Intro to C

Summary

```
int i; // variable declaration
int i = 0; // declaration and initialization
int a[5]; // 1-d array declaration
int b[5][5]; // 2-d array declaration
int c[5][5][5]; // 3-d array declaration

a[1] = 20;
b[1][2] = a[1];
c[1][2][3] = b[1][2];
```

Library functions: printf, scanf (stdio.h)

```
for (i = 0; i < 20; i++) {
  // body
}
while (i < 10) {
  // body
}
if (i < 10) {
  // if-body
}
int foo(int arg1[], char arg2, float arg3);
// function declaration

int foo(int arg1[], char arg2, float arg3) {
// function body
}
```

Example

What is this program doing?

```
#include <stdio.h>
int sum(int arr[], int n) {
   int i, s = 0;
   for (i = 0; i < n; i++) {
      s = s + arr[i];
   }
   return s;
}
int main() {
   int arr[5];
   int i, s;
   printf("Enter five numbers\n");
   for (i = 0; i < 5; i++) {
      scanf("%d", &arr[i]);
   }
   s = sum(arr, 5);
   printf("s=%d\n", s);
   return 0;
}
```

size for 1st dimension is optional

C doesn't track the size of the array. You can read/write an element that is outside the bounds of the array. The behavior of the program is undefined in this scenario.

Example

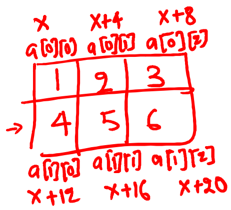
- What is this program doing?
  - main takes five integer inputs from the user in an array using a loop
  - Pass the array and the number of elements to routine sum
  - sum adds all the elements in the array using a loop and variable s
  - sum returns the final summation to the main
  - main prints the return value of sum

C doesn’t track the size of the array. You can read/write an element that is outside the bounds of the array. The behavior of the program is undefined in this scenario.

```
#include <stdio.h>
int sum(int arr[], int n) {
   int i, s = 0;
   for (i = 0; i < n; i++) {
      s = s + arr[i];
   }
   return s;
}
int main() {
   int arr[5];
   int i, s;
   printf(“Enter five numbers\n”);
   for (i = 0; i < 5; i++) {
      scanf(“%d”, &arr[i]);
   }
   s = sum(arr, 5);
   printf(“s=%d\n”, s);
   return 0;
}
```

size for 1st dimension is optional

Example



1 2 3
4 5 6

```
#include <stdio.h>

void print_2d_array(int a[2][3]) {
  printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
  printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
}

int main() {
  int a[2][3];
  a[0][0] = 1; a[0][1] = 2; a[0][2] = 3;
  a[1][0] = 4; a[1][1] = 5; a[1][2] = 6;
  print_2d_array(a);
  return 0;
}
```

What will be the output of this program?

The address of an element “a[i][j]” in a 2D array of integers “a” is calculated as X + (i * Number of columns * 4) + (4 * j), where X is the starting address of the array.

Example

error
no columns

Can you point out the issue
with this code?

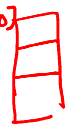
```
#include <stdio.h>

void print_2d_array(int a[][]) {
  printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
  printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
}

int main() {
  int a[2][3];
  a[0][0] = 1; a[0][1] = 2; a[0][2] = 3;
  a[1][0] = 4; a[1][1] = 5; a[1][2] = 6;
  print_2d_array(a);
  return 0;
}
```

In this case, in the print_2d_array function, we can't compute the address a[i][j] because the number of columns is missing in the prototype.

Example

1 2 3
2 3 4

a[0][0]



a[1][0]

Can you point out the issue with this code? What will be the output?

```
#include <stdio.h>

void print_2d_array(int a[][1]) {
  printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
  printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
}

int main() {
  int a[2][3];
  a[0][0] = 1; a[0][1] = 2; a[0][2] = 3;
  a[1][0] = 4; a[1][1] = 5; a[1][2] = 6;
  print_2d_array(a);
  return 0;
}
```

In this case, because of the wrong number of columns in the declaration of print_2d_array, the address of a[1][0] will be computed as X + 4. Therefore, the second line will print 2 3 4.

Example

1 2 3

4 5 6

```
#include <stdio.h>

void print_2d_array(int a[][3]) {
  printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
  printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
}

int main() {
  int a[2][3];
  a[0][0] = 1; a[0][1] = 2; a[0][2] = 3;
  a[1][0] = 4; a[1][1] = 5; a[1][2] = 6;
  print_2d_array(a);
  return 0;
}
```

What will be the output in this case?

The number of rows is not really needed for computing the address of a[i][j]. Therefore, this program is legal. The output is as expected in this case.

C languages

- You will learn more about C in the labs and tutorials
- In the class, we will discuss some more topics when they are needed

Algorithms and Data Structures

Algorithm

- Algorithm is a well-defined finite sequence of unambiguous operations that work on a given input to derive the desired output
  - i.e., an algorithm must terminate

Algorithm

- An algorithm is correct, if for all possible inputs
  - It halts
  - Finishes its computing in finite time
  - Outputs the correct solution
- All algorithms that we will discuss in this course are correct
- Some incorrect algorithms may also be helpful if you can control the error rate
  - E.g., a faster algorithm that may sometime not terminate can be used instead of a slower algorithm that always terminates

Why study algorithms and data structures

- Algorithm and data structures are important because we want to run our application faster using reasonable resources
  - An algorithm that takes an hour to search a webpage is not very useful
  - An application that takes 10 GB RAM might not work on many machines

- Widely used platforms like Google, Facebook, Amazon, IRCTC, etc., use very efficient data structures to give timely responses to our queries despite a huge volume of requests and data

Why study algorithms and data structures

- Algorithm design is not trivial, and a single algorithm will not work in all the scenarios
- In this course, we will discuss some algorithms that may work in some common scenarios; however, finding the most efficient algorithm for a given problem is very tricky
- We will also discuss the strategy to estimate the resources and time taken by the application
- Developing the skill to design good algorithms would require a lot of practice and knowledge of existing algorithms

What kind of problems can algorithms solve?

- The real-life examples are:
  - Search engines: answers your query from the millions of pages instantly
  - E-commerce: enable online purchases in a safe and secure manner
  - Social media: your updates are immediately visible to people in the same order they are made
  - Maps: instantly gives you the shortest path given the current traffic
  - Tools for compressing large files
  - IRCTC: can handle millions of requests simultaneously
  - Many applications in medical science, e.g., Genome Sequencing
  - etc.

What is data structure?

- Data structure is a way of storing and organizing data, e.g.,
  - data can be stored in consecutive addresses, or non-consecutive addresses
  - data can be stored in linear sequence or non-linear sequence

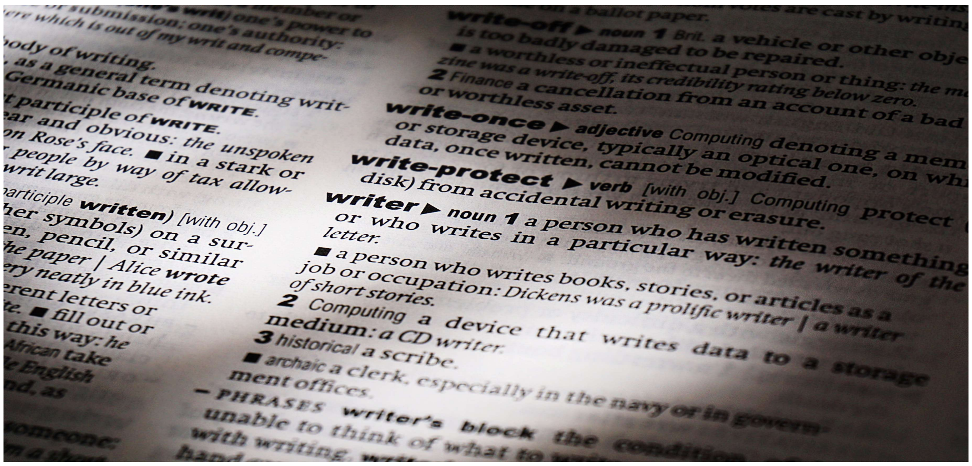
- Array, list, stack, queue, tree, graph, etc. are a few examples of data structure

Real world examples

Ticket counter

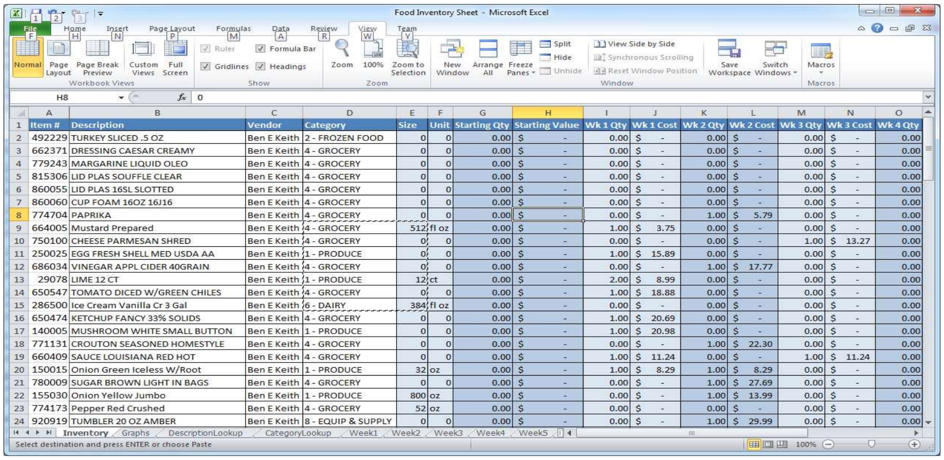
To implement a ticket booking application, we need a data-structure that is efficient for implementing first-in-first-out behavior.

Dictionary



For the dictionary, we need to store data in sorted order.

