Recursion

- In C/C++, a function may call itself either directly or indirectly.
- When a function call itself recursively, each invocation gets a fresh set of all explicit parameters and automatic (local) variables, independent of the previous set.

Example: Compute the greatest common divisor (GCD) of two numbers.

Let M > N > 0, and M = QN + R such that 0 <= R < N

- if R = 0, then M = QN, and gcd(M, N) = N
- if R > 0, then N > R, and gcd(M, N) = gcd(N, R)

```
int gcd(int M, int N)
/* given M > 0 and N > 0, compute gcd of M & N */
{
   int R = M % N;
   if (R == 0)  //base case
      return N;
   else
      return gcd(N, R);  //recursion
}
```

An equivalent non-recursive implementation.

```
int gcd(int M, int N)
/* given M > 0 and N > 0, compute gcd of M & N */
{
   int R;
   while (R = M % N)  //R = M % N and R != 0
   {  M = N;
      N = R;
   }
   return N;
}
```

Usually, looping (iteration) is more efficient than recursion.

Recursion is preferred when the problem is recursively defined, or when the data structure that the algorithm operates on is recursively defined.

Two fundamental rules of recursion:

1. **Base cases**
You must have some base cases, which can be solved without recursion.

2. **Making progress (recursion)**
For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.

Example: Fibonacci number

$f(0) = 0;$
$f(1) = 1;$
$f(n) = f(n-2) + f(n-1)$ for $n \geq 2;$

```
long long fib(long long n)
{  //  precondition: given n >= 0

   if (n == 0)     // base case #1
      return 0;

   if (n == 1)     // base case #2
      return 1;

   return fib(n-2) + fib(n-1);  // recursion

} // this program works but is unacceptably slow for n > 30
```

Computing the Fibonacci number using iteration is much more efficient.

Example: Ackermann's function

$$A(M,N)=\begin{cases} N+1 & if\ M=0 \\ A(M-1,1) & if\ N=0 \\ A(M-1,A(M,N-1)) & \text{otherwise} \end{cases}$$

```
long long Ack(long long m, long long n)
//  precondition: given m >= 0 and n >= 0
{
   if (m == 0)
      return n+1;

   if (n == 0)
      return Ack(m-1, 1);

   return Ack(m-1, Ack(m, n-1));
}
```

Example: Recursive binary search

```
//Precondition: A[] is sorted in ascending order
//A[low..high] represents the search range
int binSearch_R(int A[], int low, int high, int x)
{
   if (low > high) //search range is empty
      return -1;   //x is not found

   int mid = (low + high) / 2;  //probe the mid-point of
   if (A[mid] == x)             //the search range
      return mid;

   if (A[mid] < x) //make recursion with reduced search range
      return binSearch_R(A, mid+1, high, x);
   else
      return binSearch_R(A, low, mid-1, x);
}
```

Non-Recursive binary search

```
//Precondition: A[] is sorted in ascending order
int binSearch(int A[], int N, int x)
{
  int low = 0;
  int high = N-1;
  int loc = -1;

  while (low <= high && loc < 0)
  {
     int mid = (low + high) / 2;

     if (A[mid] < x)
       low = mid+1;
     else if (A[mid] > x)
       high = mid-1;
     else
       loc = mid; // x == A[mid]
  }

  return loc;
}
```

Example: Towers of Hanoi

- There are 3 towers, labeled L, M and R
- Initially, tower L contains a stack of N disks of graded sizes, stacked in order of size with the smallest disk at the top.
- A move consists of taking the top disk from one tower and placing it on another tower, subject to the constraint that a disk may never be placed above a disk smaller than itself.
- The problem is to move the N disks from tower L to tower R in the smallest number of moves.

