3. Recursion

- What is Recursion
- What it is good for and what it is not good for
- What are the characteristics of recursion
- How is a recursive function executed
- How is recursive function call implemented.

What is Recursion

- Self referential (defined in terms of itself)
- The laughing-cow (*la vache qui rit*) package shows a cow wearing laughing-cow packages as earrings, which show a cow wearing laughing-cow packages as earrings which …

Other examples

A **linked list** is:
a) empty, or
b) has a head (first element)
and a *tail*, which is a **linked list**

A **tree** is:
a) empty, or
b) has a *root*,
and *left* and/or *right* (sub-) **trees**

Factorial function

(the classic example!)

Factorial 5, written 5!, is:
$5 \times 4 \times 3 \times 2 \times 1$
and 6! is
$6 \times 5 \times 4 \times 3 \times 2 \times 1$, so $6 \times 5!$

Factorial function, for non-negative integers is:
a) $0! = 1$
b) if $n > 0$, then $n! = n \times (n - 1)!$

In C

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

Caution: inefficient

Useful recursion

- To be useful the recursion must *terminate*, so there must be *at least one* non-recursive case
  such as: 0!
- as well as recursive cases.
  such as: n * (n – 1)!

Infinite recursion

```
void TellStory(){
  printf("%s", "It was a dark and stormy night ");
  printf("%s", "and the captain said to the mate ");
  printf("%s", ":`Tell us a story mate' ");
  printf("%s", " and this is the story he told …");
  TellStory();
}
```


Recursive Programming

- Consider the problem of computing the sum of all the integers between 1 and any positive integer N
- This problem can be recursively defined as:

$$\begin{aligned}\sum_{i=1}^{N} i &= N + \sum_{i=1}^{N-1} i = N + N-1 + \sum_{i=1}^{N-2} i \\ &= N + N-1 + N-2 + \sum_{i=1}^{N-3} i \\ &\vdots\end{aligned}$$

Recursive Programming

```
// This method returns the sum of 1 to count
int sum (int count)
{
    if (count == 1)
        return 1;
    else
        return count + sum (count-1);
}
```


Recursive Programming

- 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 and more efficient
- 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

Indirect Recursion

- A function invoking itself is considered to be *direct recursion*
- A function could invoke another function, which invokes another, etc., until eventually the original function is invoked again
- For example, function f1 could invoke f2, which invokes f3, which in turn invokes f1 again
- This is called *indirect recursion*, and requires all the same care as direct recursion
- It is often more difficult to trace and debug

Length of a list

a) the length of an empty list is 0

b) the length of a (non-empty) list is:

1 + the length of the tail of the list

Length of a list in C

```
int length_v1 (node* p){ /* iteration */
  int countNodes = 0;
  while (p) do {
    countNodes++;
    p = p->next
  }
  return countNodes;
}

int length_v2(node* p){ /* recursion */
    if (p) return (1 + length_v2(p->next));
    else return 0;
}
```


Traversing a list: iterative

Traversing a (singly) linked list *iteratively* in the forward direction is easy:

```
void traverse (node* p){
  while (p) {
    process(p->data); /* assume a process function */
    p = p->next;
  }
}
```

Traversing iteratively in the backward direction is **hard** (no pointers, so need to *stack* return pointers)

Traversing a list: recursive, forward

Traversing a (singly) linked list *recursively* in the forward direction is easy:

```
void traverse (node* p){
    if (p){
        process(p->data);
        traverse(p->next);
    }
}
```


Traversing a list: recursive, backward

Traversing a (singly) linked list recursively in the **backward** direction is also easy:

```
void reverseTraverse (node* p){
    if (p){
        reverseTraverse(p->next);
        process(p->data);
    }
}
```


How recursion works

- When a function is *called* , its parameters, local variables and return address are *stacked* on the function-call stack.
- Nested calls lead to deeper stacking.
- A call of a function to itself is just another nested call.

When not to use recursion

- Don't use a recursive approach when a simple iterative approach is available
- Examples: searching, traversing and inserting in a list is easy to do iteratively
- Traversing a list backwards ('*backtracking*') is easy to do recursively but hard to do iteratively.

