Introduction to Programming

Week – *7*, Lecture – *3*

Solving Problems Recursively

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

Recap – Solving problems iteratively

You already know how to solve problems with the help of loops

Recap – Solving problems iteratively

You already know how to solve problems with the help of loops

The overall idea is to express the computation in terms of variables…

Recap – Solving problems iteratively

You already know how to solve problems with the help of loops

The overall idea is to express the computation in terms of variables…

- … figure out how these variables change, and

Recap – Solving problems iteratively

You already know how to solve problems with the help of loops

The overall idea is to express the computation in terms of variables…

- … figure out how these variables change, and
- … when to stop the computation

Using loops to solve problems

```
Statements before the loop
Initialisation – setting values for important variables
{
    Statements, using different values of one or more variables
    Update to the values of one or more variables
    condition over one or more variables (to go out of the loop)
}
Statements after the loop
```

Recap – Solving problems iteratively

You already know how to solve problems with the help of loops

The overall idea is to express the computation in terms of variables…

- … figure out how these variables change, and
- … when to stop the computation

If you have figured it out, you can write loops to solve your problem

Recap – Solving problems iteratively

You already know how to solve problems with the help of loops

The overall idea is to express the computation in terms of variables…

- … figure out how these variables change, and
- … when to stop the computation

If you have figured it out, you can write loops to solve your problem

We used the example of Factorial Calculation as the example to show the process

- We saw how it could be done using three different loops

The idea of recursion

Recursion is a programming concept, where a function *calls itself* to solve a problem

- We will see some examples in a minute... just grasp the concepts right now

The idea of recursion

Recursion is a programming concept, where a function *calls itself* to solve a problem

- We will see some examples in a minute... just grasp the concepts right now

In case a function does not call itself, but calls another function…

- … which then calls the first function, then too, it is called recursion

The idea of recursion

Recursion is a programming concept, where a function *calls itself* to solve a problem

- We will see some examples in a minute... just grasp the concepts right now

In case a function does not call itself, but calls another function…

- … which then calls the first function, then too, it is called recursion
- Actually, there can be any number of functions in the chain, not just two…

The idea of recursion

Recursion is a programming concept, where a function *calls itself* to solve a problem

- We will see some examples in a minute... just grasp the concepts right now

In case a function does not call itself, but calls another function…

- … which then calls the first function, then too, it is called recursion
- Actually, there can be any number of functions in the chain, not just two…

While the former type is called *direct* recursion, the latter is called *indirect* recursion

Direct and Indirect Recursion

```
function f()
{
    ...
    f();
    ...
}
```

Direct Recursion

Direct and Indirect Recursion

```
function f()
{
    ...
    f();
    ...
}
```

Direct Recursion

```
function f()
{
    ...
    g();
    ...
}

function g()
{
    ...
    f();
    ...
}
```

Indirect Recursion – with 2 functions

Direct and Indirect Recursion

```
function f()
{
    ...
    f();
    ...
}
```

Direct Recursion

```
function f()
{
    ...
    g();
    ...
}

function g()
{
    ...
    f();
    ...
}
```

Indirect Recursion – with 2 functions

```
function f()
{
    ...
    g();
    ...
}

function g()
{
    ...
    h();
    ...
}

function h()
{
    ...
    f();
    ...
}
```

Indirect Recursion – with 3 functions

Solving problems recursively

Remember the *Principle of Mathematical Induction* or PMI?

- You use it to prove that a particular premise is true for all Natural Numbers

Solving problems recursively

Remember the *Principle of Mathematical Induction* or PMI?

- You use it to prove that a particular premise is true for all Natural Numbers

The idea of PMI, is that a premise, P(*n*) can be shown to hold for all natural numbers, if

- It holds for the *base case* – i.e. P(*0*) is true

Solving problems recursively

Remember the *Principle of Mathematical Induction* or PMI?

- You use it to prove that a particular premise is true for all Natural Numbers

The idea of PMI, is that a premise, P(n) can be shown to hold for all natural numbers, if

- It holds for the *base case* – i.e. P(0) is true
- Assuming that it holds for a natural number k – i.e. P(k) is true, P($k+1$) is also true

Solving problems recursively

Remember the *Principle of Mathematical Induction* or PMI?

