Week 11

Recursion

same number on nodes ⇒ visited at same time / same iteration

Recursive graph search

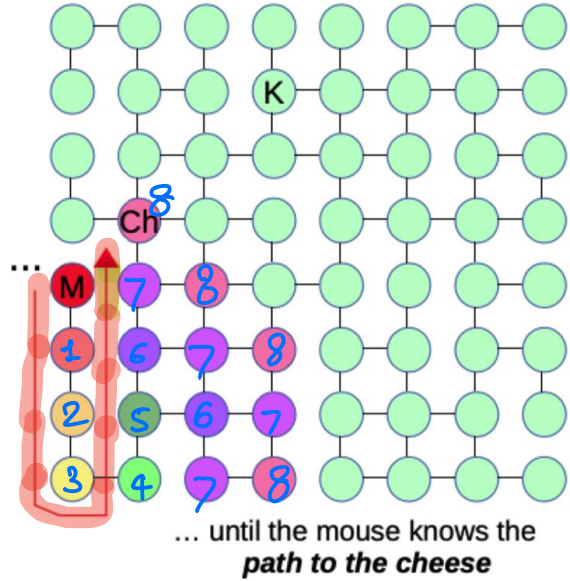
finding path from mouse to cheese
⇒ multiple path

algorithms

BFS

DFS

split + small prob + merge (backtracking)

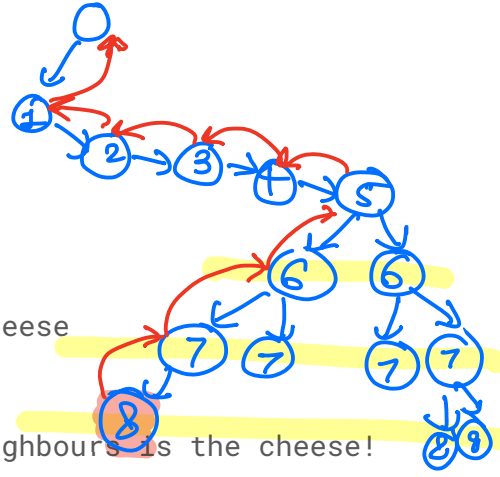


backtracking

Need way to keep track of nodes we have visited

order in which they are visited

Recursive Approach




```
find path to cheese:
Ask my neighbours: What is the path from you to cheese
    They ask their neighbours: What is the path from you to cheese
        They ask their neighbours ....
            ... until { one of the neighbours-neighbours...neighbours is the cheese!
        Now one of the cheese's neighbour knows where to go
    Then a cheese's neighbour's neighbour knows where to go
... until, the mouse knows where to go!
```

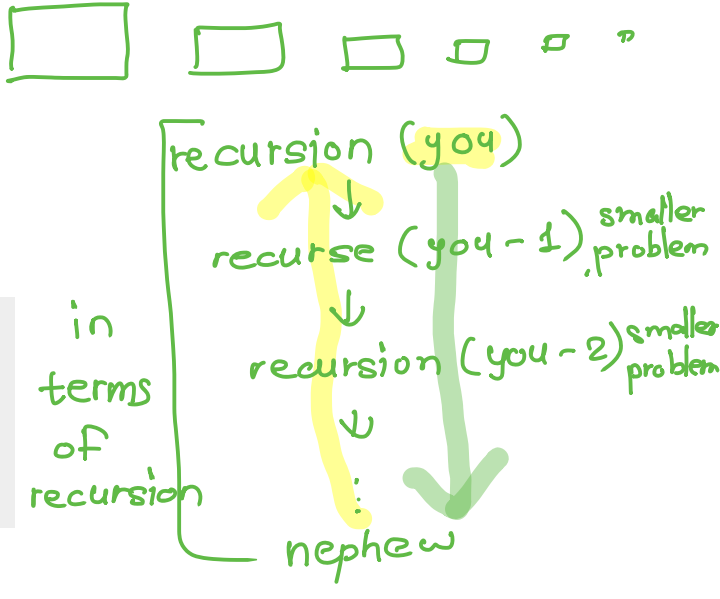
What is Recursion?

- Function calling itself!!

splitting

merging

```
int f1()
{
      …
      f1();
      …
}
```



- One of the common interview question:
- How would you explain recursion to your 5 year old nephew?

Why Use Recursion?

Intuitive understanding

- Fewer Lines of Code
  - usually results in a solution with fewer lines of code than a solution that does not utilize recursion.
- More Elegant Code
  - Intuitively easy to understand
- Increased Readability

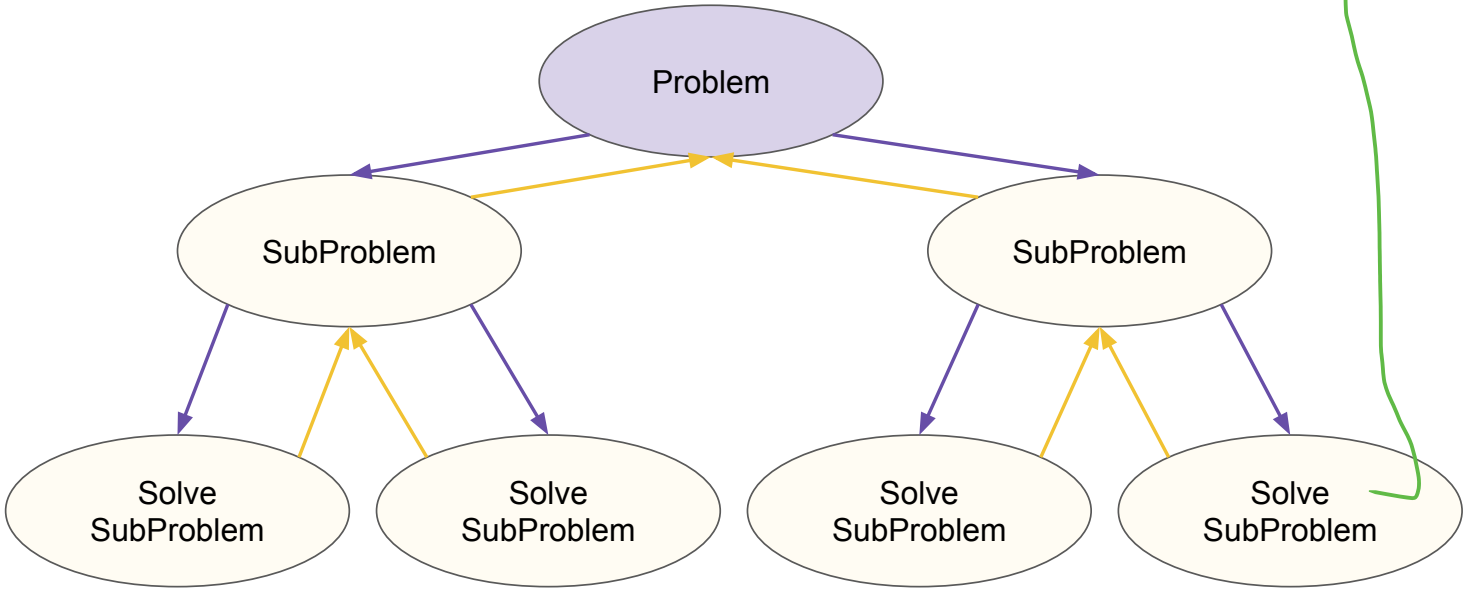
When to use recursion?

For kind of problems where

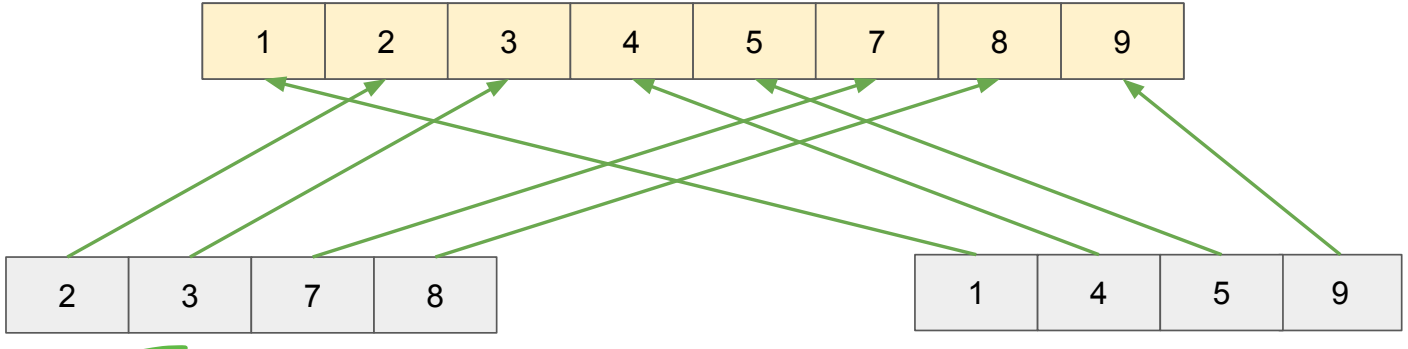
- The problem is progressively broken down, made smaller, or otherwise simplified at each step.
- Then information needed to form the solution is passed back one step at a time, until the solution for the original problem is obtained.

Divide-and-Conquer Strategy

complexity of recursive algo.



Merge Sort - Array Merging



Complexity of array merging for **n** elements?

