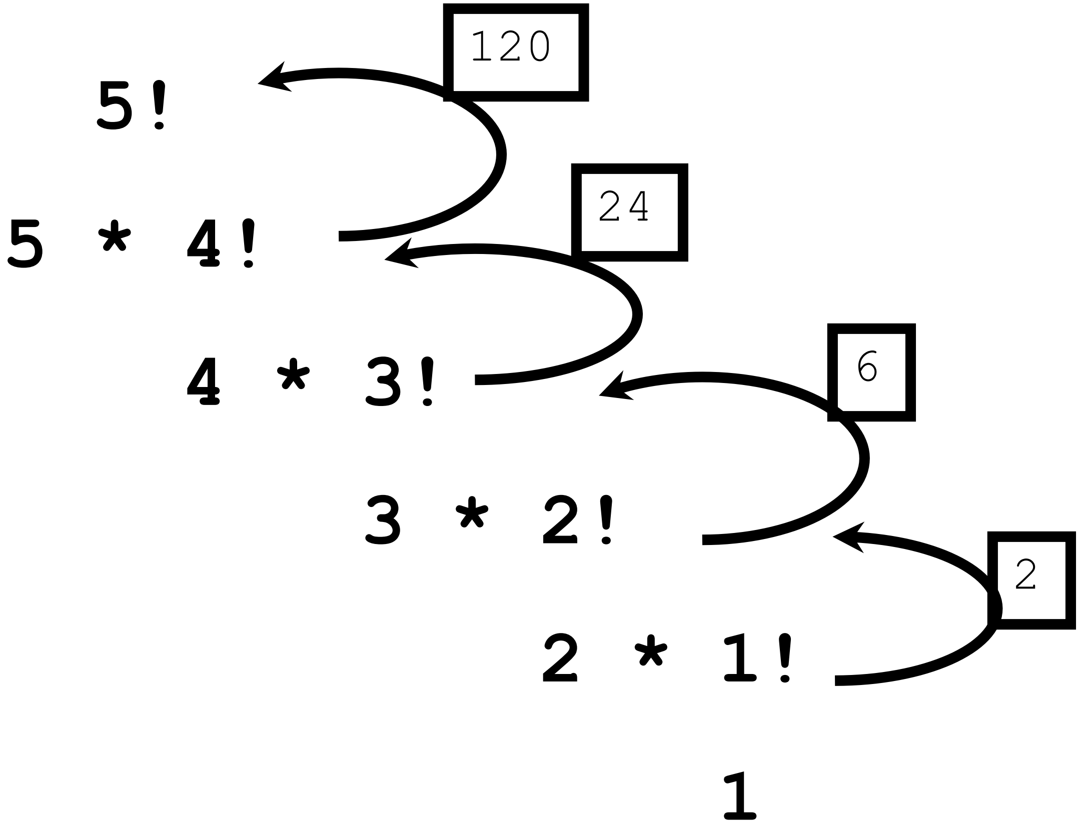
Termination Condition

```
int Factorial(int n)
{
   if (n == 0)     (base case)
    return 1;
   else        ( n>0, recursive case)
    return n*Factorial(n-1);
}
```

Loop control and recursive case both move toward termination condition

Recursive functions

Function call Factorial(5) proceeds as below:



