Recursion

- A <u>recursive function</u> contains a call to itself:

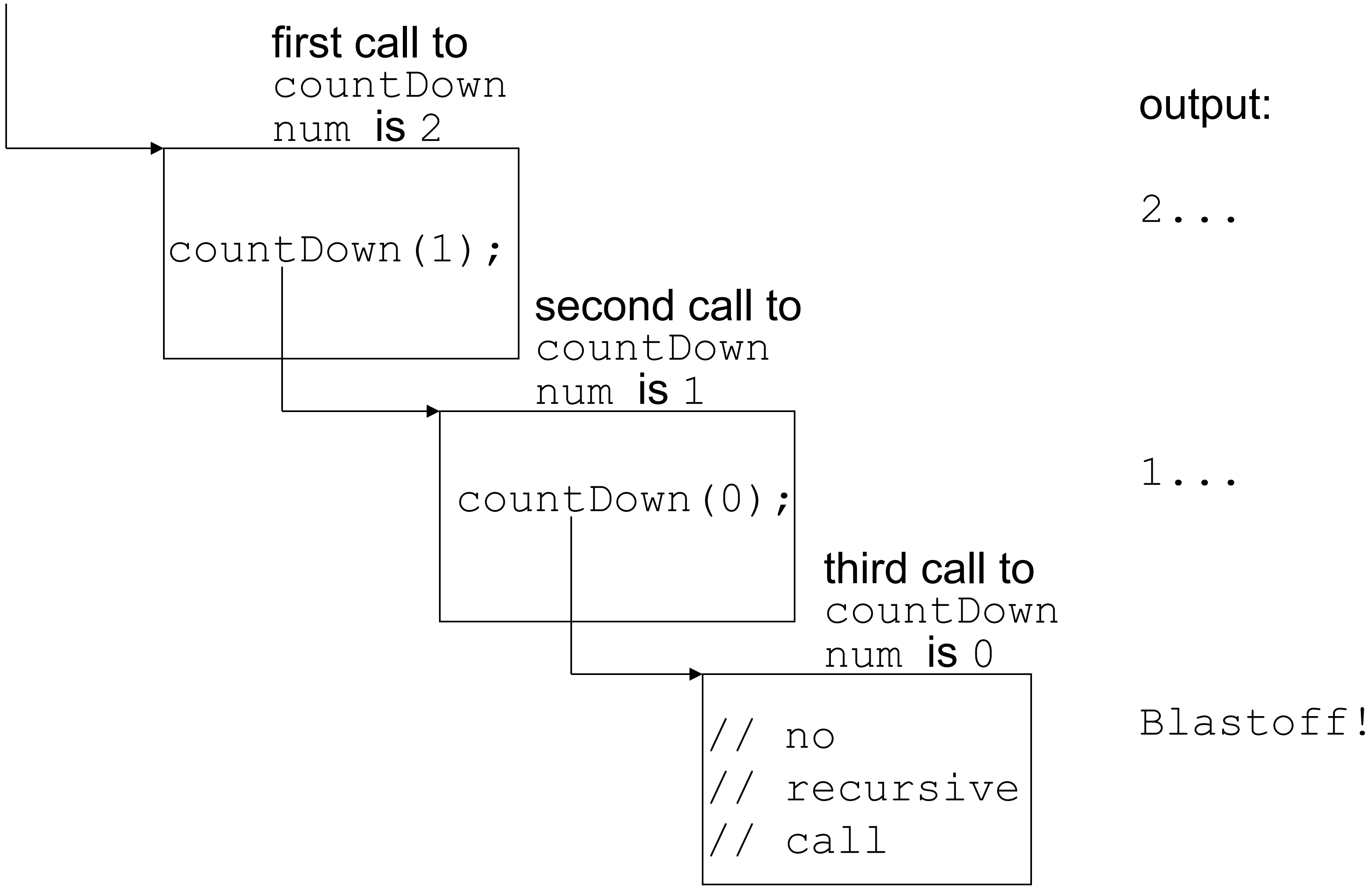
```
void countDown(int num)
{
     if (num == 0)
          cout << "Blastoff!";
     else
     {
          cout << num << "...\n";
          countDown(num-1);
     }
}
```

What happens when Called?

If a program contains a line like countDown(2);

1. countDown(2) generates the output 2..., then it calls countDown(1)
2. countDown(1) generates the output 1..., then it calls countDown(0)
3. countDown(0) generates the output Blastoff!, then returns to countDown(1)
4. countDown(1) returns to countDown(2)
5. countDown(2)returns to the calling function

What happens when called?