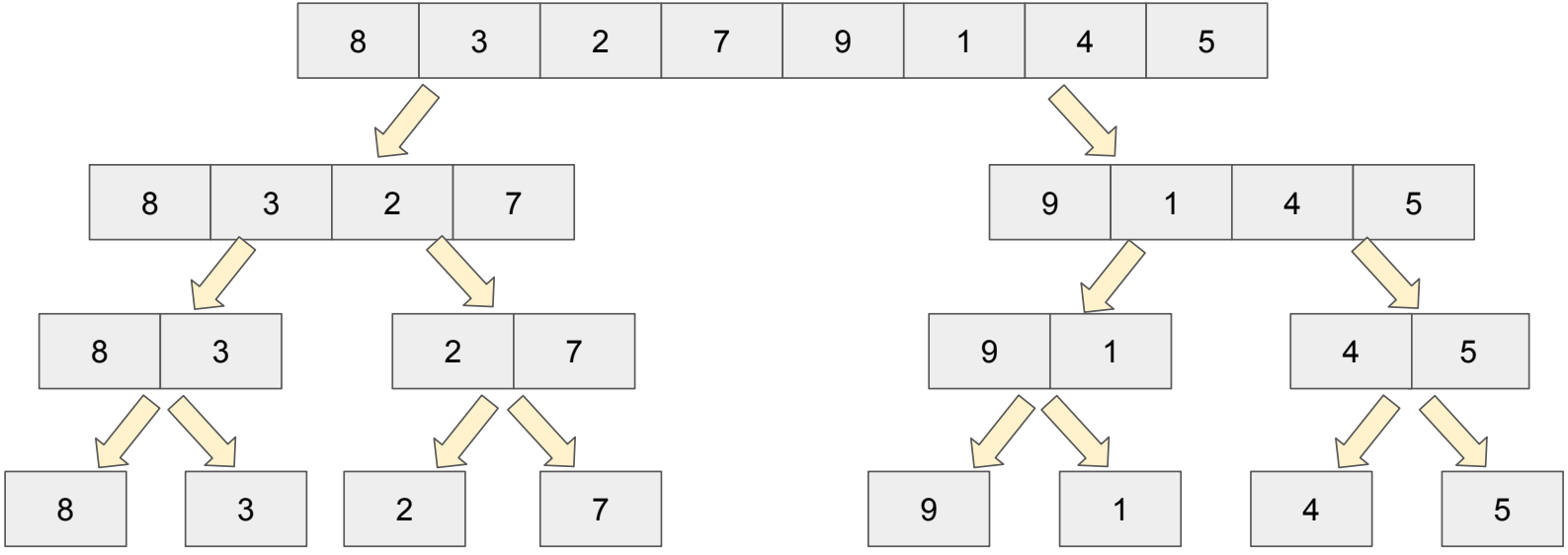
O(1)

O(log n)

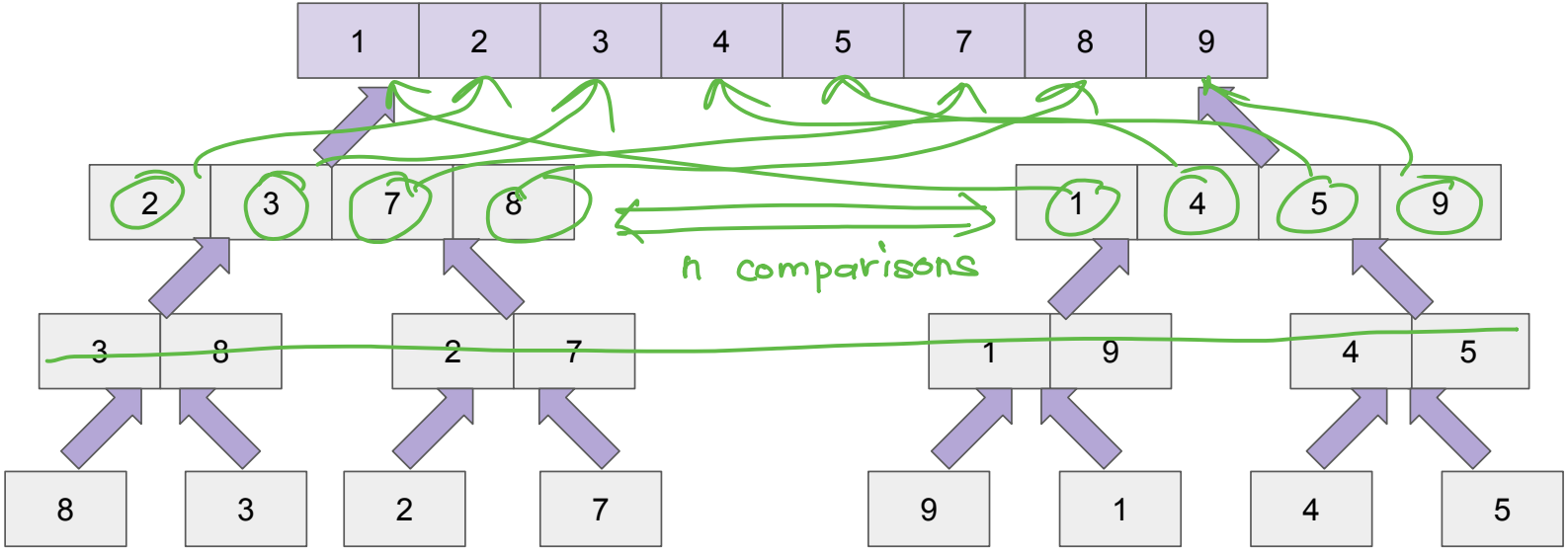
O(n)

O(n^2)

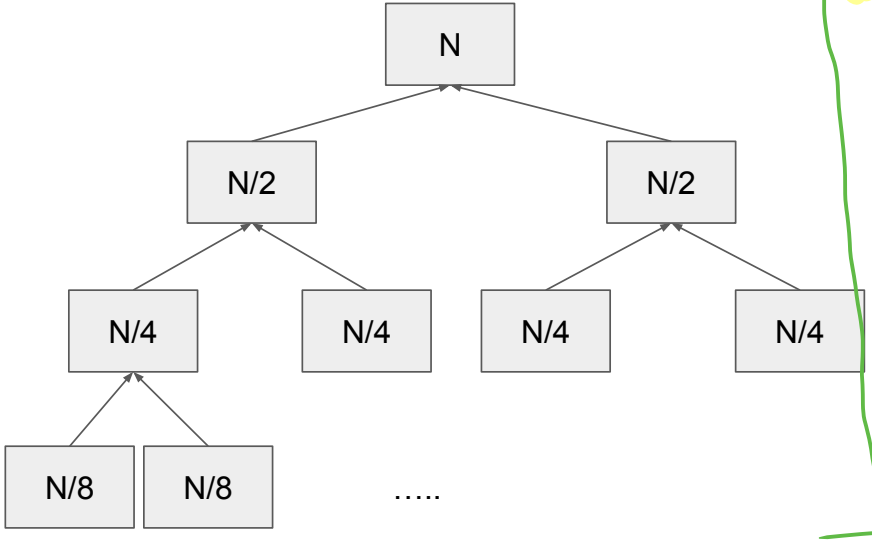
Merge Sort - Splitting



Merge Sort - Merging



Merge Sort : Order of Growth



splitting $\Rightarrow O(\log N)$

merging $\Rightarrow O(N * H)$

H = no. of levels

$= O(N \log N)$

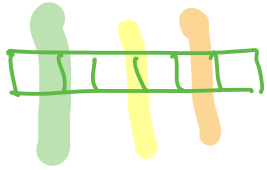
General Principle of Recursion

```
int f1()
{
       //Base case:
              //terminating condition
       //Recursive case:
           f1();
     …
}
```

Structure of Recursive Solution

- Base Case:
  - simplest form of the problem where recursion is not required (terminating condition)
    - Strings: empty string
    - Numeric arrays: array with 1 entry / array with 0 entry
    - Graphs: single node
    - Search problems: empty node/subtree/ or node with item
- Recursive Case:
  - Make the problem smaller, closer to the base case
  - This includes split, simplify, recursively calling with smaller subproblems
  - Merging/re-constructing solution to original problem
    - Strings: breaking string into chunks
    - Numeric arrays: splitting arrays into sub-arrays
    - Graphs: processing on subsets of the graph
    - Trees: splitting into subsets of tree
    - Search: searching on subsets of data structure

Designing Recursive solution



- Base case:
  - Think of the smallest version of the problem and write down the solution
  - Think of the case when the solution to the problem is not found
- Recursive case:
  - Split the problem into subproblems that will get you closer to the base case
  - Many possible ways to do this
  - Choose the splitting solution that will make the recursive case simpler(fewer steps)
- Combine base case and recursive case
- Things to keep in mind:
  - Cover all the instances for all input sizes user i/p something with 0 elements
  - Base case should not make recursive calls
  - Recursive calls should call a smaller instance of original input
  - Take that “Recursive leap of faith”

Sorting an array

Base Case:

- Empty array
- Array with 1 value

Recursive Case:

- Split the problem
- How?

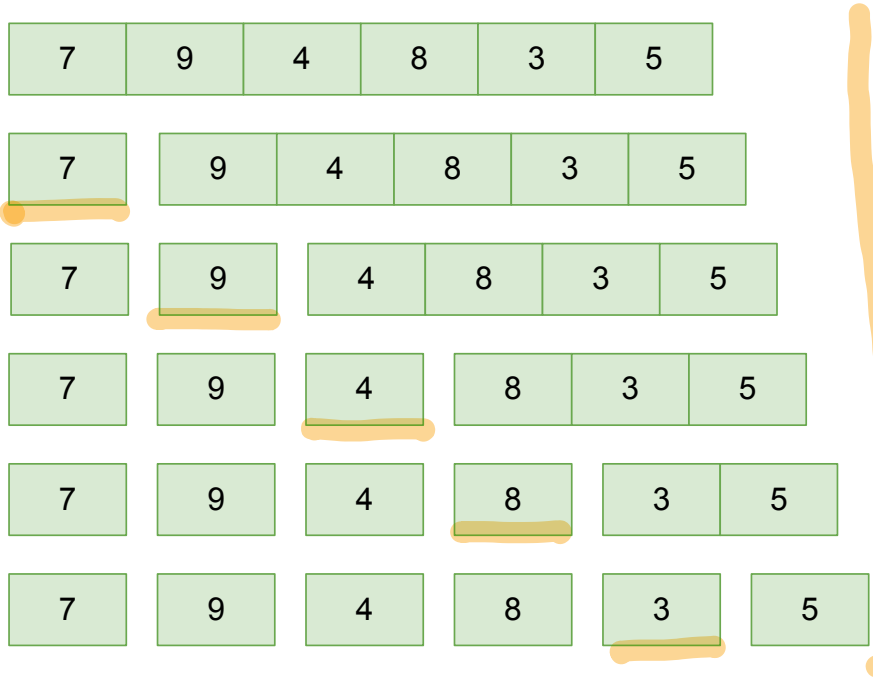
Insertion Sort NOT A GREAT ALGO

Recursive Case: → Splitting + merging

- **Split** the array such that first entry into first subarray
- Remaining into second subarray
- Recursively **sort** each subarray
- **Merge** sorted subarrays to build larger subarrays

