

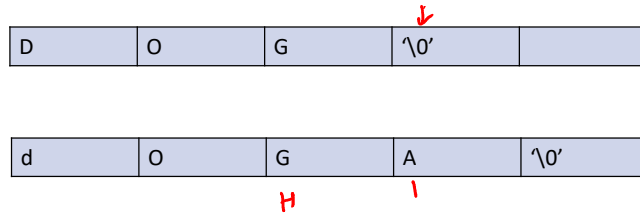
## Today's class

- Towers of Hanoi
- Selection sort
- Analysis of algorithms

# Homework

- Implement bsearch for strings
  - `int bsearch(char dict[][8], char word[], int lo, int hi)`
    - Initially
      - lo is the index of the first row = 0
      - hi is the index of the last row = number of rows - 1

## Comparing two strings



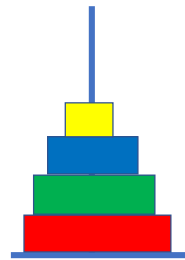
Ignore the case  
ASCII of A-Z = 65-90  
ASCII of a-z = 97-122

# Towers of Hanoi

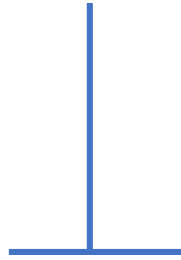
## Towers of Hanoi

- Three towers: source, middle (or temporary), destination
- $n$  discs of different sizes are stacked on the source tower
- The discs are ordered from largest (bottom) to smallest (top)
- The goal is to move all discs from source to destination
- Rules for movement
  - You can move only one disc at a time from one tower to another
  - A larger disc can't be placed over a smaller disc