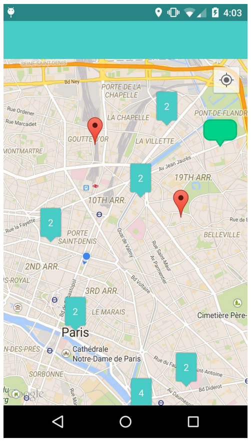
Excel sheet



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Item #	Description	Vendor	Category	Size	Unit	Starting Qty	Starting Value	Wk 1 Qty	Wk 1 Cost	Wk 2 Qty	Wk 2 Cost	Wk 3 Qty	Wk 3 Cost	Wk 4 Qty
2	492229	TURKEY SLICED .5 OZ	Ben E Keith	2 - FROZEN FOOD	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
3	662371	DRESSING CAESAR CREAMY	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
4	779243	MARGARINE LIQUID OLEO	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
5	815306	LID PLAS SOUFFLE CLEAR	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
6	860055	LID PLAS 16SL SLOTTED	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
7	860060	CUP FOAM 16OZ 16J16	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
8	774704	PAPRIKA	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	1.00	$ 5.79	0.00	$ -	0.00
9	664005	Mustard Prepared	Ben E Keith	4 - GROCERY	512	fl oz	0.00	$ -	1.00	$ 3.75	0.00	$ -	0.00	$ -	0.00
10	750100	CHEESE PARMESAN SHRED	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	1.00	$ 13.27	0.00
11	250025	EGG FRESH SHELL MED USDA AA	Ben E Keith	1 - PRODUCE	0	0	0.00	$ -	1.00	$ 15.89	0.00	$ -	0.00	$ -	0.00
12	686034	VINEGAR APPL CIDER 40GRAIN	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	1.00	$ 17.77	0.00	$ -	0.00
13	29078	LIME 12 CT	Ben E Keith	1 - PRODUCE	12	ct	0.00	$ -	2.00	$ 8.99	0.00	$ -	0.00	$ -	0.00
14	650547	TOMATO DICED W/GREEN CHILES	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	1.00	$ 18.88	0.00	$ -	0.00	$ -	0.00
15	286500	Ice Cream Vanilla Cr 3 Gal	Ben E Keith	6 - DAIRY	384	fl oz	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
16	650474	KETCHUP FANCY 33% SOLIDS	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	1.00	$ 20.69	0.00	$ -	0.00	$ -	0.00
17	140005	MUSHROOM WHITE SMALL BUTTON	Ben E Keith	1 - PRODUCE	0	0	0.00	$ -	1.00	$ 20.98	0.00	$ -	0.00	$ -	0.00
18	771131	CROUTON SEASONED HOMESTYLE	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	1.00	$ 22.30	0.00	$ -	0.00
19	660409	SAUCE LOUISIANA RED HOT	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	1.00	$ 11.24	0.00	$ -	1.00	$ 11.24	0.00
20	150015	Onion Green Iceless W/Root	Ben E Keith	1 - PRODUCE	32	oz	0.00	$ -	1.00	$ 8.29	1.00	$ 8.29	0.00	$ -	0.00
21	780009	SUGAR BROWN LIGHT IN BAGS	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	1.00	$ 27.69	0.00	$ -	0.00
22	155030	Onion Yellow Jumbo	Ben E Keith	1 - PRODUCE	800	oz	0.00	$ -	0.00	$ -	1.00	$ 13.99	0.00	$ -	0.00
23	774173	Pepper Red Crushed	Ben E Keith	4 - GROCERY	52	oz	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
24	920919	TUMBLER 20 OZ AMBER	Ben E Keith	8 - EQUIP & SUPPLY	0	0	0.00	$ -	0.00	$ -	1.00	$ 29.99	0.00	$ -	0.00

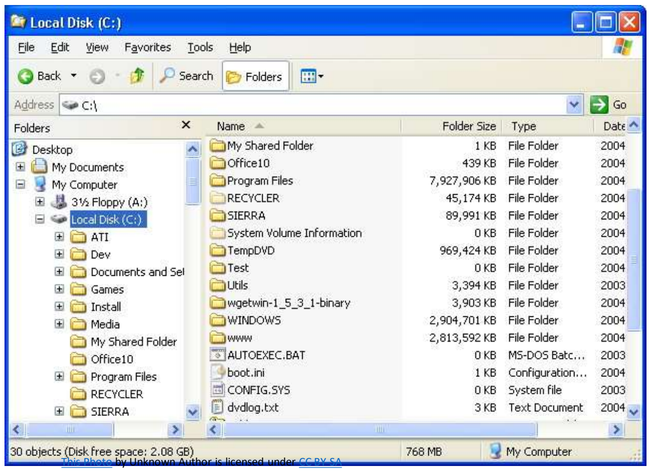
For the excel sheet, we need to store data in a way that allows us to efficiently perform various reordering operations supported by Excel.

Maps



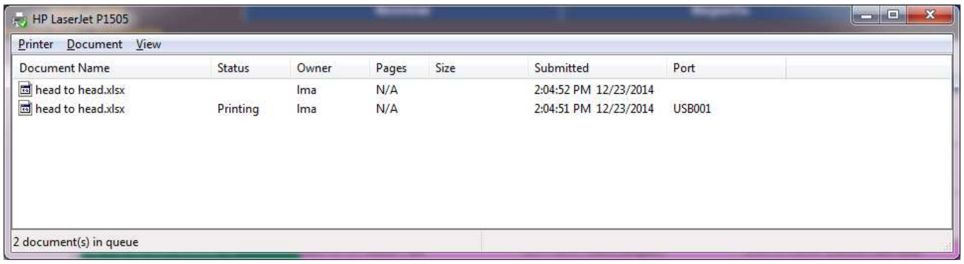
For maps, we need to store the locations and paths in a way that makes it efficient to compute the shortest distance between the two points.

Files and directories



We need an efficient data structure to locate a file or directory, or sub-directories quickly.

Printer



For the printer queue, we need a data structure to efficiently implement first-in-first-out behavior.

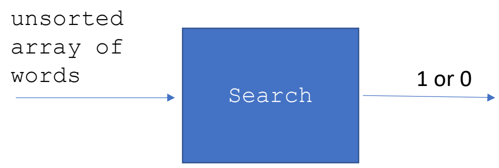
Plates

To organize a deck of plates, we need a data structure that efficiently implements last-in-first-out behavior.

Why do we need different data structures?

- A single data structure may not suffice for all purposes
- Efficiency of an algorithm depends on the underline data structures

Search



Storing words

```
char dict[12][8];
```

What are the intermediate steps during a search operation for a word, e.g., DOG?

B	A	T	\0				
D	E	S	K	\0			
B	A	L	L	\0			
A	P	P	L	E	\0		
C	O	L	D	\0			
A	I	M	\0				
C	A	L	M	\0			
D	O	L	L	\0			
A	S	K	\0				
D	O	G	\0				
A	B	O	V	E	\0		
C	A	T	\0				

To search for a word W, we can iterate all rows and check if the rows indeed contain W. Checking if a row contains W would require comparing the elements of a given row and the individual character of W until '\0' is encountered in both.

Search


```c
#include <string.h>

int search(char dict[][8], int n_words, char word[])
{
   int i;
   for (i = 0; i < n_words; i++) {
      if (strcmp(dict[i], word) == 0) {
         return 1;
      }
   }
   return 0;
}
```

Search

Can we do the search operation faster when the array of words is already sorted?

Storing sorted words

```
char dict[12][8];
```

What are the intermediate steps during a search operation for a word, e.g., DOG?

A	B	O	V	E	\0		
A	I	M	\0				
A	P	P	L	E	\0		
A	S	K	\0				
B	A	L	L	\0			
B	A	T	\0				
C	A	L	M	\0			
C	A	T	\0				
C	O	L	D	\0			
D	E	S	K	\0			
D	O	G	\0				
D	O	L	L	\0			

Storing sorted words

DOS

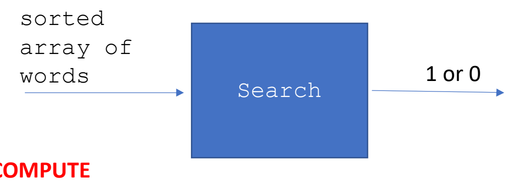
```
char dict[12][8];
```

0	A	B	O	V	E	\0		
1	A	I	M	\0				
2	A	P	P	L	E	\0		
3	A	S	K	\0				
4	B	A	L	L	\0			
5	B	A	T	\0				
6	C	A	L	M	\0			
7	C	A	T	\0				
8	C	O	L	D	\0			
9	D	E	S	K	\0			
10	D	O	G	\0				
11	D	O	L	L	\0			

idx_of[0] = 0;
idx_of[1] = 4;
idx_of[2] = 6;
idx_of[3] = 9;
idx_of[4] = 12;
idx_of[5] = 12;
…
idx_of[26] = 12;

We can pre-compute the indices of the first word that starts with ‘a’ in idx_of[0], ‘b’ in idx_of[1], ‘c’ in idx_of[2], and so on. Now, if the word we are searching starts with a ‘c’, we need to search the words stored at indices idx_of[2] to idx_of[3]-1.

Search



PRE-COMPUTE

idx_of[0] -> index of first word that starts with ‘a’.
idx_of[1] -> index of first word that starts with ‘b’.
idx_of[2] -> index of first word that starts with ‘c’.
…
idx_of[25] -> index of first word that start with ‘z’.
idx_of[26] -> Total number of words.

```
#include <string.h>

// idx_of[26] contains total number of words
int search(char dict[][8], char word[],
           unsigned int idx_of[27])
{
   unsigned int idx = word[0] - ‘a’;
   int i;
   assert(idx < 26);
   for (i = idx_of[idx]; i < idx_of[idx+1]; i++) {
      if (strcmp(dict[i], word) == 0) {
         return 1;
      }
   }
   return 0;
}
```

Abstraction

```
#include <stdio.h>

int main() {
  int num;
  printf("Enter a Number\n");
  scanf("%d", &num);
  printf("You Entered: %d\n", num);
  return 0;
}
```

Abstraction

- Abstraction hides the unnecessary implementation details from the users
- For example, a header file in `C` provides a list of function declarations that users can directly use in their program without worrying about the underline implementation
  - e.g., `printf, scanf`, etc., in `stdio.h`

Abstract data type (ADT)

- ADT specifies the user's point of view of a data structure
  - i.e., the supported operations and their semantics, the possible values

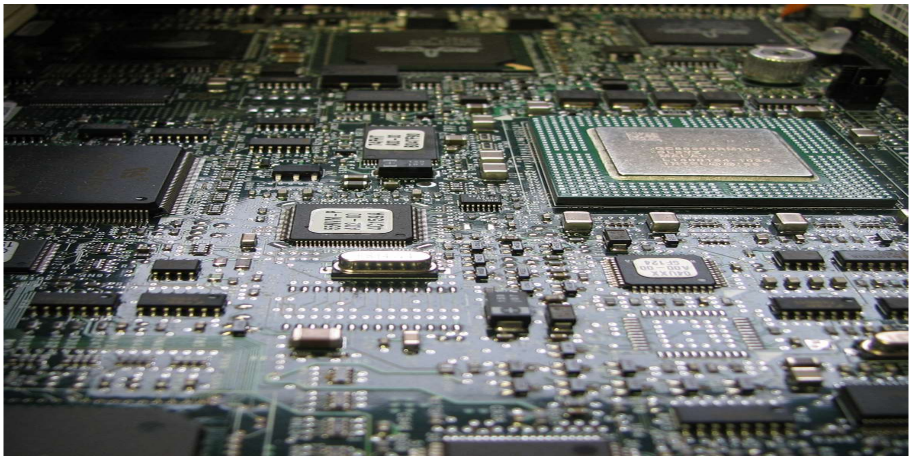
- ADT hides the implementation details from the user

- ADT can be defined using a header file in C/C++ or an interface in Java

Abstract (or user's) view

- Power On/Off
- Select a program
- Start/Pause/Resume

Implementation view



Abstract data type (ADT)

- Dictionary ADT
  - insert => insert a word
  - delete => delete a word
  - search => search a word

Example

- Train ticket booking platform
- Suppose there are thousands of requests, and you can process only one request at a time
  - In which order will you process the requests?
  - In which order will you store the requests?