Insertion Sort

Insertion Sort



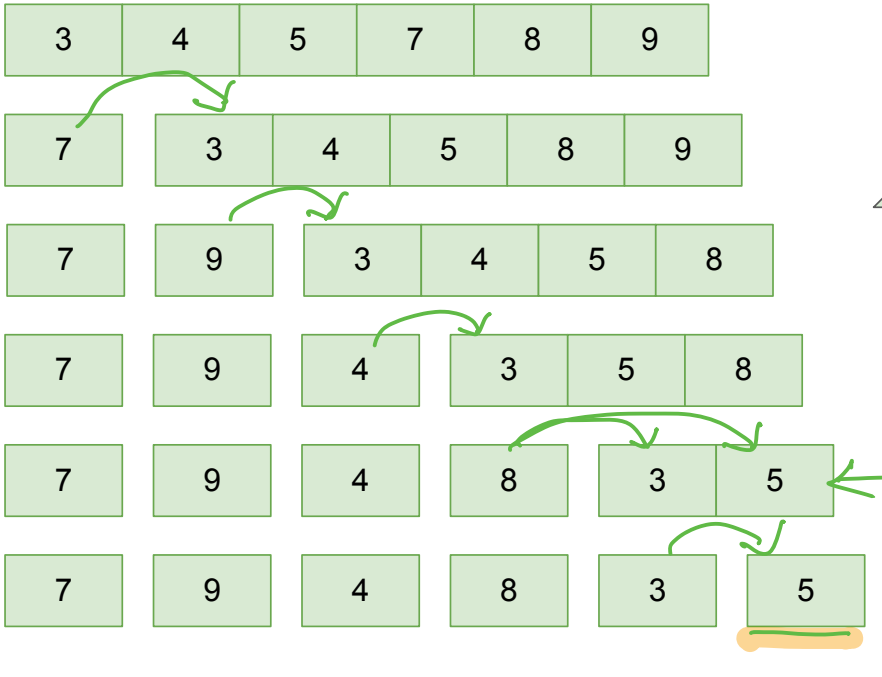
Recursive Case:

- **Split** the array such that
  - first entry into first subarray
  - Remaining into second subarray
- Recursively **sort** each subarray
- **Merge** sorted subarrays to build larger subarrays

Base case

⇒ Splitting

Insertion Sort



Recursive Case:

- **Split** the array such that
  - first entry into first subarray
  - Remaining into second subarray
- Recursively **sort** each subarray
- **Merge** sorted subarrays to build larger subarrays

merge [3|5] Insert 8

[5] is already sorted

Insert 3 into already

sorted array

Complexity of Insertion Sort

Splitting or merging (more work)

max no. of comparisons at each level ??

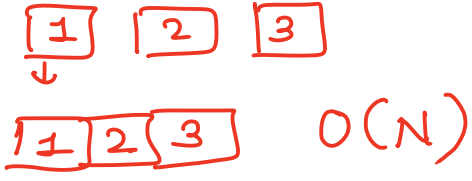
N (less than N)

∵ least upper bound

How many levels ? N

$O(N * N) = O(N^2)$

Best case

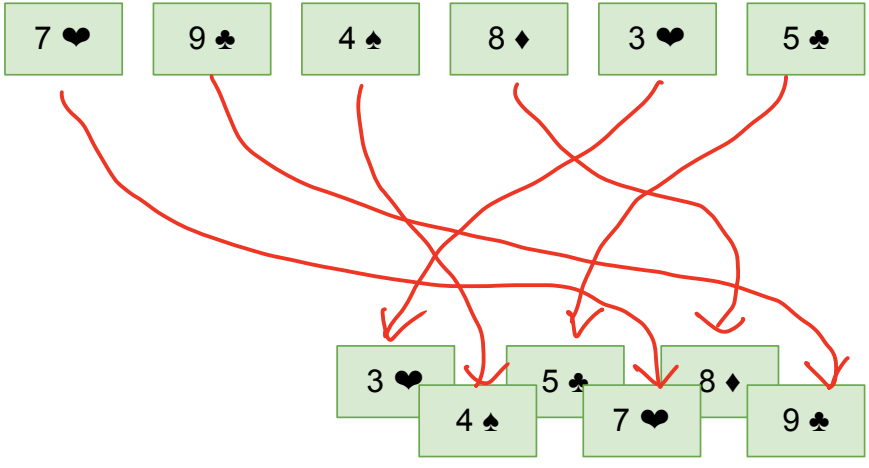


O(N)

Insertion sort intuitive idea

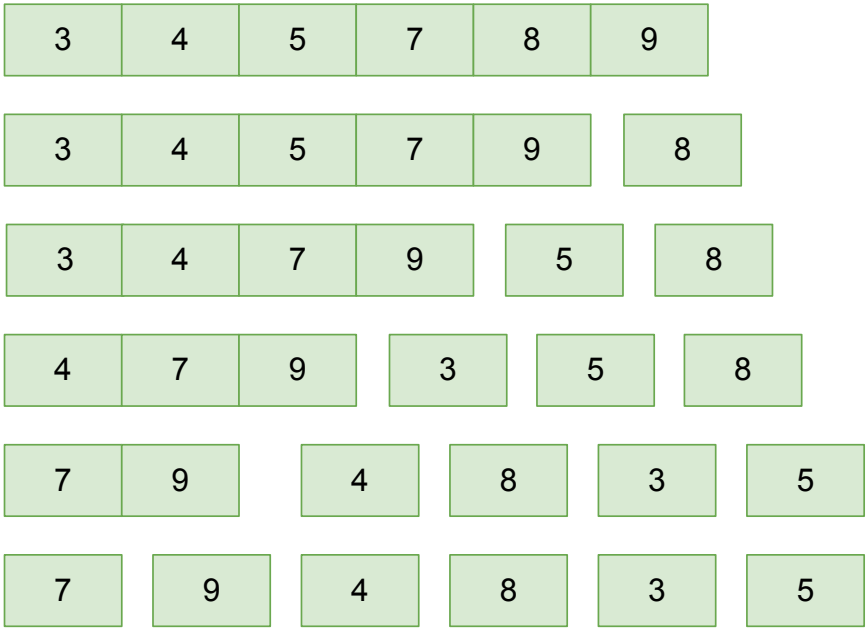
why do we still study it?
Easy to understand

Sorting set of cards in our hand



⇐ No.s you are dealt with

Insertion Sort



(In place algorithm)

Insertion Sort Pseudocode

⇒ IS(A, 1) ⇒ Go to st (3)

key = A[1] // 9

pos = 0 // A[pos] = 7

A[0+1] = 9

7	9

Solution Steps

- Base Case:
  - If array size is 1 or smaller, return.
- Recursive case:
  - Recursively sort first n-1 elements.
  - Insert the last element at its correct position in the sorted array.

IS(A, 4)
IS(A, 3)
IS(A, 2)
IS(A, 1)

name ↓ size ↓

```
InsertionSort(int arr [], int n)
    {
        // Base case
(1)     if (n <= 1)
(2)         return
        // Sort first n-1 elements
        InsertionSort( arr, n-1 )
(3)     int key = arr[n-1]
(4)     int pos = n-2
(5)     while (pos >= 0 && arr[pos] > key)
        {
(6)         arr[pos+1] = arr[pos]
(7)         pos = pos - 1
        }
(8)     arr[pos+1] = key
    }
```

control

progressively small

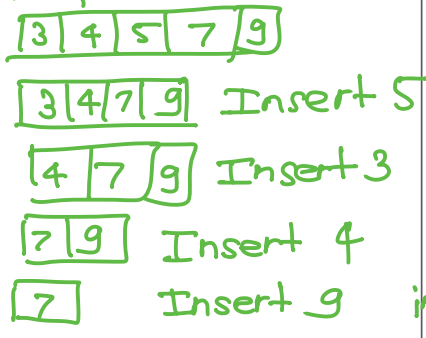
7 > 9

value is inserted into sorted array

n − 1 = 1

Example - splitting the array

```
arr = [ 7, 9, 4, 8, 3, 5 ], size = 6
insertion-sort(arr, 6)
insertion-sort(arr, 5)
insertion-sort(arr, 4)
insertion-sort(arr, 3)
insertion-sort(arr, 2)
insertion-sort(arr, 1)
```




```
InsertionSort(int arr [], int n)
        {
           // Base case
           if (n <= 1)
                return
           // Sort first n-1 elements
           InsertionSort( arr, n-1 )
           int key = arr[n-1]
           int pos = n-2
           while (pos >= 0 && arr[pos] > key)
           {
                arr[pos+1] = arr[pos]
                pos = pos - 1
           }
           arr[pos+1] = key
        }
main()
{
   InsertionSort(arr = [ 7, 9, 4, 8, 3, 5 ], 6)
}
```

for insertion

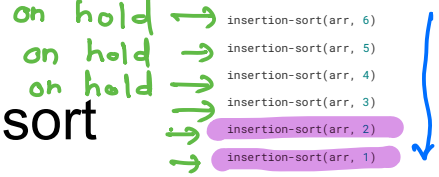
IS(A, n) ⇒ (n-1) sorted elements. Enter the nth element in sorted array

Example:

```
arr = [ 7, 9, 4, 8, 3, 5 ], size = 6
insertion-sort(arr, 1)
    insert 9
    return arr = [7, 9]
insertion-sort(arr, 2)
    insert 4
    return arr = [4, 7, 9]
insertion-sort(arr, 3)
    insert 8
    return arr = [4, 7, 8, 9]
insertion-sort(arr, 4)
    insert 3
    return arr = [3, 4, 7, 8, 9]
insertion-sort(arr, 5)
    insert 5
    return arr = [3, 4, 5, 7, 8, 9]
insertion-sort(arr, 6)
```

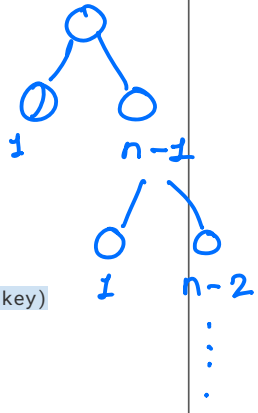
```
InsertionSort(int arr [], int n)
      {
         // Base case
          if (n <= 1)
              return
          // Sort first n-1 elements
          InsertionSort( arr, n-1 )
          int key = arr[n-1]
          int pos = n-2
          while (pos >= 0 && arr[pos] > key)
          {
              arr[pos+1] = arr[pos]
              pos = pos - 1
          }
          arr[pos+1] = key
      }
main()
{
   InsertionSort(arr = [ 7, 9, 4, 8, 3, 5 ], 6)
}
```