- You use it to prove that a particular premise is true for all Natural Numbers

The idea of PMI, is that a premise, P(*n*) can be shown to hold for all natural numbers, if

- It holds for the *base case* – i.e. P(*0*) is true
- Assuming that it holds for a natural number *k* – i.e. P(*k*) is true, P(*k+1*) is also true

PMI is an example of a simple, yet powerful problem solving technique – *Divide and Conquer*

Solving problems recursively

Remember the *Principle of Mathematical Induction* or PMI?

- You use it to prove that a particular premise is true for all Natural Numbers

The idea of PMI, is that a premise, P(*n*) can be shown to hold for all natural numbers, if

- It holds for the *base case* – i.e. P(*0*) is true
- Assuming that it holds for a natural number *k* – i.e. P(*k*) is true, P(*k+1*) is also true

PMI is an example of a simple, yet powerful problem solving technique – *Divide and Conquer*

The core idea of Divide and Conquer, is to figure out how the problem can be solved

- for a base case – usually a case when the answer is trivial, or defined by some convention

Solving problems recursively

Remember the *Principle of Mathematical Induction* or PMI?

- You use it to prove that a particular premise is true for all Natural Numbers

The idea of PMI, is that a premise, P(*n*) can be shown to hold for all natural numbers, if

- It holds for the *base case* – i.e. P(*0*) is true
- Assuming that it holds for a natural number *k* – i.e. P(*k*) is true, P(*k+1*) is also true

PMI is an example of a simple, yet powerful problem solving technique – *Divide and Conquer*

The core idea of Divide and Conquer, is to figure out how the problem can be solved

- for a base case – usually a case when the answer is trivial, or defined by some convention
- and, how solutions to smaller problems or subproblems, can be combined to give the complete solution

Solving problems recursively

Remember the *Principle of Mathematical Induction* or PMI?

- You use it to prove that a particular premise is true for all Natural Numbers

The idea of PMI, is that a premise, P(*n*) can be shown to hold for all natural numbers, if

- It holds for the *base case* – i.e. P(*0*) is true
- Assuming that it holds for a natural number *k* – i.e. P(*k*) is true, P(*k+1*) is also true

PMI is an example of a simple, yet powerful problem solving technique – *Divide and Conquer*

The core idea of Divide and Conquer, is to figure out how the problem can be solved

- for a base case – usually a case when the answer is trivial, or defined by some convention
- and, how solutions to smaller problems or subproblems, can be combined to give the complete solution

In terms of programming, it is often easy to do this using recursion

- For instance, you write a function, and call it repeatedly, to construct a solution in a bottom-up fashion

Example 1 – (The return of…) Factorial

```
#include<stdio.h>

long factorial(int number)
{
	if(number < 0)
		return -1;
	else if(number == 0 || number == 1)
		return 1;
	else
		return number * factorial(number-1);
}

int main()
{
	int number;
	do
	{
		printf("Give me a small positive integer: ");
		scanf("%d", &number);
	}
	while(number < 0 || number > 20);

	printf("Calculated Factorial: %ld\n", factorial(number));
}
```

Example 1 – (The return of…) Factorial

```
#include<stdio.h>

long factorial(int number)
{
        if(number < 0)
                return -1;
        else if(number == 0 || number == 1)
                return 1;
        else
                return number * factorial(number-1);
}

int main()
{
        int number;
        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &number);
        }
        while(number < 0 || number > 20);

        printf("Calculated Factorial: %ld\n", factorial(number));
}
```

We have a function here to calculate factorial

Example 1 – (The return of…) Factorial

```
#include<stdio.h>

long factorial(int number)
{
        if(number < 0)
                return -1;
        else if(number == 0 || number == 1)
                return 1;
        else
                return number * factorial(number-1);
}

int main()
{
        int number;
        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &number);
        }
        while(number < 0 || number > 20);

        printf("Calculated Factorial: %ld\n", factorial(number));
}
```