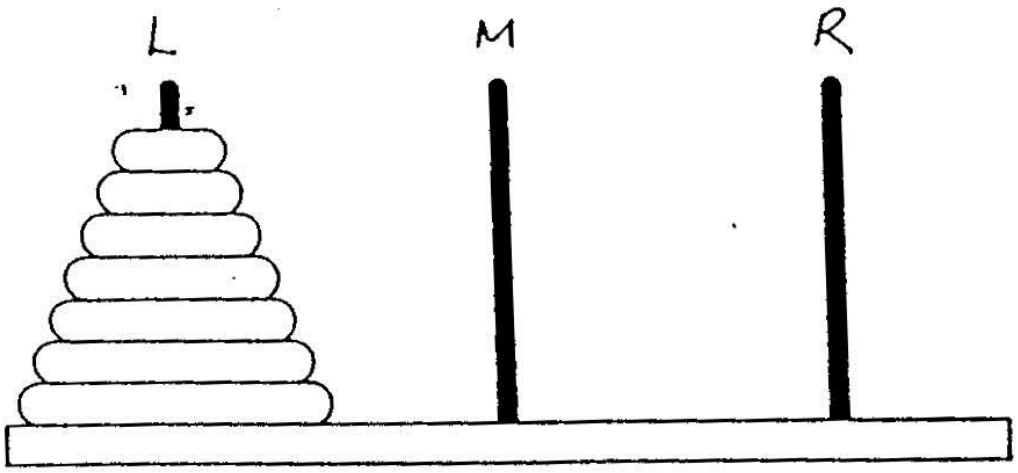


Figure 3.8. The Towers of Hanoi

The problem can be solved in 3 "steps":

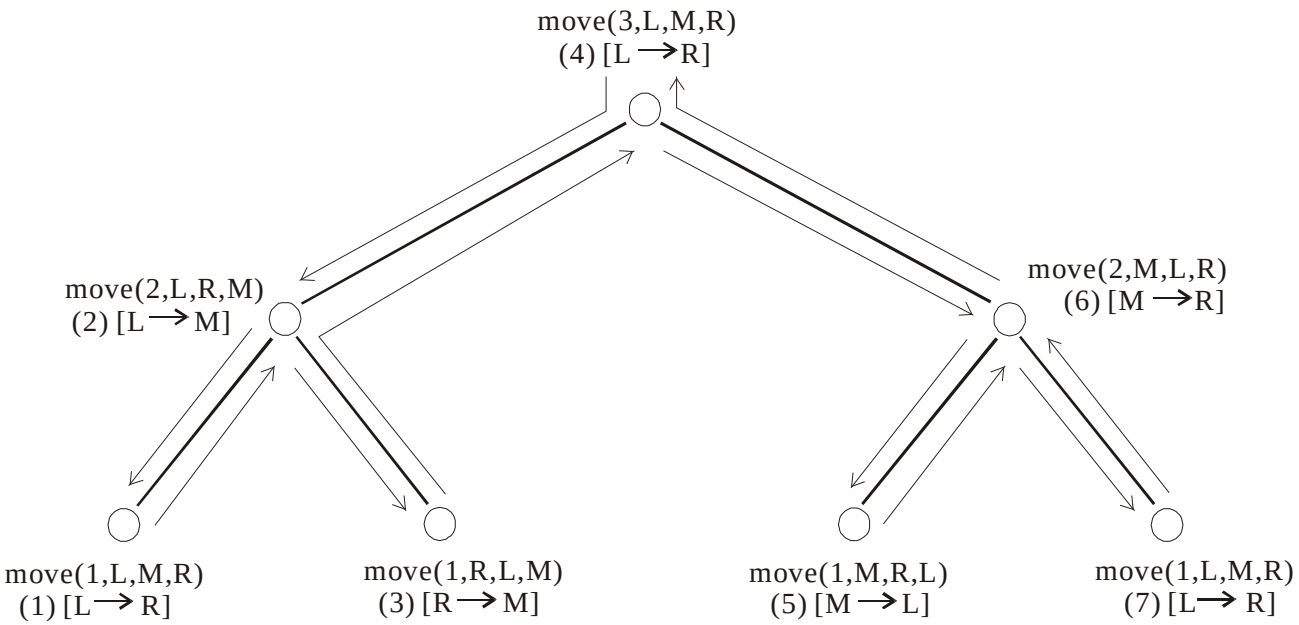
1. Move the top N-1 disks from L to M using R as a temporary working space.
2. Move the bottom disk from L to R
3. Move the remaining N-1 disks from M to R using L as a temporary working space.

Let `move(N, F, U, T)` denotes the operation of moving `N` disks from tower `F` to tower `T` using tower `U`.