When not to use recursion: example

Fibonacci Numbers:
$fib_0 = 0$
$fib_1 = 1$
$fib_n = fib_{n-1} + fib_{n-2}$, for $n > 0$

```
int fib(n: integer){ /* doubly recursive */
    if (n == 0) return 0;
   else if (n ==1) return 1;
   return  (fib(n – 1) + fib (n – 2));
}
```

Very inefficient: values repeatedly calculated, then 'forgotten'

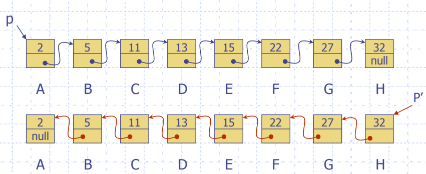
A Better way:

```
int fib (n: integer){ /* iterative */
   int i, x, y, z;

   i = 1; x = 1; y = 0;
   while (i != n) {
        z = x;
        i++;
        x = x + y;
        y = z;
    }
   return x;
}
```

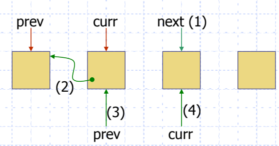

Reverse a list:

Write a function list_rev that takes a pointer to a singly linked list of nodes and reverses links, returns the pointer to the new head of the resulting list.




```
node* list_rev(node *curr){
   node *prev = NULL; *next;

   while(curr){
        next = curr->next; // (1)
        curr->next = prev; // (2)
        prev = curr;         // (3)
        curr = next;         // (4)
   }
   return prev;
}
```




```
node *list_rev_recursion(node *curr, node *prev) {
node *revHead;

   if (curr == NULL)
        revHead = prev;
   else {
        revHead = list_rev_recursion(curr->next, curr);
        curr->next = prev;
      }
   return revHead;
}
```

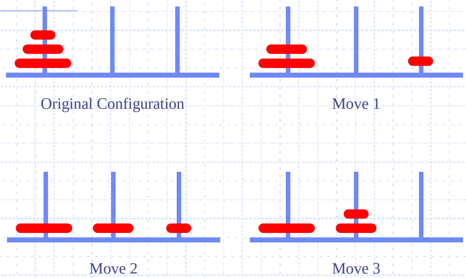
Initial method call should be

head = list_rev_recursion(head, NULL)

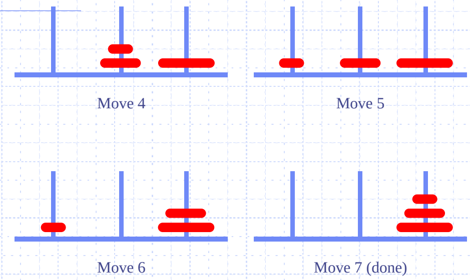
Towers of Hanoi

- The *Towers of Hanoi* is a puzzle made up of three vertical pegs and several disks that slide on the pegs
- The disks are of varying size, initially placed on one peg with the largest disk on the bottom with increasingly smaller ones on top
- The goal is to move all of the disks from one peg to another under the following rules:
  - Only one disk can be moved at a time
  - A bigger disk can never be placed on top of a smaller one

Three pegs: src, tmp, dst



Towers of Hanoi



Towers of Hanoi

- An iterative solution to the Towers of Hanoi is quite complex
- A recursive solution is much shorter and more elegant

```
if (n == 1) {
        (move one disk directly from src to dst)
} else {
        (move a tower of n-1 disks from src to tmp)
        (move one disk directly from src to dst)
        (move a tower of n-1 disk from tmp to dst)
}
```


Towers of Hanoi

```
void MoveTower(int n, char src,
               char dst, char tmp)
{
  if (n == 1) {
     MoveSingleDisk(src, dst);
  } else {
     MoveTower(n-1, src, tmp, dst);
     MoveSingleDisk(src, dst);
     MoveTower(n-1, tmp, dst, src);
  }
}
```


Three Characteristics of Recursion

- Calls itself recursively
- Has some terminating condition
- Moves “closer” to the terminating condition.