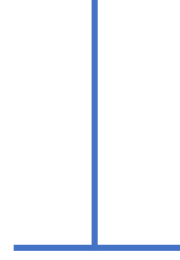
# Towers of Hanoi



TOWER-1

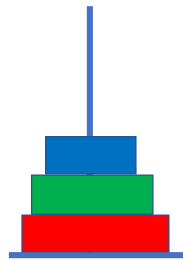


TOWER-2

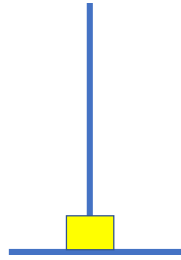


TOWER-3

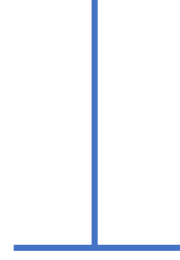
# Towers of Hanoi



TOWER-1



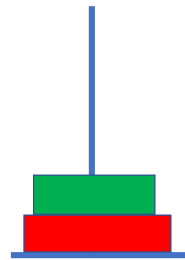
TOWER-2



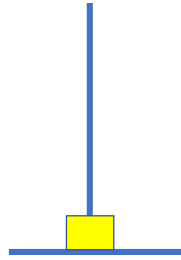
TOWER-3



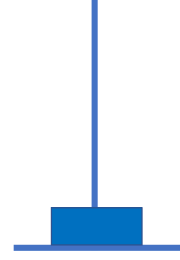
# Towers of Hanoi



TOWER-1

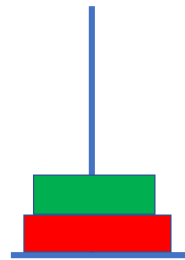


TOWER-2

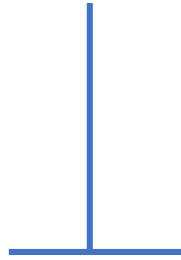


TOWER-3

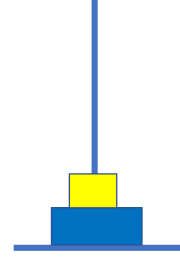
# Towers of Hanoi



TOWER-1

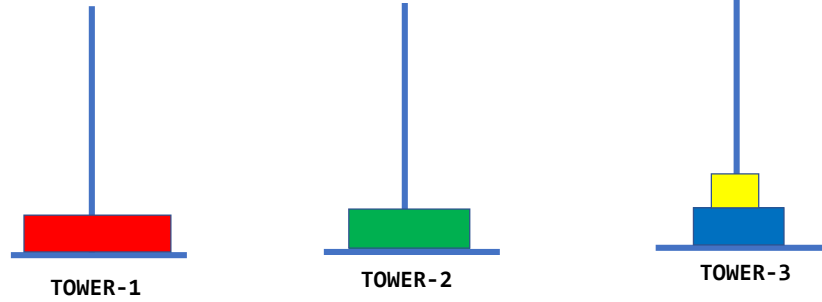


TOWER-2

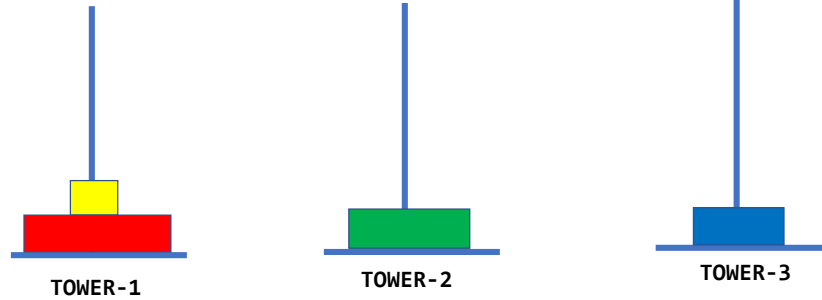


TOWER-3

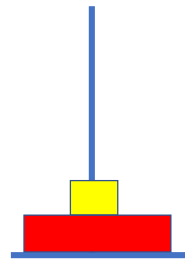
# Towers of Hanoi



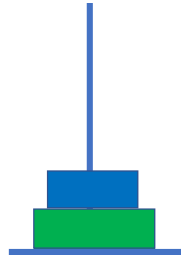
# Towers of Hanoi



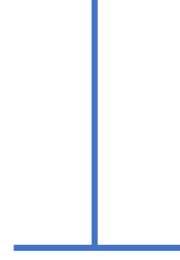
# Towers of Hanoi



TOWER-1

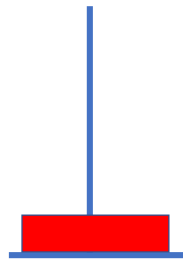


TOWER-2

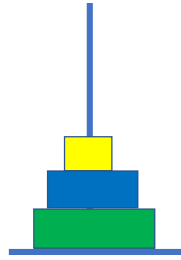


TOWER-3

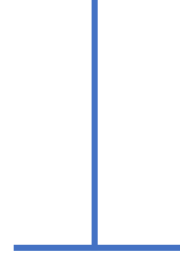
# Towers of Hanoi



TOWER-1

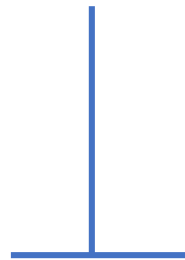


TOWER-2

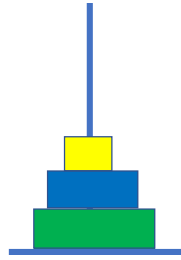


TOWER-3

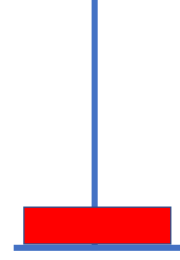
# Towers of Hanoi



TOWER-1

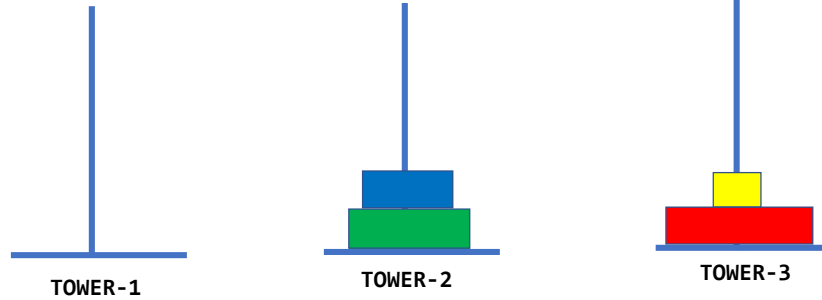


TOWER-2



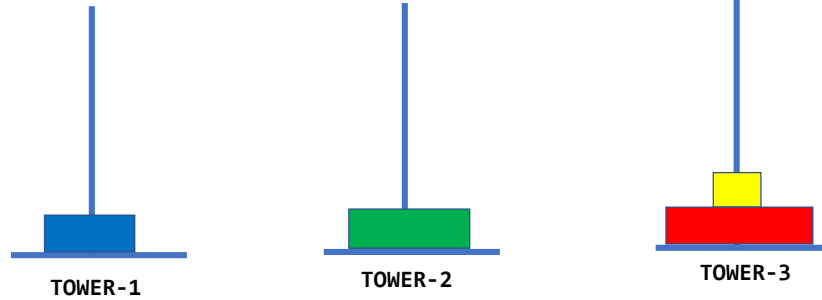
TOWER-3

# Towers of Hanoi

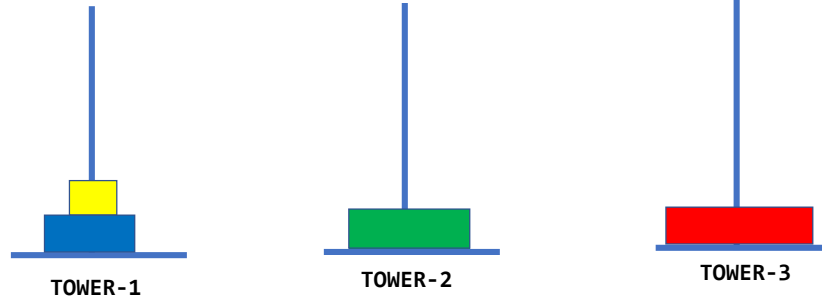




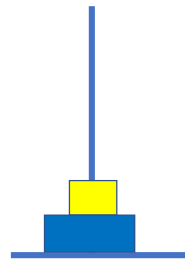
# Towers of Hanoi



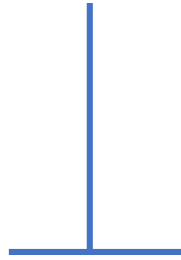
# Towers of Hanoi



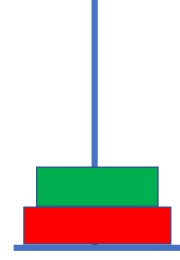
# Towers of Hanoi



TOWER-1

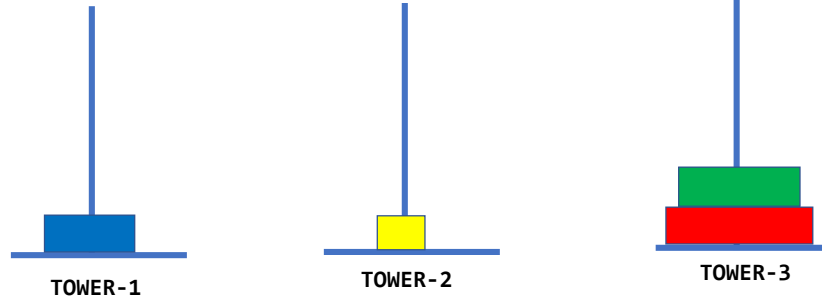


TOWER-2

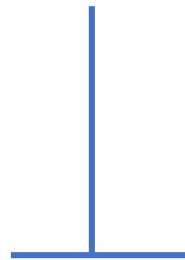


TOWER-3

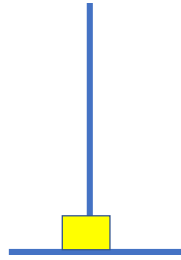
# Towers of Hanoi



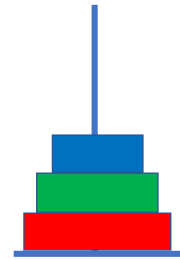
# Towers of Hanoi



TOWER-1

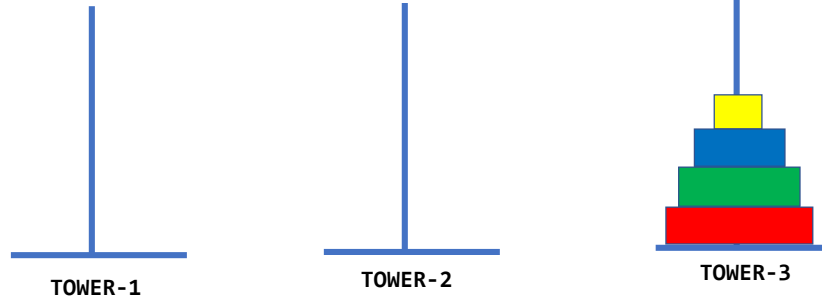


TOWER-2



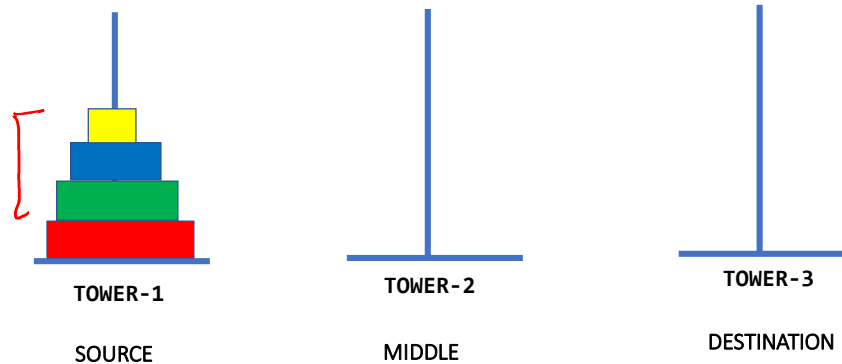
TOWER-3

# Towers of Hanoi



## Initial state

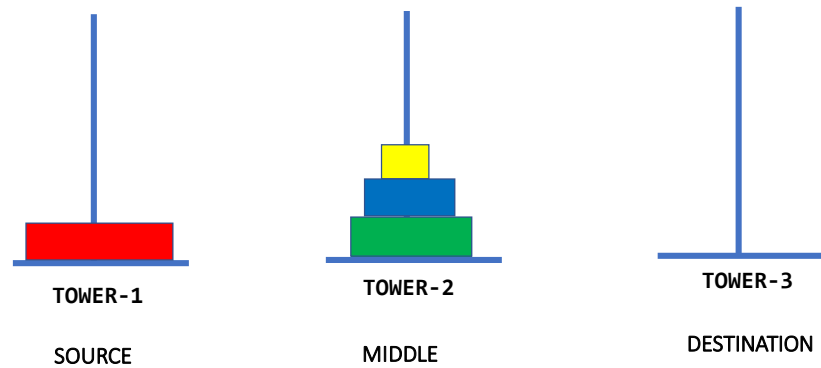
- Goal: move n discs from source to destination



The algorithm works as follows. Suppose there exists a move routine that takes four parameters: 1> the number of discs, 2> the name of the source tower, 3> the name of the destination tower, and 4> the name of the temporary tower. The move routine prints all the moves needed to move n discs from the source tower to the destination tower. Because move is a recursive procedure, we need to find a way to divide the original problem into subproblems and then combine the results of the subproblems to solve the actual problem. Because move works for an arbitrary value of n, we can first call the move routine to move top n-1 discs from the source tower to the temporary tower using the destination tower as temporary. This recursive call will print all the moves required to move top n-1 discs to the temporary tower from the source tower. The resulting state is shown on the next slide.

## Intermediate state

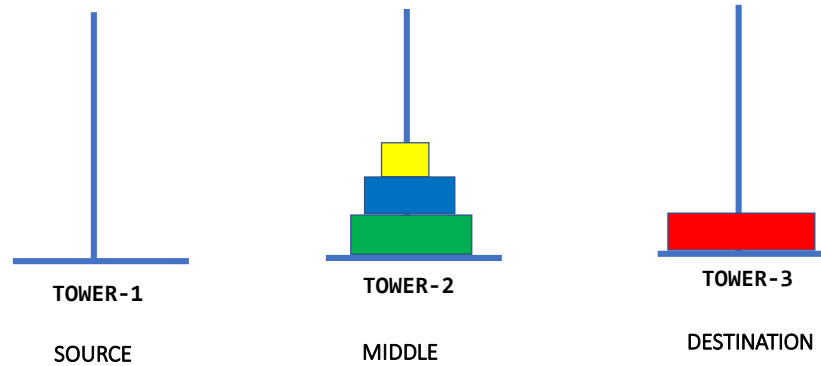
- After moving  $n-1$  discs from the source tower to the middle tower using the destination tower as temporary



After moving the top  $n-1$  discs to the temporary tower, we need to move the disc at the bottom of the source tower to the destination tower. The resulting state is shown on the next slide.



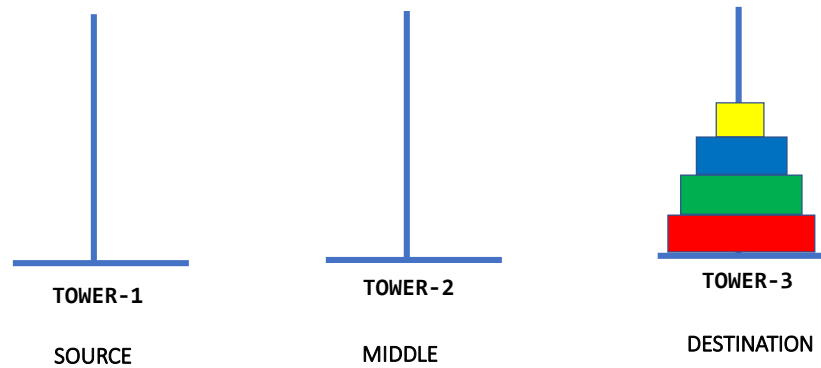
Move bottom disc from source to destination



After this step, we need to move  $n-1$  discs stacked at the temporary tower to the destination tower, using the source as the temporary tower. We can use the move procedure to do this task, which will print all the moves needed to move  $n-1$  discs from the temporary tower to the destination tower. The resulting state is shown on the next slide.

## Final state

- After moving  $n-1$  discs from the middle tower to the destination tower using the source tower as temporary



Our objective has been completed, as all the discs have been moved to the destination tower. No further action is required.

## Tower of Hanoi

```
1. void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
2.     if (n == 1) {
3.         printf("moving from %s to %s\n", src_t, dst_t);
4.     }
5.     else {
6.         move(n-1, src_t, tmp_t, dst_t);
7.         printf("moving from %s to %s\n", src_t, dst_t);
8.         move(n-1, tmp_t, dst_t, src_t);
9.     }
10.}

11.int main() {
12.    move(4, "Tower1", "Tower3", "Tower2");
13.    return 0;
14.}
```

This is the corresponding C code. Notice that the arguments of move at line-1 are 1> number of discs (n), 2> the name of the source tower (src\_t), 3> the name of the destination tower (dst\_t), and 4> the name of the temporary tower (tmp\_t). At line-6, we are calling the move routine to move top n-1 discs from the src tower to the temporary tower using the destination tower as temporary. Look at the arguments passed to move at line-6. At line-8, we are invoking the move routine again to move n-1 discs from the temporary tower to the destination tower using the source as the temporary tower.