We have a function here to calculate factorial

It takes as input, an integer, and returns its factorial

Example 1 – (The return of…) Factorial

```c
#include<stdio.h>

long factorial(int number)
{
	if(number < 0)
		return -1;
	else if(number == 0 || number == 1)
		return 1;
	else
		return number * factorial(number-1);
}

int main()
{
	int number;
	do
	{
		printf("Give me a small positive integer: ");
		scanf("%d", &number);
	}
	while(number < 0 || number > 20);

	printf("Calculated Factorial: %ld\n", factorial(number));
}
```

If it is a negative number, it returns `-1` – a convention to show an error

Example 1 – (The return of…) Factorial

```c
#include<stdio.h>

long factorial(int number)
{
        if(number < 0)
                return -1;
        else if(number == 0 || number == 1)
                return 1;
        else
                return number * factorial(number-1);
}

int main()
{
        int number;
        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &number);
        }
        while(number < 0 || number > 20);

        printf("Calculated Factorial: %ld\n", factorial(number));
}
```

If it is a negative number, it returns `-1` – a convention to show an error

This is really some validation, not actual factorial calculation

Example 1 – (The return of...) Factorial

```c
#include<stdio.h>

long factorial(int number)
{
        if(number < 0)
                return -1;
        else if(number == 0 || number == 1)
                return 1;
        else
                return number * factorial(number-1);
}

int main()
{
        int number;
        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &number);
        }
        while(number < 0 || number > 20);

        printf("Calculated Factorial: %ld\n", factorial(number));
}
```

If the number is `0` or `1`, the answer is simply, `1`

Example 1 – (The return of…) Factorial

```c
#include<stdio.h>

long factorial(int number)
{
        if(number < 0)
                return -1;
        else if(number == 0 || number == 1)
                return 1;
        else
                return number * factorial(number-1);
}

int main()
{
        int number;
        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &number);
        }
        while(number < 0 || number > 20);

        printf("Calculated Factorial: %ld\n", factorial(number));
}
```

If the number is `0` or `1`, the answer is simply, `1`

The base case here is actually when the number is `1`, we have to include `0`, because by convention, `0! = 1`

Example 1 – (The return of…) Factorial

```
#include<stdio.h>

long factorial(int number)
{
        if(number < 0)
                return -1;
        else if(number == 0 || number == 1)
                return 1;
        else
                return number * factorial(number-1);
}

int main()
{
        int number;
        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &number);
        }
        while(number < 0 || number > 20);

        printf("Calculated Factorial: %ld\n", factorial(number));
}
```

If the number is anything else, it can be calculated as:

`number * factorial(number-1)`

Example 1 – (The return of...) Factorial

```c
#include<stdio.h>

long factorial(int number)
{
        if(number < 0)
                return -1;
        else if(number == 0 || number == 1)
                return 1;
        else
                return number * factorial(number-1);
}

int main()
{
        int number;
        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &number);
        }
        while(number < 0 || number > 20);

        printf("Calculated Factorial: %ld\n", factorial(number));
}
```

If the number is anything else, it can be calculated as:
`number * factorial(number-1)`

Basically, we decide to solve the same problem (i.e., calculating factorial), but with a smaller sized input

Example 1 – (The return of...) Factorial

```c
#include<stdio.h>

long factorial(int number)
{
	if(number < 0)
		return -1;
	else if(number == 0 || number == 1)
		return 1;
	else
		return number * factorial(number-1);
}

int main()
{
	int number;
	do
	{
		printf("Give me a small positive integer: ");
		scanf("%d", &number);
	}
	while(number < 0 || number > 20);

	printf("Calculated Factorial: %ld\n", factorial(number));
}
```

Run this code in your head a few times, to convince yourself that it works !!

Example 1 – (The return of…) Factorial

```
#include<stdio.h>

long factorial(int number)
{
        if(number < 0)
                return -1;
        else if(number == 0 || number == 1)
                return 1;
        else
                return number * factorial(number-1);
}

int main()
{
        int number;
        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &number);
        }
        while(number < 0 || number > 20);

        printf("Calculated Factorial: %ld\n", factorial(number));
}
```

Run this code in your head a few times, to convince yourself that it works !!

