Recursion

Arijit Bishnu
arijit@isical.ac.in

Indian Statistical Institute, India.

August 25, 2015

Outline

Outline

Recursion

What is recursion?

Recursion

What is recursion?

- A function calling itself with certain properties.

Recursion

What is recursion?

- A function calling itself with certain properties.
- There must be a base criteria for which the function does not call itself.

Recursion

What is recursion?

- A function calling itself with certain properties.
- There must be a base criteria for which the function does not call itself.
- After each call, the function must be closer to the base criteria.

Recursion

What is recursion?

- A function calling itself with certain properties.
- There must be a base criteria for which the function does not call itself.
- After each call, the function must be closer to the base criteria.

An Example (Can you recognize the expression?)

$$\begin{aligned} f(n) &= n * f(n-1) \text{ if } n \geq 1 \\ &= 1 \text{ if } n = 0 \end{aligned}$$

Outline

Fibonacci sequence

The Definition

$$\begin{aligned} fib(n) &= fib(n-1) + fib(n-2) \text{ if } n \geq 2 \\ &= 1 \text{ if } n = 1 \\ &= 1 \text{ if } n = 0 \end{aligned}$$

Fibonacci sequence

The Definition

$$\begin{aligned} fib(n) &= fib(n-1) + fib(n-2) \text{ if } n \geq 2 \\ &= 1 \text{ if } n = 1 \\ &= 1 \text{ if } n = 0 \end{aligned}$$

Some values

$fib(2) = 2$; $fib(3) = 3$; $fib(4) = 5$; $fib(5) = 8$; $fib(6) = 13$;
$fib(7) = 21$;$fib(8) = 34$; $fib(9) = 55$; $fib(10) = 89$; $fib(11) = 144$;

A recursive implementation

```
#include<stdio.h>
```

A recursive implementation

```
#include<stdio.h>
```

```
 int main(void){
 unsigned int m,val,i;
 printf("\n What is the value of m::> "); scanf("%d",&m);
 val = f(m);
 printf("\n Fibonacci(%d) = %d \n",m,val);
 return 0;
}
```

A recursive implementation

```
#include<stdio.h>
unsigned int f(unsigned int num)
{



}
 int main(void){
 unsigned int m,val,i;
 printf("\n What is the value of m::> "); scanf("%d",&m);
 val = f(m);
 printf("\n Fibonacci(%d) = %d \n",m,val);
 return 0;
}
```

A recursive implementation

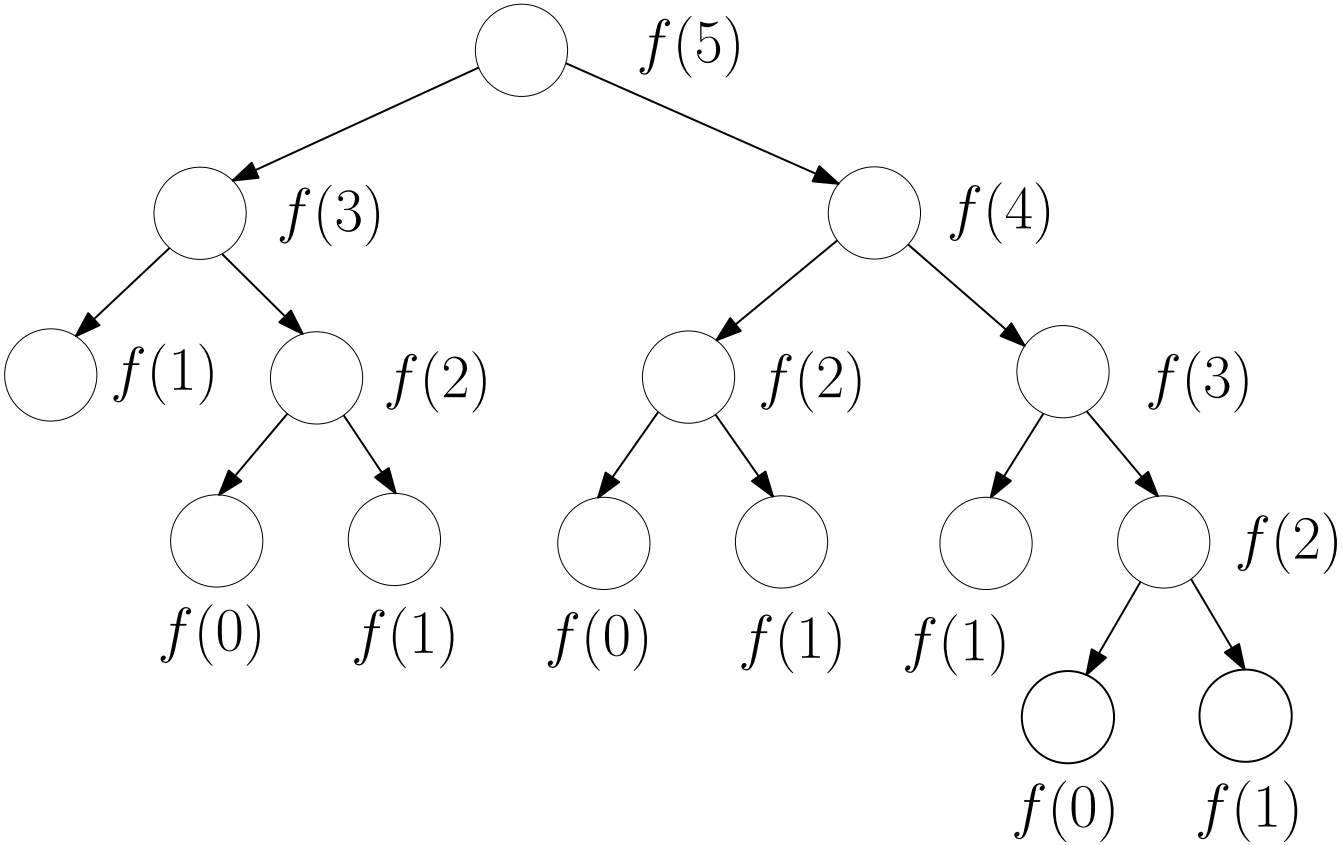
```
#include<stdio.h>
unsigned int f(unsigned int num)
{
  if(num==0 || num==1)    return 1;

}
 int main(void){
 unsigned int m,val,i;
 printf("\n What is the value of m::> "); scanf("%d",&m);
 val = f(m);
 printf("\n Fibonacci(%d) = %d \n",m,val);
 return 0;
}
```