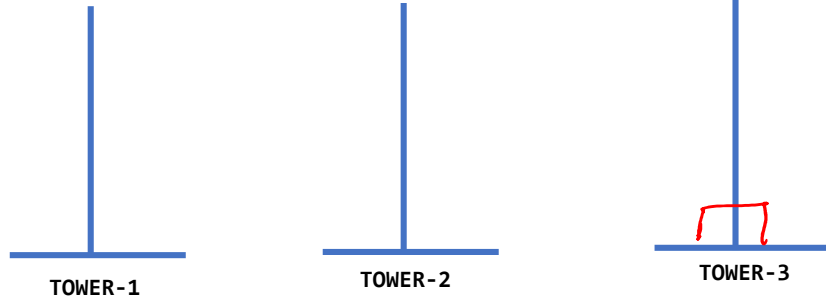
$$n = 1$$

# Towers of Hanoi

move(1, "T1", "T3", "T2")

*moving from T1 to T3*

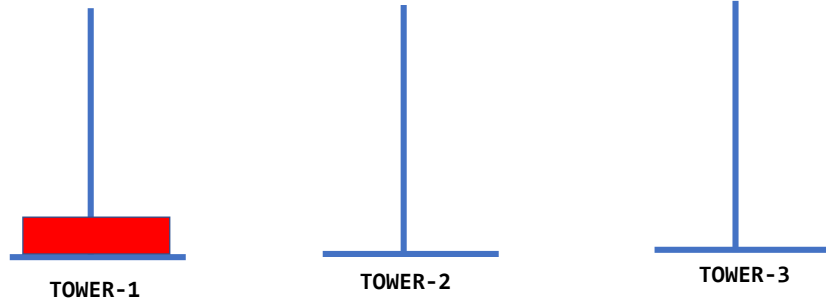
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
    if (n == 1) {  
        printf("moving from %s to %s\n", src_t, dst_t);  
    }  
    else {  
        move(n-1, src_t, tmp_t, dst_t);  
        printf("moving from %s to %s\n", src_t, dst_t);  
        move(n-1, tmp_t, dst_t, src_t);  
    }  
}
```



# Towers of Hanoi

move(1, "T1", "T3", "T2")  
moving from T1 to T3

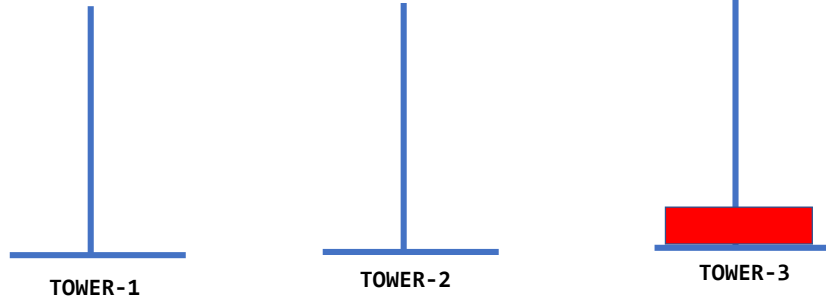
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
    if (n == 1) {  
        printf("moving from %s to %s\n", src_t, dst_t);  
    }  
    else {  
        move(n-1, src_t, tmp_t, dst_t);  
        printf("moving from %s to %s\n", src_t, dst_t);  
        move(n-1, tmp_t, dst_t, src_t);  
    }  
}
```



# Towers of Hanoi

move(1, "T1", "T3", "T2")  
moving from T1 to T3

```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
    if (n == 1) {  
        printf("moving from %s to %s\n", src_t, dst_t);  
    }  
    else {  
        move(n-1, src_t, tmp_t, dst_t);  
        printf("moving from %s to %s\n", src_t, dst_t);  
        move(n-1, tmp_t, dst_t, src_t);  
    }  
}
```



$$n = 2$$

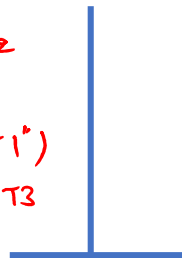


# Towers of Hanoi

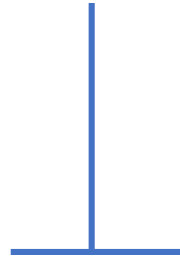
`move(2, "T1", "T3", "T2")`

`move(1, "T1", "T2", "T3")`  
moving from T1 to T2  
moving from T1 to T3  
`move(1, "T2", "T3", "T1")`  
moving from T2 to T3

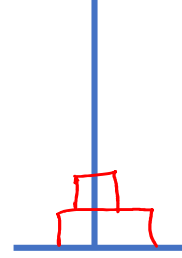
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
    if (n == 1) {  
        printf("moving from %s to %s\n", src_t, dst_t);  
    }  
    else {  
        move(n-1, src_t, tmp_t, dst_t);  
        printf("moving from %s to %s\n", src_t, dst_t);  
        move(n-1, tmp_t, dst_t, src_t);  
    }  
}
```



TOWER-1



TOWER-2

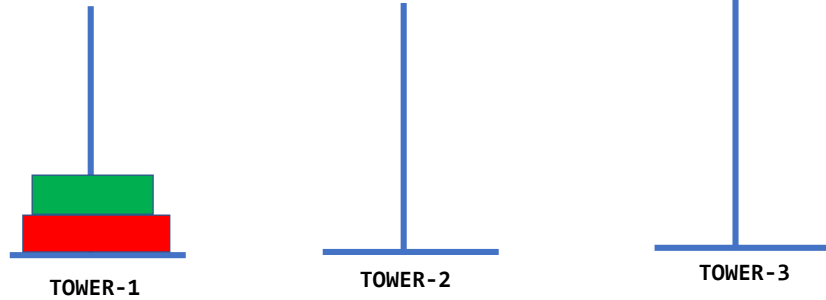


TOWER-3

# Towers of Hanoi

```
move(2, "T1", "T3", "T2")  
  move(1, "T1", "T2", "T3")  
    moving from T1 to T2
```

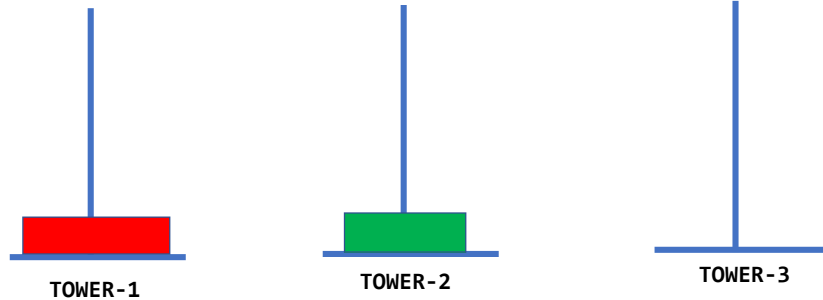
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
    if (n == 1) {  
        printf("moving from %s to %s\n", src_t, dst_t);  
    }  
    else {  
        move(n-1, src_t, tmp_t, dst_t);  
        printf("moving from %s to %s\n", src_t, dst_t);  
        move(n-1, tmp_t, dst_t, src_t);  
    }  
}
```



# Towers of Hanoi

```
move(2, "T1", "T3", "T2")  
  move(1, "T1", "T2", "T3")  
    moving from T1 to T2
```

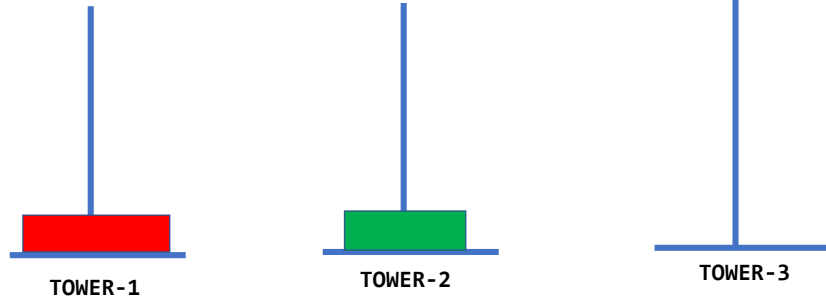
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
    if (n == 1) {  
        printf("moving from %s to %s\n", src_t, dst_t);  
    }  
    else {  
        move(n-1, src_t, tmp_t, dst_t);  
        printf("moving from %s to %s\n", src_t, dst_t);  
        move(n-1, tmp_t, dst_t, src_t);  
    }  
}
```



# Towers of Hanoi

```
move(2, "T1", "T3", "T2")  
  move(1, "T1", "T2", "T3")  
    moving from T1 to T2  
  moving from T1 to T3
```

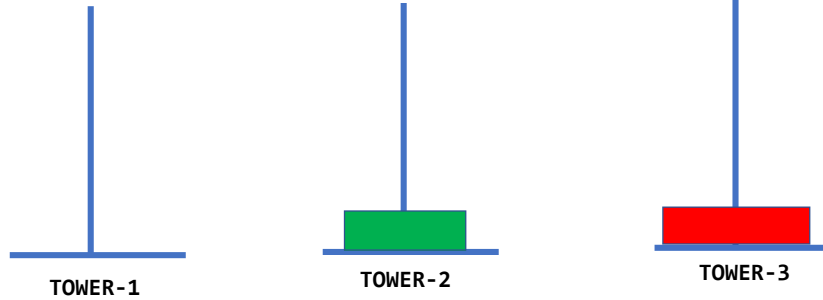
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
  if (n == 1) {  
    printf("moving from %s to %s\n", src_t, dst_t);  
  }  
  else {  
    move(n-1, src_t, tmp_t, dst_t);  
    printf("moving from %s to %s\n", src_t, dst_t);  
    move(n-1, tmp_t, dst_t, src_t);  
  }  
}
```



# Towers of Hanoi

```
move(2, "T1", "T3", "T2")  
  move(1, "T1", "T2", "T3")  
    moving from T1 to T2  
  moving from T1 to T3
```

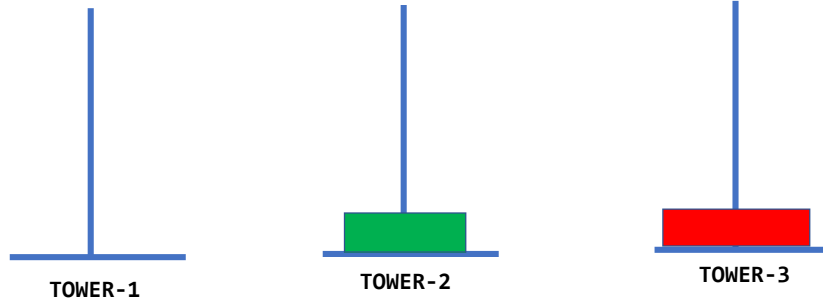
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
  if (n == 1) {  
    printf("moving from %s to %s\n", src_t, dst_t);  
  }  
  else {  
    move(n-1, src_t, tmp_t, dst_t);  
    printf("moving from %s to %s\n", src_t, dst_t);  
    move(n-1, tmp_t, dst_t, src_t);  
  }  
}
```



# Towers of Hanoi