Queue

- Follow first-in, first-out (FIFO) policy
- Queue ADT:
  - QUEUE-EMPTY => returns true if the queue is empty
  - ENQUEUE => insert an item at the end of the queue
  - DEQUEUE => remove an item from the top of the queue

Example

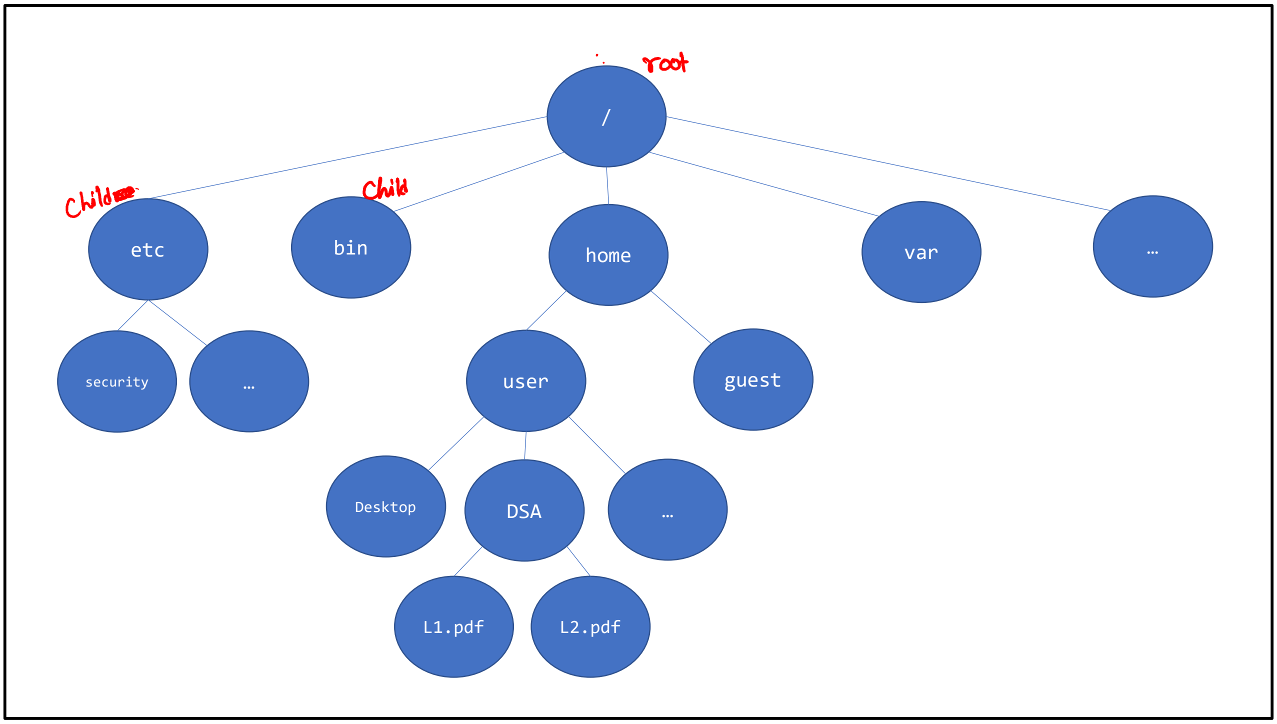
- Undo (ctrl+z) operation in power-point
- How do you store the words to perform undo efficiently?

Stack

- Follow last-in, first-out (LIFO) policy
- Stack ADT:
  - STACK-EMPTY => returns true if the stack is empty
  - PUSH => insert an item on the top of the stack
  - POP => remove an item from the top of the stack

Example

- Files and directories structure in an OS
  - How can we efficiently store the relationship between the directory and subdirectories or files?


root
Child
Child
/
etc
bin
home
var
…
security
…
user
guest
Desktop
DSA
…
L1.pdf
L2.pdf

Tree

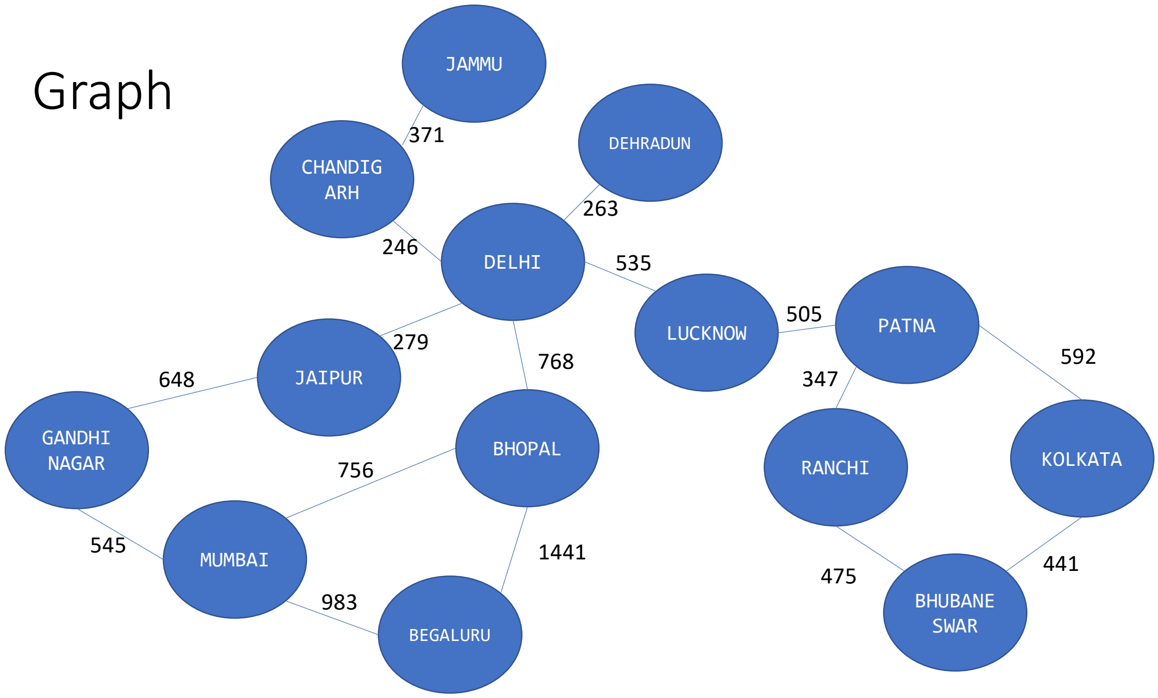
- The root node has no parent
- The other nodes have exactly one parent
- A node in the tree can have zero or more children

- Tree ADT:
  - root
  - insert
  - delete
  - children
  - parent

Example

- How can we store the list of cities and the distance between neighboring cities?

Graph



JAMMU
371
CHANDIG
ARH
DEHRADUN
263
246
DELHI
535
LUCKNOW
505
PATNA
279
JAIPUR
768
592
648
347
GANDHI
NAGAR
BHOPAL
RANCHI
KOLKATA
756
MUMBAI
545
1441
475
441
983
BEGALURU
BHUBANE
SWAR

Graph

- A set of vertices and edges
- Edges may have weights

- Graph ADT:
  - vertices
  - edges
  - addVertex
  - removeVertex
  - addEdge
  - removeEdge
  - incomingEdges
  - outgoingEdges

Data structures

- We will discuss the data structures presented in previous slides in detail in the upcoming classes

Revision of recursion

References

- Section-2.3 from the CORMEN et al.
- Chapter-2 from Narasimha Karumanchi

Recursion

- A function calling itself is called recursion

Factorial:

0! = 1

n! = n * (n-1) * (n-2) * … * 3 * 2 *1

0! = 1
1! = 1
2! = 1×2
3! = 1×2×3
4! = 1×2×3×4

Factorial

- Recursive definition of factorial

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

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

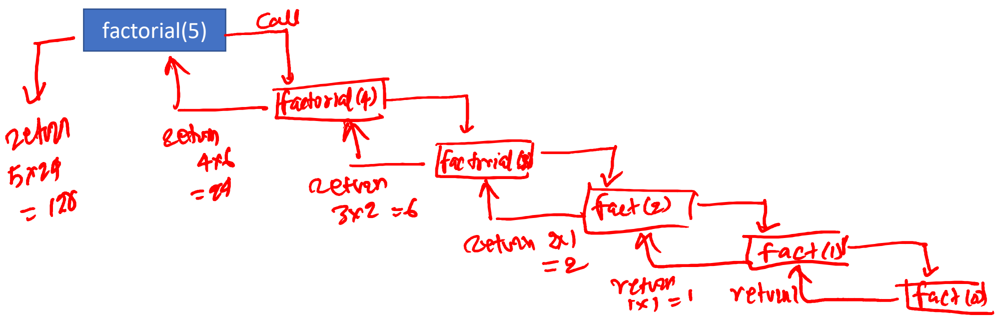
Recursion

- Divide and conquer approach
  - **Divide** the problem into sub-problems of a similar type
  - **Conquer** the subproblem by solving them recursively
  - **Combine** the results of the subproblems to compute the result

- Structure of a recursive solution
  - One or more **base cases**
    - returns a value without making a recursive call
    - e.g., n=0 case in the factorial solution
  - One or more **recursive steps**
    - make recursive calls corresponding to the sub-problems
    - e.g., n*(n-1)! step in the factorial solution

Factorial

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



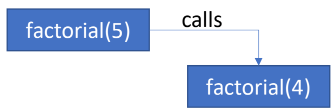
Factorial

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

factorial(5)

