Data Structures

Introduction Algorithmic Thinking and Recursion

August 11, 2016

Outline

Outline

Imagine the following

We have the following array of elements (Data Structure) $x_i \in \mathbb{R}$

1	2	3	4	5	6	7	8	9
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

- What if you are in one extreme or another?

- You need to scan from one extreme to the next... cost $O(n)$!!!

Imagine the following

We have the following array of elements (Data Structure) $x_i \in \mathbb{R}$

1	2	3	4	5	6	7	8	9
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

How do we find the peak k?

- Ideas?
- What if you are in one extreme or another?

- You need to start from one extreme to the other and walk

Imagine the following

We have the following array of elements (Data Structure) $x_i \in \mathbb{R}$

1	2	3	4	5	6	7	8	9
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

How do we find the peak k?

- Ideas?
- What if you are in one extreme or another?

Yes!!! How many steps do we need to perform?

You are in a possible worst case scenario

- You need to scan from one extreme to the next... cost $O(n)$!!!

Imagine the following

We have the following array of elements (Data Structure) $x_i \in \mathbb{R}$

1	2	3	4	5	6	7	8	9
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

How do we find the peak k?

- Ideas?
- What if you are in one extreme or another?

Yes!!! How many steps do we need to perform?

You are in a possible worst case scenario

- You need to scan from one extreme to the next... cost $O(n)$!!!

Imagine the following

We have the following array of elements (Data Structure) $x_i \in \mathbb{R}$

1	2	3	4	5	6	7	8	9
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

How do we find the peak k?

- Ideas?
- What if you are in one extreme or another?

Yes!!! How many steps do we need to perform?

You are in a possible worst case scenario

- You need to scan from one extreme to the next... cost $O(n)$!!!

What if....

Now, we change our problem

1	2	3	4	5	6	7	8	9
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

- Such that exits k, $x_1 \leq ... \leq x_i \leq x_k \geq x_j \geq ... \geq x_n$ for $i \leq k$ and $k \leq j$.

What if....

Now, we change our problem

1	2	3	4	5	6	7	8	9
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

- Such that exits k, $x_1 \leq ... \leq x_i \leq x_k \geq x_j \geq ... \geq x_n$ for $i \leq k$ and $k \leq j$.

We have then

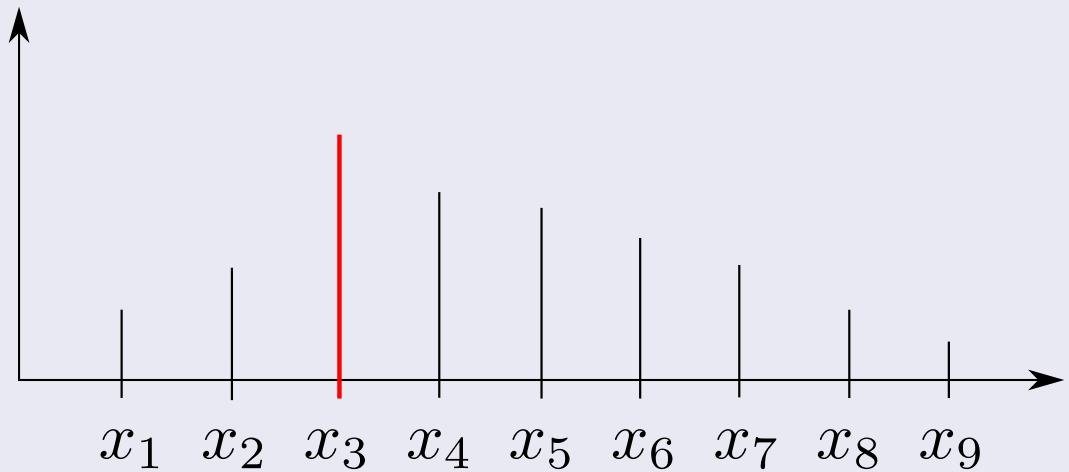
What if....

Now, we change our problem

1	2	3	4	5	6	7	8	9
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

- Such that exits k, $x_1 \leq ... \leq x_i \leq x_k \geq x_j \geq ... \geq x_n$ for $i \leq k$ and $k \leq j$.

We have then



Therefore

Thus, we can do the following - thinking algorithmically

We can position ourselves in the middle the sequence to find where we are...

Therefore

Thus, we can do the following - thinking algorithmically

We can position ourselves in the middle the sequence to find where we are...

What? if we have a new $index = \frac{n}{2}$

We can ask if:

- $x_{index-1} < x_{index}$ and $x_{index} < x_{index+1}$
- $x_{index-1} > x_{index}$ and $x_{index} > x_{index+1}$
- $x_{index-1} < x_{index}$ and $x_{index} > x_{index+1}$

Therefore

Thus, we can do the following - thinking algorithmically

We can position ourselves in the middle the sequence to find where we are...

What? if we have a new $index = \frac{n}{2}$

We can ask if:

- $x_{index-1} < x_{index}$ and $x_{index} < x_{index+1}$
- $x_{index-1} > x_{index}$ and $x_{index} > x_{index+1}$
- $x_{index-1} < x_{index}$ and $x_{index} > x_{index+1}$

Question?

Now what?

Therefore

Thus, we can do the following - thinking algorithmically

We can position ourselves in the middle the sequence to find where we are...