```
move(2, "T1", "T3", "T2")
  move(1, "T1", "T2", "T3")
    moving from T1 to T2
  moving from T1 to T3
  move(1, "T2", "T3", "T1")
    moving from T2 to T3
```

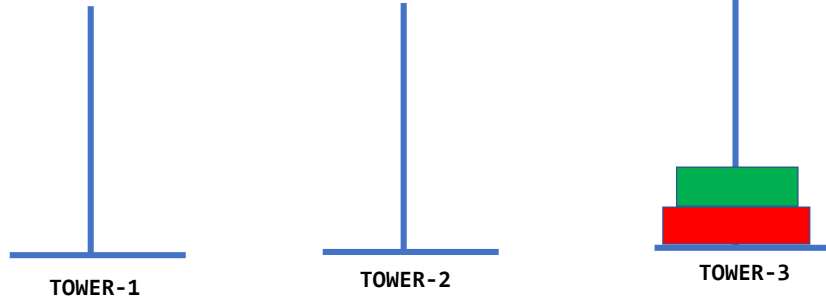
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



# Towers of Hanoi

```
move(2, "T1", "T3", "T2")
  move(1, "T1", "T2", "T3")
    moving from T1 to T2
  moving from T1 to T3
  move(1, "T2", "T3", "T1")
    moving from T2 to T3
```

```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



$$n = 3$$

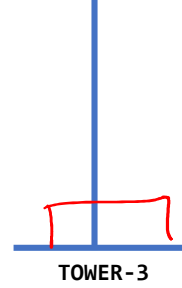
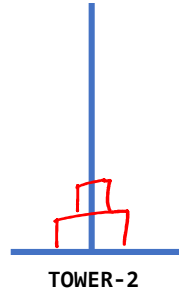


# Towers of Hanoi

```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
    if (n == 1) {
        printf("moving from %s to %s\n", src_t, dst_t);
    }
    else {
        move(n-1, src_t, tmp_t, dst_t);
        printf("moving from %s to %s\n", src_t, dst_t);
        move(n-1, tmp_t, dst_t, src_t);
    }
}
```

move(3, "T1", "T3", "T2")

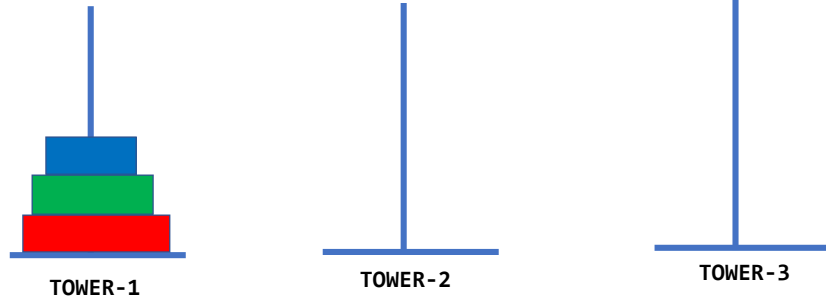
move(2, T1, T2, T3)  
 move(1, T1, T3, T2)  
 moving from T1 to T3  
 moving from T1 to T2  
 move(1, T3, T2, T1)  
 moving from T3 to T2  
 moving from T1 to T3  
 move(2, T2, T3, T1)



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")  
  move(2, "T1", "T2", "T3")  
    move(1, "T1", "T3", "T2")  
      moving from T1 to T3
```

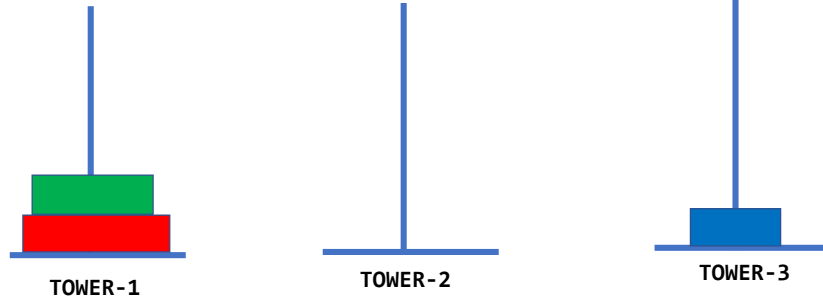
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
  if (n == 1) {  
    printf("moving from %s to %s\n", src_t, dst_t);  
  }  
  else {  
    move(n-1, src_t, tmp_t, dst_t);  
    printf("moving from %s to %s\n", src_t, dst_t);  
    move(n-1, tmp_t, dst_t, src_t);  
  }  
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
move(2, "T1", "T2", "T3")
  move(1, "T1", "T3", "T2")
    moving from T1 to T3
```

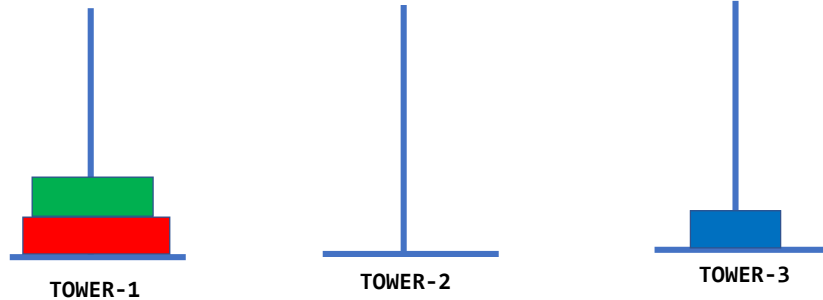
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
    if (n == 1) {
        printf("moving from %s to %s\n", src_t, dst_t);
    }
    else {
        move(n-1, src_t, tmp_t, dst_t);
        printf("moving from %s to %s\n", src_t, dst_t);
        move(n-1, tmp_t, dst_t, src_t);
    }
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")  
  move(2, "T1", "T2", "T3")  
    move(1, "T1", "T3", "T2")  
      moving from T1 to T3  
    moving from T1 to T2
```

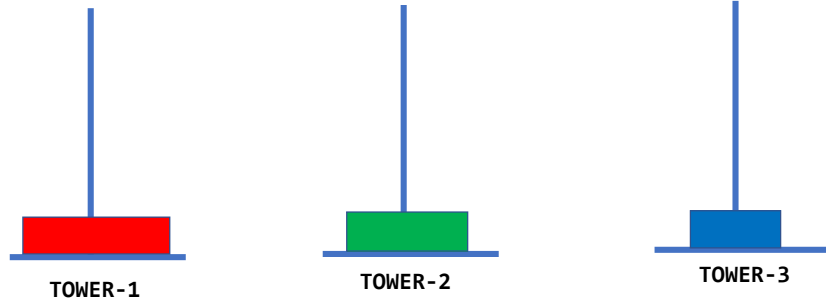
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
    if (n == 1) {  
        printf("moving from %s to %s\n", src_t, dst_t);  
    }  
    else {  
        move(n-1, src_t, tmp_t, dst_t);  
        printf("moving from %s to %s\n", src_t, dst_t);  
        move(n-1, tmp_t, dst_t, src_t);  
    }  
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")  
  move(2, "T1", "T2", "T3")  
    move(1, "T1", "T3", "T2")  
      moving from T1 to T3  
    moving from T1 to T2
```

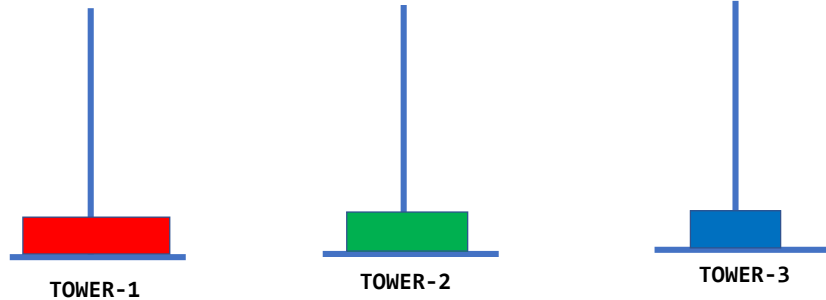
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {  
  if (n == 1) {  
    printf("moving from %s to %s\n", src_t, dst_t);  
  }  
  else {  
    move(n-1, src_t, tmp_t, dst_t);  
    printf("moving from %s to %s\n", src_t, dst_t);  
    move(n-1, tmp_t, dst_t, src_t);  
  }  
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
move(2, "T1", "T2", "T3")
move(1, "T1", "T3", "T2")
moving from T1 to T3
moving from T1 to T2
move(1, "T3", "T2", "T1")
moving from T3 to T2
```

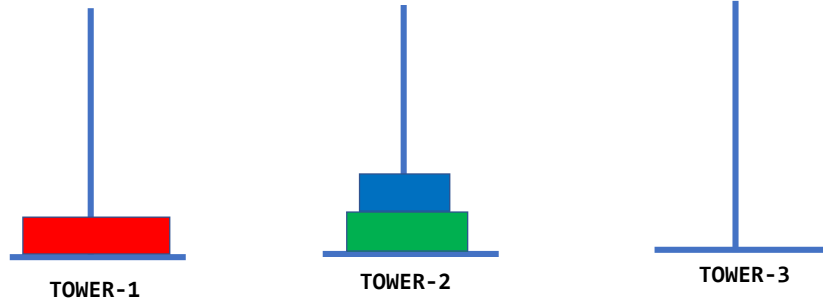
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
    if (n == 1) {
        printf("moving from %s to %s\n", src_t, dst_t);
    }
    else {
        move(n-1, src_t, tmp_t, dst_t);
        printf("moving from %s to %s\n", src_t, dst_t);
        move(n-1, tmp_t, dst_t, src_t);
    }
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
  move(1, "T3", "T2", "T1")
    moving from T3 to T2
```