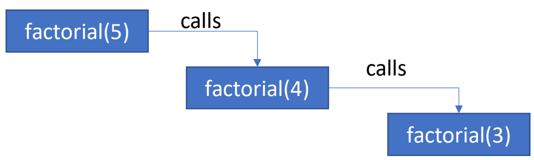
Factorial

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



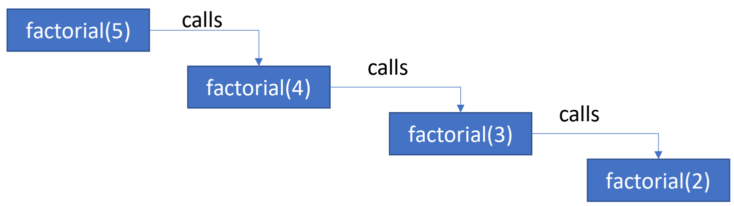
Factorial

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



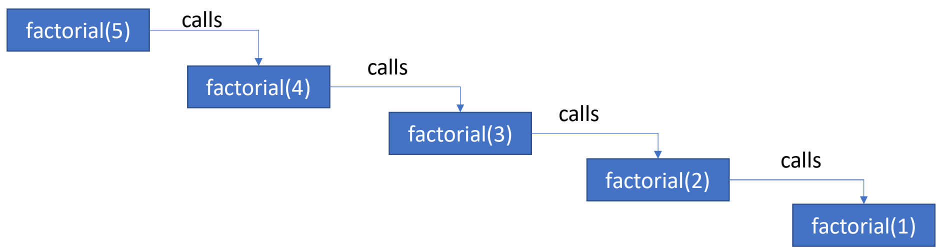
Factorial

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



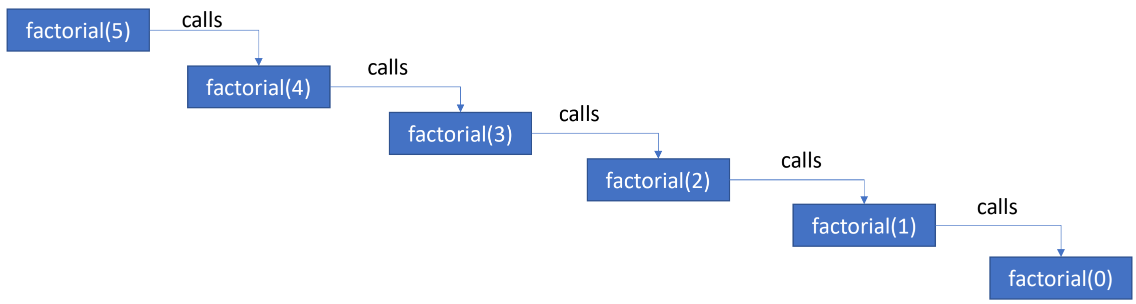
Factorial

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



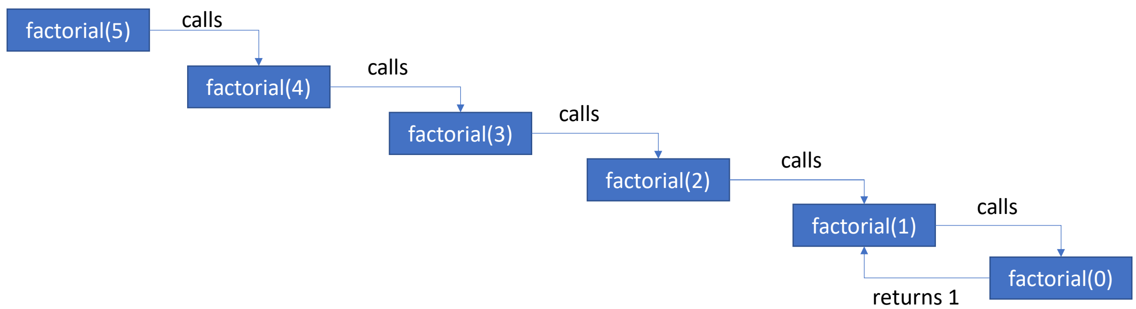
Factorial

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



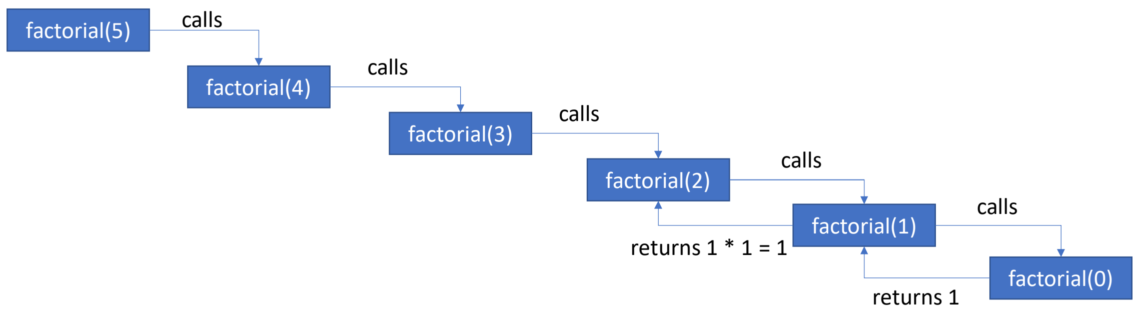
Factorial

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



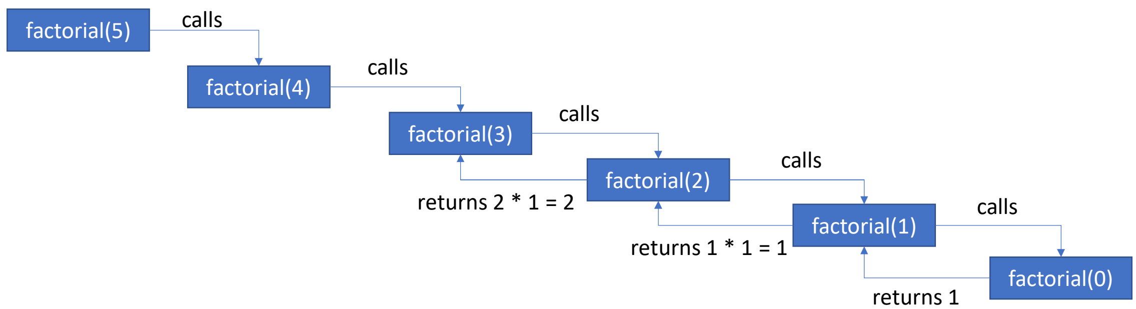
Factorial

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



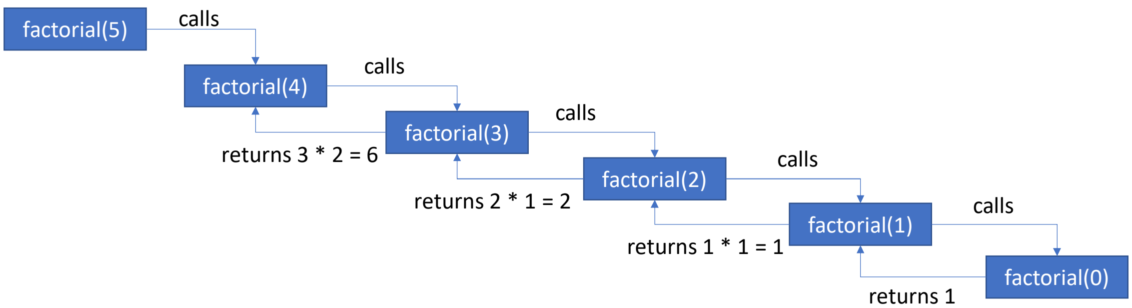
Factorial

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



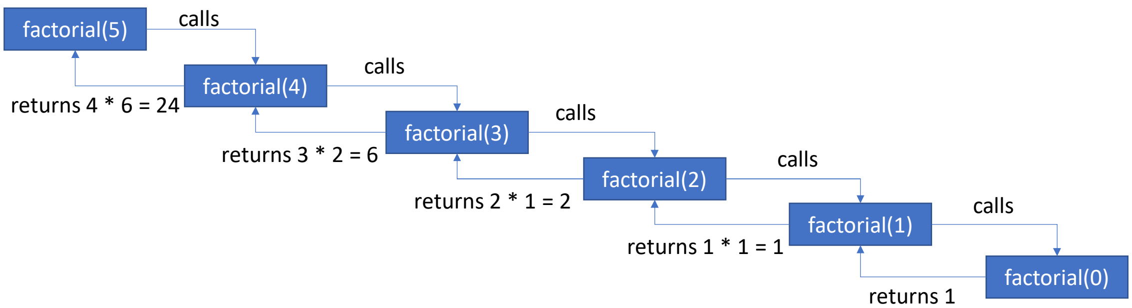
Factorial

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



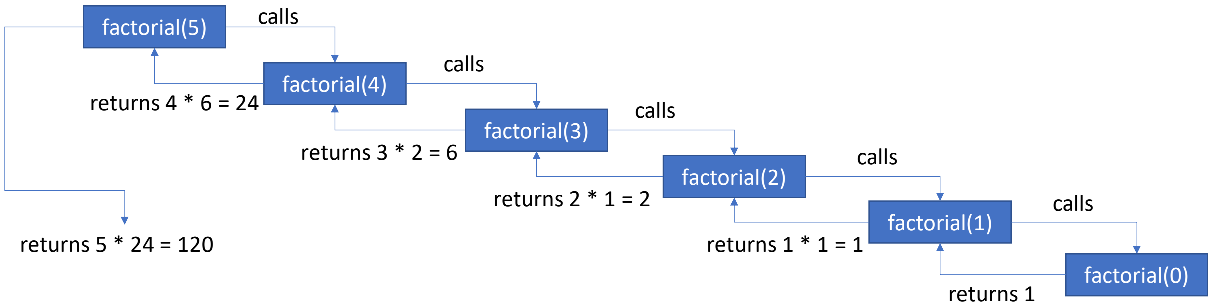
Factorial

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



Factorial

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



power :: x^n

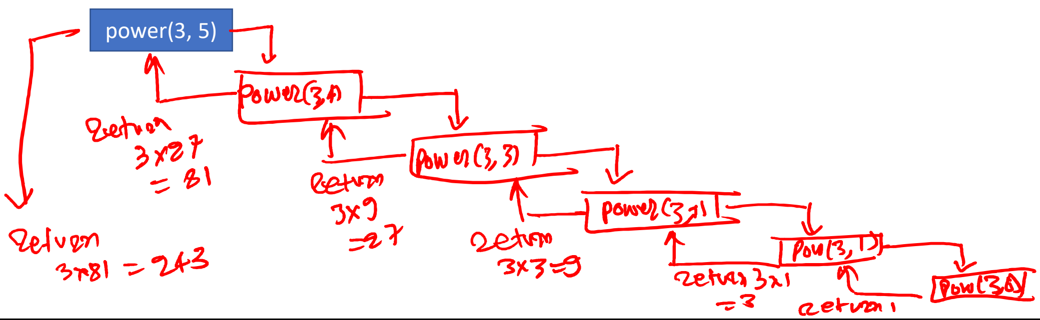
x^n

$$
\begin{matrix} 1 & n = 0 \\ x * x^{n-1} & n > 0 \end{matrix}
$$

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```

x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



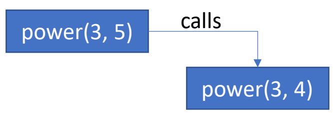
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```

power(3, 5)

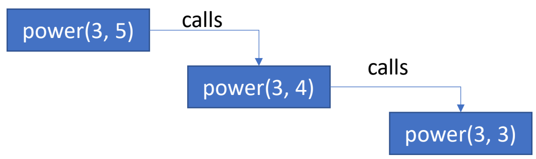
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



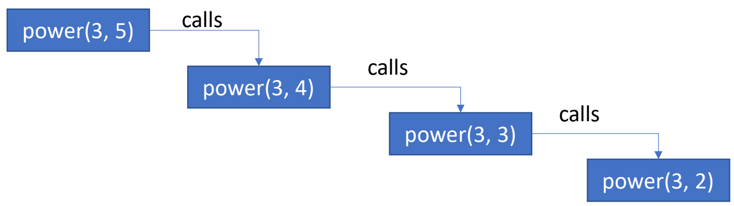
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



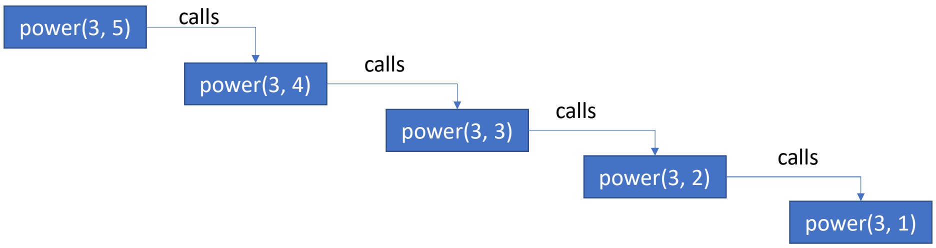
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