Insertion sort



function call stack

```
InsertionSort(int arr [], int n)
{
    // Base case
    if (n <= 1)
        return
    // Sort first n-1 elements
    InsertionSort( arr, n-1 )
    int key = arr[n-1]
    int pos = n-2
    while (pos >= 0 && arr[pos] > key)
    {
        arr[pos+1] = arr[pos]
        pos = pos - 1
    }
    arr[pos+1] = key
}
main()
{
   InsertionSort(arr = [ 7, 9, 4, 8, 3, 5 ], 6)
}
```




```
arr = [ 7, 9, 4, 8, 3, 5 ], size = 6

insertion-sort(arr, 1)
      key = arr[1] = 9
      pos = 0
      while (0 >= 0 && arr[0] > 9)
      {

      }
      arr[0+1] =  9
      arr = [7, 9]

insertion-sort(arr, 2)
      key = arr[2] = 4
      pos = 1
      while (pos >= 0 && arr[pos] > 4)
      {
          arr[pos+1] = arr[pos]
          pos = pos - 1
      }
      arr[-1+1] =  4
      arr = [4, 7, 9]
```

← Trying to place key into the sorted array

Advantages of IS ⇒ size of list to be sorted is small, runs faster

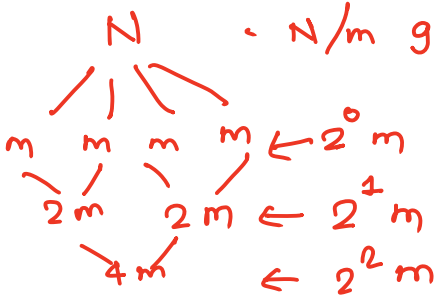
Takes O(N) time when elements are already sorted

In-place algo

Advantages of Mergesort ⇒ division of main prob. into subproblems, no major cost

Quick Sort

Combine IS + MS ⇒ $O(N(m + \log N/m))$



N/m groups of m

m is small, ⇒ IS

Divide

best case $O(m) * \frac{N}{m} = O(N)$

worst $O(m^2) * \frac{N}{m} = O(Nm)$

Merge

to step $2^i * m = N$

$i = \log\left(\frac{N}{m}\right)$

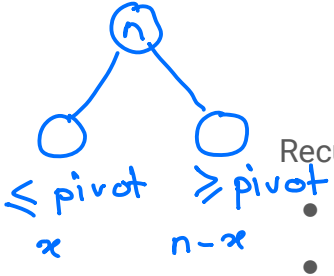
Quicksort

nearly sorted

size m $m < N$

Cost of merging = No. iterations * cost of it

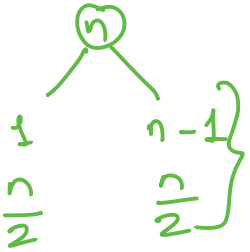
$= \left(N * \log \frac{N}{m}\right)$



7	9	4	8	3	5

Recursive Case:

- **Pivot** is chosen at random
- Two subarrays are created:
  - One subarray with all entries ≤ **Pivot**
  - Another subarray with entries > **Pivot**
- Recursively **sort** each subarray until the base case is reached
- **Merge** sorted subarrays to build larger subarrays



depending on pivot

divide + merging

$O(N) + O(N * \log \frac{N}{m})$

$O(Nm) + O(N * \log \frac{N}{m})$

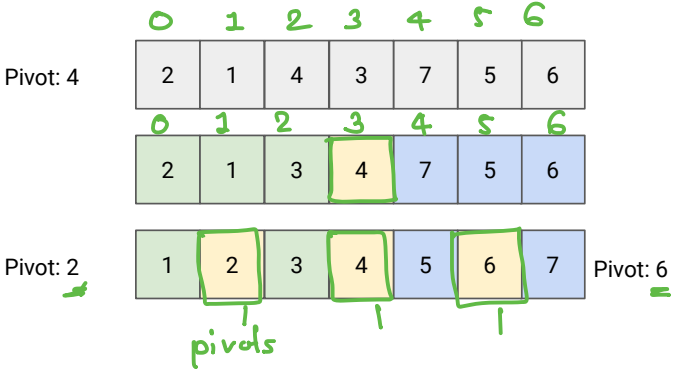
Quicksort

- Idea of Quicksort
  - Partition array A into A[1..q − 1], A[q], and A[q + 1..n] such that
    - Each element in A[1..q − 1] is ≤ A[q].
    - Each element in A[q + 1..n] is > A[q].
  - The element A[q] is called **pivot**.
  - Recursively sort (in place) each subarray.
- Average complexity:
  - $O(n \log n)$
- Worst case complexity:
  - $O(n^2)$

In-place algo.

Quicksort Example

2 functions



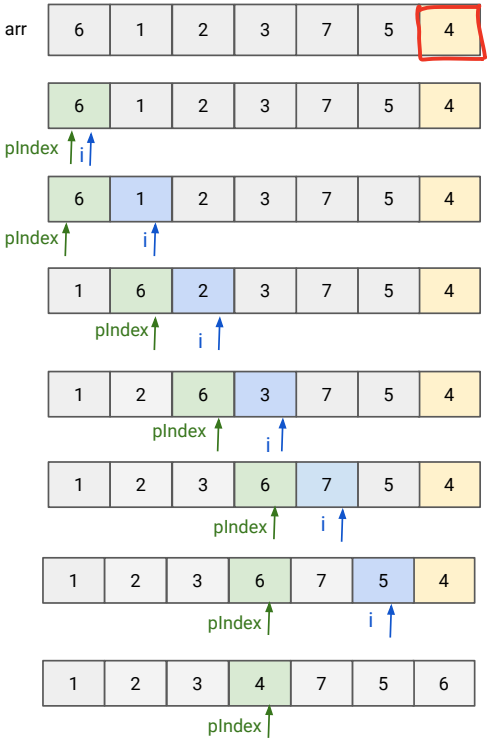
Final result:

1	2	3	4	5	6	7

```
quickSort(arr[], start, end)
{
    if (start < end)
    {
        /* pIndex partitioning index, arr[pIndex] is now at right place */
        pIndex = partition(arr, start, end);

        quickSort(arr, start, pIndex - 1);  // Before pIndex
        quickSort(arr, pIndex + 1, end); // After pIndex
    }
}
```

base condition

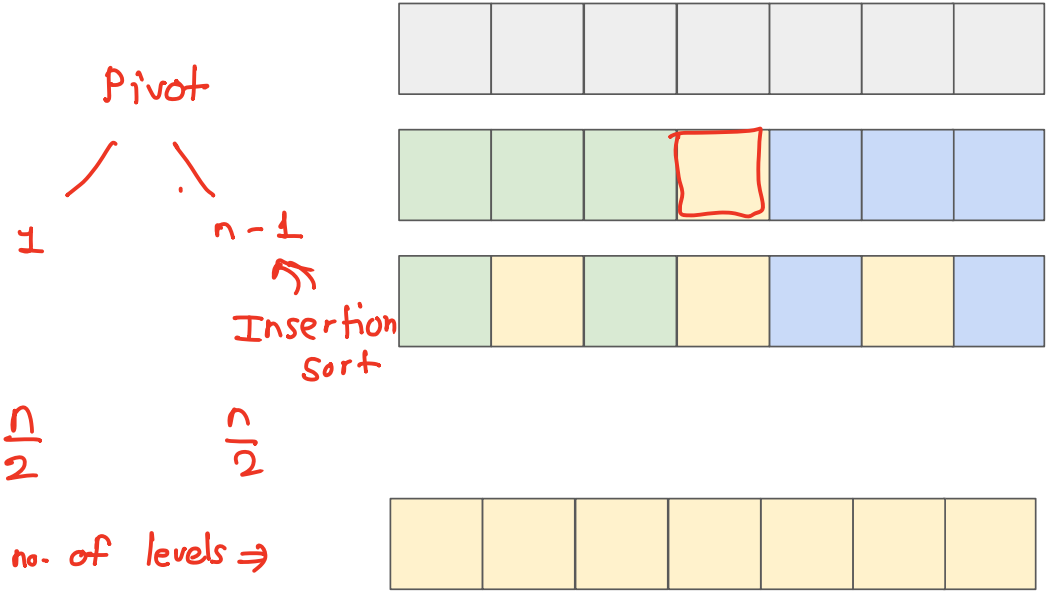
Quicksort Example



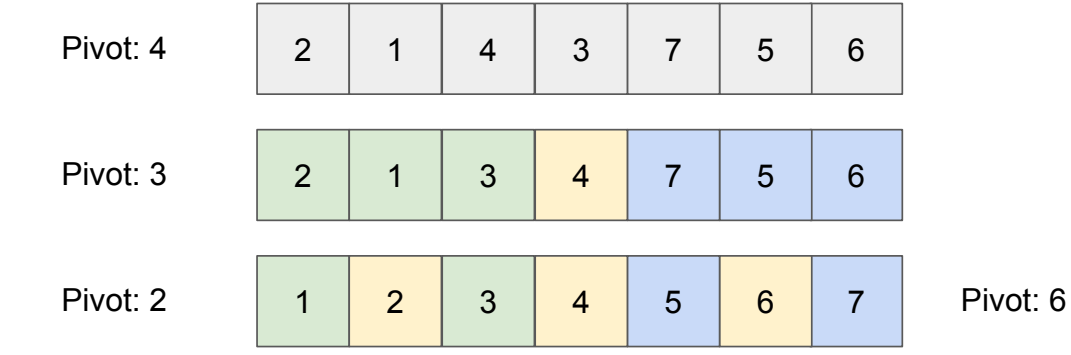

```
quickSort(arr[], start, end)
{
    if (start < end)
    {
        /* pIndex partitioning index, arr[pIndex] is now at right place */
        pIndex = partition(arr, start, end);

        quickSort(arr, start, pIndex - 1);  // Before pIndex
        quickSort(arr, pIndex + 1, end); // After pIndex
    }
}
```

```
partition (arr[], start, end)
{
    pivot = arr[end]
    pIndex = start
    for (i = start; i < end - 1; i++)
    {
        // If current element is smaller than the pivot
        if (arr[i] < pivot)
        {
            swap arr[pIndex] and arr[i]
            pIndex ++
        }
    }
    swap arr[pIndex] and pivot
    return pIndex
}
```