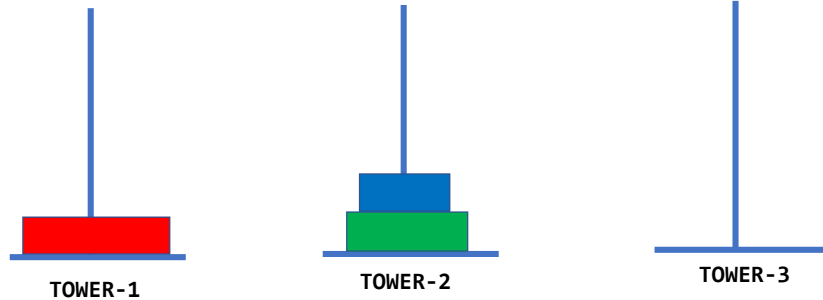
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
  move(1, "T3", "T2", "T1")
    moving from T3 to T2
  moving from T1 to T3
```

```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```

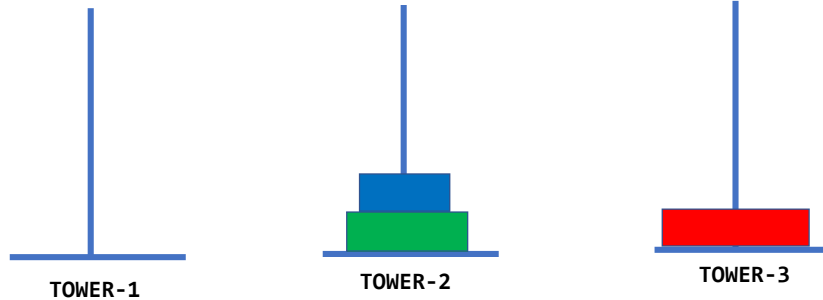




# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
    move(1, "T3", "T2", "T1")
      moving from T3 to T2
  moving from T1 to T3
```

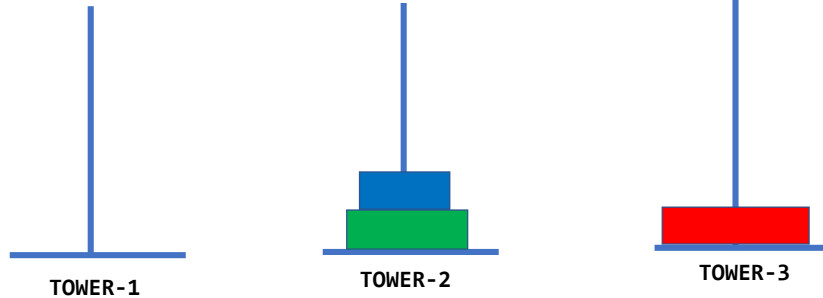
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
  move(1, "T3", "T2", "T1")
    moving from T3 to T2
  moving from T1 to T3
move(2, "T2", "T3", "T1")
  move(1, "T2", "T1", "T3")
    moving from T2 to T1
```

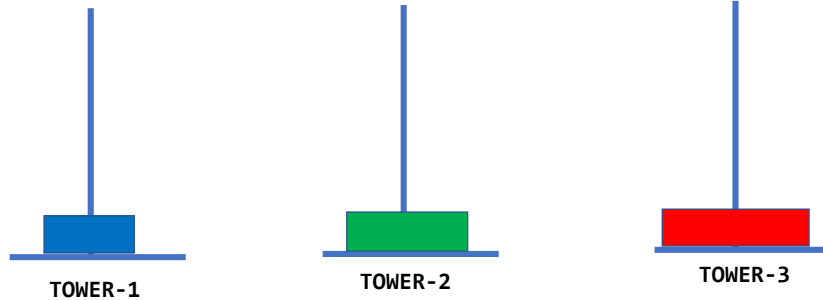
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
  move(1, "T3", "T2", "T1")
    moving from T3 to T2
  moving from T1 to T3
move(2, "T2", "T3", "T1")
  move(1, "T2", "T1", "T3")
    moving from T2 to T1
```

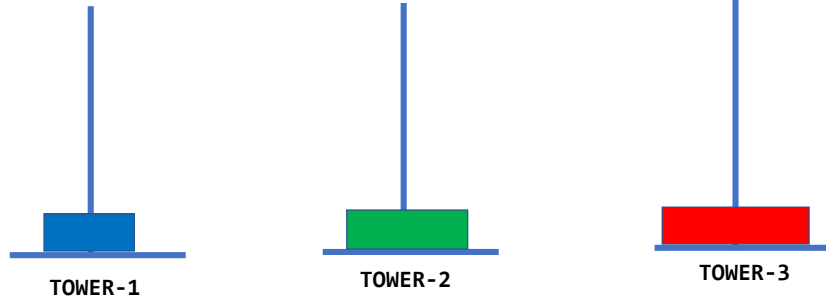
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
  move(1, "T3", "T2", "T1")
    moving from T3 to T2
  moving from T1 to T3
  move(2, "T2", "T3", "T1")
    move(1, "T2", "T1", "T3")
      moving from T2 to T1
    moving from T2 to T3
```

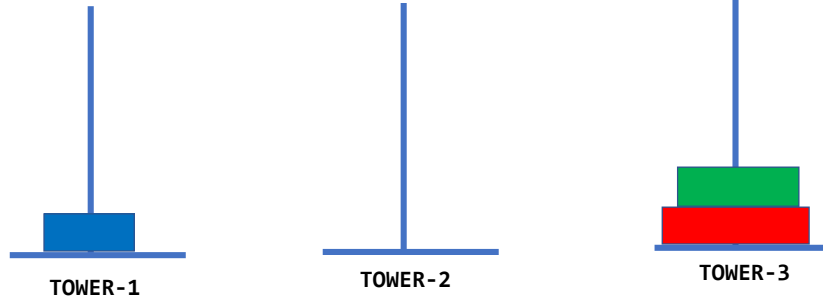
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
    move(1, "T3", "T2", "T1")
      moving from T3 to T2
    moving from T1 to T3
  move(2, "T2", "T3", "T1")
    move(1, "T2", "T1", "T3")
      moving from T2 to T1
    moving from T2 to T3
```

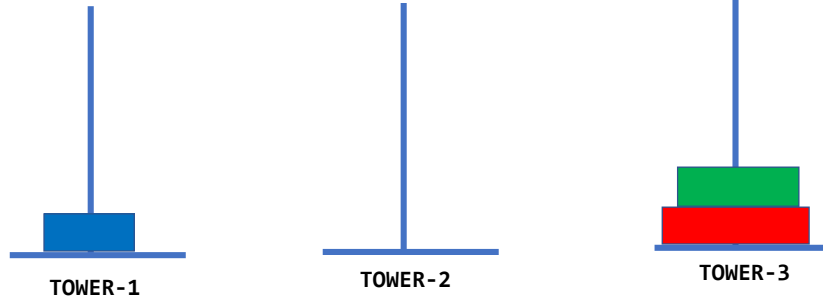
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
    move(1, "T3", "T2", "T1")
      moving from T3 to T2
    moving from T1 to T3
  move(2, "T2", "T3", "T1")
    move(1, "T2", "T1", "T3")
      moving from T2 to T1
    moving from T2 to T3
  move(1, "T1", "T3", "T2")
    moving from T1 to T3
```

```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



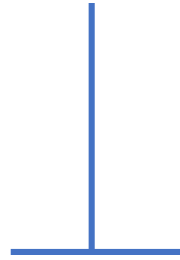
# Towers of Hanoi