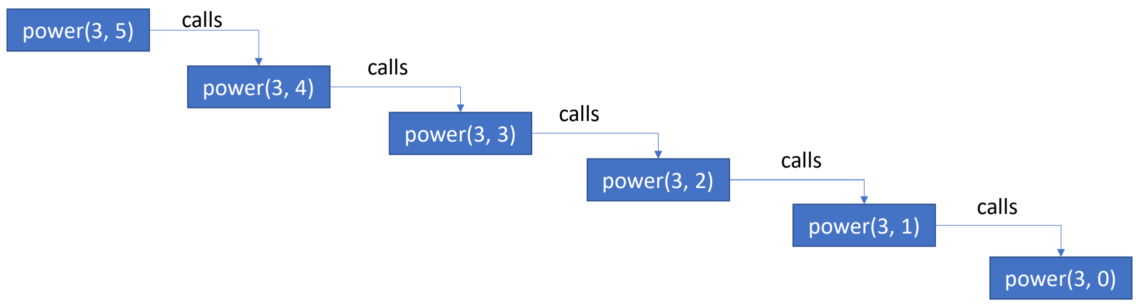
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



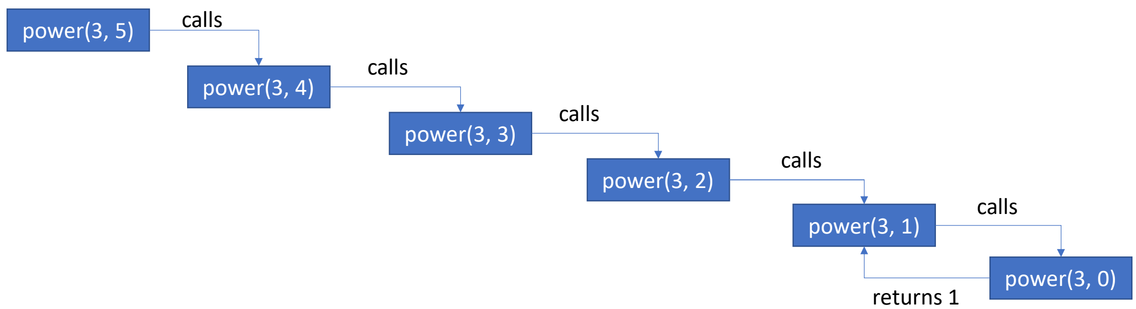
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



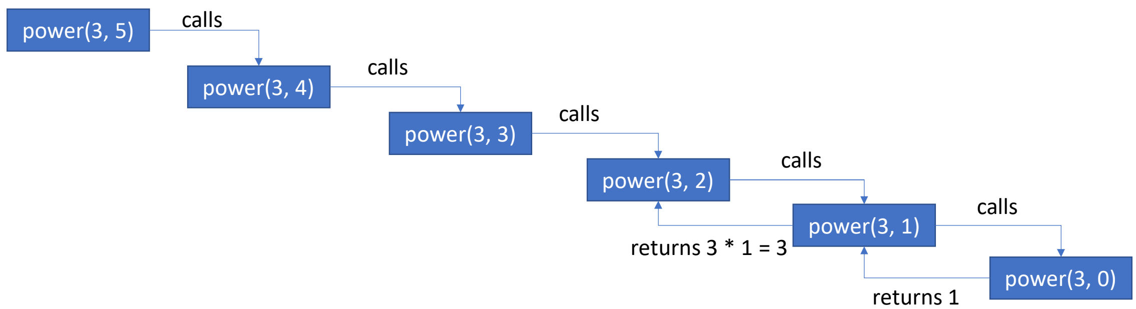
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



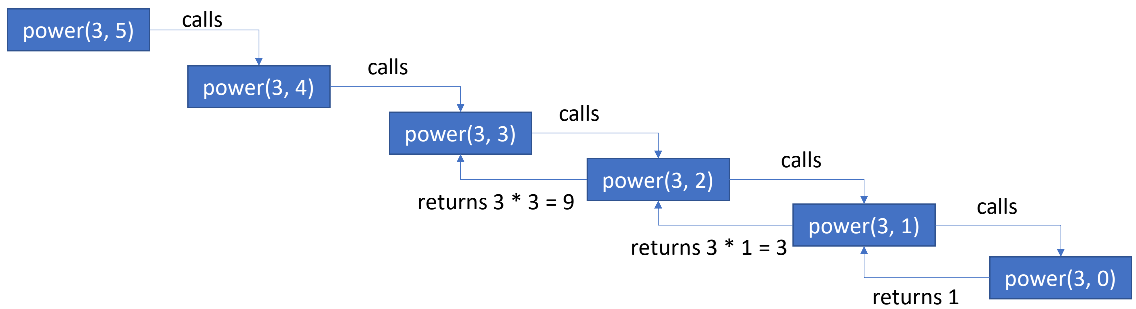
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



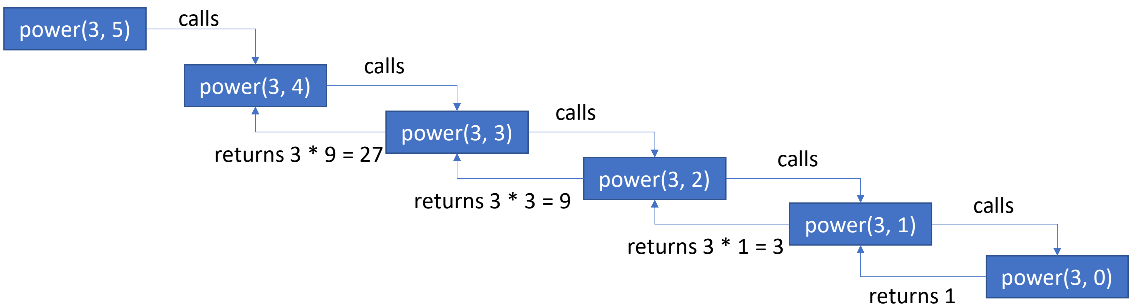
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



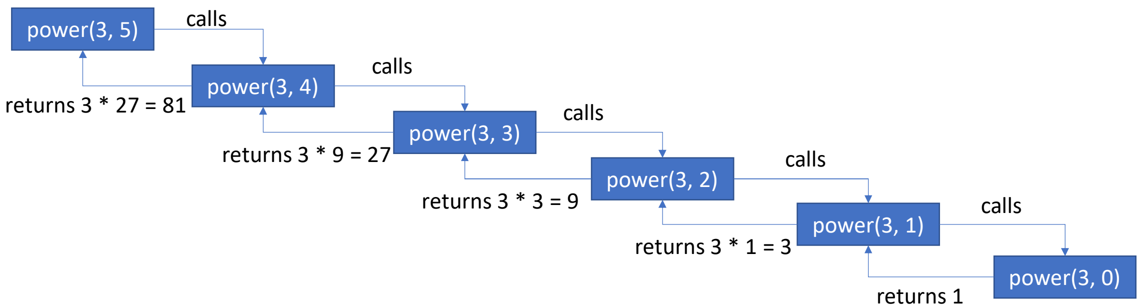
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



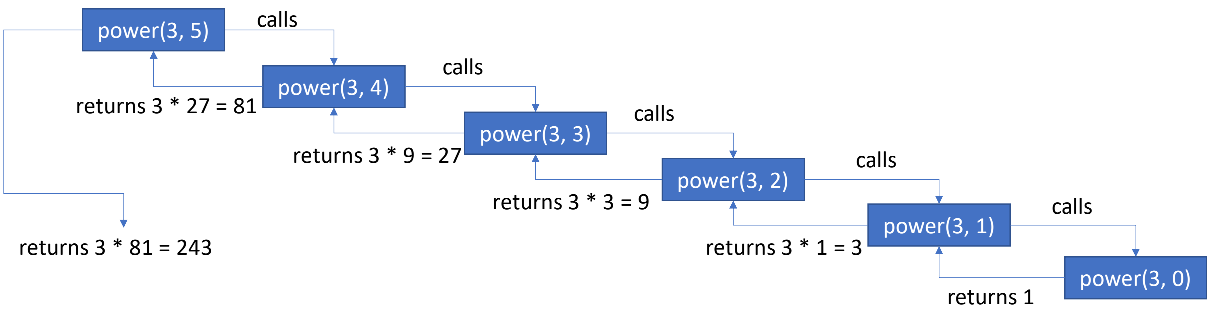
x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



x^n

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```



Faster algorithm

Faster algorithm for x^n

- Can we reduce computation?

Faster algorithm for x^n

- Properties of power

$$x^{100} = x^{50} * x^{50}$$
$$x^{101} = x^{50} * x^{50} * x$$

Faster algorithm for x^n

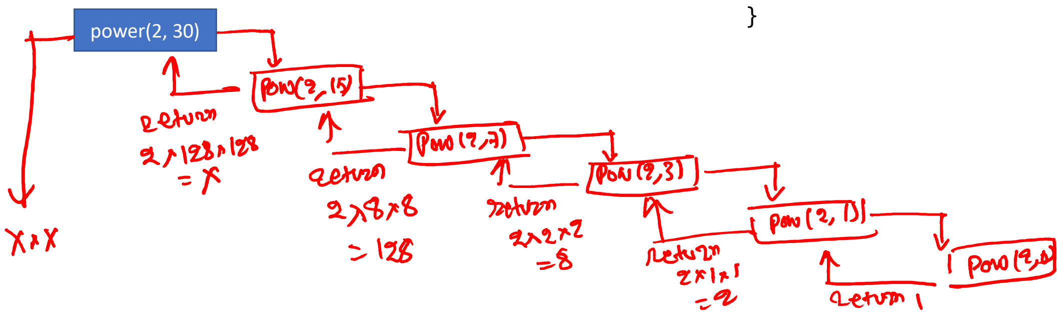
$$
\begin{array}{ll}
1 & if\ n = 0 \\
x^{\frac{n}{2}} * x^{\frac{n}{2}} & if\ n \,\%\, 2 = 0 \\
x * x^{\frac{n-1}{2}} * x^{\frac{n-1}{2}} & if\ n \,\%\, 2 = 1
\end{array}
$$

Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```

Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



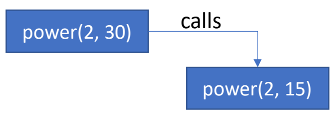
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```

power(2, 30)

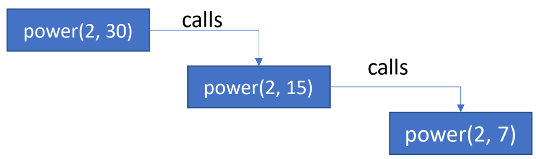
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



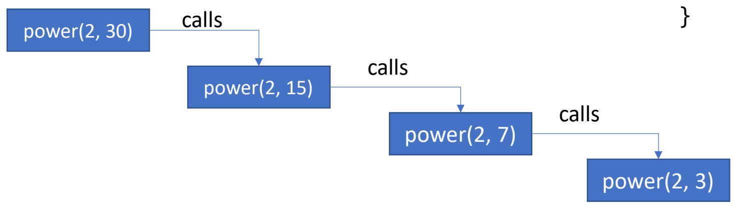
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