Then try to understand how we solved a large problem by solving “smaller problems recursively”, and then “combining their solutions appropriately”

Example 2 – Binary Search (let's code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
        int mid;
        if(beg > end)
                return -1;
        mid = beg + (end - beg)/2;
        if(sorted_array[mid] == key)
                return mid;
        else if(sorted_array[mid] < key)
                return binary_search(sorted_array, mid+1, end, key);
        else
                return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
        int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int i, search_index, key;

        printf("Here's the sorted array that we have\n");
        for(i = 0; i < 10; i++)
                printf("%d ", array[i]);

        printf("\nEnter a key to search in the array: ");
        scanf("%d", &key);

        search_index = binary_search(array, 0, 9, key);
        if(search_index < 0)
                printf("The key is not in this array\n");
        else
                printf("The key is found on index %d\n", search_index);
}
```

Here, we have a function for performing a binary search in a part of a sorted integer array

Example 2 – Binary Search (let’s code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
	int mid;
	if(beg > end)
		return -1;
	mid = beg + (end - beg)/2;
	if(sorted_array[mid] == key)
		return mid;
	else if(sorted_array[mid] < key)
		return binary_search(sorted_array, mid+1, end, key);
	else
		return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
	int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
	int i, search_index, key;

	printf("Here's the sorted array that we have\n");
	for(i = 0; i < 10; i++)
		printf("%d ", array[i]);

	printf("\nEnter a key to search in the array: ");
	scanf("%d", &key);

	search_index = binary_search(array, 0, 9, key);
	if(search_index < 0)
		printf("The key is not in this array\n");
	else
		printf("The key is found on index %d\n", search_index);
}
```

Here, we have a function for performing a binary search in a <u>part</u> of a sorted integer array

The function expects the following inputs:

1. The sorted array
2. The beginning index of the part of the array to search
3. The ending index of the part of the array to search
4. The value to search

Example 2 – Binary Search (let’s code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
        int mid;
        if(beg > end)
                return -1;
        mid = beg + (end - beg)/2;
        if(sorted_array[mid] == key)
                return mid;
        else if(sorted_array[mid] < key)
                return binary_search(sorted_array, mid+1, end, key);
        else
                return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
        int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int i, search_index, key;

        printf("Here's the sorted array that we have\n");
        for(i = 0; i < 10; i++)
                printf("%d ", array[i]);

        printf("\nEnter a key to search in the array: ");
        scanf("%d", &key);

        search_index = binary_search(array, 0, 9, key);
        if(search_index < 0)
                printf("The key is not in this array\n");
        else
                printf("The key is found on index %d\n", search_index);
}
```

Here, we have a function for performing a binary search in a <u>part</u> of a sorted integer array

The function expects the following inputs:
1. The sorted array
2. The beginning index of the part of the array to search
3. The ending index of the part of the array to search
4. The value to search

If the value is found in the part, it returns the index at which it was found, otherwise, it returns -1 (another convention)

Example 2 – Binary Search (let's code it !)

```c
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
	int mid;
	if(beg > end)
		return -1;
	mid = beg + (end - beg)/2;
	if(sorted_array[mid] == key)
		return mid;
	else if(sorted_array[mid] < key)
		return binary_search(sorted_array, mid+1, end, key);
	else
		return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
	int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
	int i, search_index, key;

	printf("Here's the sorted array that we have\n");
	for(i = 0; i < 10; i++)
		printf("%d ", array[i]);

	printf("\nEnter a key to search in the array: ");
	scanf("%d", &key);

	search_index = binary_search(array, 0, 9, key);
	if(search_index < 0)
		printf("The key is not in this array\n");
	else
		printf("The key is found on index %d\n", search_index);
}
```

If the beginning index is greater than the ending index, it means we have a part of size 0

Example 2 – Binary Search (let’s code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
        int mid;
        if(beg > end)
                return -1;
        mid = beg + (end - beg)/2;
        if(sorted_array[mid] == key)
                return mid;
        else if(sorted_array[mid] < key)
                return binary_search(sorted_array, mid+1, end, key);
        else
                return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
        int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int i, search_index, key;

        printf("Here's the sorted array that we have\n");
        for(i = 0; i < 10; i++)
                printf("%d ", array[i]);

        printf("\nEnter a key to search in the array: ");
        scanf("%d", &key);

        search_index = binary_search(array, 0, 9, key);
        if(search_index < 0)
                printf("The key is not in this array\n");
        else
                printf("The key is found on index %d\n", search_index);
}
```