A recursive implementation

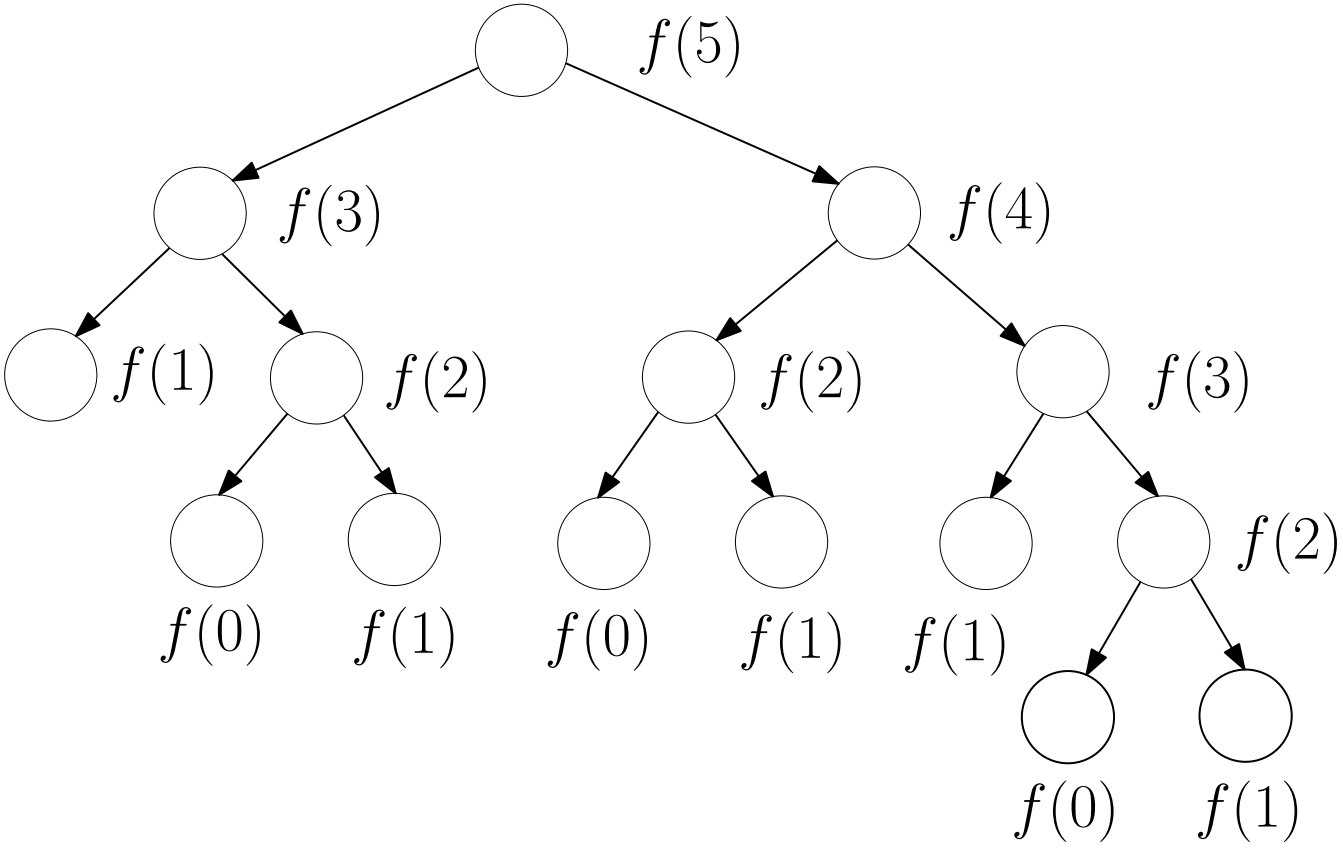
```
#include<stdio.h>
unsigned int f(unsigned int num)
{
  if(num==0 || num==1)    return 1;
  else return(f(num-1)+f(num-2));
}
 int main(void){
 unsigned int m,val,i;
 printf("\n What is the value of m::> "); scanf("%d",&m);
 val = f(m);
 printf("\n Fibonacci(%d) = %d \n",m,val);
 return 0;
}
```

The recursion tree

The recursion tree



The recursion tree



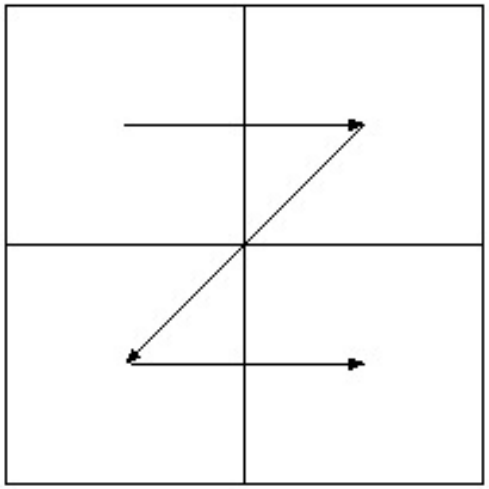
How many function calls?

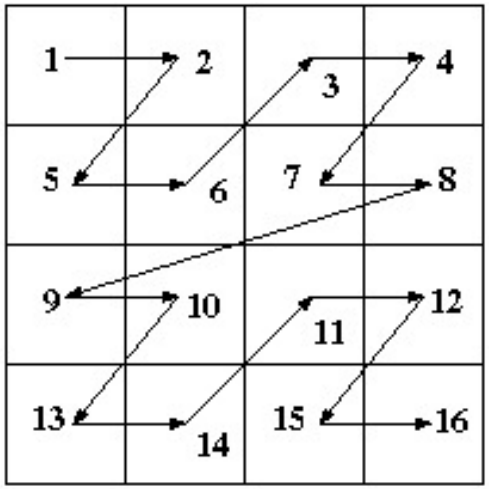
$f(5) : 1$, $f(4) : 1$, $f(3) : 2$, $f(2) : 3$, $f(1) : 5$, $f(0) : 3$

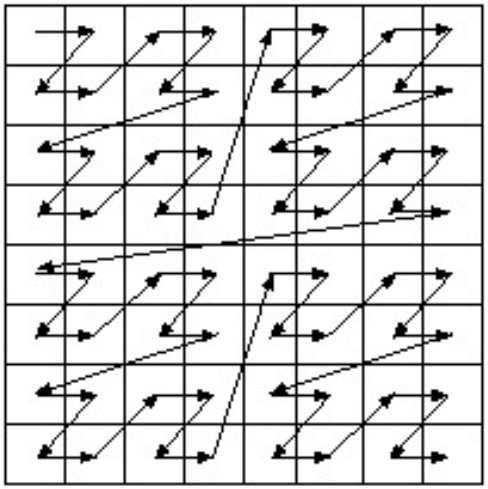
Exercise: Find it experimentally and analytically.