```
move(3, "T1", "T3", "T2")
  move(2, "T1", "T2", "T3")
    move(1, "T1", "T3", "T2")
      moving from T1 to T3
    moving from T1 to T2
    move(1, "T3", "T2", "T1")
      moving from T3 to T2
    moving from T1 to T3
  move(2, "T2", "T3", "T1")
    move(1, "T2", "T1", "T3")
      moving from T2 to T1
    moving from T2 to T3
  move(1, "T1", "T3", "T2")
    moving from T1 to T3
```

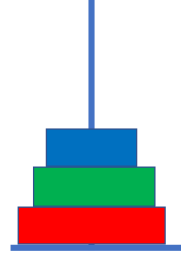
```
void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
  if (n == 1) {
    printf("moving from %s to %s\n", src_t, dst_t);
  }
  else {
    move(n-1, src_t, tmp_t, dst_t);
    printf("moving from %s to %s\n", src_t, dst_t);
    move(n-1, tmp_t, dst_t, src_t);
  }
}
```



TOWER-1



TOWER-2



TOWER-3

## Homework

- Extend the Towers of Hanoi solution to use four towers
  - Two temporaries instead of one
- Compare the number of moves in both approaches for a given number of discs



Selection sort

## Sorting

- Input: a sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$
- Output: a permutation of input sequence  $\langle a'_1, a'_2, \dots, a'_n \rangle$  such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

## Selection sort

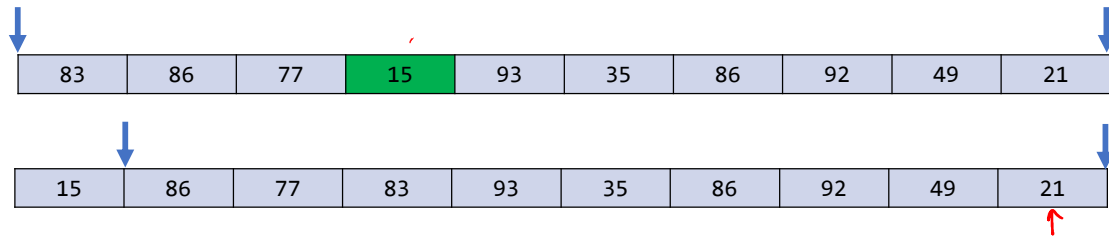
- Find the minimum element and swap it with the first element
- Find the second minimum and swap it with the second element
- Find the third minimum and swap it with the third element
  
- and so on.
  
- Find the  $(n - 1)th$  minimum and swap it with the  $(n - 1)th$  element

## Selection sort

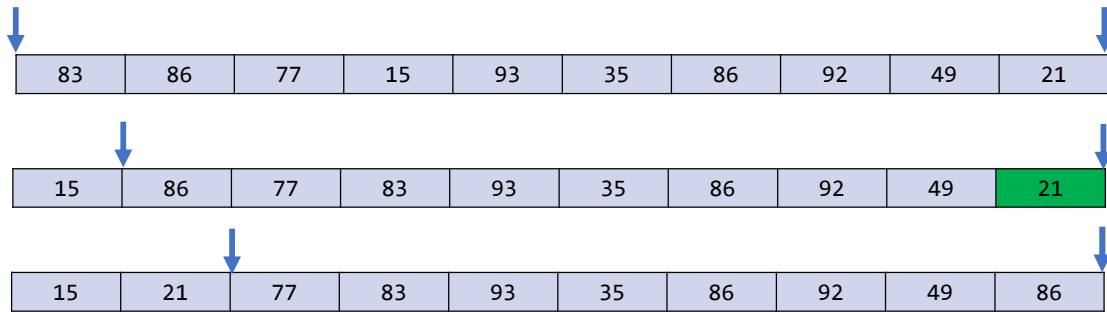
83	86	77	15	93	35	86	92	49	21
----	----	----	----	----	----	----	----	----	----



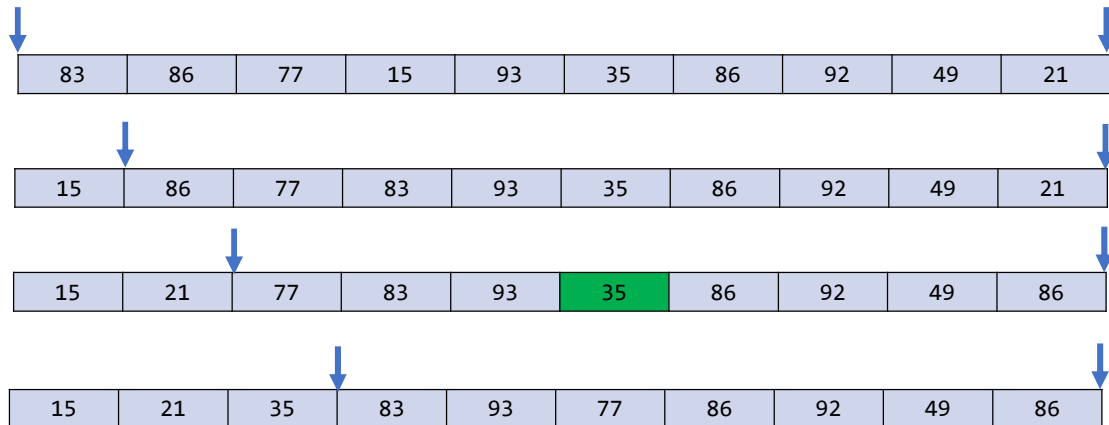
## Selection sort



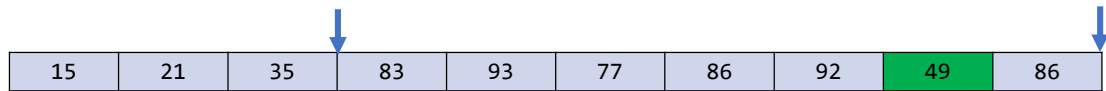
## Selection sort



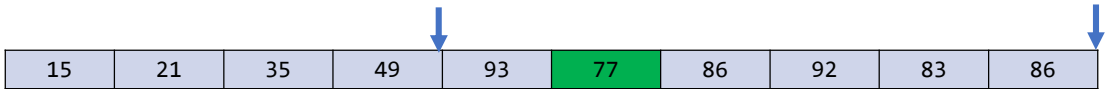
## Selection sort



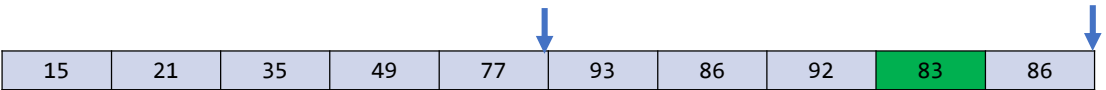
## Selection sort



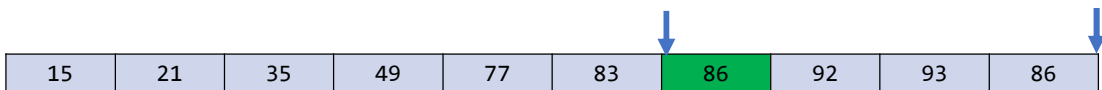
15	21	35	83	93	77	86	92	49	86
----	----	----	----	----	----	----	----	----	----



15	21	35	49	93	77	86	92	83	86
----	----	----	----	----	----	----	----	----	----



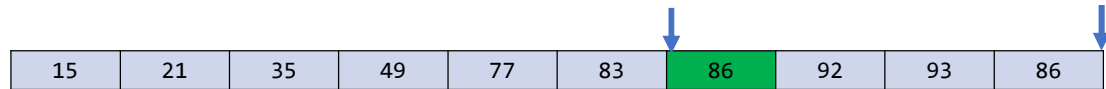
15	21	35	49	77	93	86	92	83	86
----	----	----	----	----	----	----	----	----	----



15	21	35	49	77	83	86	92	93	86
----	----	----	----	----	----	----	----	----	----

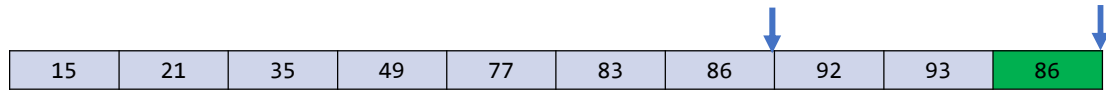


## Selection sort



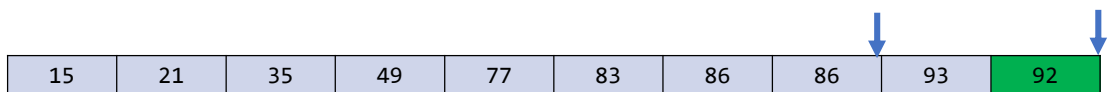
15	21	35	49	77	83	86	92	93	86
----	----	----	----	----	----	----	----	----	----

The diagram shows the first step of selection sort. The array is [15, 21, 35, 49, 77, 83, 86, 92, 93, 86]. The element 86 at index 6 is highlighted in green. Blue arrows point to the element at index 6 and the element at index 9.



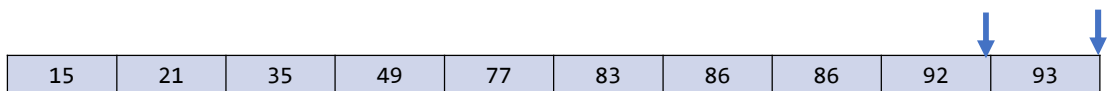
15	21	35	49	77	83	86	92	93	86
----	----	----	----	----	----	----	----	----	----

The diagram shows the second step of selection sort. The array is [15, 21, 35, 49, 77, 83, 86, 92, 93, 86]. The element 86 at index 9 is highlighted in green. Blue arrows point to the element at index 7 and the element at index 9.



15	21	35	49	77	83	86	86	93	92
----	----	----	----	----	----	----	----	----	----

The diagram shows the third step of selection sort. The array is [15, 21, 35, 49, 77, 83, 86, 86, 93, 92]. The element 92 at index 9 is highlighted in green. Blue arrows point to the element at index 8 and the element at index 9.



15	21	35	49	77	83	86	86	92	93
----	----	----	----	----	----	----	----	----	----

The diagram shows the fourth step of selection sort. The array is [15, 21, 35, 49, 77, 83, 86, 86, 92, 93]. The element 92 at index 8 is highlighted in green. Blue arrows point to the element at index 8 and the element at index 9.

## Selection sort

```
void selection_sort(int arr[], int n)
{
    int i, idx;

    for (i = 0; i < n - 1; i++) {
        idx = find_min(arr, i, n);
        if (idx != i) {
            swap(arr, i, idx);
        }
    }
}
```

```
int find_min(int arr[], int start, int end)
{
    int i, min, idx;

    min = arr[start];
    idx = start;
    for (i = start + 1; i < end; i++) {
        if (arr[i] < min) {
            min = arr[i];
            idx = i;
        }
    }
    return idx;
}

void swap(int arr[], int i, int j)
{
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