What? if we have a new $index = \frac{n}{2}$

We can ask if:

- $x_{index-1} < x_{index}$ and $x_{index} < x_{index+1}$
- $x_{index-1} > x_{index}$ and $x_{index} > x_{index+1}$
- $x_{index-1} < x_{index}$ and $x_{index} > x_{index+1}$

Question?

Now what?

Therefore

Thus, we can do the following - thinking algorithmically

We can position ourselves in the middle the sequence to find where we are...

What? if we have a new $index = \frac{n}{2}$

We can ask if:

- $x_{index-1} < x_{index}$ and $x_{index} < x_{index+1}$
- $x_{index-1} > x_{index}$ and $x_{index} > x_{index+1}$
- $x_{index-1} < x_{index}$ and $x_{index} > x_{index+1}$

Question?

Now what?

Therefore

Thus, we can do the following - thinking algorithmically

We can position ourselves in the middle the sequence to find where we are...

What? if we have a new $index = \frac{n}{2}$

We can ask if:

- $x_{index-1} < x_{index}$ and $x_{index} < x_{index+1}$
- $x_{index-1} > x_{index}$ and $x_{index} > x_{index+1}$
- $x_{index-1} < x_{index}$ and $x_{index} > x_{index+1}$

Question?

Now what?

Rules

$x_{index-1} < x_{index}$ and $x_{index} < x_{index+1}$

Go to the Right for the Peak!!!

$x_{index-1} > x_{index}$ and $x_{index} > x_{index+1}$

Go to the Left for the Peak!!!

You have the Peak!!!

$x_{index-1} < x_{index}$ and $x_{index} > x_{index+1}$

Rules

$x_{index-1} < x_{index}$ and $x_{index} < x_{index+1}$

Go to the Right for the Peak!!!

$x_{index-1} > x_{index}$ and $x_{index} > x_{index+1}$

Go to the Left for the Peak!!!

You have the Peak!!!

$x_{index-1} < x_{index}$ and $x_{index} > x_{index+1}$

Rules

$x_{index-1} < x_{index}$ and $x_{index} < x_{index+1}$

Go to the Right for the Peak!!!

$x_{index-1} > x_{index}$ and $x_{index} > x_{index+1}$

Go to the Left for the Peak!!!

You have the Peak!!!

$x_{index-1} < x_{index}$ and $x_{index} > x_{index+1}$

What is the complexity of this simple Strategy?

We have the following

$$T(n) = \underbrace{1+1+1+1+1+...+1}_{\text{Some Number of Times}}$$

Then what?

Imagine that you divide your input by 2 every time, then you stop when

$$\frac{n}{2^h} = 1$$

Therefore

$$T(n) = c\log_2 n = O(\log_2 n)$$

What is the complexity of this simple Strategy?

We have the following

$$T(n) = \underbrace{1+1+1+1+1+...+1}_{\text{Some Number of Times}}$$

Then what?

Imagine that you divide your input by 2 every time, then you stop when

$$\frac{n}{2^h} = 1$$

Therefore

$$T(n) = c\log_2 n = O(\log_2 n)$$

What is the complexity of this simple Strategy?

We have the following

$$T(n) = \underbrace{1+1+1+1+1+...+1}_{\text{Some Number of Times}}$$

Then what?