Recursive functions - purpose

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
- The simpler-to-solve problem is known as the base case
- Recursive calls stop when the base case is reached
- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call
- Recursion uses a process of breaking a problem down into smaller problems until the problem can be solved

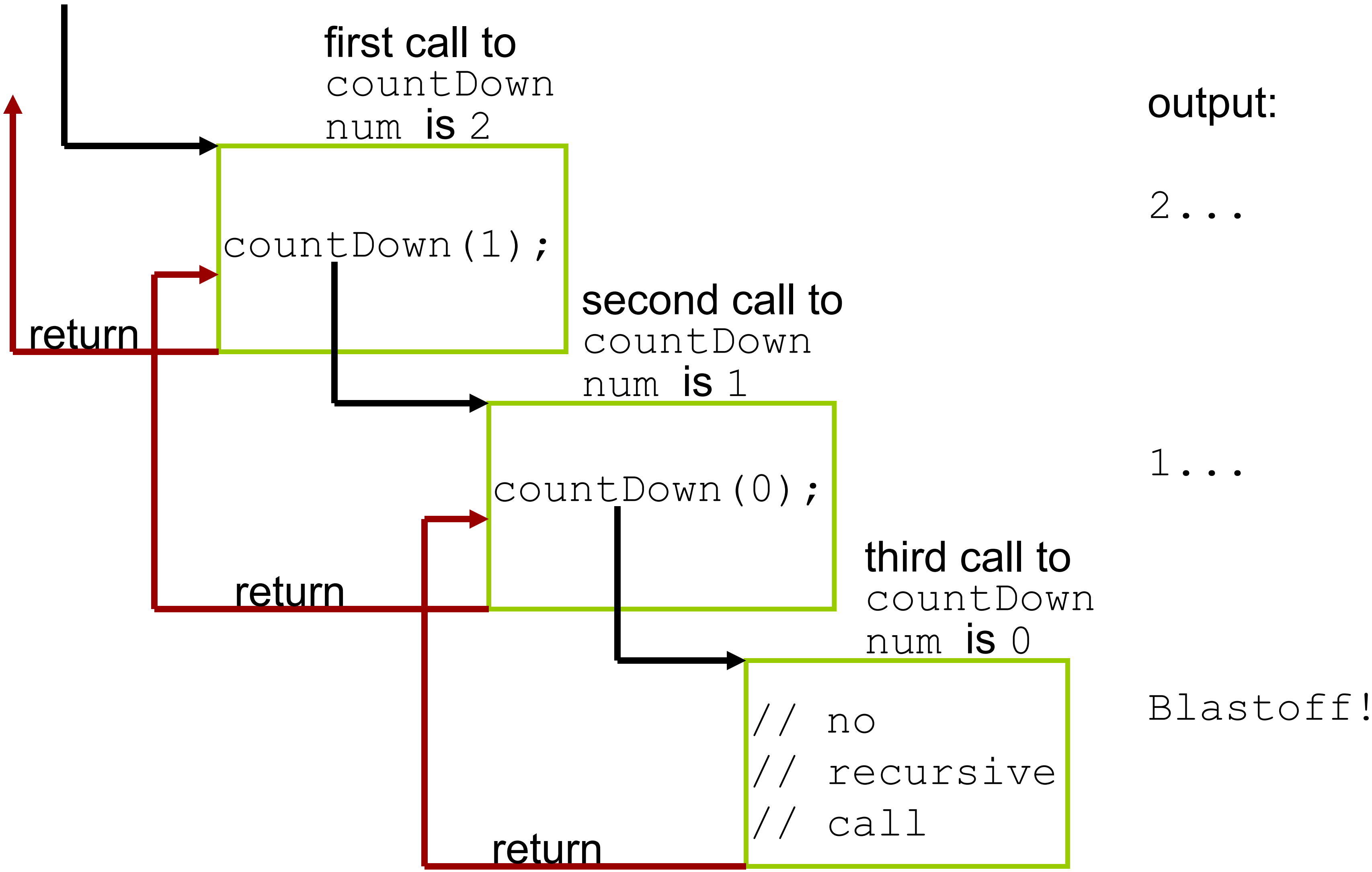
```
void countDown(int num)
{
     if (num == 0)
        cout << "Blastoff!";
     else
     {
        cout << num << "...\n";
        countDown(num-1);
     }
}
```

- In the countDown function, a different value is passed to the function each time it is called Eventually, the parameter reaches the value in the test, and the recursion stops

What Happens When Called?

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

What happens when called?