The recursive function to print out the sequence of moves:

```
void move(int N, char F, char U, char T)
{
   if (N == 1)
      cout << F << " -> " << T << endl;
   else if (N > 1)
   {
      move(N-1, F, T, U);
      cout << F << " -> " << T << endl;
      move(N-1, U, F, T);
   }
}
```

Recursion tree for moving 3 disks



depth of the recursion = number of disks to be moved = N

number of moves = number of nodes in the recursion tree

$$= 1 + 2 + 4 + \ldots + 2^{N-1} = 2^N - 1$$

```
//version 2, without the else statement

void move_v2(int N, char F, char U, char T)
{
   if (N > 0)
   {
      move_v2(N-1, F, T, U);
      cout << F << " -> " << T << endl;
      move_v2(N-1, U, F, T);
   }
}
```

Recursive function to print the permutations of an array of symbols

```
//Consider the case for N=3
char x[3] = {'a', 'b', 'c'};

Permutations of the 3 symbols are:
a b c
a c b
b a c
b c a
c b a
c a b
```

Pseudo code:

```
void permute(char x[], int s, int e)
// Enumerate the permutations of the symbols in x[s..e].
// The symbols in x[0..s-1] have already been fixed.
{
   if (s == e) //base case
      //symbols in x[0..e] are fixed
      print the contents of x[];
   else //recursion
   {
      //x[s] can have e-s+1 choices
      for each choice of x[s]
         find the permutations of x[s+1..e] by recursion;
   }
}
```

```
//Detailed codes

void swap(char x[], int i, int j) // swap x[i] with x[j]
{
   char t = x[i];
   x[i] = x[j];
   x[j] = t;
}

void permute(char x[], int s, int e)
// Enumerate the permutations of the symbols in x[s..e]
{
   if (s == e)  //base case: symbols in x[0..e] are fixed
   {
      for (int k = 0; k <= e; k++)
         cout << x[k] << " ";
      cout << endl;
   }
   else  //recursion
   {
      for (int k = s; k <= e; k++)
      {
         //1. there are e-s+1 choices for the first symbol
         swap(x, s, k);

         //2. once we fix the first symbol,
         //   find the permutations of the remaining e-s
         //   symbols by recursion
         permute(x, s+1, e);

         //3. restore the original order of the array
         swap(x, s, k);
       }
    }
}

void testPermute()
{
   char x[4] = {'a', 'b', 'c', 'd'};

   permute(x, 0, 3); //print permutations of 4 symbols
}
```