Imagine that you divide your input by 2 every time, then you stop when

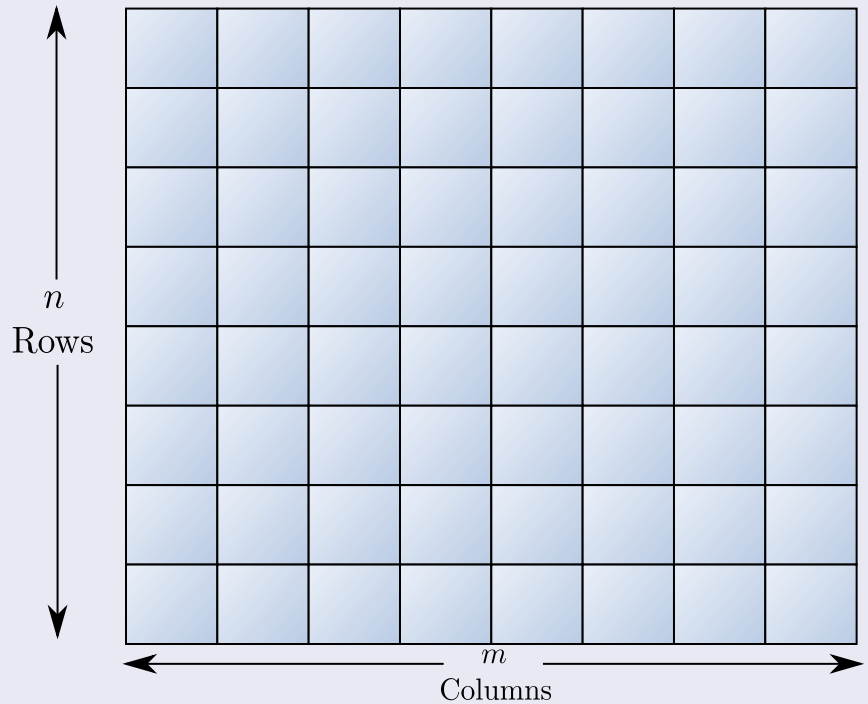
$$\frac{n}{2^h} = 1$$

Therefore

$$T(n) = c\log_2 n = O(\log_2 n)$$

What about a more complex case

2D Case



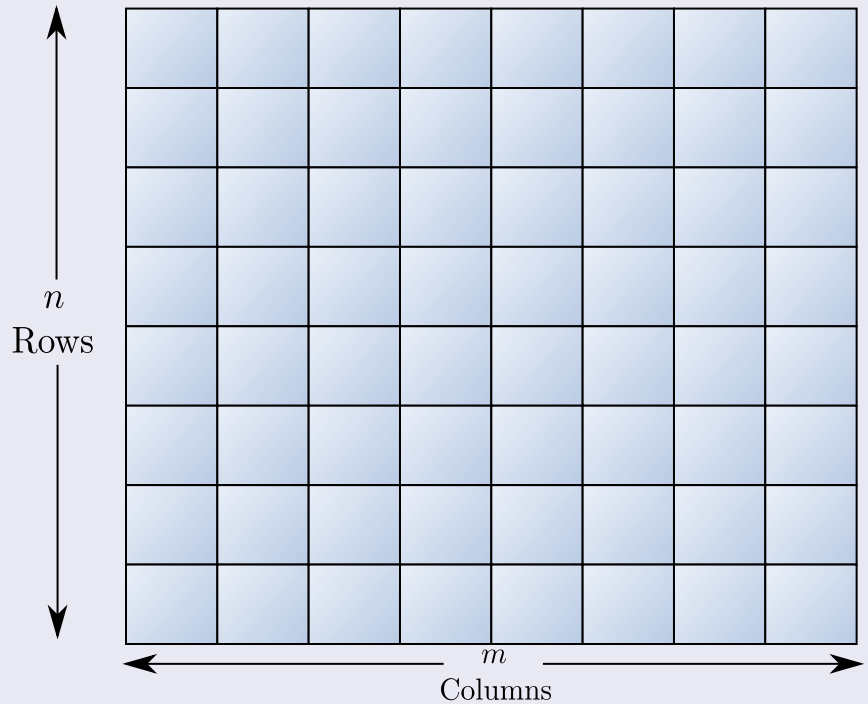
Ideas of what to do?

A simple Greedy method

- Look for the larges one will take $O(nm) = O(n^2)$ if $n = m$

What about a more complex case

2D Case



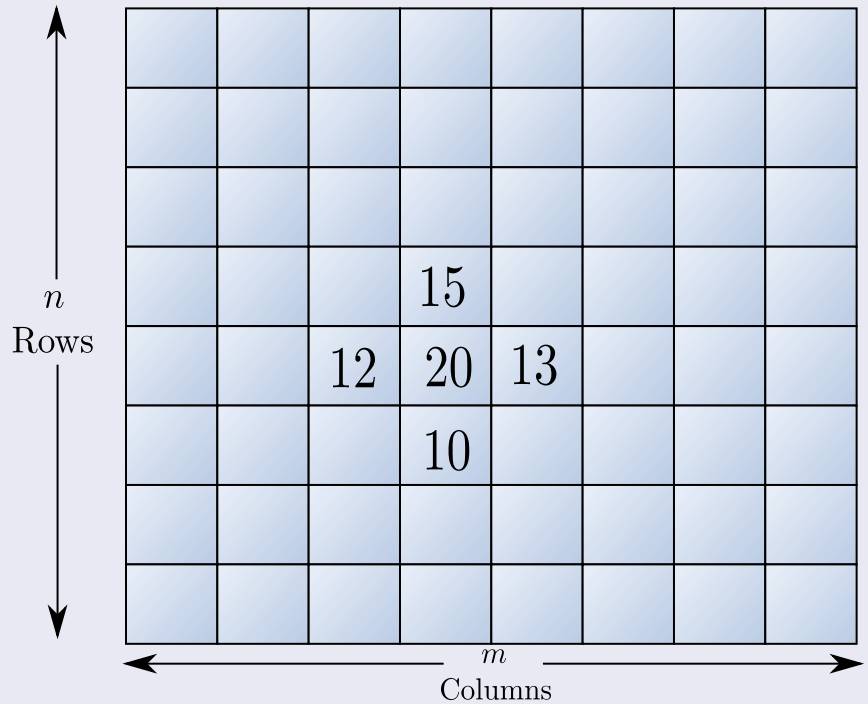
Ideas of what to do?

A simple Greedy method

- Look for the larges one will take $O(nm) = O(n^2)$ if $n = m$

By the way How do we define a 2D-peak?