Two Flavors of Recursion

```
if (terminating condition){
  do final actions
} else {
  move one step closer to terminating condition
  recursive call(s)
}
```

- or -

```
if (!(terminating condition)){
  move one step closer to terminating condition
  recursive call(s)
}
```

Analysis of a Recursive Algorithm

- To evaluate the computing costs of an algorithm, we need to analyze it.
- The first step in algorithm analysis is to find out the number of times the "primitive operation" is executed.
- In doing so, we express the number of times as a function of n, which is the input size.

Analysis of a Recursive Algorithm

Steps in analyzing a recursive algorithm:

1. Decide on the input size n.
2. Identify the primitive operation.
3. Set up a recurrence relation, with an appropriate initial condition, for the number of times the primitive operation is executed.
4. Solve the recurrence.

Analysis of a Recursive Algorithm

Example: recursive computing of factorial

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

Input size: n

Primitive operation: multiplication

Analysis of a Recursive Algorithm

Example: recursive computing of factorial

Recurrence relation for the number of times multiplication is executed and initial condition:

$M(n) = M(n-1) + 1$ for $n > 0$

$M(0) = 0$

Analysis of a Recursive Algorithm

Solve the recurrence relation using the method of backward substitution:

$M(n) = M(n-1) + 1$ — substitute $M(n-1) = M(n-2)+1$

$= [M(n-2)+1] + 1 = M(n-2)+2$ — substitute $M(n-2) = M(n-3)+1$

$= [M(n-3)+1] + 2 = M(n-3)+3$ — substitute $M(n-3) = M(n-4)+1$

…

$M(n) = M(n-1) +1 = \ldots = M(n-i) + i = \ldots$

$M(n) = M(n-i) + i$, $M(0) = 0$

To eliminate the M term from the right hand side, let $i = n$ and use the initial condition, we have

$M(n) = n$

Analysis of a Recursive Algorithm

Example: Hanoi tower

```
void MoveTower(int n, char src,
               char dst, char tmp)
{
  if (n == 1) {
     MoveSingleDisk(src, dst);
   } else {
     MoveTower(n-1, src, tmp, dst);
     MoveSingleDisk(src, dst);
     MoveTower(n-1, tmp, dst, src);
   }
}
```

Analysis of a Recursive Algorithm

Example: Hanoi tower

1. Input size: n (the number of disks)
2. Primitive operation: Moving a disk
3. Recurrence and initial condition:

$$M(n) = M(n-1) + 1 + M(n-1)$$
$$= 2M(n-1) + 1 \qquad n>1$$
$$M(1) = 1$$

Analysis of a Recursive Algorithm

Example: Hanoi tower

$M(n)=2M(n-1)+1$ sub. $M(n-1)=2M(n-2)+1$
$=2[2M(n-2)+1]+1$
$=2^2M(n-2)+2+1$ sub. $M(n-2)=2M(n-3)+1$
$=2^2[2M(n-3)+1]+2+1$
$=2^3M(n-3)+2^2+2+1$
…
$M(n)= 2^iM(n-i)+2^{i-1}+2^{i-2}+\cdots+2+1$
$= 2^iM(n-i)+2^i-1$

Using $\sum_{j=0}^{j=i-1} 2^j = 2^i-1$

Analysis of a Recursive Algorithm

Example: Hanoi tower

$M(n)= 2M(n-1)+1 = 2^iM(n-i)+2^i-1 \quad n>1$
$M(1)= 1$

To use the initial condition, let i=n-1. We have

$M(n)= 2^{n-1}M[n-(n-1)]+2^{n-1}-1$
$= 2^{n-1}M(1)+2^{n-1}-1$
$= 2^{n-1}+2^{n-1}-1$
$= 2^n-1$

That is, to move n disks, the number of movements is 2^n-1.

Tracing The Recursion

To keep track of recursive execution, do what a computer does: maintain information on an activation stack.

Each stack frame contains:
- Module identifier and variables
- Any unfinished business

ModuleID:	Data values	Unfinished business

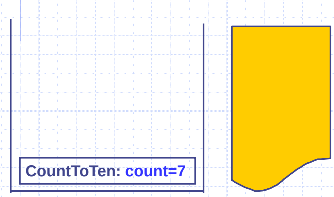
Work and Recursion

Problem: Count from N to 10.

```
void CountToTen(int count){
  if (count <= 10){
     printf("%d\n", count);   // work
     CountToTen(count + 1);   // recurse
  }
} //CountToTen
```

First do the work and then the recursive call!