Quicksort Example - Best Case ⇒ Pivot always ends up in the middle of the array



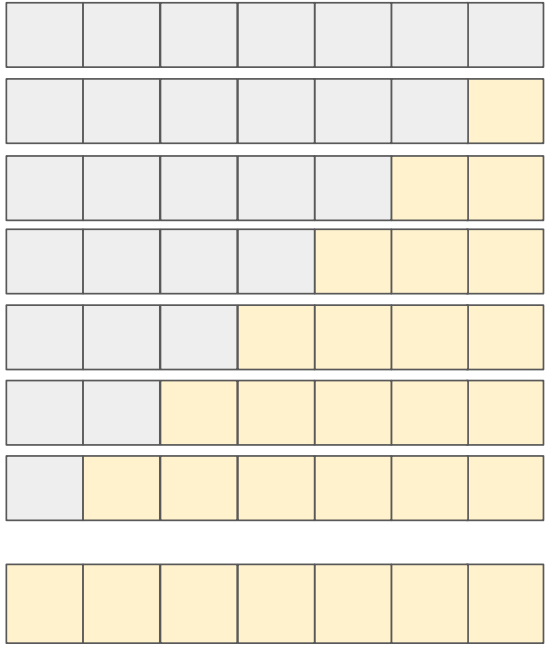
Quicksort Example - Best Case



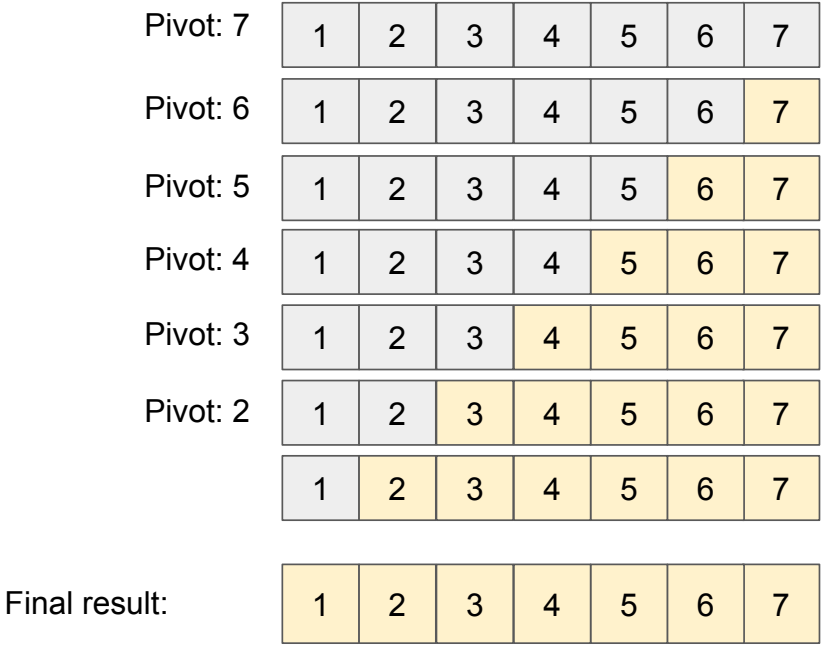
Quicksort Example - Worst Case

Best case

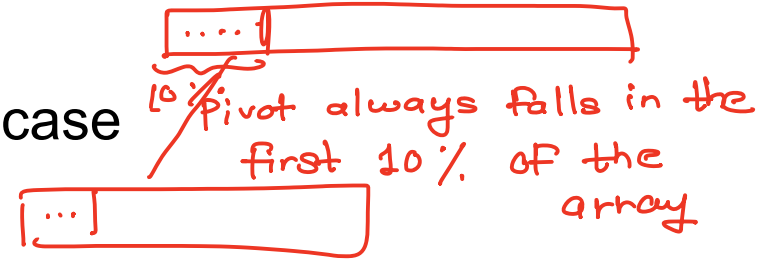
array is sorted in ascending or descending order

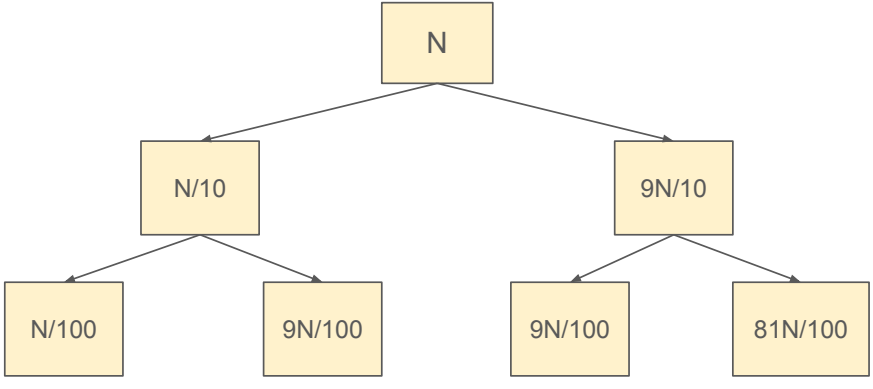


Quicksort Example - Worst Case



Quicksort Example - Average case





- Complexity: O(N * H)
- Height
  - $\log_{10/9} N$
- Complexity on Average:
  - O(N log N)

Randomization of Pivot

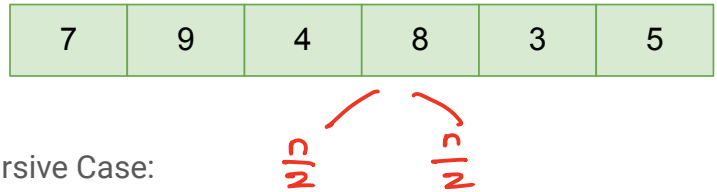
Randomization of array

Merge Sort

Merge Sort

Not In-Place algo

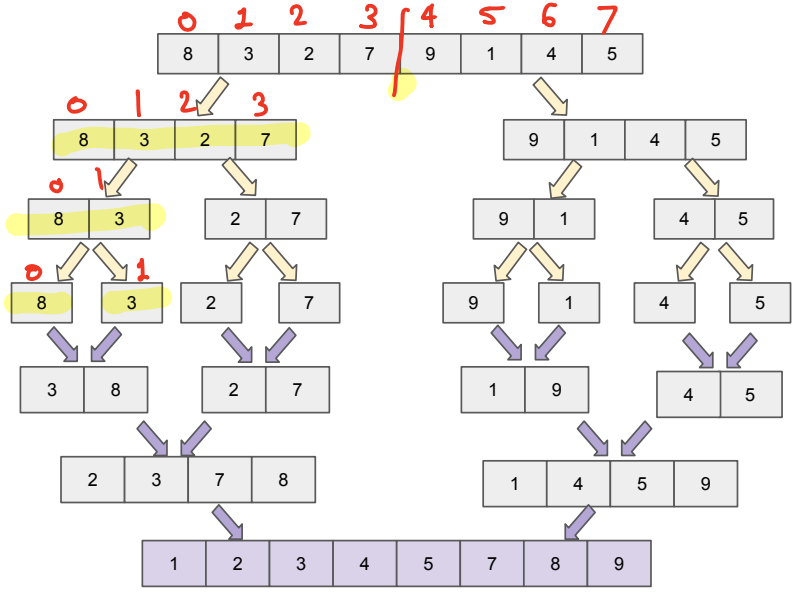
Requires extra space



Recursive Case:

- **Split** the input array such that
- Each subarray has half entries of the original
- Recursively **sort** each subarray until the base case is reached
- **Merge** sorted subarrays to build larger subarrays

Pseudocode



mid = 3 MS(A, 0, 7) on hold
mid = 1 MS(A, 0, 3) on hold
mid = 0 MS(A, 0, 1) on hold (ongoing)
MS(A, 0, 0) done!
MS(A, 1, 1) done!
merge(A, 0, 0, 1)
x
done!

indices

```
void mergeSort(int arr[], int left, int right)
{                                   0       1
      //Base case                   0       0
      if left ≥ right
        return
      //Recursive case
      mid = (left+right)/2
      // Sort first and second halves
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
                          1       1

      merge(arr, left, mid, right);
}
```

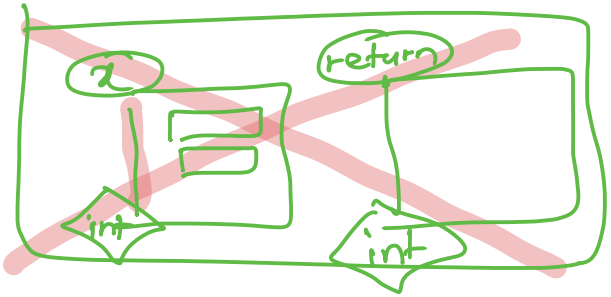
sorting is done in merge function

Performance

- Merge Sort:
  - Best case: O(N log N)
  - Worst case: O(N log N)
- Quick Sort:
  - Best case: O(N log N)
  - Worst case: $O(N^2)$
  - Randomly chosen array size: (Average Case) O(N log N)
- Which one is faster?
  - Quicksort
  - Requires empirical analysis.