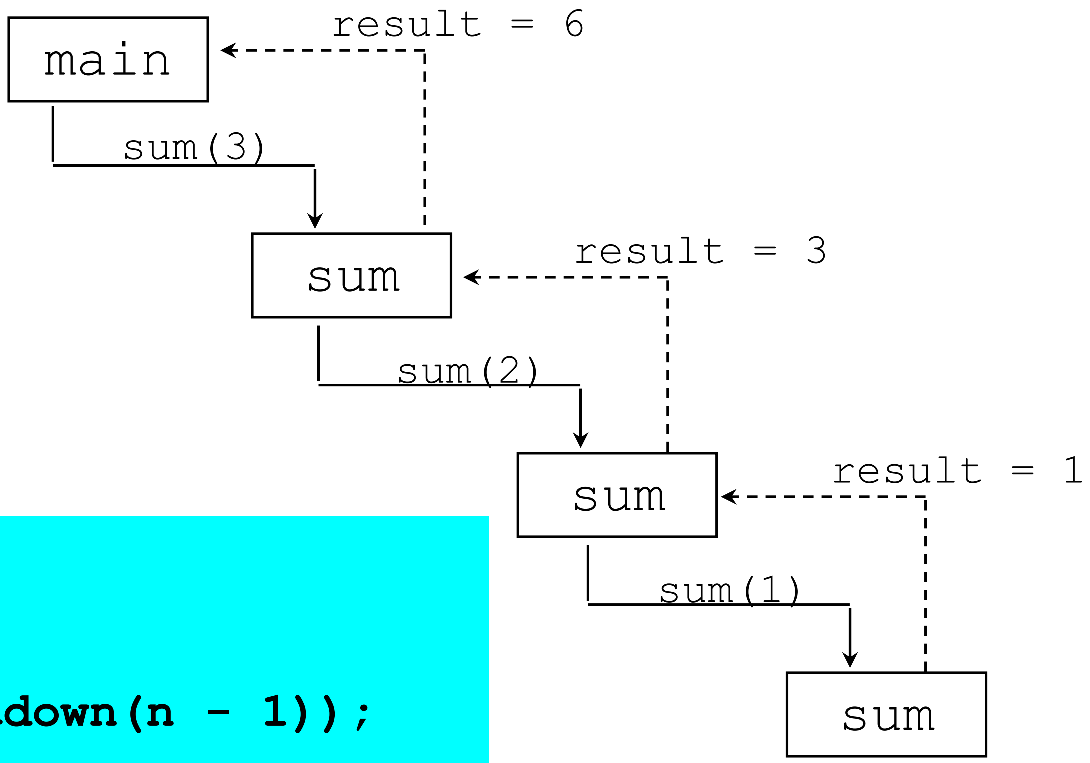
Recursive functions

- Consider the problem of computing the sum of all the numbers between 1 and any positive integer N

- This problem can be recursively defined as:

$$\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + (N-1) + \sum_{i=1}^{N-2} i$$

Recursive functions




```
int countdown (int n){
   if (n > 1)
      return (n + countdown(n - 1));
   else
      return 1;
}
```

Recursive functions

- Note that just because we can use recursion to solve a problem, doesn't mean we should
- For instance, we usually would not use recursion to solve the sum of 1 to N problem, because the iterative version is easier to understand
- However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version
- You must carefully decide whether recursion is the correct technique for any problem

Recursive functions

- Fibonacci numbers
  - 0, 1, 1, 2, 3, 5, 8...
  - Each number sum of the previous two

  fib(n) = fib(n - 1) + fib(n - 2)

  - Base case: **fib(0) = 0** and **fib(1) = 1**

Recursive functions

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

Recursive functions

```
int main(){
int i, array[32];
for( i = 0 ; i<32; i++)     {
        if( i == 0 )
                array[i]= 0;
        else if( i == 1)
                array[i] = 1;
        else
                array[i] = array[i-2] + array[i-1];
        }
        cout<<array[31];
        return 0;
}
```

Divide and conquer

- An effective approach to designing fast algorithm in sequential computation is the method known as **divide and conquer.**
- The problem to be solved is broken into a number of subprograms (typically two) of the same form as the original problem; this is the **divide** step.
- The subproblems are then solved independently, usually recursively; this is the **conquer** step.
- Finally, the solutions to the subproblems are combined to provide the answer to the original problem.

Divide and conquer

•The sorting algorithms Mergesort and Quicksort are both based on the divide-and-conquer approach.

•Example:Let us consider the task of finding the maximum(or minimum) of N items stored in an array.