Outline

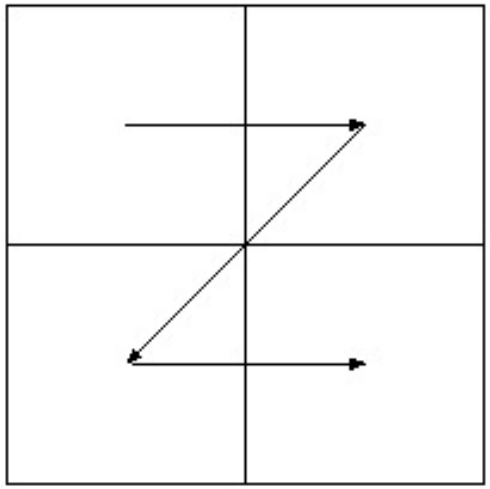
Space Filling Curve

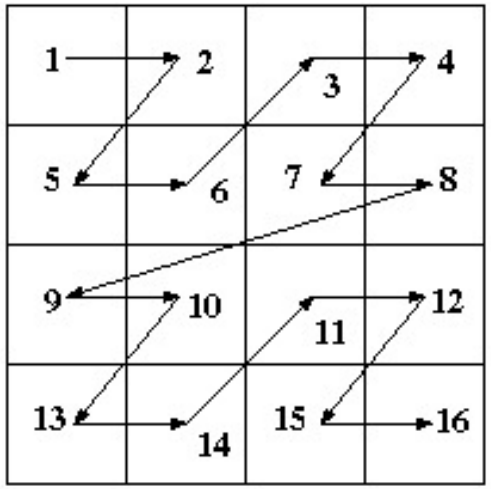


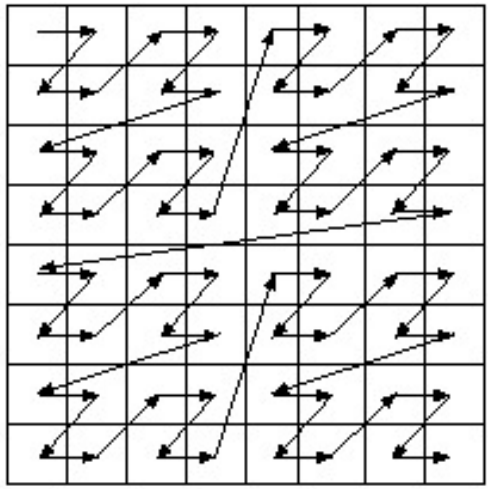




Space Filling Curve

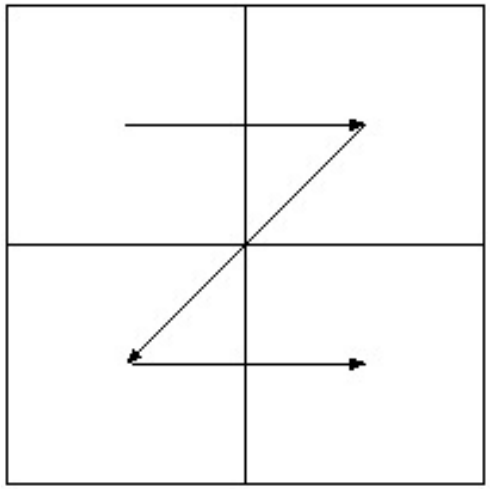


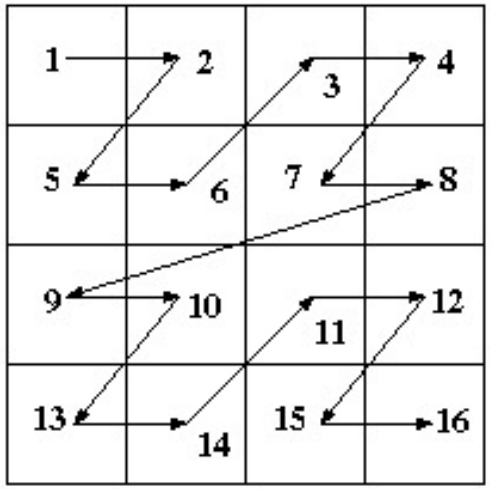


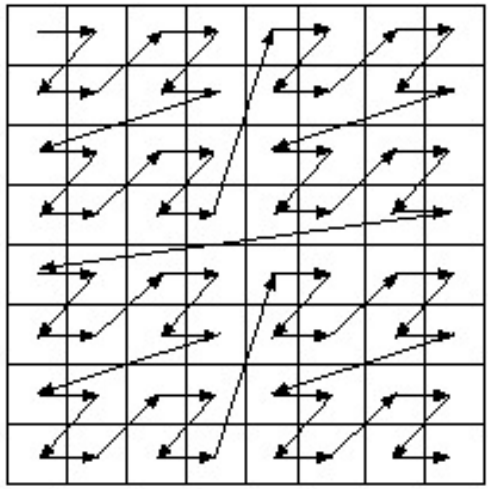


- Z Curve order: 1 2 5 6 3 4 7 8 9 10 13 14 11 12 15 16

Space Filling Curve



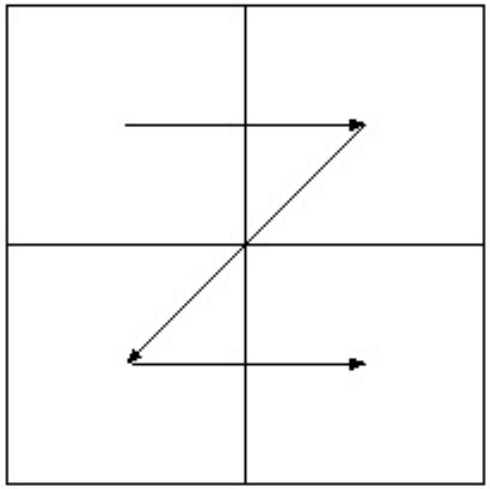


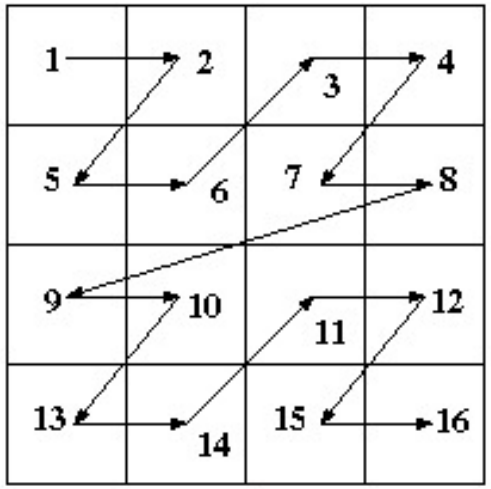


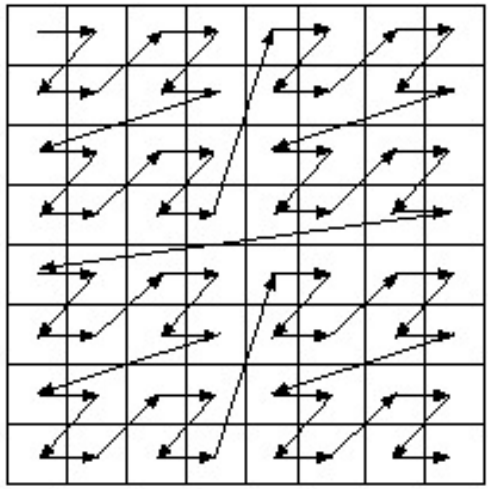
- Z Curve order: 1 2 5 6 3 4 7 8 9 10 13 14 11 12 15 16

Problem

Space Filling Curve





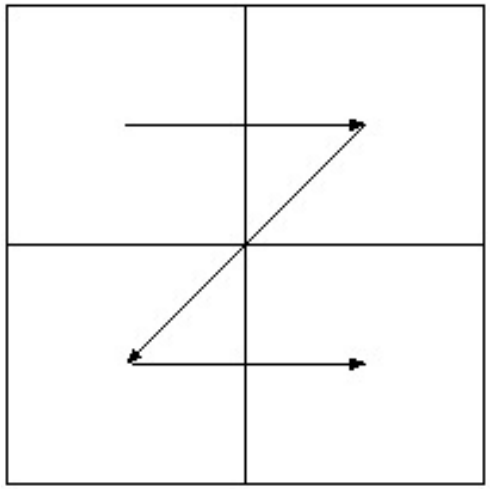


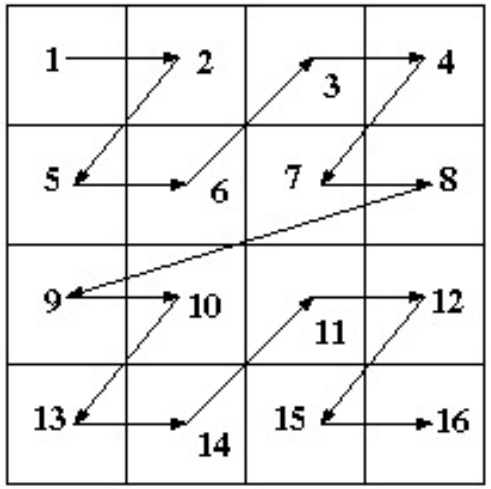
- Z Curve order: 1 2 5 6 3 4 7 8 9 10 13 14 11 12 15 16

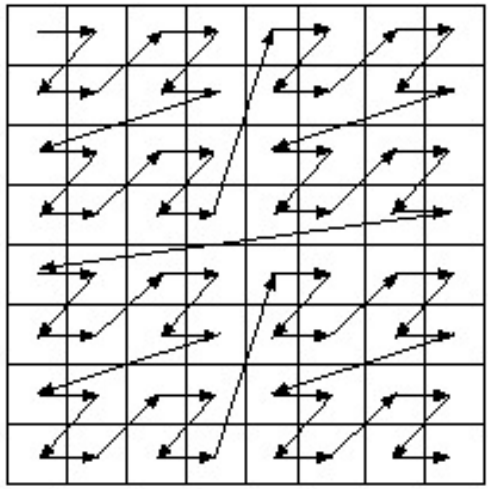
Problem

- Allocate a 2-D matrix of size $2^m \times 2^m$.