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```



```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=7

7


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=7

7


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=8

CountToTen: count=7

7


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=8

CountToTen: count=7

7
8


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=8

CountToTen: count=7

7
8


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=9

CountToTen: count=8

CountToTen: count=7

7
8

```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=9
CountToTen: count=8
CountToTen: count=7

7
8
9


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=9
CountToTen: count=8
CountToTen: count=7

7
8
9


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7

7
8
9


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7

7
8
9
10


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7

7
8
9
10


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=11
CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7

7
8
9
10

```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=10

CountToTen: count=9

CountToTen: count=8

CountToTen: count=7

7
8
9
10


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=9

CountToTen: count=8

CountToTen: count=7

7
8
9
10


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=8

CountToTen: count=7

7
8
9
10


```
void CountToTen(int count){
  if (count < = 10){
    printf("%d\n", count);  // work
    CountToTen(count + 1);  // recurse
}
} //CountToTen
```

CountToTen: count=7

7
8
9
10

Return to the algorithm.

7
8
9
10

Reversing the Work and Recursion

Problem: Count from N to 10.

```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

Now the work will happen as the frames pop off the stack!

```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=7


```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=7


```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=8

CountToTen: count=7


```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=8

CountToTen: count=7


```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=9

CountToTen: count=8

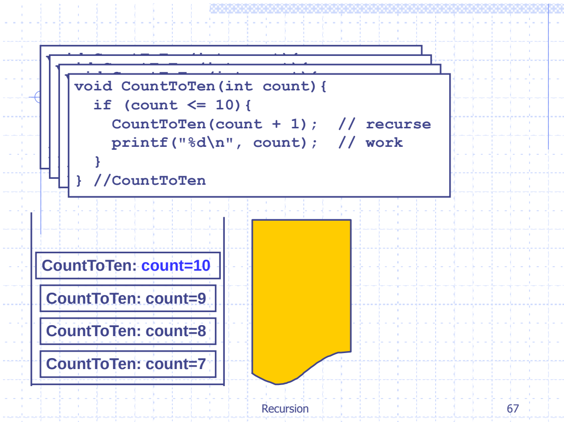
CountToTen: count=7

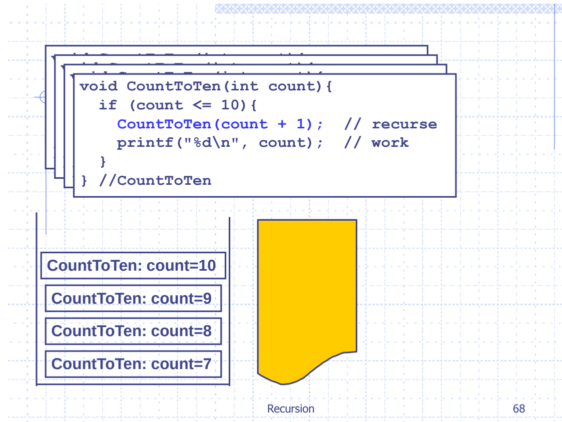