Look at this

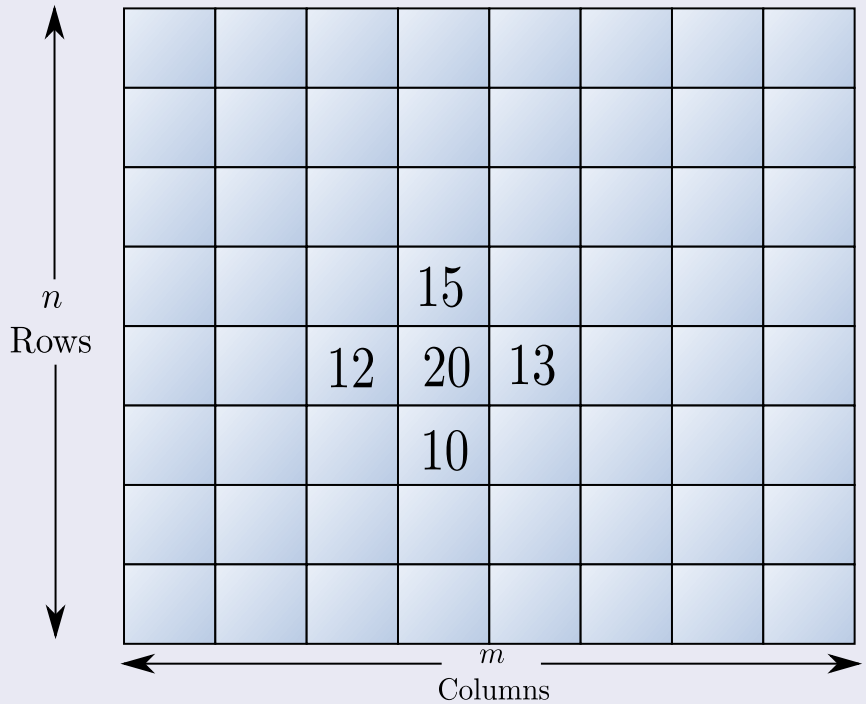


How do we define the peak

Ideas?

By the way How do we define a 2D-peak?

Look at this



How do we define the peak

Ideas?

Restrict Our Problem

We decide to have something like this

- Each column has a increasing-decreasing sequence!!!

Thus

We have that $x_{1,j} \leq x_{2,j} \leq ... \leq x_{k-1,j} \leq x_{k,j} \geq x_{k+1,j} \geq ... \geq x_{n,j}$

Restrict Our Problem

We decide to have something like this

- Each column has a increasing-decreasing sequence!!!

Thus

We have that $x_{1,j} \leq x_{2,j} \leq ... \leq x_{k-1,j} \leq x_{k,j} \geq x_{k+1,j} \geq ... \geq x_{n,j}$

Therefore

You can try something like

- Pick Middle Column $j = \frac{m}{2}$.
- Find a 1D-peak at column j.
- Using (i, j) as a start point to find a row maximum in row i.

Therefore

You can try something like

- Pick Middle Column $j = \frac{m}{2}$.
- Find a 1D-peak at column j.
- Using (i, j) as a start point to find a row maximum in row i.

Example - Here the algorithms fails

10	1	20	10
14	13	19	13
16	15	17	15
2	11	10	14
1	9	2	8

Therefore

You can try something like

- Pick Middle Column $j = \frac{m}{2}$.
- Find a 1D-peak at column j.
- Using (i, j) as a start point to find a row maximum in row i.

Example - Here the algorithms fails

10	1	20	10
14	13	19	13
16	15	17	15
2	11	10	14
1	9	2	8

Therefore

You can try something like

- Pick Middle Column $j = \frac{m}{2}$.
- Find a 1D-peak at column j.
- Using (i, j) as a start point to find a row maximum in row i.

Example - Here the algorithms fails

10	**1**	20	10
14	**13**	19	13
16	**15**	**17**	**15**
2	**11**	10	14
1	**9**	2	8

Another Attempt

Second Attempt

Ideas?

What Complexity?

Ideas?

Another Attempt

Second Attempt

Ideas?

What Complexity?

Ideas?

What about this?

We use the new constraint

- **Each column has a increasing-decreasing sequence.**

I propose the following

Find the peak of each column then?

Then

What do we need to find the peak between all the column-peaks?

What about this?

We use the new constraint

- **Each column has a increasing-decreasing sequence.**

I propose the following

Find the peak of each column then?

Then

What do we need to find the peak between all the column-peaks?

What about this?

We use the new constraint

- **Each column has a increasing-decreasing sequence.**

I propose the following

Find the peak of each column then?

Then

What do we need to find the peak between all the column-peaks?

What Complexity

Basically

of steps for finding column peaks + # steps for finding the global peak

Then, we have

$$c_1 m \log n + c_2 m$$

In Big O notation

What do we have?

What Complexity

Basically

of steps for finding column peaks + # steps for finding the global peak

Then, we have

$$c_1 m \log n + c_2 m$$

In Big O notation

What do we have?

What Complexity

Basically

\# of steps for finding column peaks + \# steps for finding the global peak

Then, we have

$$c_1 m \log n + c_2 m$$

In Big O notation

What do we have?

Outline

Therefore... Algorithms are

We have that

Algorithms are procedures operating in data structures!!!

Therefore... given a data structure

We need to think about the algorithms operating on it!!!

Therefore... Algorithms are

We have that

Algorithms are procedures operating in data structures!!!

Therefore... given a data structure

We need to think about the algorithms operating on it!!!

Outline

What is recursion?

Fact

Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first.

What is recursion?

Fact

Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first.

Example

- You can build a house by hiring a contractor.
- The contractor in turn hires several subcontractors to complete portions of the house.
- Each subcontractor might hire other subcontractors to help.

What is recursion?

Fact

Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first.

Example

- You can build a house by hiring a contractor.
- The contractor in turn hires several subcontractors to complete portions of the house.
- Each subcontractor might hire other subcontractors to help.