If the beginning index is greater than the ending index, it means we have a part of size 0

So, we return -1 irrespective of what the key is (we cannot find anything in a zero-sized array)

Example 2 – Binary Search (let's code it !)

```c
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
        int mid;
        if(beg > end)
                return -1;
        mid = beg + (end - beg)/2;
        if(sorted_array[mid] == key)
                return mid;
        else if(sorted_array[mid] < key)
                return binary_search(sorted_array, mid+1, end, key);
        else
                return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
        int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int i, search_index, key;

        printf("Here's the sorted array that we have\n");
        for(i = 0; i < 10; i++)
                printf("%d ", array[i]);

        printf("\nEnter a key to search in the array: ");
        scanf("%d", &key);

        search_index = binary_search(array, 0, 9, key);
        if(search_index < 0)
                printf("The key is not in this array\n");
        else
                printf("The key is found on index %d\n", search_index);
}
```

As discussed before, a good point to check the sorted array for the key, is the mid-point (or almost the mid-point) of the part of the array to search

SAURABH SRIVASTAVA | DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING | IIT KANPUR

Example 2 – Binary Search (let’s code it !)

```c
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
        int mid;
        if(beg > end)
                return -1;
        mid = beg + (end - beg)/2;
        if(sorted_array[mid] == key)
                return mid;
        else if(sorted_array[mid] < key)
                return binary_search(sorted_array, mid+1, end, key);
        else
                return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
        int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int i, search_index, key;

        printf("Here's the sorted array that we have\n");
        for(i = 0; i < 10; i++)
                printf("%d ", array[i]);

        printf("\nEnter a key to search in the array: ");
        scanf("%d", &key);

        search_index = binary_search(array, 0, 9, key);
        if(search_index < 0)
                printf("The key is not in this array\n");
        else
                printf("The key is found on index %d\n", search_index);
}
```

As discussed before, a good point to check the sorted array for the key, is the mid-point (or almost the mid-point) of the part of the array to search

So, we find the mid-point of the array part

Example 2 – Binary Search (let's code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
        int mid;
        if(beg > end)
                return -1;
        mid = beg + (end - beg)/2;
        if(sorted_array[mid] == key)
                return mid;
        else if(sorted_array[mid] < key)
                return binary_search(sorted_array, mid+1, end, key);
        else
                return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
        int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int i, search_index, key;

        printf("Here's the sorted array that we have\n");
        for(i = 0; i < 10; i++)
                printf("%d ", array[i]);

        printf("\nEnter a key to search in the array: ");
        scanf("%d", &key);

        search_index = binary_search(array, 0, 9, key);
        if(search_index < 0)
                printf("The key is not in this array\n");
        else
                printf("The key is found on index %d\n", search_index);
}
```

Then we check if the key is found at the mid-point

Example 2 – Binary Search (let’s code it !)

```c
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
	int mid;
	if(beg > end)
		return -1;
	mid = beg + (end - beg)/2;
	if(sorted_array[mid] == key)
		return mid;
	else if(sorted_array[mid] < key)
		return binary_search(sorted_array, mid+1, end, key);
	else
		return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
	int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
	int i, search_index, key;

	printf("Here's the sorted array that we have\n");
	for(i = 0; i < 10; i++)
		printf("%d ", array[i]);

	printf("\nEnter a key to search in the array: ");
	scanf("%d", &key);