•Iterative Findmax:

```
for( max =a[0], i =1 ; i < N; i++)
        if(a[i] > max)
              max = a[i];
```

T(n) = n-1

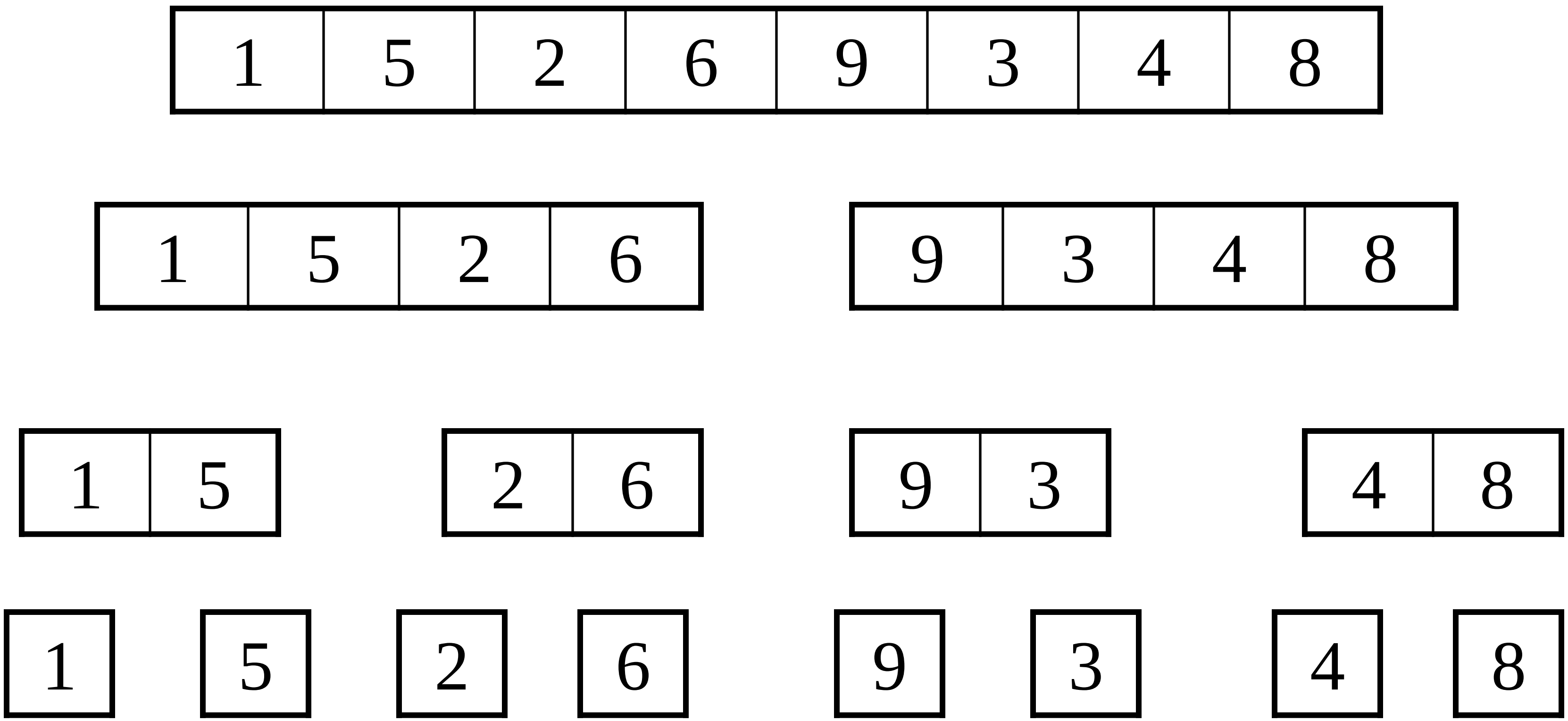
Divide and conquer

Divide and Conquer solution of finding max of N integers.