What is recursion?

Fact

Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first.

Example

- You can build a house by hiring a contractor.
- The contractor in turn hires several subcontractors to complete portions of the house.
- Each subcontractor might hire other subcontractors to help.

Another Example

Example: Principal Idea You do your part until your friend finishes and tells you

Outline

A formal definition

Definition (Successor Function)

We define the successor operator:

- the function $S(x)$ that takes a number x to its successor.
- This gives one the nonnegative integers $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$.

A formal definition

Definition (Successor Function)

We define the successor operator:

- the function $S(x)$ that takes a number x to its successor.
- This gives one the nonnegative integers $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$.

Definition (Addition)

We define the add function as:

$$add(x, z) = \begin{cases} x & \text{if } z = 0 \\ S(add(x, y)) & \text{if } z = S(y) \end{cases}$$

A formal definition

Definition (Successor Function)

We define the successor operator:

- the function $S(x)$ that takes a number x to its successor.
- This gives one the nonnegative integers $\mathbb{N}_0 = \{0, 1, 2, ...\}$.

Definition (Addition)

We define the add function as:

$$add(x, z) = \begin{cases} x & \text{if } z = 0 \\ S(add(x, y)) & \text{if } z = S(y) \end{cases}$$

A formal definition

Definition (Successor Function)

We define the successor operator:

- the function $S(x)$ that takes a number x to its successor.
- This gives one the nonnegative integers $\mathbb{N}_0 = \{0, 1, 2, ...\}$.

Definition (Addition)

We define the add function as:

$$add(x, z) = \begin{cases} x & \text{if } z = 0 \\ S(add(x, y)) & \text{if } z = S(y) \end{cases}$$

A formal definition

Definition (Successor Function)

We define the successor operator:

- the function $S(x)$ that takes a number x to its successor.
- This gives one the nonnegative integers $\mathbb{N}_0 = \{0, 1, 2, ...\}$.

Definition (Addition)

We define the add function as:

$$add(x, z) = \begin{cases} x & \text{if } z = 0 \\ S(add(x, y)) & \text{if } z = S(y) \end{cases}$$

Example

We can prove that $2 + 2 = 4$

$$\begin{aligned} add\,(2,2) &= S\,(add\,(2,1)) \\ &= S\,(S\,(add\,(2,0))) \\ &= S\,(S\,(2)) \\ &= S\,(3) \\ &= 4 \end{aligned}$$

Example

We can prove that $2 + 2 = 4$

$$\begin{aligned} add\,(2,2) &= S\,(add\,(2,1)) \\ &= S\,(S\,(add\,(2,0))) \\ &= S\,(S\,(2)) \\ &= S\,(3) \\ &= 4 \end{aligned}$$

Example

We can prove that $2 + 2 = 4$

$$\begin{aligned} add\,(2,2) &= S\,(add\,(2,1)) \\ &= S\,(S\,(add\,(2,0))) \\ &= S\,(S\,(2)) \\ &= S\,(3) \\ &= 4 \end{aligned}$$

Example

We can prove that $2 + 2 = 4$

$$\begin{aligned} add\,(2,2) &= S\,(add\,(2,1)) \\ &= S\,(S\,(add\,(2,0))) \\ &= S\,(S\,(2)) \\ &= S\,(3) \\ &= 4 \end{aligned}$$

Example

We can prove that $2 + 2 = 4$

$$\begin{aligned} add\,(2,2) &= S\,(add\,(2,1)) \\ &= S\,(S\,(add\,(2,0))) \\ &= S\,(S\,(2)) \\ &= S\,(3) \\ &= 4 \end{aligned}$$

Now

Something Notable

Primitive recursive functions are built up from three basic functions using two operations.

Now

Something Notable

Primitive recursive functions are built up from three basic functions using two operations.

The Basic Functions are

1. **Zero**. $Z \equiv 0$.
2. **Successor**. $S(x) = x + 1$
3. **Projection**. A projection function select one of the arguments. Specifically:

$$P_1(x, y) \equiv x \text{ and } P_2(x, y) \equiv y \tag{1}$$

Now

Something Notable

Primitive recursive functions are built up from three basic functions using two operations.

The Basic Functions are

1. **Zero**. $Z \equiv 0$.
2. **Successor**. $S(x) = x + 1$
3. **Projection**. A projection function select one of the arguments. Specifically:

$$P_1(x, y) \equiv x \text{ and } P_2(x, y) \equiv y \quad (1)$$

Now

Something Notable

Primitive recursive functions are built up from three basic functions using two operations.

The Basic Functions are

1. **Zero**. $Z \equiv 0$.
2. **Successor**. $S(x) = x + 1$
3. **Projection**. A projection function select one of the arguments. Specifically:

$$P_1(x, y) \equiv x \text{ and } P_2(x, y) \equiv y \tag{1}$$

Composition Operations

Something Notable

There are two operations that make new functions from old:

- Composition and Primitive Recursion.

Definition (Composition)

Composition replaces the arguments of a function by another. For example,

$$f(x, y) = g(h_1(x, y), h_2(x, y))$$

where one supplies the functions g_1, g_2 and h.

Composition Operations

Something Notable

There are two operations that make new functions from old:

- Composition and Primitive Recursion.