Types of Recursion

- Direct
  - a function calls itself
- Indirect
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls …, which calls function A

Recursive factorial functions

- The factorial function:

  $n! = n*(n-1)*(n-2)*...*3*2*1$ if $n > 0$

  $n! = 1$ if $n = 0$
- Can compute factorial of n if the factorial of (n-1) is known:

  $n! = n * (n-1)!$
- n = 0 is the base case

```
int factorial (int num)
{
    if (num > 0)
            return num * factorial(num - 1);
 else
      return 1;
}
```

```
int main()
{
  int x = 5;
  cout << "Factorial of "
       << x << " is "
       << factorial(x)
       << endl;
    return 0;
}
```

Factorial of 5 is 120

Recursive GCD functions

- Greatest common divisor (gcd) is the largest factor that two integers have in common
- Computed using Euclid's algorithm:

  gcd(x, y) = y if y divides x evenly

  gcd(x, y) = gcd(y, x % y) otherwise
- gcd(x, y) = y is the base case

```
int gcd(int x, int y)
{
      if (x % y == 0)
            return y;
      else
            return gcd(y, x % y);
}
```

GCD(8,12)=4

```
int main()
{
   int x = 8, y = 12;
   cout << "GCD("
         << x  <<  ","  <<  y  <<  ")="
         << gcd(x,y)
         << endl;

   return 0;
}
```

Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:

  0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- After the starting 0, 1, each number is the sum of the two preceding numbers
- Recursive solution:

  fib(n) = fib(n – 1) + fib(n – 2);

- Base cases: n <= 0, n == 1

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

Recursive Linked List Operations

- Recursive functions can be members of a linked list class
- Some applications:
  - Compute the size of (number of nodes in) a list
  - Traverse the list in reverse order
  - Print the elements of the array in reversed order

```
template<class T>
void LinkedList<T>::PrintBackward(Node<T> *ptr) const
{
    T x = ptr->info;
    if (ptr->next != nullptr)
        PrintBackward(ptr->next);

    std::cout << x << std::endl;
}

template<class T>
void LinkedList<T>::print(){
    PrintBackward(head);
}
```

```
int main()
{
    LinkedList<std::string> lst;
    lst.push_back("FCI");
    lst.push_back("FOE");
    lst.push_back("FOM");

    lst.print();

    return 0;
}
```

```
FOM
FOE
FCI
```

Count elements of Linked List Recursively

- Uses a pointer to visit each node
- Algorithm:
  - pointer starts at head of list
  - If pointer is null pointer, return 0 (base case)

    else, return 1 + number of nodes in the list pointed to by current node

```
template<class T>
int LinkedList<T>::CountI() const {
    Node<T> *ptr = head;
    int counter = 0;
    while (ptr!=nullptr) {
        counter++;
        ptr=ptr->next;
    }
    return counter;
}
```

```
template<class T>
int LinkedList<T>::Count(Node<T> *ptr) const {
    if (ptr != nullptr)
        return 1+Count(ptr->next);
    else
        return 0;
}
template<class T>
int LinkedList<T>::Counter(){
    return Count(head);
}
```

```
int main()
{
    LinkedList<std::string> lst;
    lst.push_back("FCI");
    lst.push_back("FOE");
    lst.push_back("FOM");
    std::cout << "size = "
              << lst.Counter()
              << std::endl;
    return 0;
}
```

```
size = 3
```

Recursive Binary Search Function

- Binary search algorithm can easily be written to use recursion
- Base cases: desired value is found, or no more array elements to search
- Algorithm (array in ascending order):
  - If middle element of array segment is desired value, then done
  - Else, if the middle element is too large, repeat binary search in first half of array segment
  - Else, if the middle element is too small, repeat binary search on the second half of array segment

Recursive Binary Search Function

```
int BSearchR(int a[], int first, int last, int key)
{
    if (first <= last) {
        int mid = (first + last) / 2;
        if (key == a[mid])
            return mid;
        else if (key < a[mid])
            return BSearchR(a, first, mid-1, key);
        else
            return BSearchR(a, mid+1, last, key);
    }
    return -1;
}
```

```
int main() {
 int a[]={1,2,3,4,5,6,7,8,9,10,11,12,13};
 std::cout << BSearchR(a,0,13,3)
              << std::endl;
 std::cout << BSearchI(a,13,3)
              << std::endl;
 return 0;
 }
```