Implementation considerations for recursive methods

Memory Model



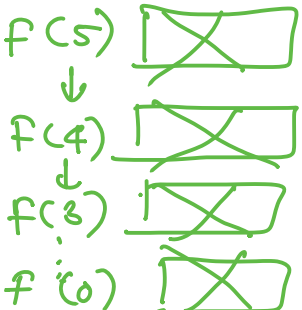
- Memory model of functions:
  - Non-recursive functions: Calling a function
    - Reserves space for function's parameters, variables, and return value
    - Release the space after the return value is obtained
  - Recursive functions: Calling function
    - Reserves space for function's parameters, variables, and return value
    - Release the space after each recursive call is completed

f() {

f()

}

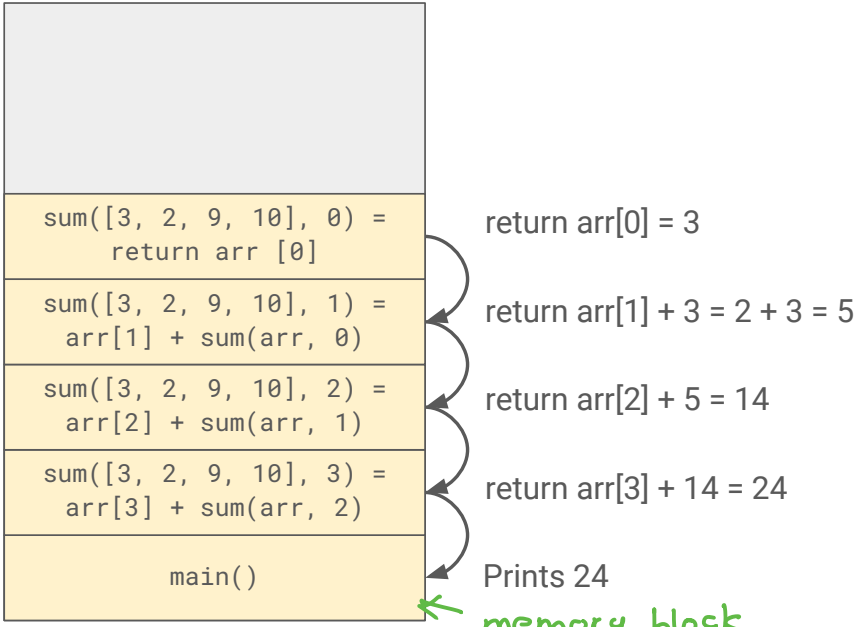
6 blocks



memory requirement ⇒ 50 blocks

Example

function call stack



Function call stack

memory block or stack frame

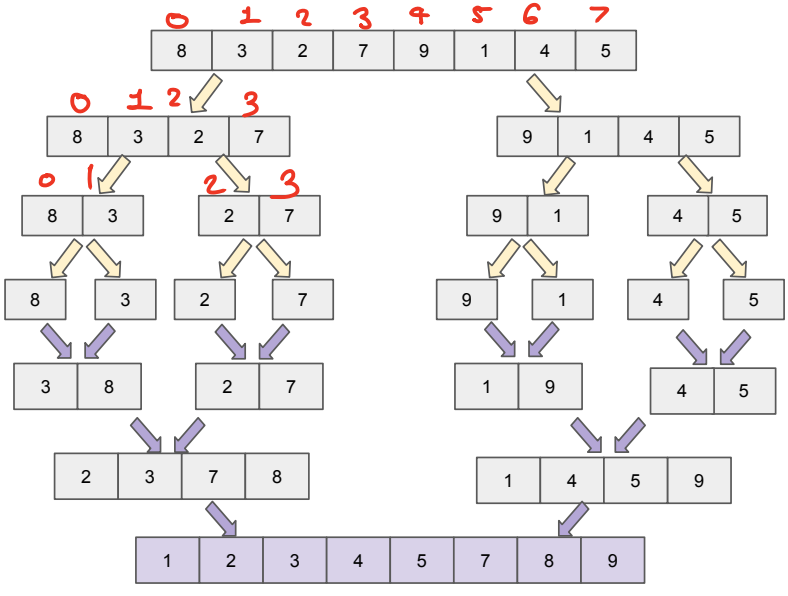
Stack frame

Stack ⇒ data structure

```
void sum(int arr[], int n)
{
      if (n == 0)
            return arr[0];
      else
            return arr[n] + sum(arr, n-1);
}

int main()
{                    0  1  2  3
      int arr = [3, 2, 9, 10];
      printf("%d", sum(arr, 3));
}
```

Index of last position

Call Stack for Merge Sort




```
void mergeSort(int arr[], int left, int right)
{
      //Base case
      if left > right
        return
      //Recursive case
      mid = (left+right)/2
      // Sort first and second halves
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);

      merge(arr, left, mid, right);
}
```

If 0 > 0 → No, so control returns to next call to mergeSort

Call mergeSort ([3], 1, 1)

If 1 > 1 → No, so control returns to next call to mergeSort

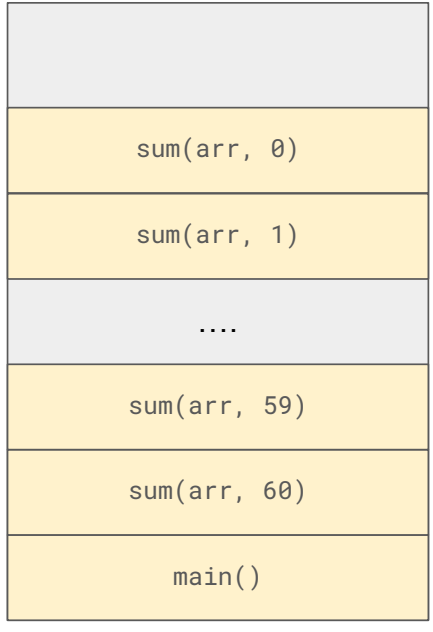
Call merge ([8, 3], 0, 0, 1)

merge([8, 3], 0, 0, 1) returns sorted array arr [3, 8]

`merge([2, 7], 2, 2, 3)`
`merge([8, 3, 2, 7], 0, 1, 3)`
`mergeSort([8, 3, 2, 7], 0, 3)`
`mergeSort([8, 3, 2, 7, 9, 1, 4, 5], 0, 7)`

Things to consider

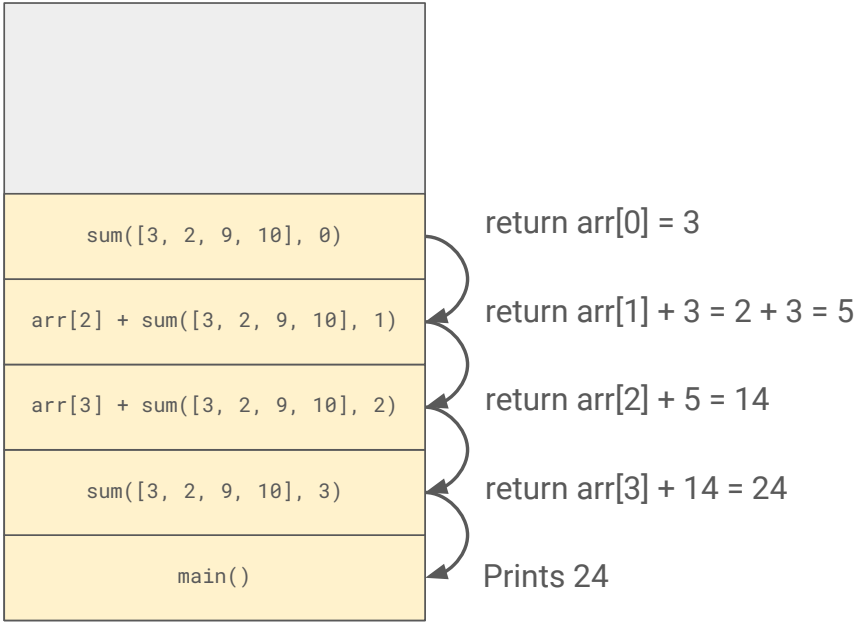
- Recursion call stack grows upwards
- Depending on the input size, call stacks with individual stack frames for each recursive call can grow large!
- Code can run out of stack space
- Program can terminate abruptly!
- Solution:
  - **Tail-recursive optimization**
- What is tail recursion?
  - A tail recursive function is a function where recursive call is the last thing executed by the function.



Function call stack

Non-tail recursive

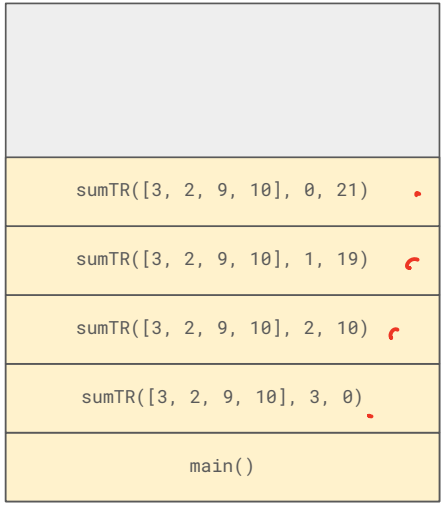
~~Tail recursive~~



Function call stack

```
void sum(int arr[], int n)
{
      if (n == 0)
            return arr[0];
      else
            return arr[n] + sum(arr, n-1);
}

int main()
{
      int arr = [3, 2, 9, 10];
      printf("%d", sum(arr, 3));
}
```

Tail Recursive Optimization



Function call stack

part_sum = 21 + 3 = 24

part_sum = 19 + 2 = 21

part_sum = 10 + 9 = 19

part_sum = 0 + 10 = 10

A tail recursive function is a function where recursive call is the last thing executed by the function.

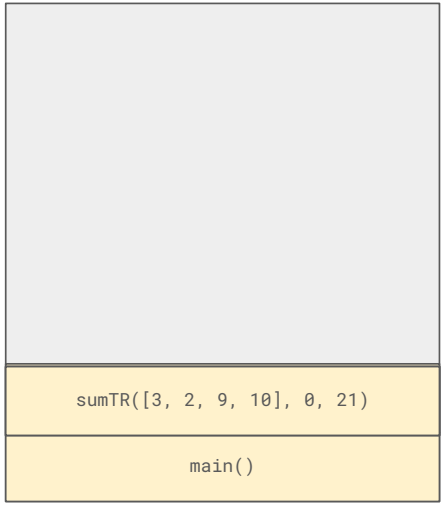
acts like an accumulator

```
void sumTR(int arr[], int n, part_sum)
{
      if (n == 0)
            return part_sum + arr[0];
      else
            return sum(arr, n-1, part_sum+arr[n]);
}

int main()
{
      int arr = [3, 2, 9, 10];
      printf("%d", sumTR(arr, 3, 0));
}
```

Compilers recognize tail-recursive functions and optimize use of stack frames

Tail Recursive Optimization

A tail recursive function is a function where recursive call is the last thing executed by the function.