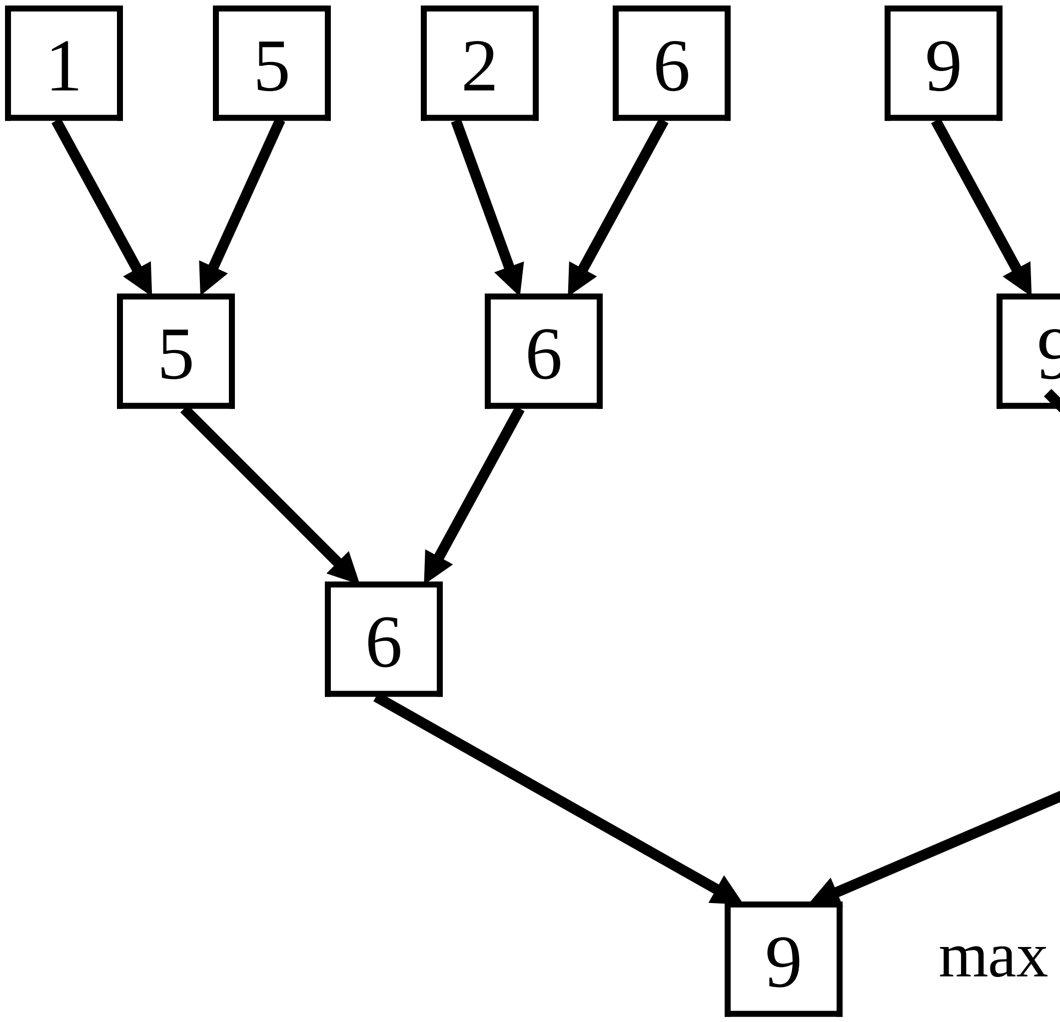
```
int max(int a[], int l, int r){
   int u, v;
   int m = (l+r)/2;
   if (l == r)
        return a[l];
   u = max(a, l, m);
   v = max(a, m+1, r);
   if (u > v)
        return u;
   else
        return v;
}
```

Divide and conquer

- Assume that $N = 2^k$
- $T(n) = 2\,T(n/2) + 1$



Divide and conquer



Divide and conquer

$T(n) = 2\,T(n/2) + 1$

$T(n/2) = 2\,T(n/4) + 1$

$T(n) = 2\,(2\,T(n/4) + 1) + 1$

$= 2^2 T(n/4) + 2 + 1$

$T(n/4) = 2\,T(n/8) + 1$

$T(n) = 2^2 (2\,T(n/8) + 1) + 2 + 1$

$= 2^3 T(n/8) + 2^2 + 2 + 1$

$= 2^3 T(n/2^3) + 2^2 + 2 + 1$

…..

$= 2^k T(n/2^k) + 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2 + 1$

$$= 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i$$

Divide and conquer

From the base case where T(1) = 0

$(n/2^k) = 1 \rightarrow n = 2^k \quad k = \log n$

and we know that $$\sum_{i=0}^{k-1} 2^i = \frac{x^k - 1}{x - 1}$$

$$T(n) = 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i = 2^k T(n/2^k) + \frac{2^k - 1}{2 - 1}$$

$T(n) = 2^k * 0 + 2^k - 1 = 2^{\log n} - 1 = \mathbf{n-1}$