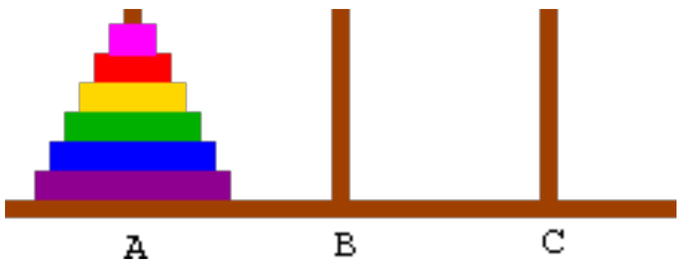
2
2

```
int BSearchI(int a[], int size, int value){
    int low = 0;
    int high = size - 1;
    int mid;
    while(low <= high){
        mid = (low + high)/2;

        if (value == a[mid]){
            return mid;
        }
        else if (value > a[mid]){
            low = mid + 1;
        }
        else
        {
            high = mid - 1;
        }
    }
    return -1;
}
```

9
–15

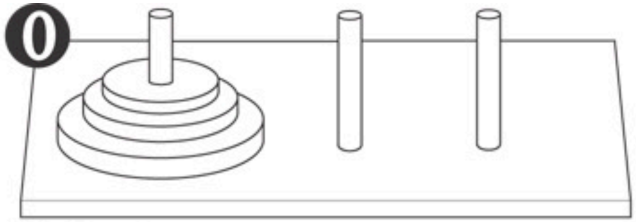
Recursive Towers of Hanoi

- The Towers of Hanoi is a mathematical game that is often used to demonstrate the power of recursion.
- The game uses three pegs and a set of discs, stacked on one of the pegs.

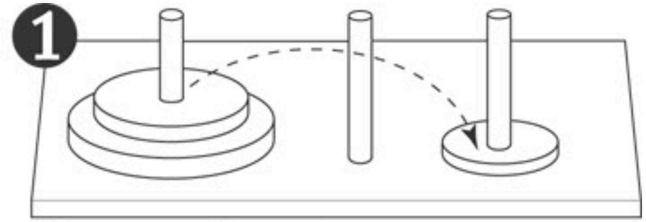


- The object of the game is to move the discs from the first peg to the third peg. Here are the rules:
  - Only one disc may be moved at a time.
  - A disc cannot be placed on top of a smaller disc.
  - All discs must be stored on a peg except while being moved.

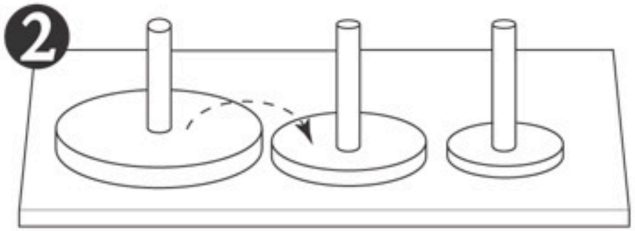
Recursive Towers of Hanoi



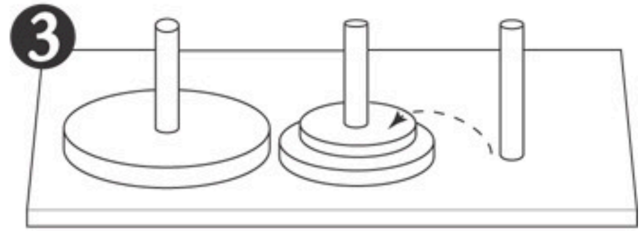
Original setup.



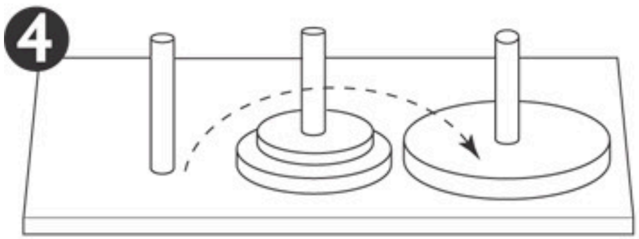
First move: Move disc 1 to peg 3.



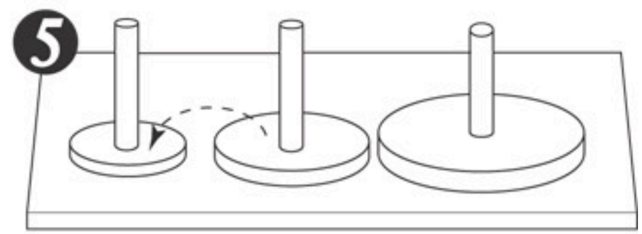
Second move: Move disc 2 to peg 2.



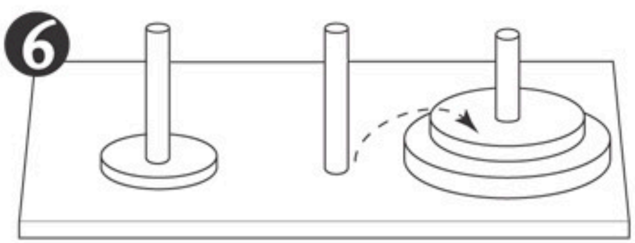
Third move: Move disc 1 to peg 2.