Definition (Composition)

Composition replaces the arguments of a function by another. For example,

$$f(x, y) = g(h_1(x, y), h_2(x, y))$$

where one supplies the functions g_1, g_2 and h.

Finally

Definition (Primitive Recursive Function - Turing Computable)

A function is primitive recursive if it can be built up using the base functions and the operations of composition and primitive recursion.

Classic Structure

$$f(x,0) = g_1(x)$$
$$f(x,S(y)) = h(g_2(x,y), f(x,y))$$

Finally

Definition (Primitive Recursive Function - Turing Computable)

A function is primitive recursive if it can be built up using the base functions and the operations of composition and primitive recursion.

Classic Structure

$$f(x,0) = g_1(x)$$
$$f(x,S(y)) = h(g_2(x,y), f(x,y))$$

Example

Multiplication

$$mul\left(x,0\right)=0$$
$$mul\left(x,S\left(y\right)\right)=add\left(x,mul\left(x,y\right)\right)$$

Outline

Recursive Method

Definition

- A method that calls itself is a **recursive method**.
- The invocation is a **recursive call** or **recursive invocation**.

The previous example can be seen in code as follow

Recursive Method

Definition

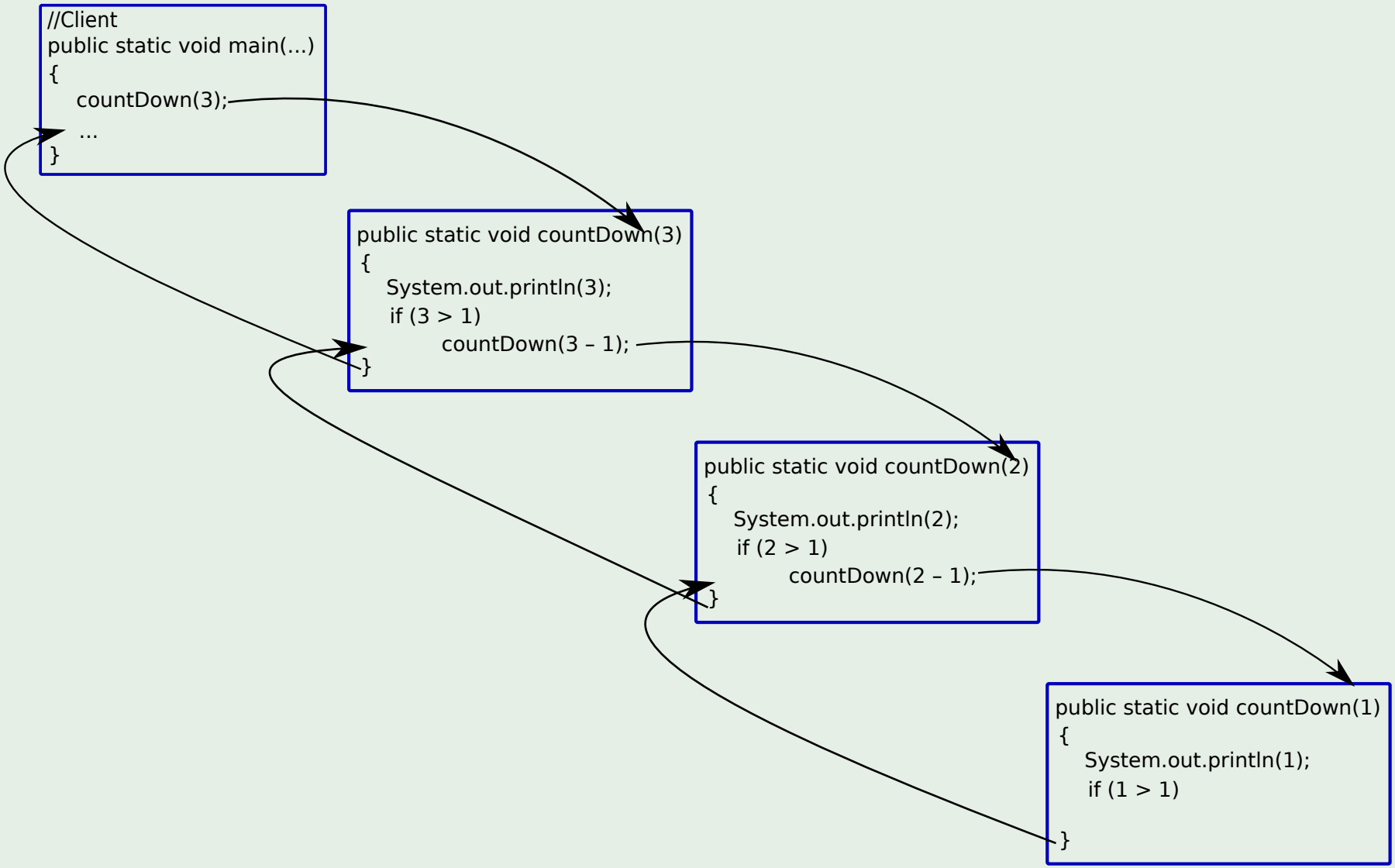
- A method that calls itself is a **recursive method**.
- The invocation is a **recursive call** or **recursive invocation**.