	search_index = binary_search(array, 0, 9, key);
	if(search_index < 0)
		printf("The key is not in this array\n");
	else
		printf("The key is found on index %d\n", search_index);
}
```

Then we check if the key is found at the mid-point

If so, we return the mid-point of the array part as the output

Example 2 – Binary Search (let's code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
        int mid;
        if(beg > end)
                return -1;
        mid = beg + (end - beg)/2;
        if(sorted_array[mid] == key)
                return mid;
        else if(sorted_array[mid] < key)
                return binary_search(sorted_array, mid+1, end, key);
        else
                return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
        int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int i, search_index, key;

        printf("Here's the sorted array that we have\n");
        for(i = 0; i < 10; i++)
                printf("%d ", array[i]);

        printf("\nEnter a key to search in the array: ");
        scanf("%d", &key);

        search_index = binary_search(array, 0, 9, key);
        if(search_index < 0)
                printf("The key is not in this array\n");
        else
                printf("The key is found on index %d\n", search_index);
}
```

Otherwise, if the key greater than the element at the mid-point, we can be sure that "if at all" the key is present in this part of the array, it can only be on the right of the midpoint

Example 2 – Binary Search (let's code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
	int mid;
	if(beg > end)
		return -1;
	mid = beg + (end - beg)/2;
	if(sorted_array[mid] == key)
		return mid;
	else if(sorted_array[mid] < key)
		return binary_search(sorted_array, mid+1, end, key);
	else
		return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
	int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
	int i, search_index, key;

	printf("Here's the sorted array that we have\n");
	for(i = 0; i < 10; i++)
		printf("%d ", array[i]);

	printf("\nEnter a key to search in the array: ");
	scanf("%d", &key);

	search_index = binary_search(array, 0, 9, key);
	if(search_index < 0)
		printf("The key is not in this array\n");
	else
		printf("The key is found on index %d\n", search_index);
}
```

Otherwise, if the key greater than the element at the mid-point, we can be sure that "if at all" the key is present in this part of the array, it can only be on the right of the midpoint

So, we find the result of the binary search in the part of array which starts at `mid+1` and ends at `end`, and pass on the result as the result of binary search in the whole part

Example 2 – Binary Search (let's code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
	int mid;
	if(beg > end)
		return -1;
	mid = beg + (end - beg)/2;
	if(sorted_array[mid] == key)
		return mid;
	else if(sorted_array[mid] < key)
		return binary_search(sorted_array, mid+1, end, key);
	else
		return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
	int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
	int i, search_index, key;

	printf("Here's the sorted array that we have\n");
	for(i = 0; i < 10; i++)
		printf("%d ", array[i]);

	printf("\nEnter a key to search in the array: ");
	scanf("%d", &key);

	search_index = binary_search(array, 0, 9, key);
	if(search_index < 0)
		printf("The key is not in this array\n");
	else
		printf("The key is found on index %d\n", search_index);
}
```

Or else, i.e., the key is smaller than the element at the mid-point, the only possibility of finding it in the current part is to the left of the mid-point

Example 2 – Binary Search (let’s code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
	int mid;
	if(beg > end)
		return -1;
	mid = beg + (end - beg)/2;
	if(sorted_array[mid] == key)
		return mid;
	else if(sorted_array[mid] < key)
		return binary_search(sorted_array, mid+1, end, key);
	else
		return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
	int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
	int i, search_index, key;

	printf("Here's the sorted array that we have\n");
	for(i = 0; i < 10; i++)
		printf("%d ", array[i]);

	printf("\nEnter a key to search in the array: ");
	scanf("%d", &key);

	search_index = binary_search(array, 0, 9, key);
	if(search_index < 0)
		printf("The key is not in this array\n");
	else
		printf("The key is found on index %d\n", search_index);
}
```

Or else, i.e., the key is smaller than the element at the mid-point, the only possibility of finding it in the current part is to the left of the mid-point

So, we find the result of the binary search in the part of array which starts at `beg` and ends at `mid-1`, and pass on the result as the result of binary search in the whole part

Example 2 – Binary Search (let’s code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
        int mid;
        if(beg > end)
                return -1;
        mid = beg + (end - beg)/2;
        if(sorted_array[mid] == key)
                return mid;
        else if(sorted_array[mid] < key)
                return binary_search(sorted_array, mid+1, end, key);
        else
                return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
        int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int i, search_index, key;

        printf("Here's the sorted array that we have\n");
        for(i = 0; i < 10; i++)
                printf("%d ", array[i]);

        printf("\nEnter a key to search in the array: ");
        scanf("%d", &key);

        search_index = binary_search(array, 0, 9, key);
        if(search_index < 0)
                printf("The key is not in this array\n");
        else
                printf("The key is found on index %d\n", search_index);
}
```