Function call stack

part_sum = 0+10 = 10

part_sum = 10+9 = 19

part_sum = 19+2 = 21

```
void sumTR(int arr[], int n, part_sum)
{
      if (n == 0)
            return part_sum + arr[0];
      else
            return sumTR(arr, n-1, part_sum+arr[n]);
}

int main()
{
      int arr = [3, 2, 9, 10];
      printf("%d", sumTR(arr, 3, 0));
}
```

base case
n=0

recursive case

n * fact(n-1)

fact(n)
= n * fact(n - 1)

Factorial

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

$$1! = 1$$
$$2! = 2 \times 1 = 2$$
$$3! = 3 \times 2 \times 1 = 6$$
$$4! = 4 \times 3 \times 2 \times 1 = 24$$
$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Iteration vs Recursion vs Tail recursion

Iteration

Every recursive program can be modeled into iterative program

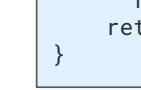
Recursion

- Advantage:
  - More elegant
  - Shorter code
  - Intuitive approach
- Disadvantage:
  - Larger problems can occupy more space
  - Function call stack can grow large

Tail Recursion

Better performance since stack space is saved by the compiler

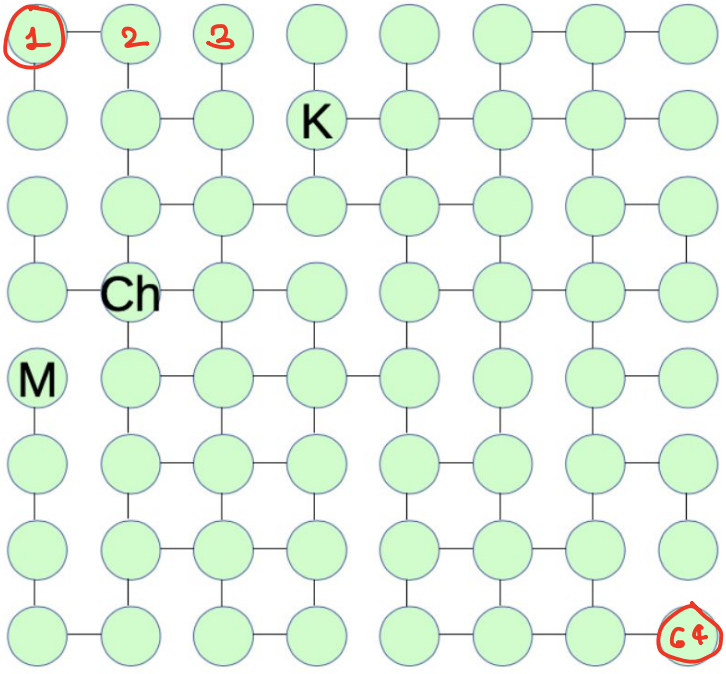
```
int factorial(int n)
{
    int fact = 1, i;
    for (i = 2; i <= n; i++)
      fact *= i;
    return fact;
}
```




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

```
int factTR(int n, int part_fact)
{
    if (n == 1)
      return part_fact;
    return factTR(n-1, n * part_fact);
}
```

Back to Graphs

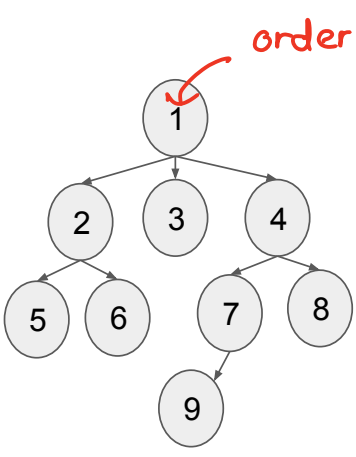


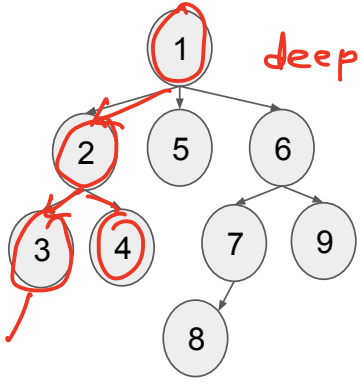
- Represent with Adjacency Matrix:
  - Size 64 × 64

- Problem: Find a path, which consists of a sequence of connected nodes, leading from the start node to the destination.

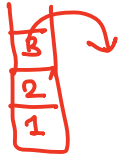
- Base Case:
  - Reached destination node
  - No nodes left
- Recursive Case:
  - Finding the path from given node
  - Removing current node
  - Finding path from from each neighbour of the current node to the destination
  - Once destination is identified, Attach current node to start of path