The previous example can be seen in code as follow

```
/** Counts down from a given positive integer.
@param integer an integer > 0 */
public static void countDown(int integer)
{
    System.out.println(integer);
    if (integer > 1)
        countDown(integer - 1);
} // end countDown
```

Example with 3

Tracing the recursive call countDown(3)



Outline

Questions to answer when designing a recursive solution

First

What part of the solution can you contribute directly?

Second

What smaller but identical problem has a solution that, when taken with your contribution, provides the solution to the original problem?

Third

When does the process end? That is, what smaller but identical problem has a known solution, and have you reached this problem, or base case?

Questions to answer when designing a recursive solution

First

What part of the solution can you contribute directly?

Second

What smaller but identical problem has a solution that, when taken with your contribution, provides the solution to the original problem?

Third

When does the process end? That is, what smaller but identical problem has a known solution, and have you reached this problem, or base case?

Questions to answer when designing a recursive solution

First

What part of the solution can you contribute directly?

Second

What smaller but identical problem has a solution that, when taken with your contribution, provides the solution to the original problem?

Third

When does the process end? That is, what smaller but identical problem has a known solution, and have you reached this problem, or base case?

For countDown

Answer 1

The method countDown displays the given integer first.

Answer 2

The smaller problem is counting down from **integer - 1**.

Answer 3

The **if** statement asks if the process has reached the base case. In our case **integer = 1**.

For countDown

Answer 1

The method countDown displays the given integer first.

Answer 2

The smaller problem is counting down from **integer - 1**.

Answer 3

The **if** statement asks if the process has reached the base case. In our case **integer = 1**.

For countDown

Answer 1

The method countDown displays the given integer first.

Answer 2

The smaller problem is counting down from **integer - 1**.

Answer 3

The **if** statement asks if the process has reached the base case. In our case **integer = 1**.

Remember Discrete Mathematics

During the definition of structural induction

A version of mathematical induction that is used to prove properties of recursively defined sets.

Remember Discrete Mathematics

During the definition of structural induction

A version of mathematical induction that is used to prove properties of recursively defined sets.

For example Boolean expressions

1. BASE: Each symbol of the alphabet is a Boolean expression.
2. RECURSION: If P and Q are Boolean expressions, then so are
   1. $P \wedge Q$
   2. $P \vee Q$
   3. $\neg P$
3. RESTRICTION: There are no Boolean expressions over the alphabet other than those obtained from 1 and 2.

Remember Discrete Mathematics

During the definition of structural induction

A version of mathematical induction that is used to prove properties of recursively defined sets.

For example Boolean expressions

1. BASE: Each symbol of the alphabet is a Boolean expression.
2. RECURSION: If P and Q are Boolean expressions, then so are
   1. $P \wedge Q$
   2. $P \vee Q$
   3. $\neg P$
3. RESTRICTION: There are no Boolean expressions over the alphabet other than those obtained from 1 and 2.

Remember Discrete Mathematics

During the definition of structural induction

A version of mathematical induction that is used to prove properties of recursively defined sets.

For example Boolean expressions

1. BASE: Each symbol of the alphabet is a Boolean expression.
2. RECURSION: If P and Q are Boolean expressions, then so are
   1. $P \wedge Q$
   2. $P \vee Q$
   3. $\neg P$
3. RESTRICTION: There are no Boolean expressions over the alphabet other than those obtained from 1 and 2.

Remember Discrete Mathematics

During the definition of structural induction

A version of mathematical induction that is used to prove properties of recursively defined sets.

For example Boolean expressions

1. BASE: Each symbol of the alphabet is a Boolean expression.
2. RECURSION: If P and Q are Boolean expressions, then so are
   1. $P \wedge Q$
   2. $P \vee Q$
   3. $\neg P$
3. RESTRICTION: There are no Boolean expressions over the alphabet other than those obtained from 1 and 2.

Remember Discrete Mathematics

During the definition of structural induction

A version of mathematical induction that is used to prove properties of recursively defined sets.

For example Boolean expressions

1. BASE: Each symbol of the alphabet is a Boolean expression.
2. RECURSION: If P and Q are Boolean expressions, then so are
   1. $P \wedge Q$
   2. $P \vee Q$
   3. $\neg P$
3. RESTRICTION: There are no Boolean expressions over the alphabet other than those obtained from 1 and 2.

Remember Discrete Mathematics

During the definition of structural induction

A version of mathematical induction that is used to prove properties of recursively defined sets.

For example Boolean expressions

1. BASE: Each symbol of the alphabet is a Boolean expression.
2. RECURSION: If P and Q are Boolean expressions, then so are
   1. $P \wedge Q$
   2. $P \vee Q$
   3. $\neg P$
3. RESTRICTION: There are no Boolean expressions over the alphabet other than those obtained from 1 and 2.

Now, Recursive Functions

Recursive Functions

A function is said to be defined recursively or to be a recursive function if its rule of definition refers to itself.

We can use this functions to define the number of steps taken by recursive algorithms

For example:

$$T(n) = \begin{cases} 1 & \text{if } n \text{ is } 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 & \text{if } n > 1 \end{cases} \tag{2}$$

It describes

The total number of steps of the binary search in an ordered set of number using recursion.

Now, Recursive Functions

Recursive Functions

A function is said to be defined recursively or to be a recursive function if its rule of definition refers to itself.