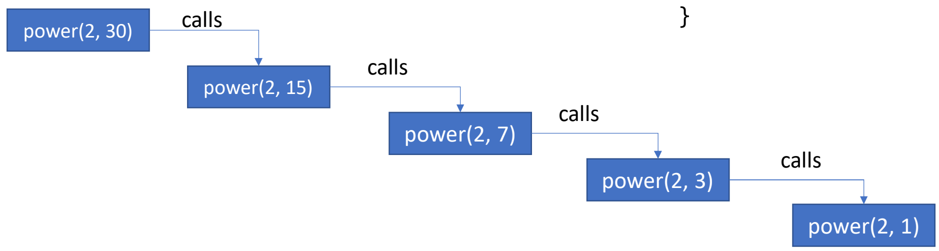
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



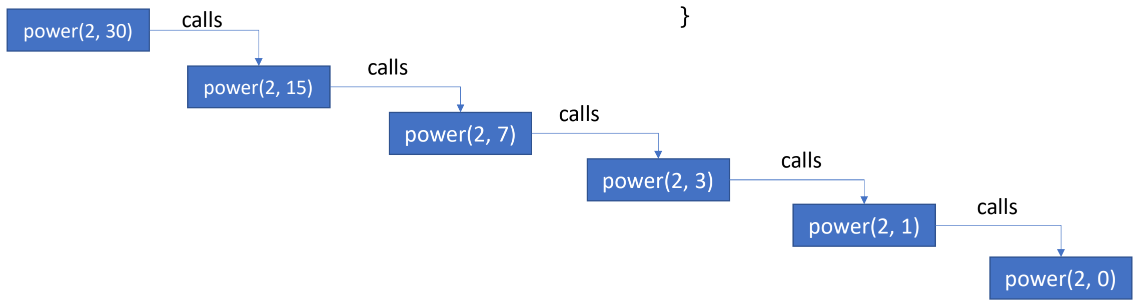
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



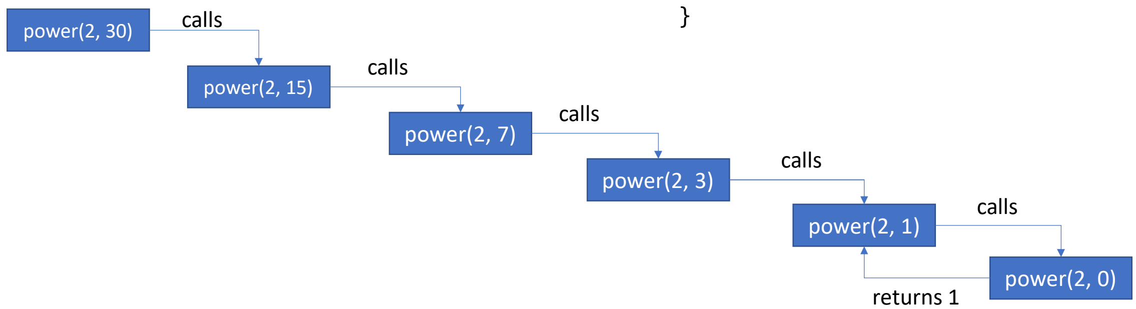
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



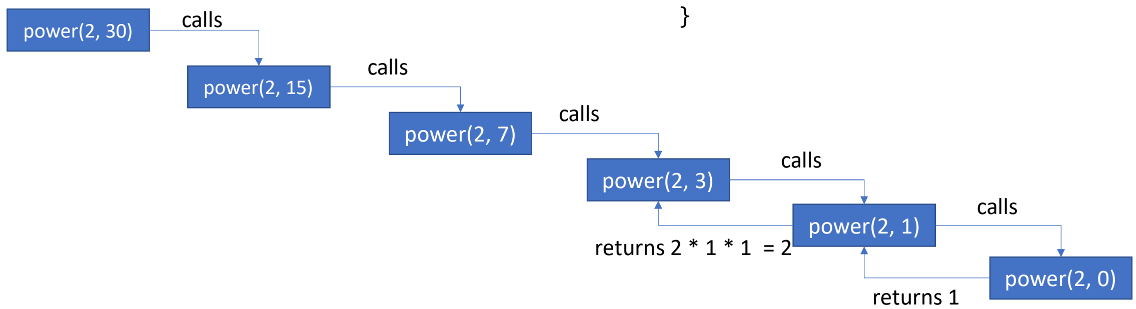
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



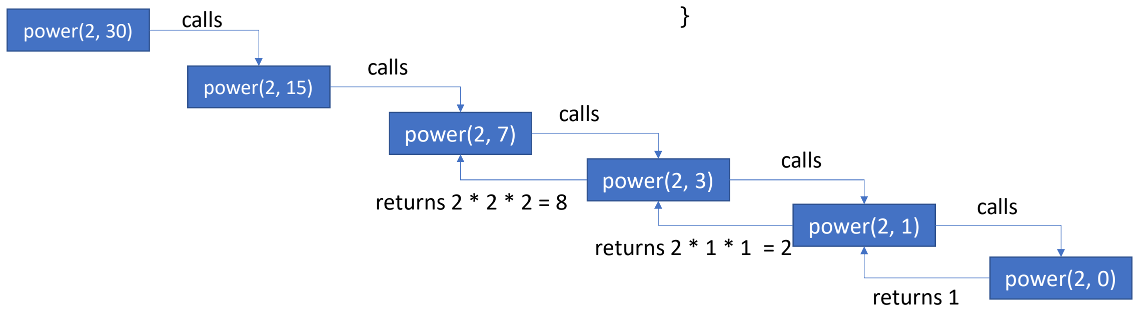
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



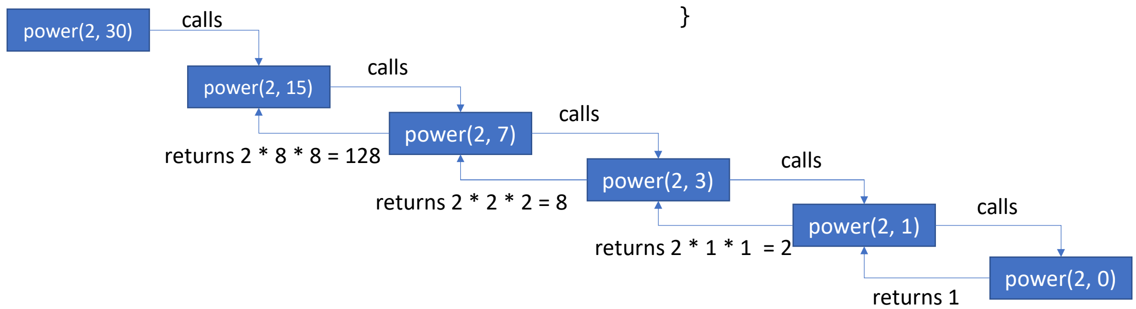
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



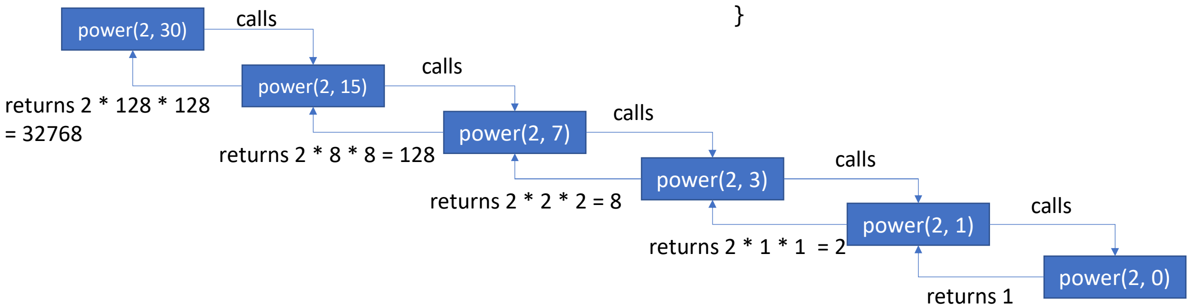
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



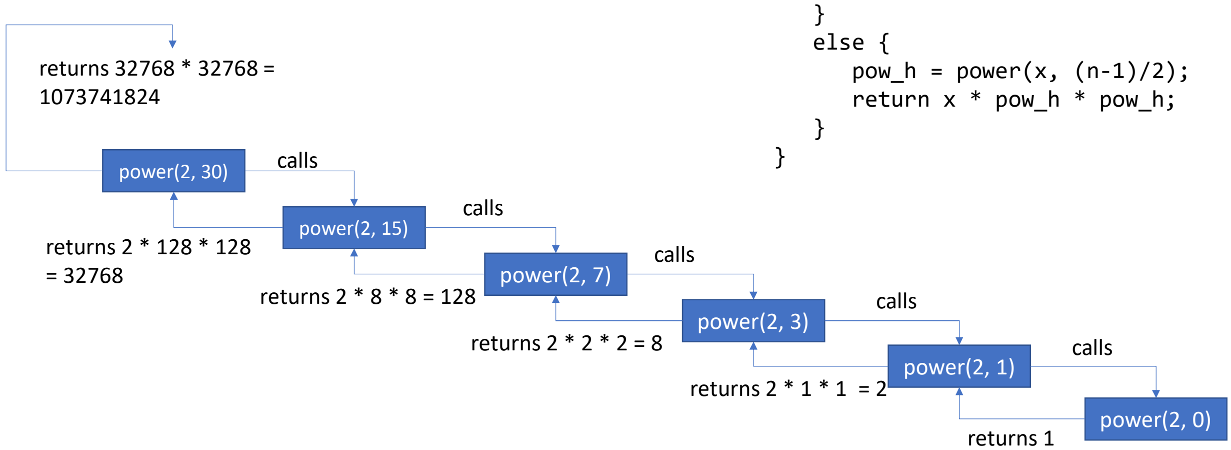
Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



Faster algorithm for x^n

```
int power(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```



Fibonacci numbers

Fibonacci numbers

- Fibonacci numbers are a sequence of numbers in which each number is the sum of the two preceding numbers

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
```

Fibonacci numbers in nature

trillium ~ 3 petals

wild rose ~ 5 petals

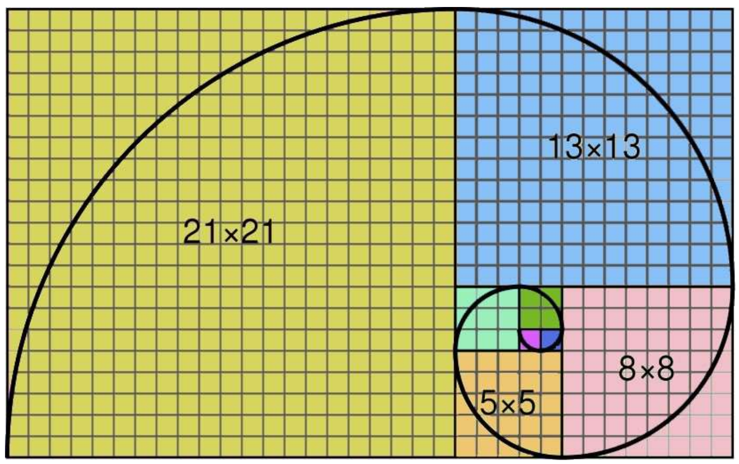
bloodroot ~ 8 petals

black eyed susan ~ 13 petals

chamomile ~ 21 petals

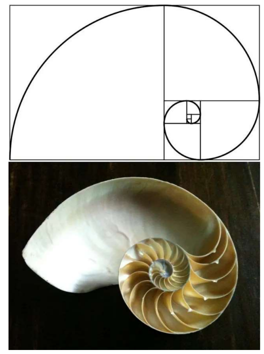
gaillardia ~ 34 petals

Fibonacci spiral



Fibonacci spiral

Galaxy



Sea shell

Golden ratio

2 / 1	2
3 / 2	1.5
5 / 3	1.666
8 / 5	1.6
13 / 8	1.625
21 / 13	1.615
34 / 21	1.619
55 / 34	1.617
89 / 55	1.618
144 / 89	1.617
233 / 144	1.618
377 / 233	1.618

Golden ratio

- The golden ratio (ϕ) is 1.618033988749 …
- Many patterns based on the golden ratio exist in the nature

Golden ratio

Ancient temple in Greece almost fits into a golden rectangle.
We don’t know for sure if the temple was designed that way.