## Selection sort recursion

```
void selection_sort_r(int arr[], int pos, int n)
{
    int idx;

    if (pos >= n - 1) {
        return;
    }

    idx = find_min(arr, pos, n);
    if (idx != pos) {
        swap(arr, pos, idx);
    }
    selection_sort(arr, pos+1, n);
}

int find_min(int arr[], int start, int end)
{
    int i, min, idx;

    min = arr[start];
    idx = start;
    for (i = start + 1; i < end; i++) {
        if (arr[i] < min) {
            min = arr[i];
            idx = i;
        }
    }
    return idx;
}

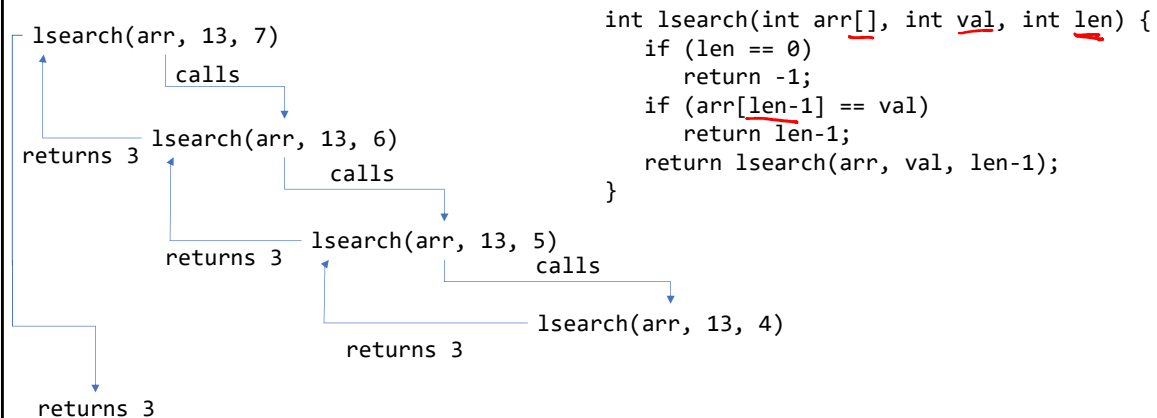
void swap(int arr[], int i, int j)
{
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

We can also write selection sort in a recursive manner. Initially, pos is zero, which is also the start index. The recursive procedure sorts the elements in the range pos to n, where n is the number of elements in the array.

Tail recursion

## Tail recursion

12	11	23	13	41	19	25
----	----	----	----	----	----	----



Look at the recursive definition of the `lsearch` routine. To search an element, we might need to execute a chain function calls before we reach an index at which the value we are searching is stored. After the value is found, we need to return through the chain of function calls until we reach the outermost function. Notice that during a function call, the compiler needs to allocate the space for all local variables and the parameters at function entry and deallocate all the space before returning to the caller. In contrast, none of this is needed in the corresponding iterative algorithm. There is just one function that iterates through the array elements in a loop. Does this mean that the iterative algorithm will always be better than the recursive algorithm?

## Function arguments and local variables

- Space for function arguments and local variables are automatically allocated by the compiler
- The allocation strategy may differ across compilers
- Most compilers use a runtime stack to store a part of local variables, function arguments, and the caller info
- The caller info is required to return to the caller

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20


## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

lsearch(arr, 20, 7)

In this case, 24 bytes are required to store the parameters and the caller info (needed to return to the caller) when lsearch is called. This memory is automatically allocated and deallocated by the compiler. The compiler deallocates the memory just before lsearch returns.



## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

lsearch(arr, 20, 7)

lsearch(arr, 20, 6)

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

24 BYTES

lsearch(arr, 20, 7)

lsearch(arr, 20, 6)

lsearch(arr, 20, 5)

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

24 BYTES

24 BYTES

lsearch(arr, 20, 7)

lsearch(arr, 20, 6)

lsearch(arr, 20, 5)

lsearch(arr, 20, 4)

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

24 BYTES

24 BYTES

24 BYTES

lsearch(arr, 20, 7)

lsearch(arr, 20, 6)

lsearch(arr, 20, 5)

lsearch(arr, 20, 4)

lsearch(arr, 20, 3)

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

24 BYTES

24 BYTES

24 BYTES

24 BYTES

lsearch(arr, 20, 7)

lsearch(arr, 20, 6)

lsearch(arr, 20, 5)

lsearch(arr, 20, 4)

lsearch(arr, 20, 3)

lsearch(arr, 20, 2)

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

24 BYTES

24 BYTES

24 BYTES

24 BYTES

24 BYTES

lsearch(arr, 20, 7)

lsearch(arr, 20, 6)

lsearch(arr, 20, 5)

lsearch(arr, 20, 4)

lsearch(arr, 20, 3)

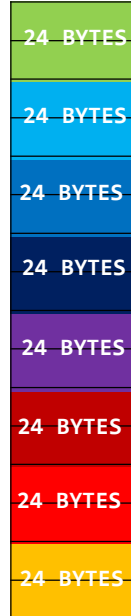
lsearch(arr, 20, 2)

lsearch(arr, 20, 1)

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

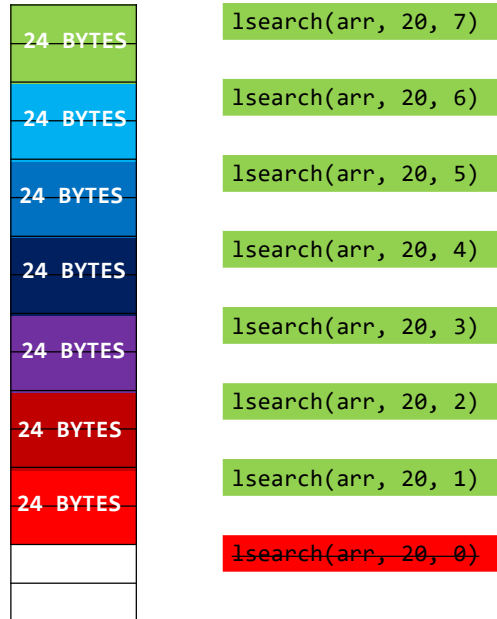


lsearch(arr, 20, 7)  
lsearch(arr, 20, 6)  
➤ lsearch(arr, 20, 5)  
lsearch(arr, 20, 4)  
lsearch(arr, 20, 3)  
- lsearch(arr, 20, 2)  
- lsearch(arr, 20, 1)  
lsearch(arr, 20, 0)

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

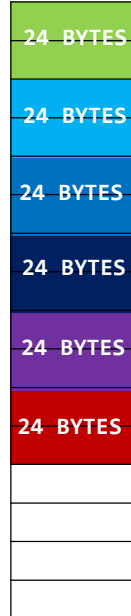




## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20



`lsearch(arr, 20, 7)`  
`lsearch(arr, 20, 6)`  
`lsearch(arr, 20, 5)`  
`lsearch(arr, 20, 4)`  
`lsearch(arr, 20, 3)`  
`lsearch(arr, 20, 2)`  
`lsearch(arr, 20, 1)`  
`lsearch(arr, 20, 0)`

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

24 BYTES

24 BYTES

24 BYTES

`lsearch(arr, 20, 7)`

`lsearch(arr, 20, 6)`

`lsearch(arr, 20, 5)`

`lsearch(arr, 20, 4)`

`lsearch(arr, 20, 3)`

`lsearch(arr, 20, 2)`

`lsearch(arr, 20, 1)`

`lsearch(arr, 20, 0)`

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

24 BYTES

24 BYTES

`lsearch(arr, 20, 7)`

`lsearch(arr, 20, 6)`

`lsearch(arr, 20, 5)`

`lsearch(arr, 20, 4)`

`lsearch(arr, 20, 3)`

`lsearch(arr, 20, 2)`

`lsearch(arr, 20, 1)`

`lsearch(arr, 20, 0)`

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

24 BYTES

`lsearch(arr, 20, 7)`

`lsearch(arr, 20, 6)`

`lsearch(arr, 20, 5)`

`lsearch(arr, 20, 4)`

`lsearch(arr, 20, 3)`

`lsearch(arr, 20, 2)`

`lsearch(arr, 20, 1)`

`lsearch(arr, 20, 0)`

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

24 BYTES

`lsearch(arr, 20, 7)`

`lsearch(arr, 20, 6)`

`lsearch(arr, 20, 5)`

`lsearch(arr, 20, 4)`

`lsearch(arr, 20, 3)`

`lsearch(arr, 20, 2)`

`lsearch(arr, 20, 1)`

`lsearch(arr, 20, 0)`

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

24 BYTES

`lsearch(arr, 20, 7)`

`lsearch(arr, 20, 6)`

`lsearch(arr, 20, 5)`

`lsearch(arr, 20, 4)`

`lsearch(arr, 20, 3)`

`lsearch(arr, 20, 2)`

`lsearch(arr, 20, 1)`

`lsearch(arr, 20, 0)`

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20


~~lsearch(arr, 20, 7)~~

~~lsearch(arr, 20, 6)~~

~~lsearch(arr, 20, 5)~~

~~lsearch(arr, 20, 4)~~

~~lsearch(arr, 20, 3)~~

~~lsearch(arr, 20, 2)~~

~~lsearch(arr, 20, 1)~~

~~lsearch(arr, 20, 0)~~

## Tail recursion

```
int lsearch(int arr[], int val, int len) {
    if (len == 0)
        return -1;
    if (arr[len-1] == val)
        return len-1;
    return lsearch(arr, val, len-1);
}
```

The caller simply returns after the recursive call.

```
arr: 13, 79, 87, 32, 65, 23, 11
search : 20
```

[illegible]

```
lsearch(arr, 20, 7)
```

```
lsearch(arr, 20, 6)
```

```
lsearch(arr, 20, 5)
```

```
lsearch(arr, 20, 4)
```

```
lsearch(arr, 20, 3)
```

```
lsearch(arr, 20, 2)
```

```
lsearch(arr, 20, 1)
```

```
lsearch(arr, 20, 0)
```



## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
Start: if (len == 0)  
        return -1;  
        if (arr[len-1] == val)  
            return len-1;  
len-1: return lsearch(arr, val, len-1);  
        }  
Stop: }
```

The caller simply returns after the recursive call.

Can we remove the recursive call?

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

## Tail recursion

```
int lsearch(int arr[], int val, int len) {  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    return lsearch(arr, val, len-1);  
}
```

The caller simply returns after the recursive call.

arr: 13, 79, 87, 32, 65, 23, 11  
search : 20

```
int lsearch(int arr[], int val, int len) {  
    start:  
    if (len == 0)  
        return -1;  
    if (arr[len-1] == val)  
        return len-1;  
    len = len - 1;  
    goto start;  
}
```

After eliminating recursive call.

We can remove the function call and simply add a goto statement if we are just returning the return value of the target function. This can be done automatically by the compiler. The resulting code is as efficient as its iterative counterpart. The tail recursion enables programmers to write concise and elegant recursive routines without worrying about performance.