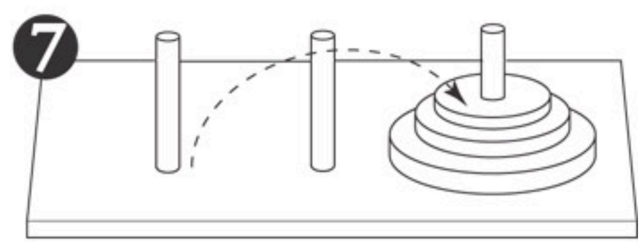
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.



Seventh move: Move disc 1 to peg 3.

Recursive Towers of Hanoi

- The following statement describes the overall solution to the problem:
  - Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.

- Algorithm
  - *To move n discs from peg A to peg C, using peg B as a temporary peg:*
    If n > 0 Then
    Move n – 1 discs from peg A to peg B, using peg C as a temporary peg.

    Move the remaining disc from the peg A to peg C.

    Move n – 1 discs from peg B to peg C, using peg A as a temporary peg.

    End If

Recursive Towers of Hanoi

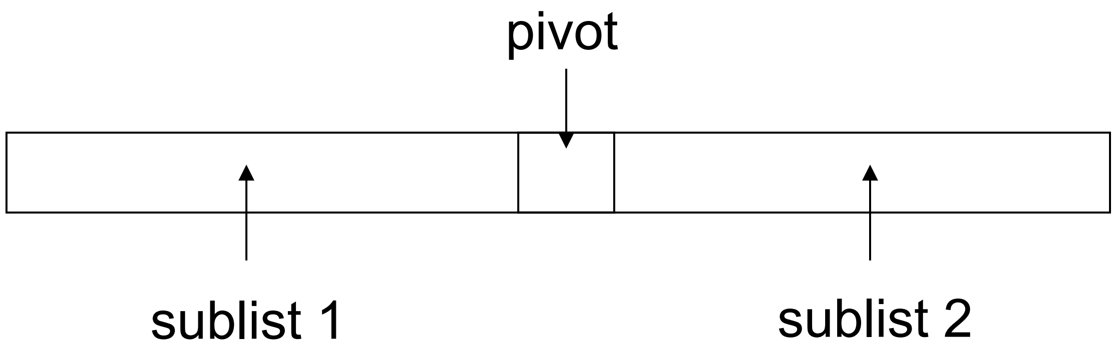
```
int moves(0);

void Hanoi(int m, char a, char b, char c) {
    moves++;
    if (m == 1) {
        cout << "Move disc " << m << " from " << a << " to " << c << endl;
    } else {
        Hanoi (m-1, a,c,b);
        cout << "Move disc " << m << " from " << a << " to " << c << endl;
        Hanoi (m-1,b,a,c);
    }
}
```

```
int main()
{
    int discs;
    cout << "Enter the number of discs: " << endl;
    cin >> discs;
    Hanoi(discs, 'A', 'B', 'C');
    cout << "It took " << moves << " moves. " << endl;
    return 0;
}
```

```
Enter the number of discs: 3
Program ended with exit code: 03
Move disc 1 from A to C
Move disc 2 from A to B
Move disc 1 from C to B
Move disc 3 from A to C
Move disc 1 from B to A
Move disc 2 from B to C
Move disc 1 from A to C
It took 7 moves.
```

Recursive Quick sort Algorithm

- Recursive algorithm that can sort an array or a linear linked list
- Determines an element/node to use as pivot value:



- Once pivot value is determined, values are shifted so that elements in sublist1 are < pivot and elements in sublist2 are > pivot
- Algorithm then sorts sublist1 and sublist2
- Base case: sublist has size 1

Recursive Quick sort Algorithm

```
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition (int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high- 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

Sorted array: 1:5:7:8:9:10:

```
void quickSort(int arr [ ], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
void printArray(int arr[], int size) {
    int i;
    for (i=0; i < size; i++)
          cout << arr[i] << ":";
    cout << endl;
}
```

```
int main() {
    int arr [ ] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
```

Recursive vs Iteration

- Benefits (+), disadvantages(-) for recursion:
  - \+ Models certain algorithms most accurately
  - \+ Results in shorter, simpler functions
  - May not execute very efficiently
- Benefits (+), disadvantages(-) for iteration:
  - \+ Executes more efficiently than recursion
  - Often is harder to code or understand