Fibonacci numbers

- Recursive definition of Fibonacci numbers

$$\mathrm{f}(n) = \begin{array}{cc} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{array}$$

$f(0) = 0$
$f(1) = 1$
$f(2) = 1$
$f(3) = 2$
$f(4) = 3$
$\vdots$

Fibonacci numbers

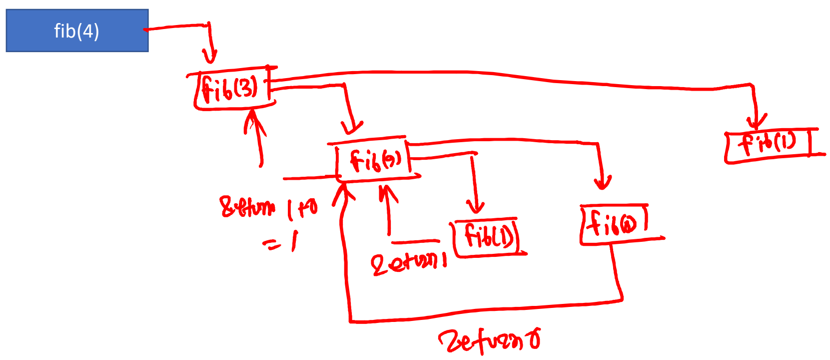
- Recursive definition of Fibonacci numbers

$$\mathrm{f}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



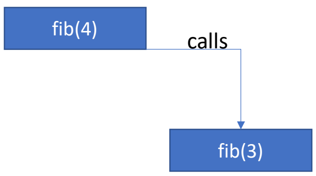
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

fib(4)

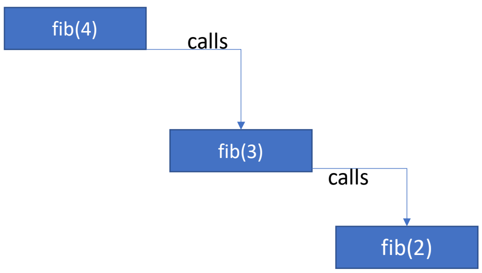
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



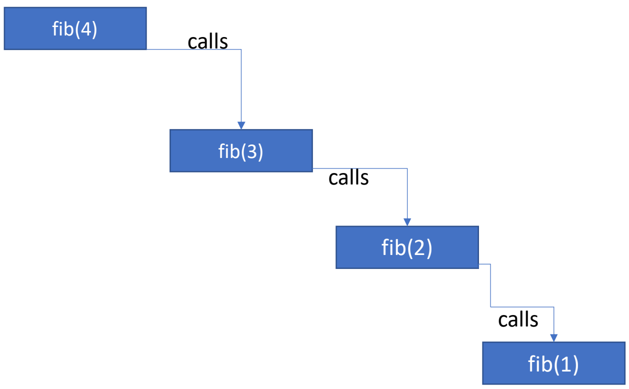
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



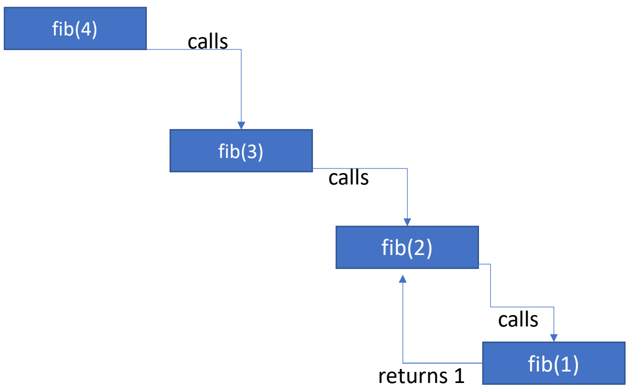
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



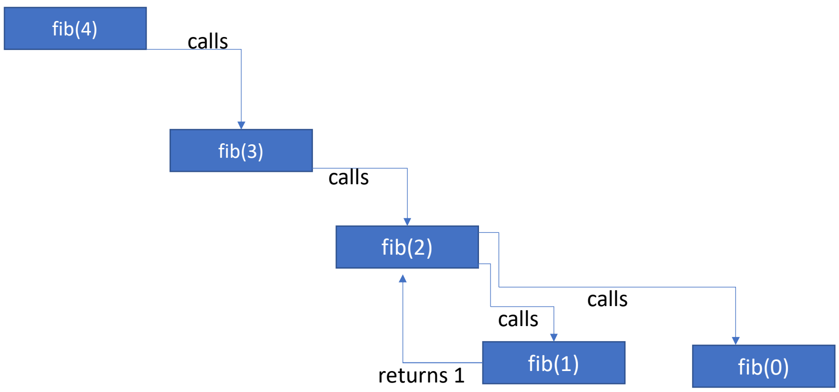
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



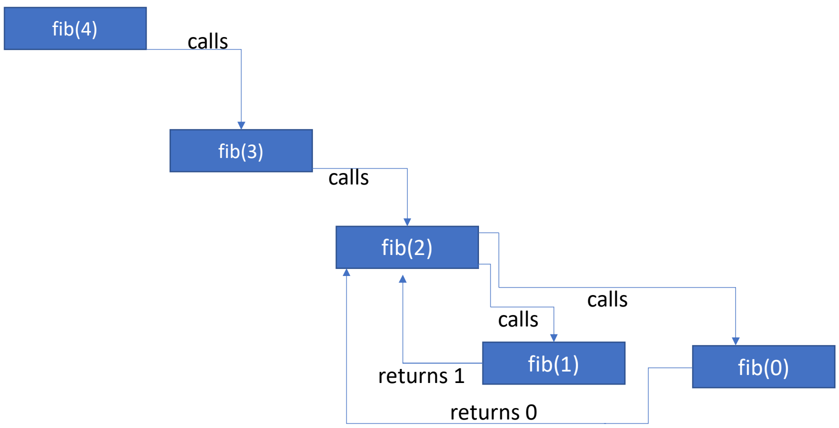
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



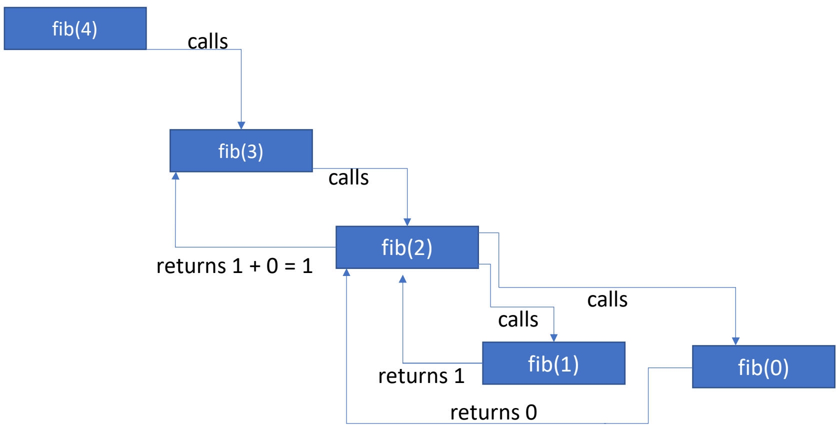
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



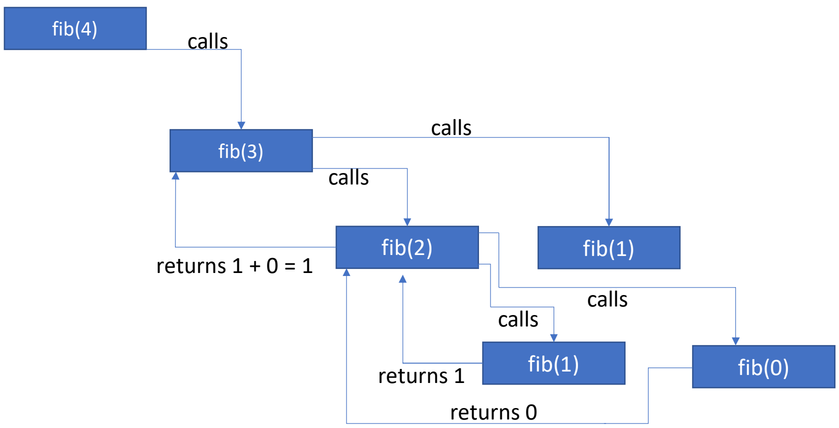
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



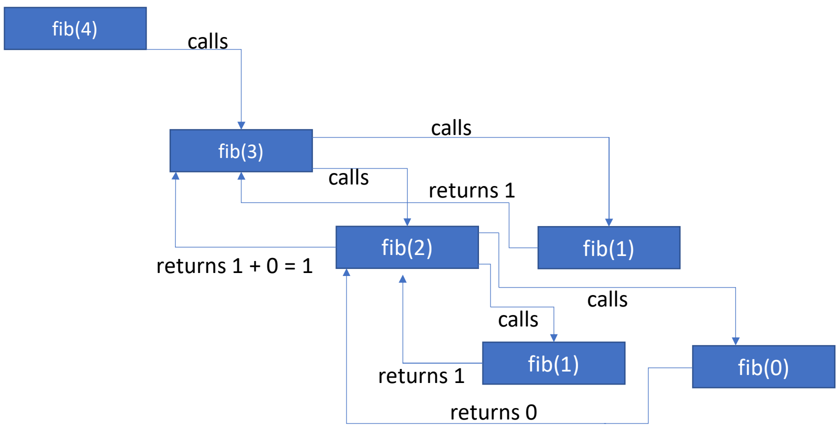
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