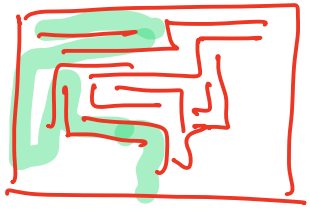
Graph Traversal





deep into the tree

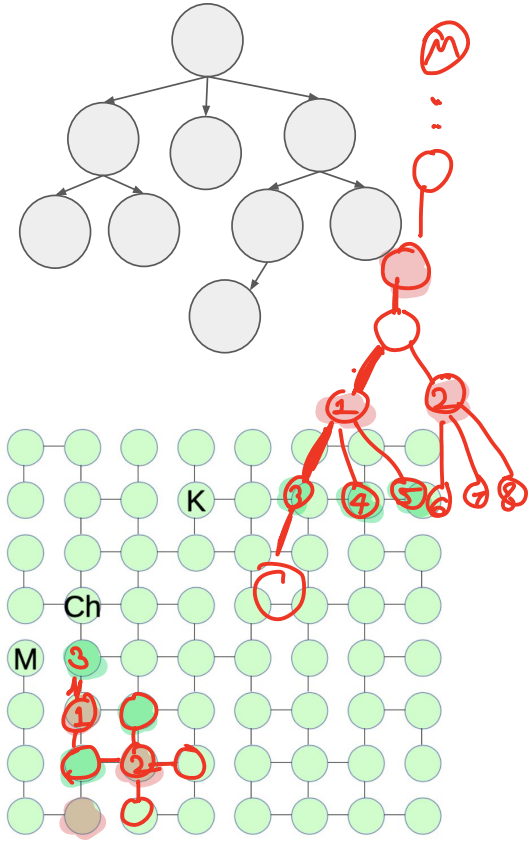




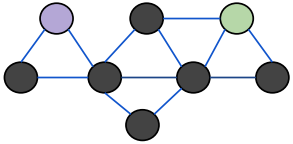
BFS : visit all neighbours

visit all neighbours

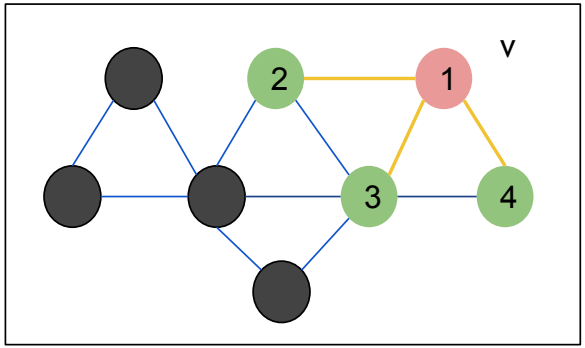
Given G

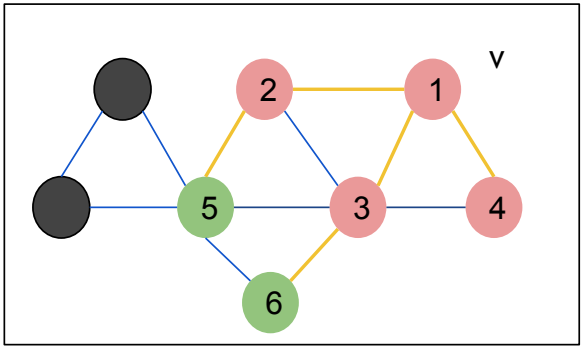


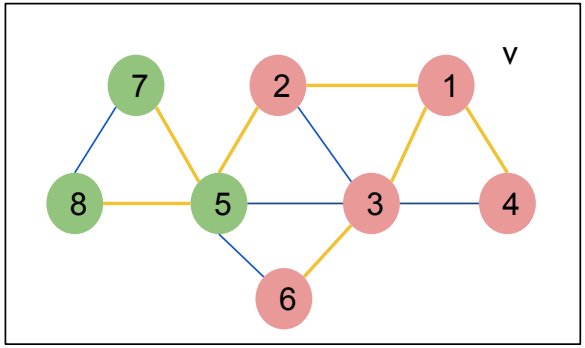
Graph Traversal
For search

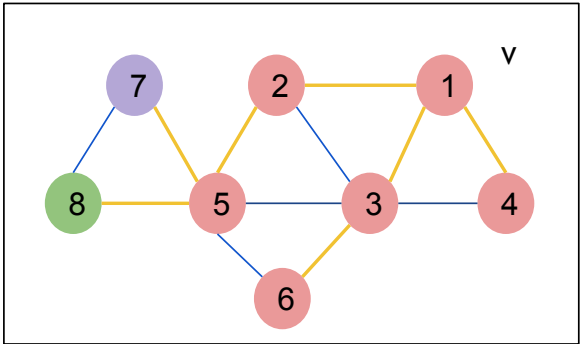


Breadth First Search

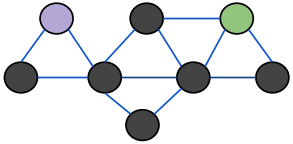




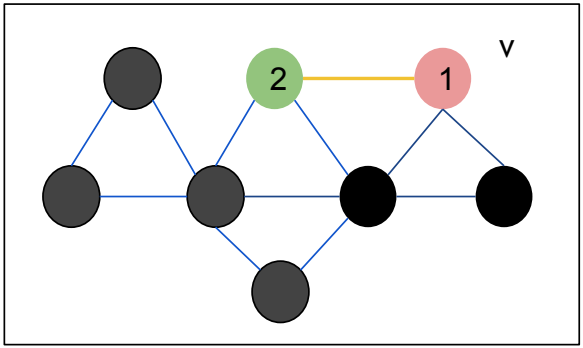


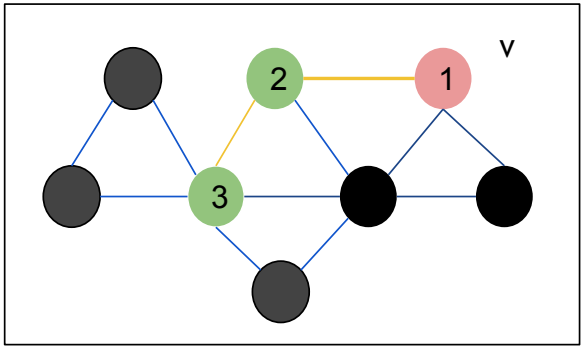


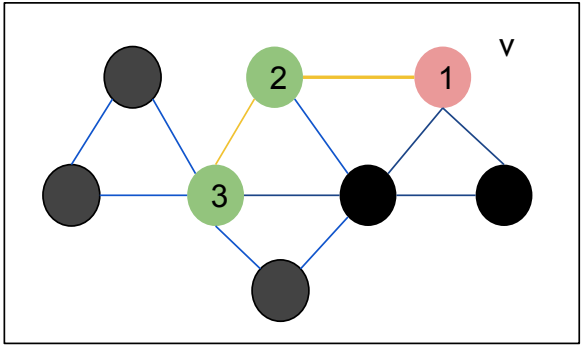
Graph Traversal for search

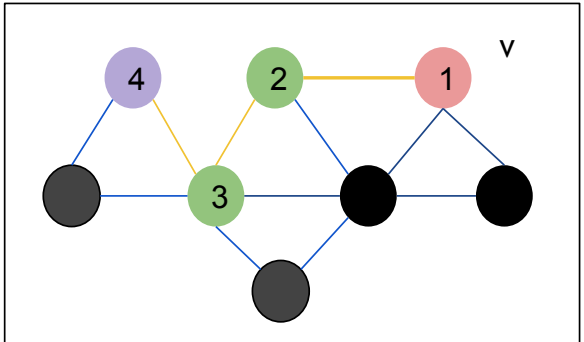


Depth First Search

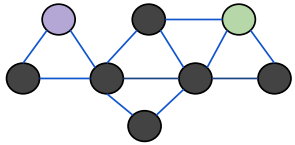






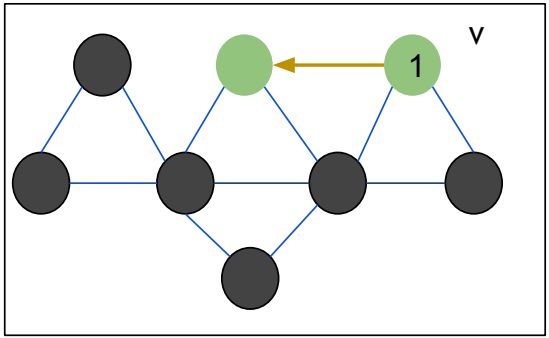


DFS - finding path

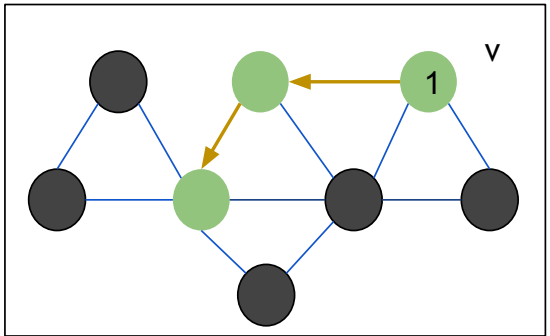


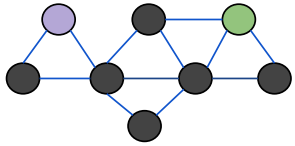
```
// Inputs: current node (its index in the graph)
// destination node (its index in the graph)
// Adjacency matrix (to figure out a node's neighbours)

DFS(current, destination, A)
    if current==destination
       return destination // We got to base case!
    else
       for each neighbour of current
         subpath = DFS(neighbour, destination, A)
         if subpath found
            // prepend current to subpath to build full path
            return {current -> subpath}
       return (nil) // Second base case, no subpath from current
```

1



2

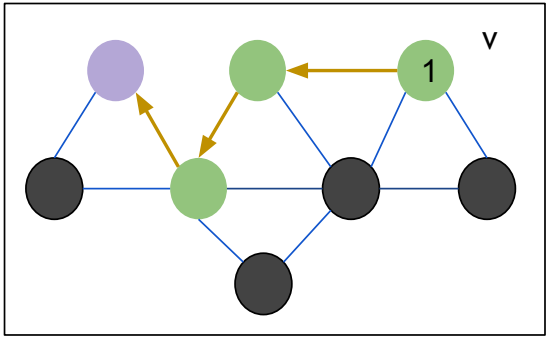




DFS - finding path

```
// Inputs: current node (its index in the graph)
// destination node (its index in the graph)
// Adjacency matrix (to figure out a node's neighbours)

DFS(current, destination, A)
    if current==destination
       return destination // We got to base case!
    else
       for each neighbour of current
         subpath = DFS(neighbour, destination, A)
         if subpath found
            // prepend current to subpath to build full path
            return {current -> subpath}
       return (nil) // Second base case, no subpath from current
```

3



Tracing recursive calls

- Which recursive call is not doing the right thing?
- Add another variable (depth or level) for tracing the recursive calls at each level
- Function definition:
  - DFS(current, destination, Adj, depth)
- The first call to DFS will look like this:
  - DFS(15, 1, Adj, 0);
  - (Find a path from node 15 to node 1, with adjacency matrix Adj, initial recursion depth is 0).
- Within the DFS code, the recursive calls will look like this:
  - subpath=DFS(neighbour, destination, Adj, depth+1);
- Use either gdb or printf() to trace which recursion level these variables belong to

How does this help?

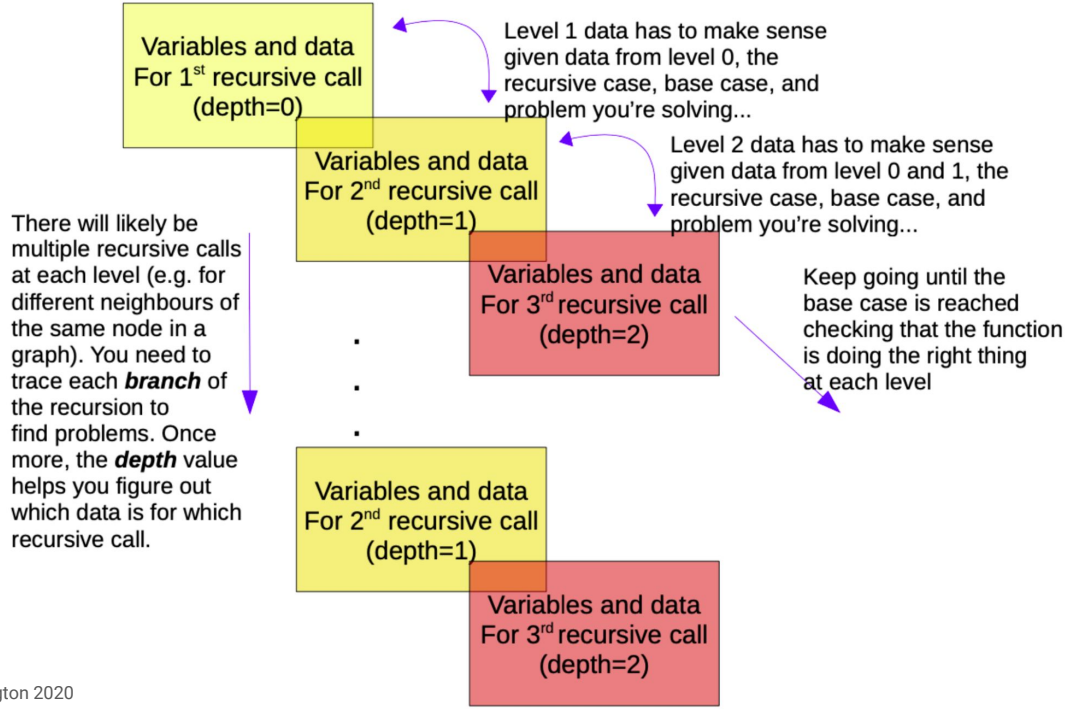
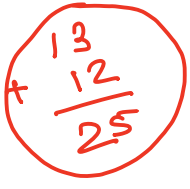


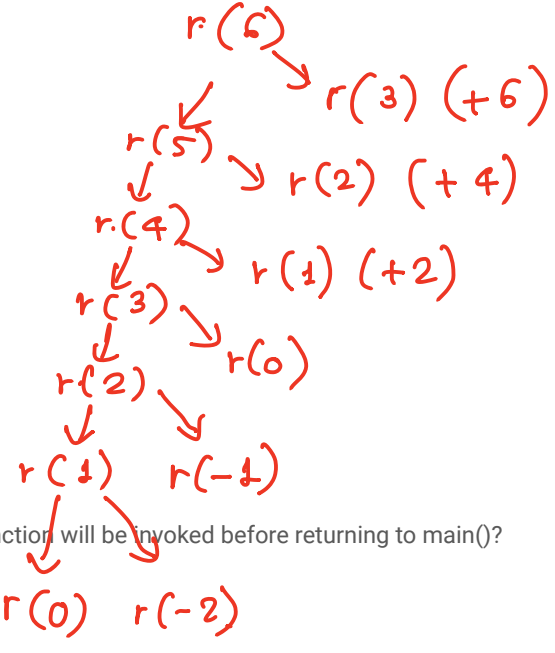
Image taken from Unit 5 notes

Recursion Practice Problem

Consider following recursive function:

```
void recur(int n)
{
  if(n<1)
      return;
  recur(n-1);
  recur(n-3);
}
```





If recur(6) is called from main() function, how many calls to recur() function will be invoked before returning to main()?

Recursion Practice Problem

Consider following recursive function:

```
void recur(int n)
{
  int i = 0;
  if(n>1)
      recur(n-1);
  for(i=0; i<n; i++)
      printf("*");
}
```

How many times * is printed for recur(n)?

Generalize o/p

n

n(n+1)/2

n*n

for n = 2

r(2)
r(1)
r(0)

* 1
* * 2

3

n(n+1)/2

n=3

r(3)
r(2)
r(1)
r(0)

* 1
* * 2
* * * 3

6

n= 4

* 1
* * 2
* * * 3
* * * * 4

10

n = 5

* 1
* * 2
* * * 3
* * * * 4
* * * * * 5

15

for n = k

1+2+3+4+ ... +k

- k(k+1)

2

Take Away from Unit 5

- Recursion is an intuitive approach
  - Utilize when problem can be broken down in self similar manner
  - Optimize stack space using tail recursion
- Common recursion errors
  - Never reaches base case
  - Runs out of stack space
  - Doesn’t return the correct solution
  - Avoid such errors with the help of auxiliary variables
- Recursion might look challenging, but more examples you practice, more comfortable you will get with the idea!