```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

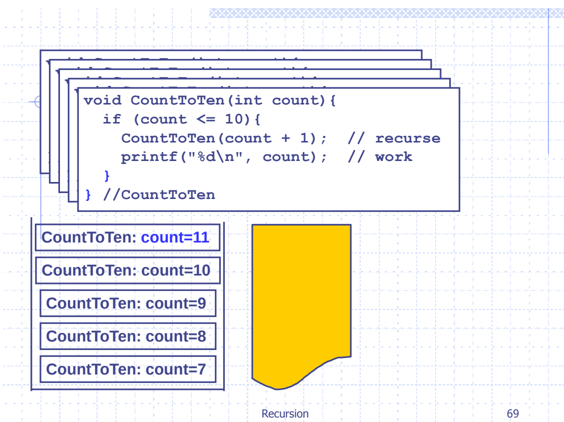
CountToTen: count=9

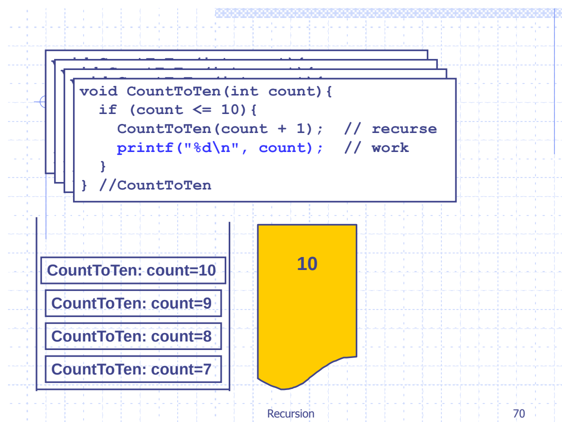
CountToTen: count=8

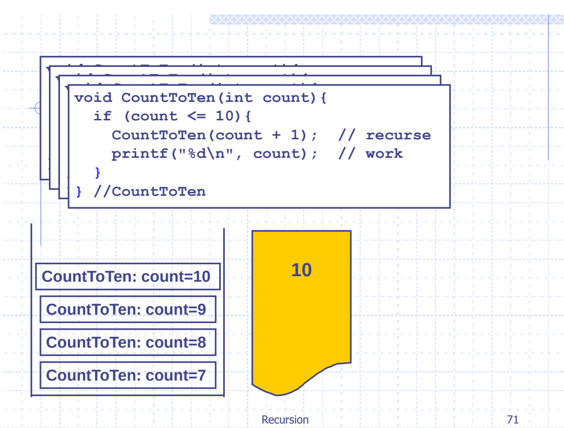
CountToTen: count=7

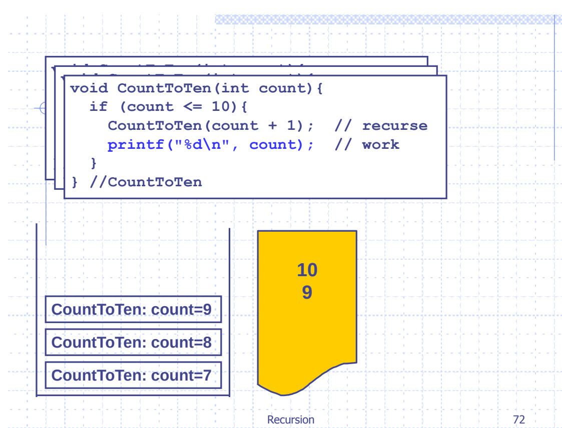

void CountToTen(int count){
if (count <= 10){
CountToTen(count + 1); // recurse
printf("%d\n", count); // work
}
} //CountToTen
CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7
Recursion
67


void CountToTen(int count){
if (count <= 10){
CountToTen(count + 1); // recurse
printf("%d\n", count); // work
}
} //CountToTen
CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7
Recursion
68


void CountToTen(int count){
if (count <= 10){
CountToTen(count + 1); // recurse
printf("%d\n", count); // work
}
} //CountToTen
CountToTen: count=11
CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7
Recursion
69


void CountToTen(int count){
if (count <= 10){
CountToTen(count + 1); // recurse
printf("%d\n", count); // work
}
} //CountToTen
CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7
10
Recursion
70


void CountToTen(int count){
if (count <= 10){
CountToTen(count + 1); // recurse
printf("%d\n", count); // work
}
} //CountToTen
CountToTen: count=10
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7
10
Recursion
71


void CountToTen(int count){
if (count <= 10){
CountToTen(count + 1); // recurse
printf("%d\n", count); // work
}
} //CountToTen
CountToTen: count=9
CountToTen: count=8
CountToTen: count=7
10
9
Recursion
72

```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=9

CountToTen: count=8

CountToTen: count=7

10
9


```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=8

CountToTen: count=7

10
9
8


```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=8

CountToTen: count=7

10
9
8


```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=7

10
9
8
7


```
void CountToTen(int count){
  if (count <= 10){
    CountToTen(count + 1);  // recurse
    printf("%d\n", count);  // work
  }
} //CountToTen
```

CountToTen: count=7

10
9
8
7

Return to the algorithm.

10
9
8
7