Space Filling Curve





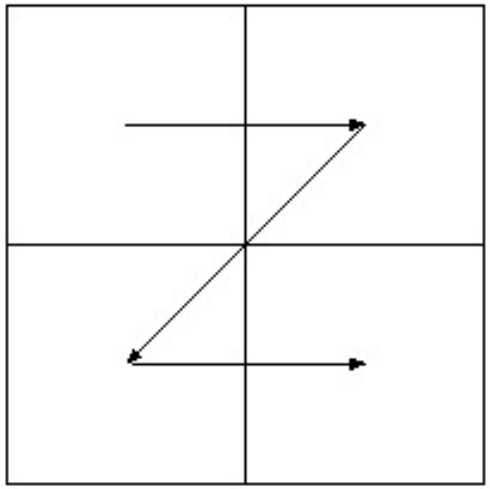


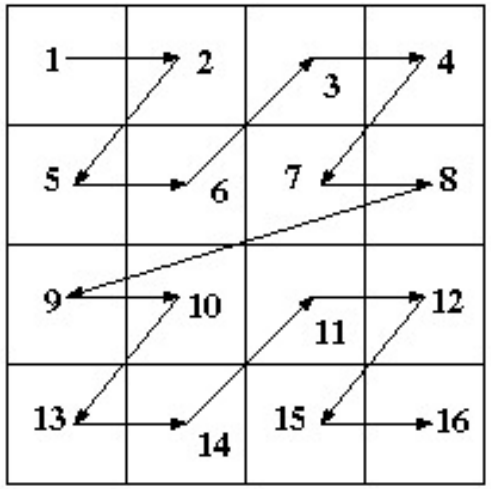
- Z Curve order: 1 2 5 6 3 4 7 8 9 10 13 14 11 12 15 16

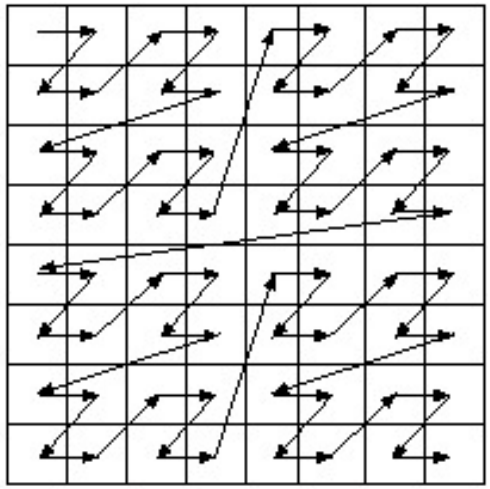
Problem

- Allocate a 2-D matrix of size $2^m \times 2^m$.
- Fill it with integers $1, \ldots, 2^{2m}$ in a row-major fashion.

Space Filling Curve







- Z Curve order: 1 2 5 6 3 4 7 8 9 10 13 14 11 12 15 16

Problem

- Allocate a 2-D matrix of size $2^m \times 2^m$.
- Fill it with integers $1, \ldots, 2^{2m}$ in a row-major fashion.
- Traverse the matrix in a z-curve order.

The main program

```
int main(void){
int m,N, i,j,k=1,**matrix;
printf(" \n What is the value of m::>"); scanf("%d",&m);
N=(int)(pow(2.0,m));
matrix = allocate2D(N,N);









}
```

The main program

```
int main(void){
int m,N, i,j,k=1,**matrix;
printf(" \n What is the value of m::>"); scanf("%d",&m);
N=(int)(pow(2.0,m));
matrix = allocate2D(N,N);
/*Fill up and print the matrix */
 for(i=0;i<N;i++)
  for(j=0;j<N;j++){matrix[i][j]=k; k++;}
 for(i=0;i<N;i++){ printf("\n");
  for(j=0;j<N;j++)  printf("%d \t",matrix[i][j]);
  }



}
```

The main program

```
int main(void){
int m,N, i,j,k=1,**matrix;
printf(" \n What is the value of m::>"); scanf("%d",&m);
N=(int)(pow(2.0,m));
matrix = allocate2D(N,N);
/*Fill up and print the matrix */
 for(i=0;i<N;i++)
  for(j=0;j<N;j++){matrix[i][j]=k; k++;}
 for(i=0;i<N;i++){ printf("\n");
  for(j=0;j<N;j++)  printf("%d \t",matrix[i][j]);
  }
printf("\n \nReading the matrix in z-curve fashion::>");
zee(0,N-1,0,N-1,matrix);
printf("\n"); return 0;
}
```

Allocating the 2-D matrix

```
int** allocate2D(int row,int col){
int **S,k;

S=(int **)calloc(row,sizeof(int *));
if(S==NULL){printf("\n No space \n"); exit(0);}

for(k=0;k<row;k++){
  S[k]=(int *)calloc(col,sizeof(int));
  if(S[k]==NULL){ printf("\n No space \n"); exit(0);}
  }
  return S;
}
```

The recursive function

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }




return;
}
```

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }
zee(
      );




return;
}
```

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }
zee(
      );
zee(
      );



return;
}
```

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }
zee(
      );
zee(
      );
zee(
      );


return;
}
```

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }
zee(
      );
zee(
      );
zee(
      );
zee(
       );
return;
}
```

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }
zee(r_b,(r_e+r_b)/2,c_b,(c_e+c_b)/2,
matrix);
zee(
      );
zee(
      );
zee(
       );
return;
}
```

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }
zee(r_b,(r_e+r_b)/2,c_b,(c_e+c_b)/2,
matrix);
zee(r_b,(r_e+r_b)/2,(c_e+c_b)/2+1,c_e,
matrix);
zee(
      );
zee(
       );
return;
}
```

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }
zee(r_b,(r_e+r_b)/2,c_b,(c_e+c_b)/2,
matrix);
zee(r_b,(r_e+r_b)/2,(c_e+c_b)/2+1,c_e,
matrix);
zee((r_e+r_b)/2+1,r_e,c_b,(c_e+c_b)/2,
matrix);
zee(
       );
return;
}
```

The recursive function

```
void zee(int r_b,int r_e,int c_b,int c_e,
         int **matrix){
    if((r_b==r_e)&&(c_b==c_e)){
       printf(" %2d ",matrix[r_b][c_b]);
       return;
     }
zee(r_b,(r_e+r_b)/2,c_b,(c_e+c_b)/2,
matrix);
zee(r_b,(r_e+r_b)/2,(c_e+c_b)/2+1,c_e,
matrix);
zee((r_e+r_b)/2+1,r_e,c_b,(c_e+c_b)/2,
matrix);
zee((r_e+r_b)/2+1,r_e,(c_e+c_b)/2+1,c_e,
matrix));
return;
}
```

Outline

Towers of Hanoi

The Problem

Towers of Hanoi

The Problem

- Three pegs A, B and C are given and on peg A there are n disks with decreasing size.

Towers of Hanoi

The Problem

- Three pegs A, B and C are given and on peg A there are n disks with decreasing size.
- We have to move disks from peg A to peg C using peg B as an auxiliary peg following some rules.

Towers of Hanoi

The Problem

- Three pegs A, B and C are given and on peg A there are n disks with decreasing size.
- We have to move disks from peg A to peg C using peg B as an auxiliary peg following some rules.
  - The top disk on any peg can be moved to any other peg.

Towers of Hanoi

The Problem

- Three pegs A, B and C are given and on peg A there are n disks with decreasing size.
- We have to move disks from peg A to peg C using peg B as an auxiliary peg following some rules.
  - The top disk on any peg can be moved to any other peg.
  - A larger disk can never be placed on a smaller disk.

Towers of Hanoi

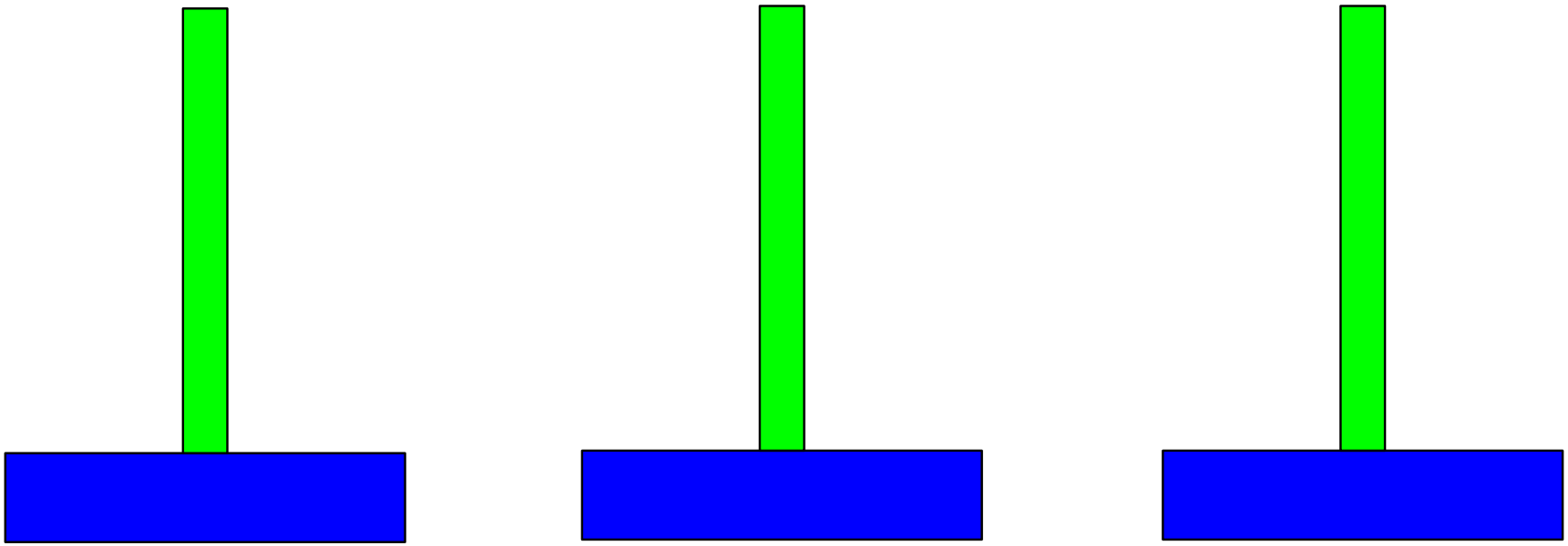
The Problem

- Three pegs A, B and C are given and on peg A there are n disks with decreasing size.
- We have to move disks from peg A to peg C using peg B as an auxiliary peg following some rules.
  - The top disk on any peg can be moved to any other peg.
  - A larger disk can never be placed on a smaller disk.