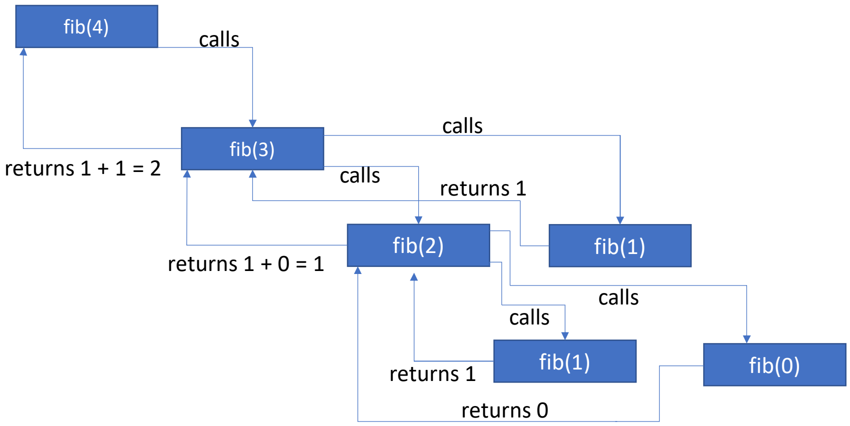
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



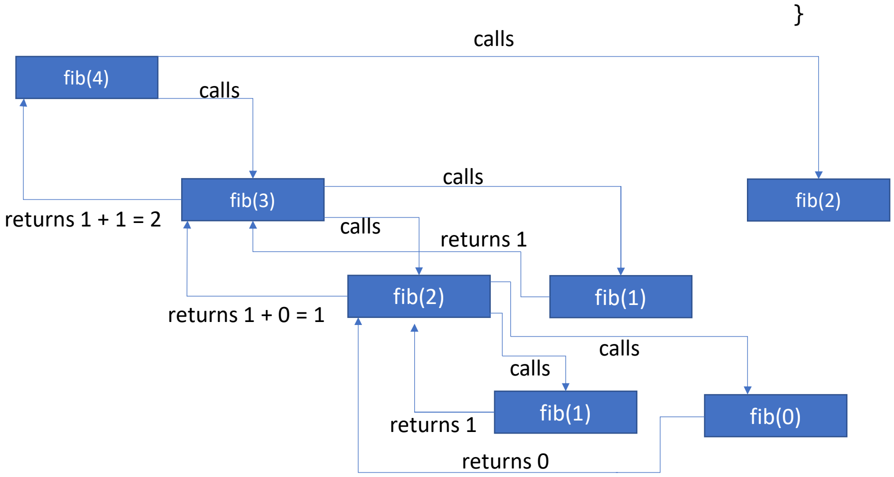
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



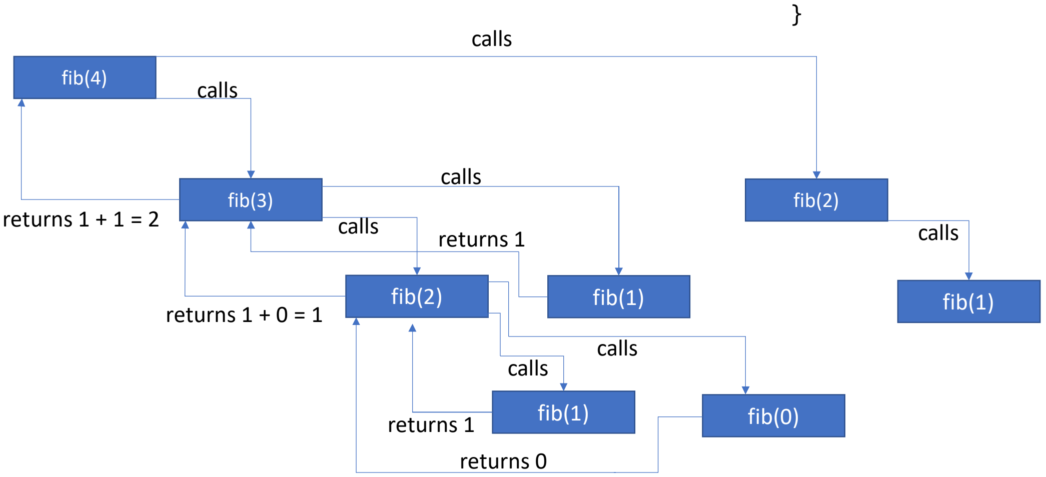
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



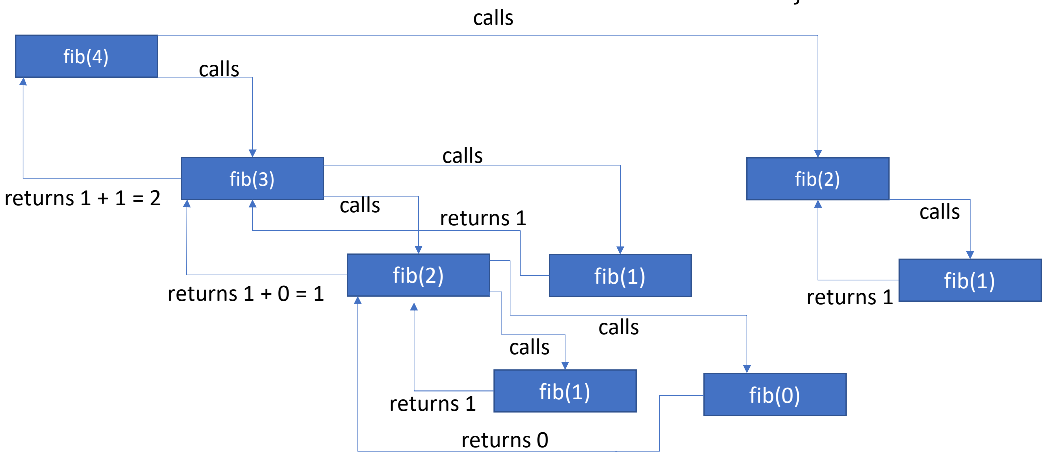
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



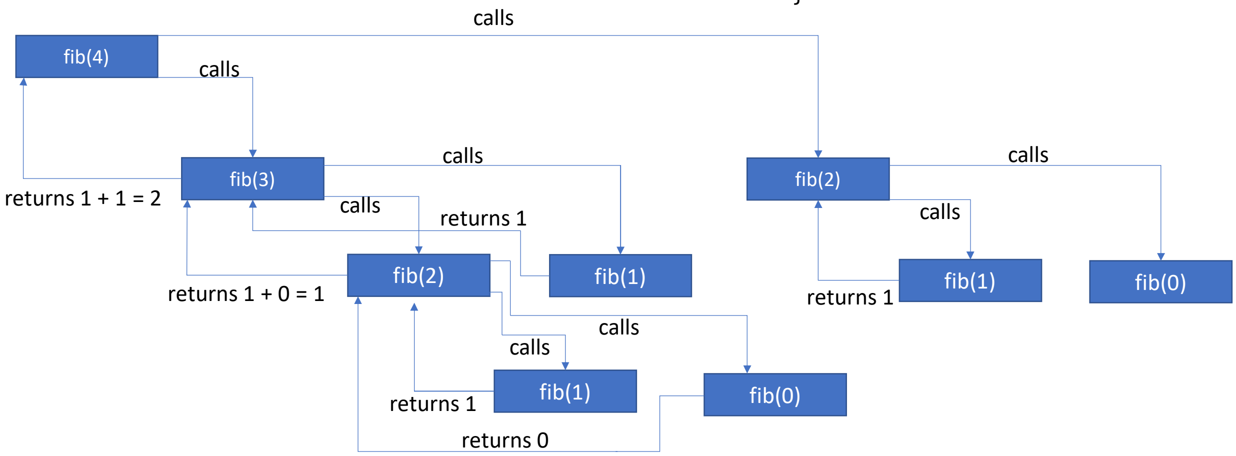
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



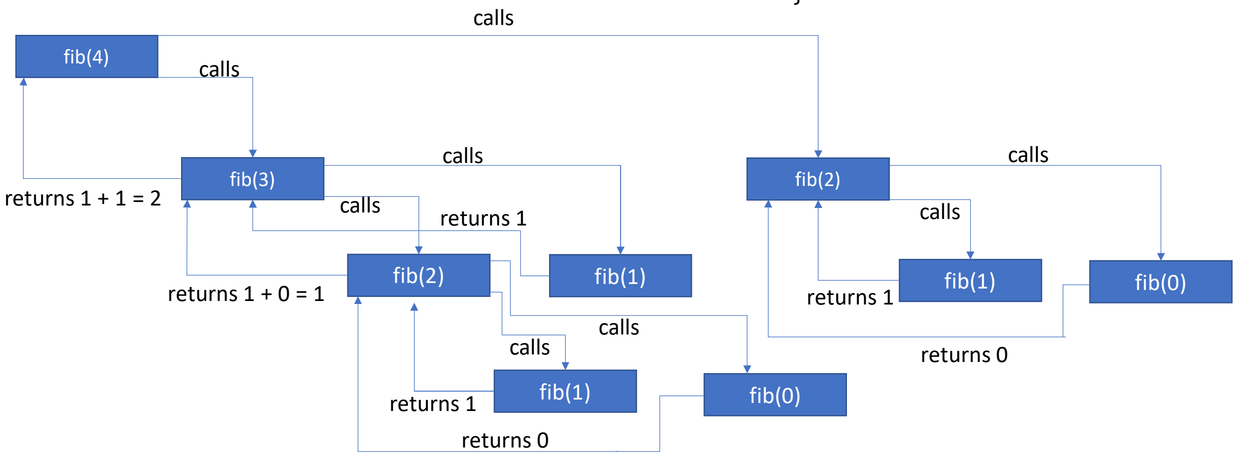
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



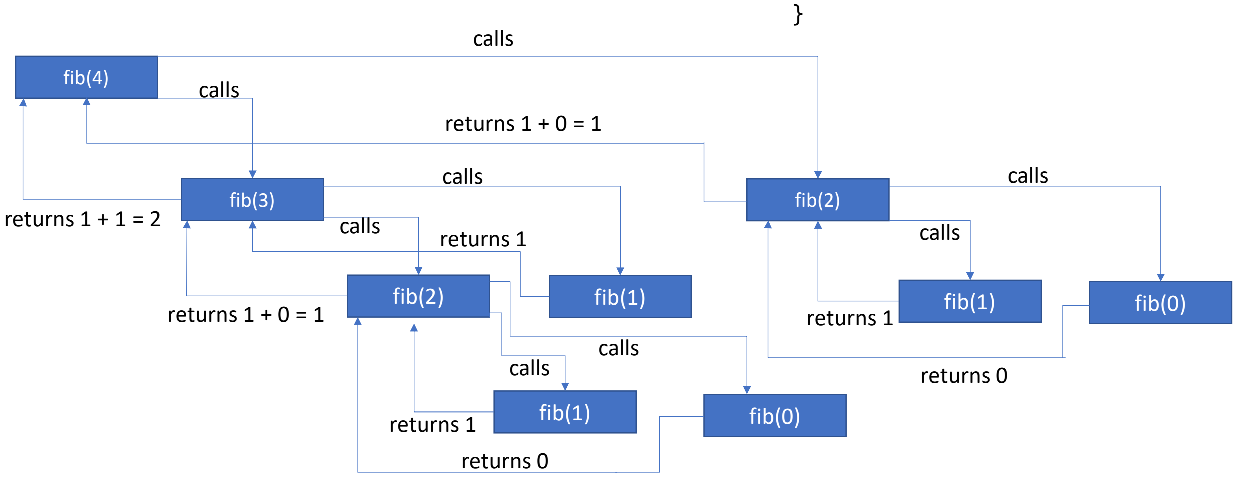
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```