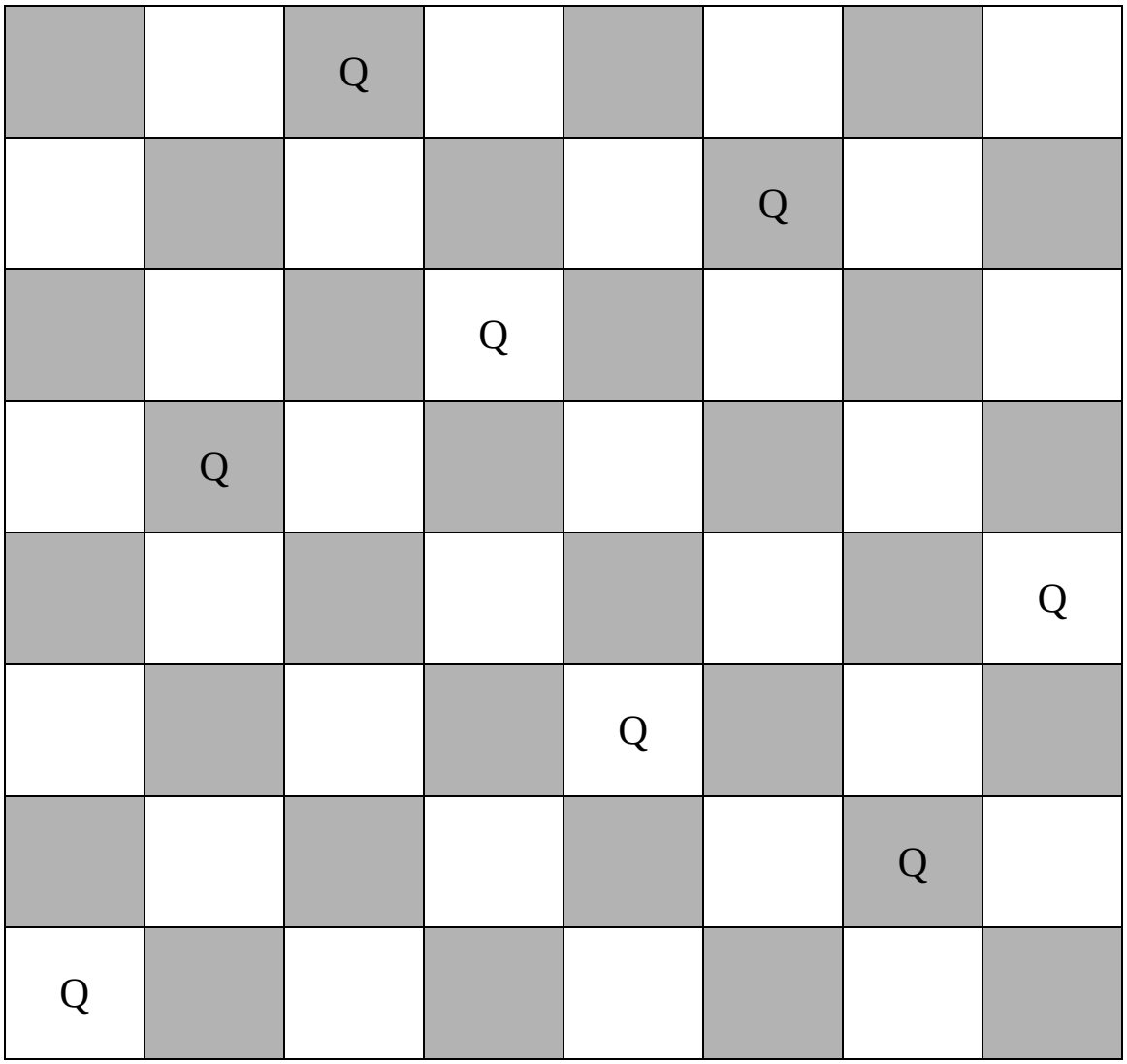
Recursion and Backtracking

Search space is a set of possible right answers to be explored.

Backtracking is a technique whereby

- the search space is explored by systematically trying each possible path,
- back up to try another whenever a dead end is encountered.

The Eight-Queens Problem:
Put eight queens on the chessboard such that no queen is being attacked by the others.



Rather than viewing the chessboard as consisting of 64 squares, it can be seen as being comprised of eight columns, each with eight rows.

- To solve the problem, we need to place one Queen in each column.
- In each column we will place a Queen in one of the eight rows, and then take a step forward by making a recursive call.
- This call will return having either **succeeded** in finding a solution, or having **failed** (meaning that there is no solution given the current placement of Queens).
- If the recursive call fails, we move the Queen to another row and try again.

Outline of the solution in pseudo code

```
bool EightQueen(ChessBoard square, int col)
{
    int row;
    bool success = false;

    if (col >= 8)
        success = true;
    else
    {
        for (row = 0; row < 8 and not success; row++)
        {
            if (square[row, col] is safe)
            {
                put a queen in square[row, col];
                success = EightQueen(col+1);
                if (not success)
                    remove queue from square[row, col];
            }
        }
    }
    return success;
}
```

```
#define NumRows 8
#define NumCols 8
typedef int ChessBoard[NumRows][NumCols];

bool safePos(ChessBoard square, int row, int col)
/* The poistion is safe if there is not another Queen in
   the given row and diagonals. */
{
    bool conflict = false;
    for (int j = col-1; j >= 0 && !conflict; j--)
    {
        if (square[row][j] == 1) // check the row
            conflict = true;

        int i = row + col - j; // check upper diagonal
        if (i < NumRows)
            if (square[i][j] == 1)
                conflict = true;

        i = row - (col - j); // check lower diagonal
        if (i >= 0)
             if (square[i][j] == 1)
                conflict = true;
    }
    return !conflict;
}

bool EightQueen(ChessBoard square, int col)
{
    bool success = false;
    if (col >= 8)  //base case: solution found
        success = true;
    else
    {
        for (int row = 0; row < NumRows && !success; row++)
        {
            if (safePos(square, row, col))
            {   //put a queue at square[row][col]
                square[row][col] = 1;
                success = EightQueen(square, col+1);
                if (!success) //remove the queue
                    square[row][col] = 0;
            }
        }
    }
    return success;
}
```

```
void solveEightQueen(ChessBoard square)
{
    // initialize the chessboard
    for (int i = 0; i < NumRows; i++)
        for (int j = 0; j < NumCols; j++)
            square[i][j] = 0;

    if (EightQueen(square, 0))
    {    // print solution
         for (int i = NumRows-1; i >= 0; i--)
         {
             for (int j = 0; j < NumCols; j++)
                 cout << square[i][j];
             cout << endl;
         }
    }
    else
        cout << "no solution found" << endl;
}
```