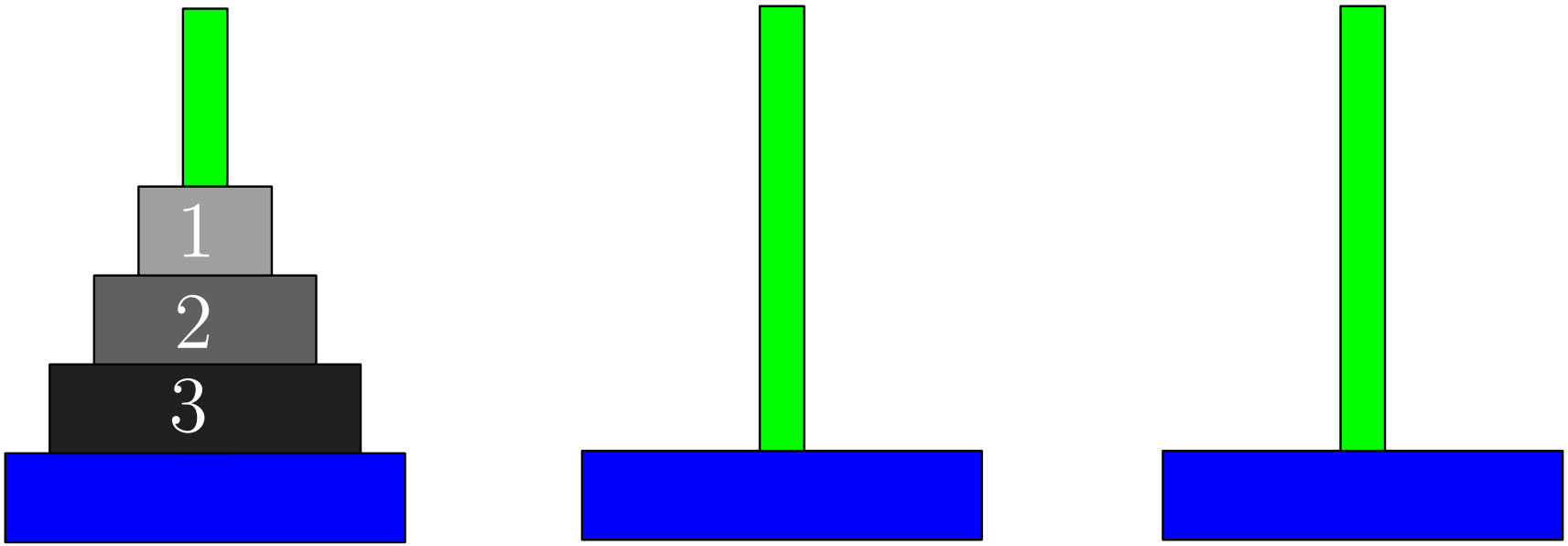
Exercise

Try with $n = 3, 4, \ldots$ and start thinking recursively.