To use the function, we call it with the `beg` value set to `0`, and the `end` value set to `capacity-1`

Example 2 – Binary Search (let’s code it !)

```
#include<stdio.h>

int binary_search(int *sorted_array, int beg, int end, int key)
{
	int mid;
	if(beg > end)
		return -1;
	mid = beg + (end - beg)/2;
	if(sorted_array[mid] == key)
		return mid;
	else if(sorted_array[mid] < key)
		return binary_search(sorted_array, mid+1, end, key);
	else
		return binary_search(sorted_array, beg, mid-1, key);
}

int main()
{
	int array[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
	int i, search_index, key;

	printf("Here's the sorted array that we have\n");
	for(i = 0; i < 10; i++)
		printf("%d ", array[i]);

	printf("\nEnter a key to search in the array: ");
	scanf("%d", &key);

	search_index = binary_search(array, 0, 9, key);
	if(search_index < 0)
		printf("The key is not in this array\n");
	else
		printf("The key is found on index %d\n", search_index);
}
```

Again, make sure you get the flow here, i.e., you understand why this works !!

How we applied *Divide and Conquer* here?

Factorial

- At each step, we reduce the problem size by 1

Binary Search

- At each step, we reduce the problem size by roughly half

How we applied *Divide and Conquer* here?

Factorial

- At each step, we reduce the problem size by 1
- At each step, the solution to the larger problem is either trivial (if the number is 0 or 1), or, given by combining the solution of a smaller problem with a trivial element (the number itself)

Binary Search

- At each step, we reduce the problem size by roughly half
- At each step, the solution to the larger problem is either trivial (if the element is found at the mid-point, or the part of array to search is of size 0), or, given by the solution to either of the smaller problems

How we applied *Divide and Conquer* here?

Factorial

- At each step, we reduce the problem size by 1
- At each step, the solution to the larger problem is either trivial (if the number is 0 or 1), or, given by combining the solution of a smaller problem with a trivial element (the number itself)
- We know we have the final solution, when the base case is found (i.e. the number is 1 or 0)

Binary Search

- At each step, we reduce the problem size by roughly half
- At each step, the solution to the larger problem is either trivial (if the element is found at the mid-point, or the part of array to search is of size 0), or, given by the solution to either of the smaller problems
- We know we have the solution, when either of the two base cases is found (element found or array exhausted)

How we applied *Divide and Conquer* here?

Factorial

- At each step, we reduce the problem size by 1
- At each step, the solution to the larger problem is either trivial (if the number is 0 or 1), or, given by combining the solution of a smaller problem with a trivial element (the number itself)
- We know we have the final solution, when the base case is found (i.e. the number is 1 or 0)
- It will take n steps to find the final solution, when the input size is n (i.e. number is n)

Binary Search

- At each step, we reduce the problem size by roughly half
- At each step, the solution to the larger problem is either trivial (if the element is found at the mid-point, or the part of array to search is of size 0), or, given by the solution to either of the smaller problems
- We know we have the solution, when either of the two base cases is found (element found or array exhausted)
- It will take, at max, $\lceil \log_2 n \rceil$ steps to find the final solution, when the input size is n (i.e. the array size)

Homework !!

While Divide and Conquer and Recursion are orthogonal concepts, they are often used together

- You can, theory, write Divide and Conquer algorithms iteratively as well
- Write a recursive as well as iterative Divide and Conquer algorithm to find the smallest element in an array

Read more about the overall Divide and Conquer approach

- You may start here: https://www.geeksforgeeks.org/divide-and-conquer/