We can use this functions to define the number of steps taken by recursive algorithms

For example:

$$T(n) = \begin{cases} 1 & \text{if } n \text{ is } 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + 1 & \text{if } n > 1 \end{cases} \tag{2}$$

It describes

The total number of steps of the binary search in an ordered set of number using recursion.

Now, Recursive Functions

Recursive Functions

A function is said to be defined recursively or to be a recursive function if its rule of definition refers to itself.

We can use this functions to define the number of steps taken by recursive algorithms

For example:

$$T(n) = \begin{cases} 1 & \text{if } n \text{ is } 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + 1 & \text{if } n > 1 \end{cases} \tag{2}$$

It describes

The total number of steps of the binary search in an ordered set of number using recursion.

Thus

We can use our theory of solving recursive functions

In order to calculate the number of steps in many algorithms and data structures, if not all

Because

Since you can build a Turing complete language (Computable) using strictly iterative structures and a Turning complete language (Computable) using only recursive structures, then the two are therefore equivalent.

Thus

We can use our theory of solving recursive functions

In order to calculate the number of steps in many algorithms and data structures, if not all

Because

Since you can build a Turing complete language (Computable) using strictly iterative structures and a Turning complete language (Computable) using only recursive structures, then the two are therefore equivalent.

Outline

Design guidelines for successful recursion

First

The method must be given an input value.

Second

The method definition must contain logic that involves this input value and leads to different cases.

Third

One or more of these cases should provide a solution that does not require recursion. These are the base cases, or stopping cases.

Fourth

One or more cases must include a recursive invocation of the method using a "smaller" argument.

Design guidelines for successful recursion

First

The method must be given an input value.

Second

The method definition must contain logic that involves this input value and leads to different cases.

Third

One or more of these cases should provide a solution that does not require recursion. These are the base cases, or stopping cases.

Fourth

One or more cases must include a recursive invocation of the method using a "smaller" argument.

Design guidelines for successful recursion

First

The method must be given an input value.

Second

The method definition must contain logic that involves this input value and leads to different cases.

Third

One or more of these cases should provide a solution that does not require recursion. These are the base cases, or stopping cases.

Fourth

One or more cases must include a recursive invocation of the method using a "smaller" argument.

Design guidelines for successful recursion

First

The method must be given an input value.

Second

The method definition must contain logic that involves this input value and leads to different cases.

Third

One or more of these cases should provide a solution that does not require recursion. These are the base cases, or stopping cases.

Fourth

One or more cases must include a recursive invocation of the method using a "smaller" argument.

So, be careful

Infinite recursion

A recursive method that does not check for a base case, or that misses the base case, will execute "forever." This situation is known as infinite recursion.

Which is known as

This situation is known as infinite recursion.

So, be careful

Infinite recursion

A recursive method that does not check for a base case, or that misses the base case, will execute "forever." This situation is known as infinite recursion.

Which is known as

This situation is known as infinite recursion.

Outline

Example of code with infinite recursion

Code

```
/** Counts down from a given positive integer.
@param integer an integer > 0 */
public static void countDown(int integer)
{
    System.out.println(integer);
    countDown(integer - 1);
} // end countDown
```

Recursive Methods That Return a Value

The nice stuff about recursion

You can return values

For example

We can compute the sum $1 + 2 + 3 + \ldots + n$

Recursive Methods That Return a Value

The nice stuff about recursion

You can return values

For example

We can compute the sum $1 + 2 + 3 + ... + n$

How the code will look

Partial Code

```
/** @param n an integer > 0
    @return the sum 1 + 2 + ... + n */
public static int sumOf(int n)
{
int sum;

// What do we put here?

return sum;
} // end sumOf
```

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

How To Debugging a Recursive Method

GENERAL RULES

1. Does the method have at least one input value?
2. Does the method contain a statement that tests an input value and leads to different cases?
3. Did you consider all possible cases? Does at least one of these cases cause at least one recursive call?
4. Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?
5. If these recursive calls produce or return correct results, will the method produce or return a correct result?
6. Is at least one of the cases a base case that has no recursive call?
7. Are there enough base cases?
8. Does each base case produce a result that is correct for that case?
9. If the method returns a value, does each of the cases return a value?

Outline

Now, what if we solve a recursive function

What we want to solve

n choose k (combinations)

We have

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \text{ with } 1 < k < n \tag{3}$$

Base Case

$$\binom{n}{1} = n(k=1), \binom{n}{n} = 1(k=n) \tag{4}$$

Now, what if we solve a recursive function

What we want to solve

n choose k (combinations)

We have

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \text{ with } 1 < k < n \tag{3}$$

Base Case

$$\binom{n}{1} = n\,(k = 1), \binom{n}{n} = 1(k = n) \tag{4}$$

Now, what if we solve a recursive function

What we want to solve

n choose k (combinations)

We have

$$\begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} n-1 \\ k \end{pmatrix} + \begin{pmatrix} n-1 \\ k-1 \end{pmatrix}, \text{ with } 1 < k < n \tag{3}$$

Base Case

$$\begin{pmatrix} n \\ 1 \end{pmatrix} = n\,(k=1)\,, \begin{pmatrix} n \\ n \end{pmatrix} = 1(k=n) \tag{4}$$

Base Code so you can start

Base Code

```
class Combinations{
        public static int Combinations(int n, int k)
        {

        }

        public static void main(String[] args)
        {

        }
}
```