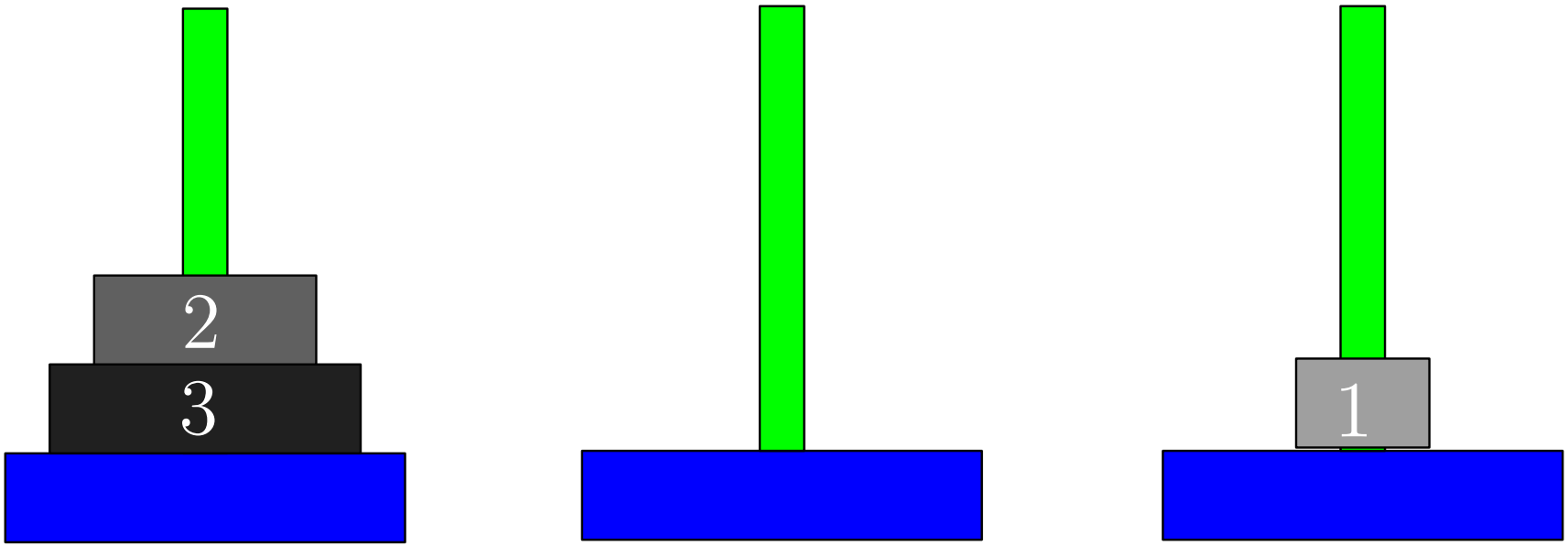
Towers of Hanoi



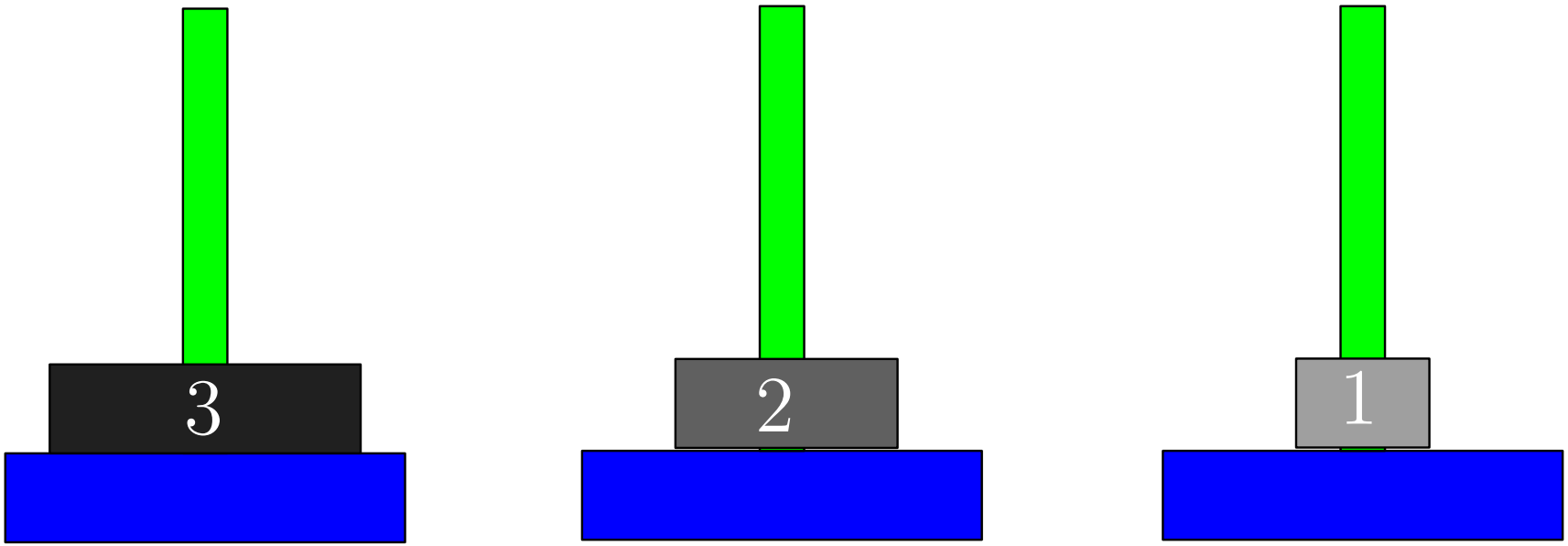
Towers of Hanoi



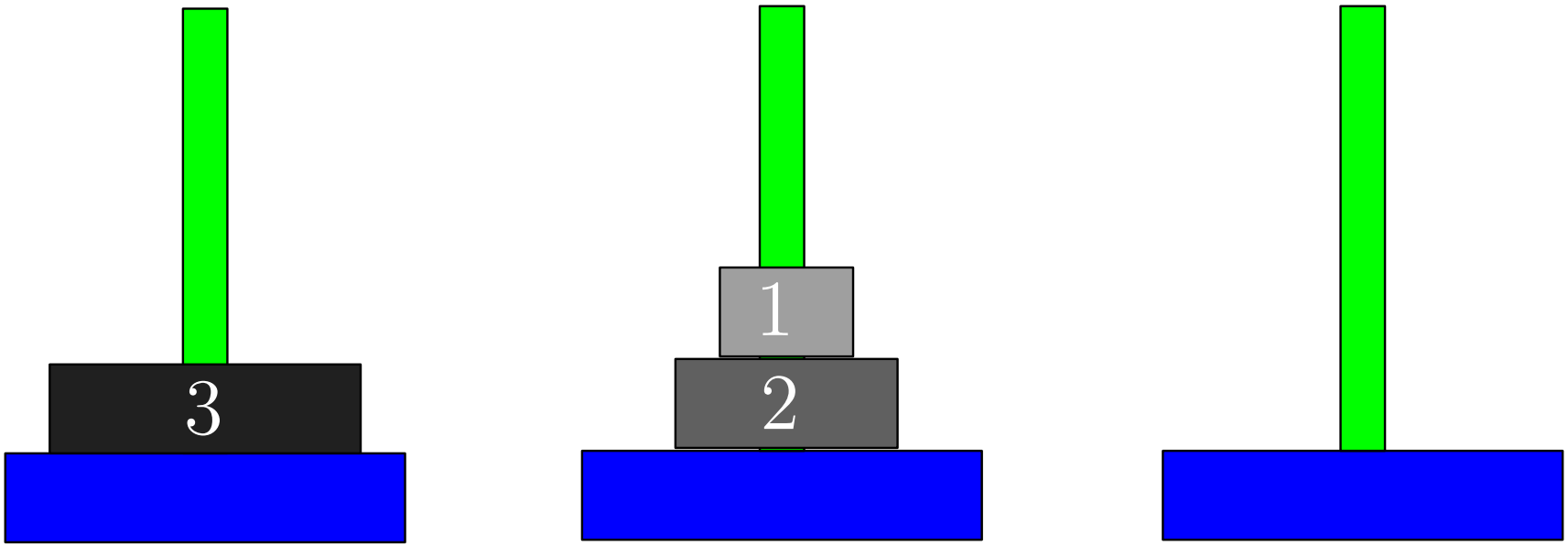
Towers of Hanoi



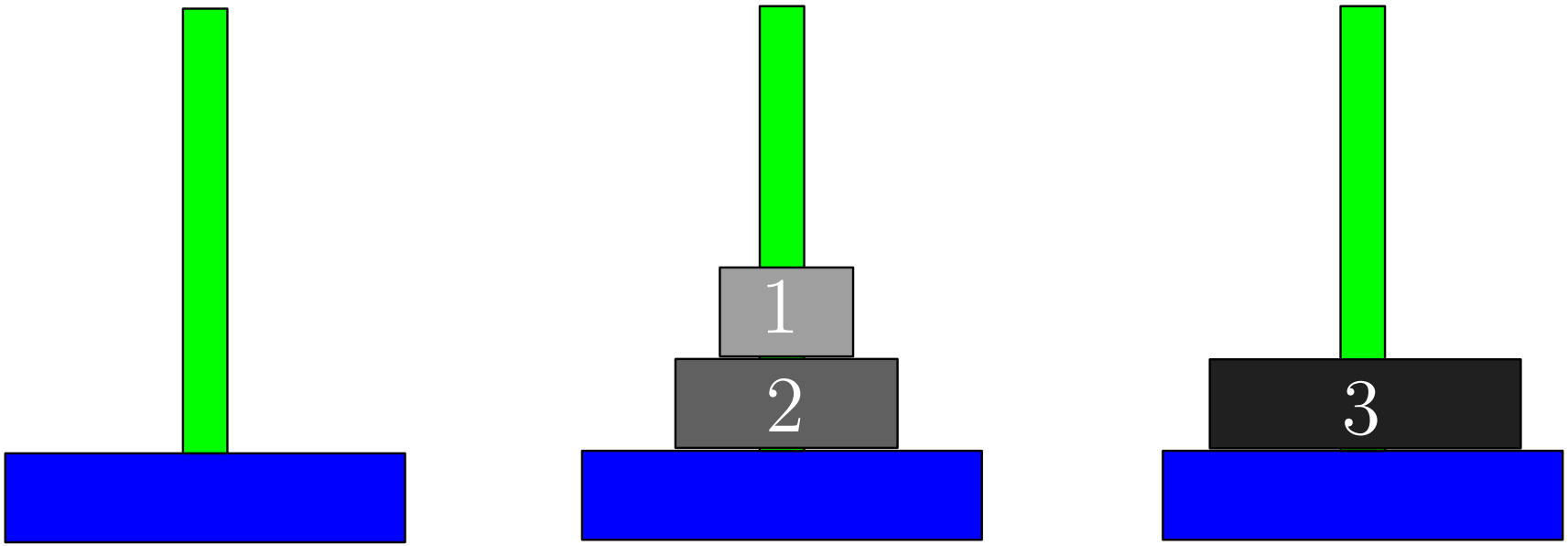
Towers of Hanoi



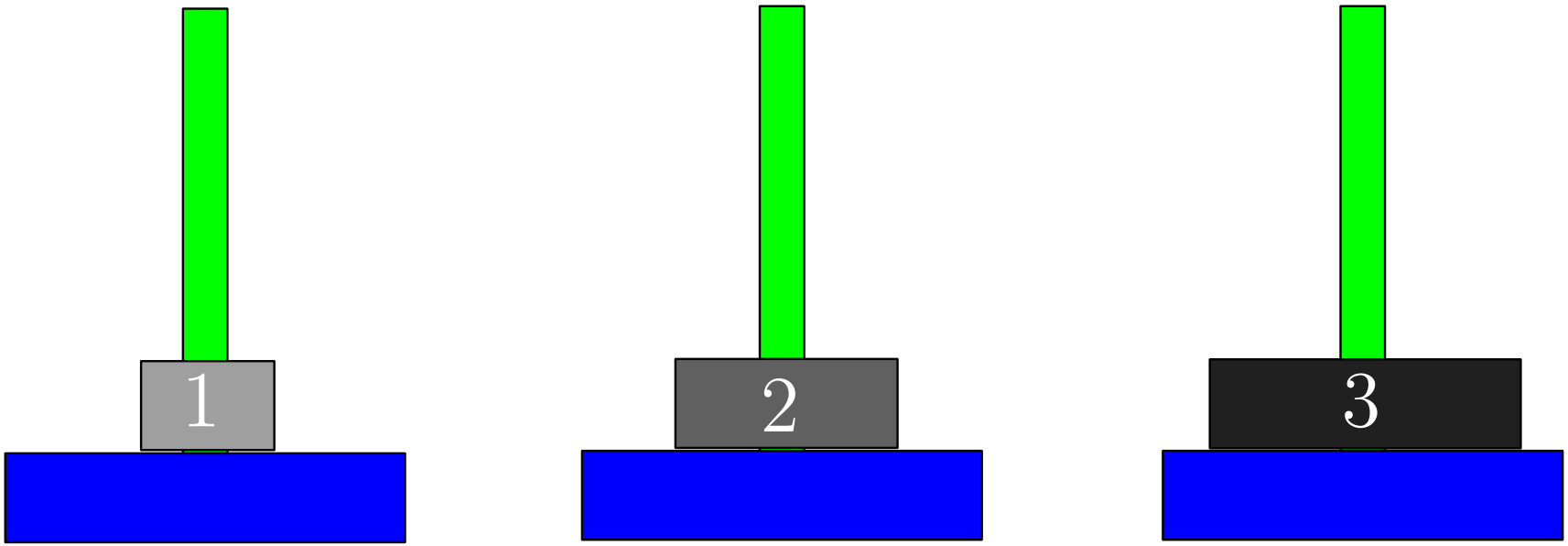
Towers of Hanoi



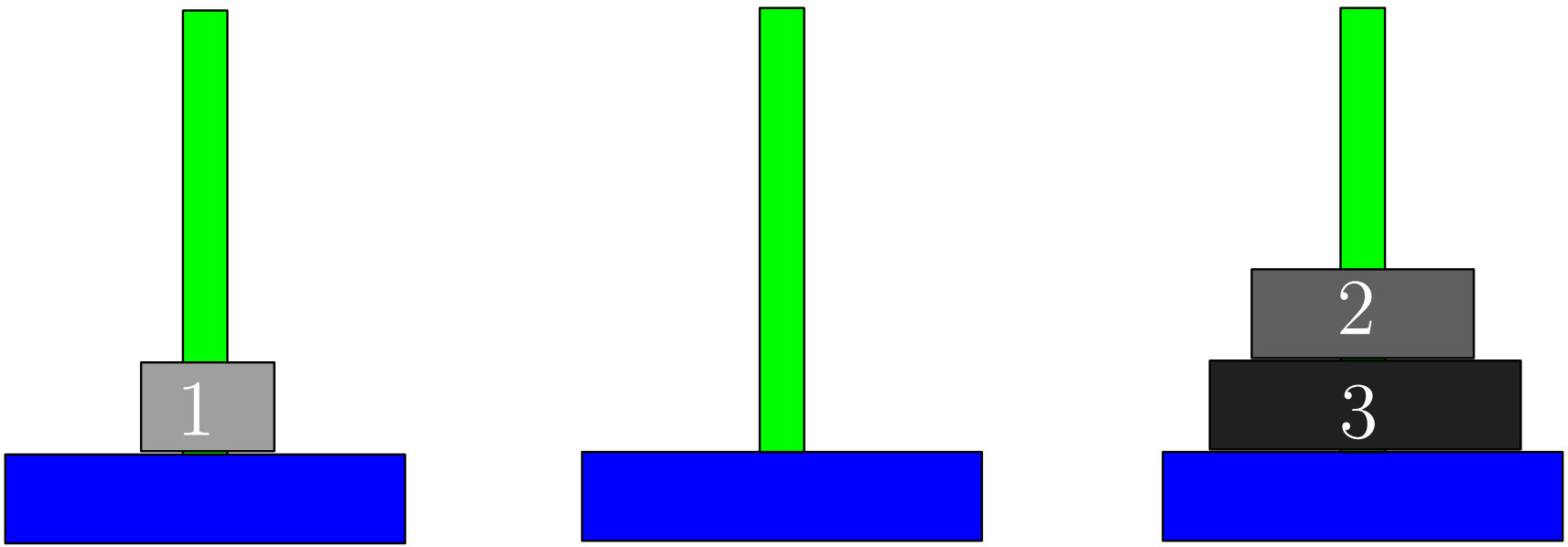
Towers of Hanoi



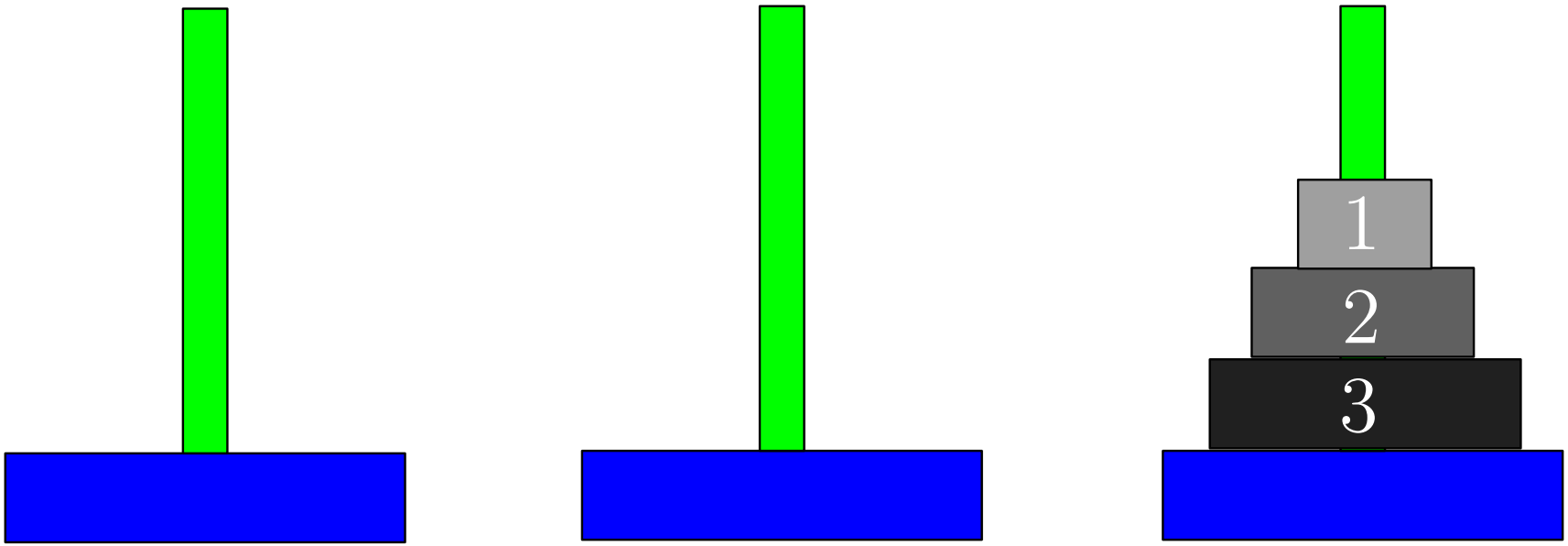
Towers of Hanoi



Towers of Hanoi



Towers of Hanoi



The algorithm

The recursive method

The algorithm

The recursive method

- Move the top $n-1$ disks from peg A to peg B using peg C as auxiliary.

The algorithm

The recursive method

- Move the top $n-1$ disks from peg A to peg B using peg C as auxiliary.
- Move the top disk from peg A to peg C.

The algorithm

The recursive method

- Move the top $n-1$ disks from peg A to peg B using peg C as auxiliary.
- Move the top disk from peg A to peg C.
- Move the top $n-1$ disks from peg B to peg C using peg A as auxiliary.

The main program

```
int main(void) {
int n;
printf("Enter the number of disks : ");
scanf("%d",&n);
printf("The Tower of Hanoi involves the moves :\n \n");


return 0;
}
```

The main program

```
int main(void) {
int n;
printf("Enter the number of disks : ");
scanf("%d",&n);
printf("The Tower of Hanoi involves the moves :\n \n");
towers(n,'A','C','B');

return 0;
}
```

The main program

```
int main(void) {
int n;
printf("Enter the number of disks : ");
scanf("%d",&n);
printf("The Tower of Hanoi involves the moves :\n \n");
towers(n,'A','C','B');
printf("\n \n");
return 0;
}
```

The recursive function

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){




}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */




}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */



/* Move top n-1 disks from A to B, using C as auxiliary */



}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */



/* Move top n-1 disks from A to B, using C as auxiliary */

/* Move remaining disks from A to C*/



}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */




/* Move top n-1 disks from A to B, using C as auxiliary */

/* Move remaining disks from A to C*/


/* Move n-1 disks from B to C using A as auxiliary */

}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */
  if(n==1) {


    }
/* Move top n-1 disks from A to B, using C as auxiliary */

/* Move remaining disks from A to C*/


/* Move n-1 disks from B to C using A as auxiliary */

}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */
  if(n==1) { printf("\n Move disk 1 from peg %c
                          to peg %c", frompeg,topeg);
             return;
    }
/* Move top n-1 disks from A to B, using C as auxiliary */

/* Move remaining disks from A to C*/


/* Move n-1 disks from B to C using A as auxiliary */

}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */
  if(n==1) { printf("\n Move disk 1 from peg %c
                          to peg %c", frompeg,topeg);
             return;
    }
/* Move top n-1 disks from A to B, using C as auxiliary */
  towers(n-1,frompeg,auxpeg,topeg);
/* Move remaining disks from A to C*/


/* Move n-1 disks from B to C using A as auxiliary */

}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */
  if(n==1) { printf("\n Move disk 1 from peg %c
                          to peg %c", frompeg,topeg);
             return;
    }
/* Move top n-1 disks from A to B, using C as auxiliary */
  towers(n-1,frompeg,auxpeg,topeg);
/* Move remaining disks from A to C*/
   printf("\n Move disk %d from peg %c to peg %c",
          n,frompeg,topeg);
/* Move n-1 disks from B to C using A as auxiliary */

}
```

The recursive function

```
void towers(int n,char frompeg,char topeg,char auxpeg){
/* If only 1 disk, make the move and return */
  if(n==1) { printf("\n Move disk 1 from peg %c
                          to peg %c", frompeg,topeg);
             return;
    }
/* Move top n-1 disks from A to B, using C as auxiliary */
  towers(n-1,frompeg,auxpeg,topeg);
/* Move remaining disks from A to C*/
   printf("\n Move disk %d from peg %c to peg %c",
          n,frompeg,topeg);
/* Move n-1 disks from B to C using A as auxiliary */
  towers(n-1,auxpeg,topeg,frompeg);
}
```

Outline

Generating Permutations

- Suppose we can generate permutations of all $n-1$ numbers.

Generating Permutations

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:

Generating Permutations

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:
- Generate permutations of all the numbers $2, 3, \ldots, n$ and add 1 to the beginning.

Generating Permutations

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:
- Generate permutations of all the numbers $2, 3, \ldots, n$ and add 1 to the beginning.
- Generate permutations of all the numbers $1, 3, 4, \ldots, n$ and add 2 to the beginning.

Generating Permutations

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:
- Generate permutations of all the numbers $2, 3, \ldots, n$ and add 1 to the beginning.
- Generate permutations of all the numbers $1, 3, 4, \ldots, n$ and add 2 to the beginning.
- $\ldots$

Generating Permutations

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:
- Generate permutations of all the numbers $2, 3, \ldots, n$ and add 1 to the beginning.
- Generate permutations of all the numbers $1, 3, 4, \ldots, n$ and add 2 to the beginning.
- $\ldots$
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the beginning.

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{




}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{
int i;









}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{
int i;
if(k==n) {

}

}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{
int i;
if(k==n) {
  PrintPermutation(matrix,n); (*number_per)++;
}




}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{
int i;
if(k==n) {
  PrintPermutation(matrix,n); (*number_per)++;
}
else {









}

}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{
int i;
if(k==n) {
  PrintPermutation(matrix,n); (*number_per)++;
}
else {
  for(i=k;i<n;i++){



}
}

}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{
int i;
if(k==n) {
  PrintPermutation(matrix,n); (*number_per)++;
}
else {
  for(i=k;i<n;i++){
    swap(matrix,i,k);

    swap(matrix,k,i);
}
}

}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{
int i;
if(k==n) {
  PrintPermutation(matrix,n); (*number_per)++;
}
else {
  for(i=k;i<n;i++){
    swap(matrix,i,k);
    Permute(matrix,k+1,n,number_per);
    swap(matrix,k,i);
}
}

}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per)
{
int i;
if(k==n) {
  PrintPermutation(matrix,n); (*number_per)++;
}
else {
  for(i=k;i<n;i++){
    swap(matrix,i,k);
    Permute(matrix,k+1,n,number_per);
    swap(matrix,k,i);
}
}
return;
}
```

The main function call

```
int main(void){

 return 0;
}
```

The main function call

```
int main(void){
int i, m, *matrix, number_per=0;









 return 0;
}
```

The main function call

```
int main(void){
int i, m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);









return 0;
}
```

The main function call

```
int main(void){
int i, m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);
matrix = allocate1D(m);









return 0;
}
```

The main function call

```
int main(void){
int i, m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);
matrix = allocate1D(m);
/*Fill the matrix with numbers from 1 to m */
for(i=0;i<m;i++)
  matrix[i] = i+1;



return 0;
}
```

The main function call

```
int main(void){
int i, m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);
matrix = allocate1D(m);
/*Fill the matrix with numbers from 1 to m */
for(i=0;i<m;i++)
  matrix[i] = i+1;

Permute(matrix,0,m,&number_per);

return 0;
}
```

The main function call

```
int main(void){
int i, m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);
matrix = allocate1D(m);
/*Fill the matrix with numbers from 1 to m */
for(i=0;i<m;i++)
  matrix[i] = i+1;

Permute(matrix,0,m,&number_per);
printf("\n Total permutations::>%d\n",number_per);
return 0;
}
```

Generating Permutations – Another way

- Suppose we can generate permutations of all $n-1$ numbers.

Generating Permutations – Another way

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:

Generating Permutations – Another way

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the 1st position.

Generating Permutations – Another way

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the 1st position.
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the 2nd position.

Generating Permutations – Another way

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the 1st position.
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the 2nd position.
- ...

Generating Permutations – Another way

- Suppose we can generate permutations of all $n-1$ numbers.
- To generate permutations of n numbers, do the following:
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the 1st position.
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the 2nd position.
- ...
- Generate permutations of all the numbers $1, 2, \ldots, n-1$ and add n to the last position.

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per){
int i;









return;
}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per){
int i;
if(k==0) {
   PrintPermutation(matrix,n); (*number_per)++;
}









return;
}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per){
int i;
if(k==0) {
   PrintPermutation(matrix,n); (*number_per)++;
}
  else {









}
return;
}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per){
int i;
if(k==0) {
   PrintPermutation(matrix,n); (*number_per)++;
}
  else {
   for(i=0;i<n;i++){









  }
}
return;
}
```

The Basic Function

```
void Permute(int *matrix, int k,int n, int *number_per){
int i;
if(k==0) {
   PrintPermutation(matrix,n); (*number_per)++;
}
  else {
   for(i=0;i<n;i++){
     if(matrix[i]==0) {
                matrix[i]=k;
                Permute(matrix,k-1,n,number_per);
                matrix[i]=0;
      }
  }
}
return;
}
```

The main function call

```
int main(void){




return 0;
}
```

The main function call

```
int main(void){
int m, *matrix, number_per=0;




return 0;
}
```

The main function call

```
int main(void){
int m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);



return 0;
}
```

The main function call

```
int main(void){
int m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);
matrix = allocate1D(m);


return 0;
}
```

The main function call

```
int main(void){
int m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);
matrix = allocate1D(m);
Permute(matrix,m,m,&number_per);

return 0;
}
```

The main function call

```
int main(void){
int m, *matrix, number_per=0;
printf("\n How many numbers you want to permute::>");
scanf("%d",&m);
matrix = allocate1D(m);
Permute(matrix,m,m,&number_per);
printf("\n Total permutations::>%d\n",number_per);
return 0;
}
```