## Tail recursion

- An algorithm uses tail recursion if it simply returns after recursive calls without doing additional computation
- The compiler can automatically transform such algorithms into iterative algorithms
- Therefore, the tail-recursive algorithms are equally efficient as their iterative counterparts

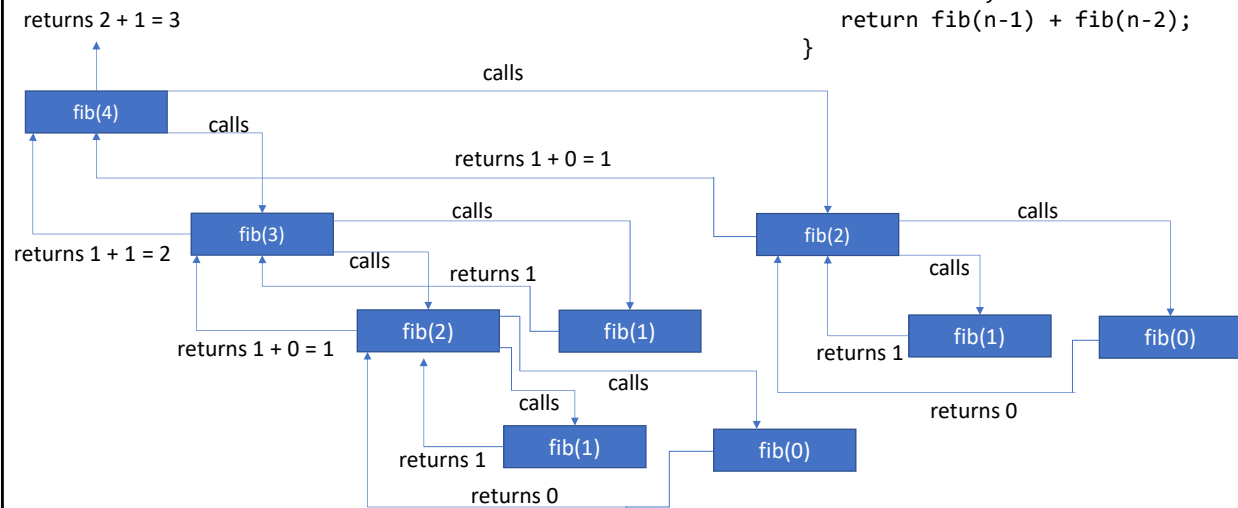
## Recursion vs iteration

- Recursive programs usually turn out to be **concise** and **elegant**
- Recursive solutions can get extremely slow if not designed carefully
  - e.g.,  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$  solution
- Iterative solution is generally faster than the recursive counterpart
  - However, for many problems, iterative solutions are not obvious
    - e.g., the Towers of Hanoi problem
- Tail recursion can eliminate the overhead of additional calls
  - However, tail-recursive solutions do not exist for all problems

# Analysis of algorithms

## Algorithm-1

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



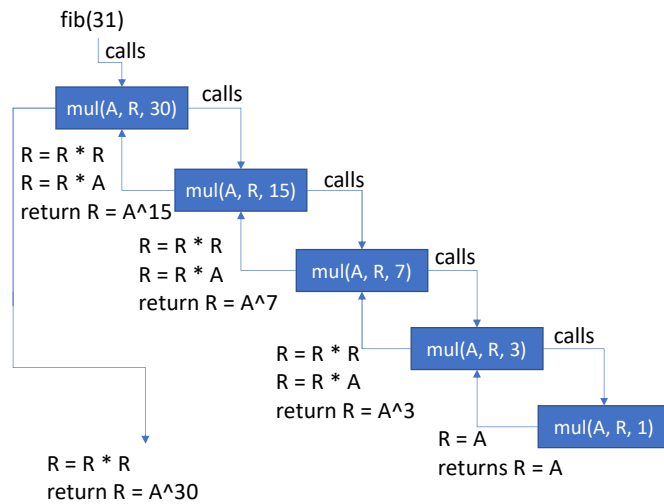
This algorithm is inefficient for computing the  $n$ th Fibonacci number, as discussed in the last class.

## Algorithm-2

<pre>compute fib(5):  prev = 1 pprev = 0  iteration 1 : i = 2; i &lt;= 5 res = 1 + 0 = 1 pprev = 1 prev = 1 i = 3  iteration2: i = 3; i &lt;= 5 res = 1 + 1 = 2 pprev = 1 prev = 2 i = 4</pre>	<pre>iteration 3 : i = 4; i &lt;= 5 res = 2 + 1 = 3 pprev = 2 prev = 3 i = 5  iteration4: i = 5; i &lt;= 5 res = 3 + 2 = 5 pprev = 3 prev = 5 i = 6  return res = 5</pre>	<pre>int fib(int n) {     if (n == 0    n == 1) {         return n;     }      int prev = 1;     int pprev = 0;     int res, i;      for (i = 2; i &lt;= n; i++) {         res = prev + pprev;         pprev = prev;         prev = res;     }     return res; }</pre>
--	---	--

This is an iterative algorithm to compute the nth Fibonacci. This algorithm is reasonably fast.

## Algorithm-3



```

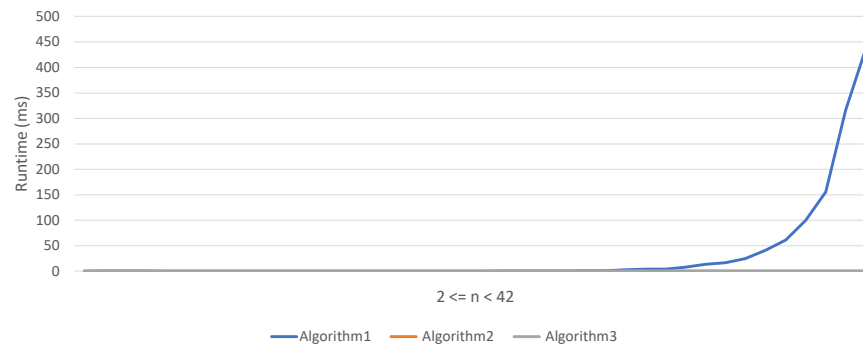
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
    if (n == 1) {
        R[0][0] = A[0][0]; R[0][1] = A[0][1];
        R[1][0] = A[1][0]; R[1][1] = A[1][1];
        return;
    }
    if (n % 2 == 0) {
        mul(A, R, n/2);
        // mul2 takes two 2x2 matrices as input and
        // returns the multiplication in the first
        // matrix
        mul2(R, R); // R <- R * R
    }
    else {
        mul(A, R, (n-1)/2);
        mul2(R, R); // R <- R * R
        mul2(R, A); // R <- R * A
    }
}
  
```

This is the fastest algorithm to compute the nth Fibonacci number. It takes only five function calls to compute the value of fib(31), whereas the iterative algorithm will take around 30 iterations.



## Runtime statistics

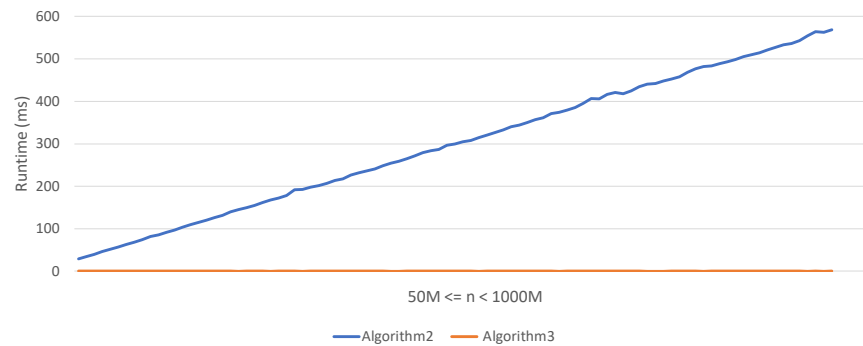
- For various number of  $n$



Even for a small value of  $n$ , e.g., 42, the first algorithm takes around 450 ms. On the other hand, the other two algorithms hardly take any time from such a small  $n$ .

## Runtime statistics

- For various number of n



For very large values of n, e.g., 50 million to 1000 million, the third algorithm takes very little time compared to the second algorithm.

## Analysis of algorithms

- How do we say that one algorithm is better than the other?
- Possible strategy

# Analysis of algorithms

- How do we say that one algorithm is better than the other?
- Possible strategy
  - Implement both algorithms
  - Run both algorithms for various inputs
  - Analyse the runtime and memory usage of both implementations
  - What is the drawback of this approach?

## Analysis of algorithms

- Experiments are performed on a set of test inputs that may not include all possible behaviors
- The algorithm may perform differently on different platforms
  - Some CPUs are faster than others, e.g., Intel i7 vs Intel i3
- The algorithm needs to be fully implemented before the runtime analysis

## An example

Generate a set of test inputs needed to cover all the paths in the example routine at runtime.

```
int example(int arr[], int n) {  
    int ret = -1;  
    if (n > 1000) {  
        if (n > 5000) { 5000  
            ret = sort(arr, n);  
        } else if (n > 3000) { 4000  
            ret = set_zero(arr, n);  
        } else { 3000  
            ret = search_zero(arr, n);  
        }  
    } else {  
        if (n > 900) { 1000  
            ret = sum(arr, n);  
        } else if (n < 700) { 400  
            ret = reverse(arr, n);  
        } else { 800  
            ret = sum_all(arr, n, 2);  
        }  
    }  
    return ret;  
}
```

One possible strategy to test our program is ensuring all program paths execute at runtime. If we test our program when the values of  $n$  are 5000, 4000, 3000, 1000, 400, and 800, it will cover all program paths.

## Analysis of algorithms

- Even though we are successful in generating a set of test inputs that cover all program paths at runtime, the runtime when the program takes a given path may vary depending on the input, e.g.,
  - Some sorting algorithms are fast for small-size arrays but not for large arrays
  - Some sorting algorithms work faster if the input array is mostly sorted
- It's very hard to test for all possible kinds of inputs
  - Think about all possible inputs for a sorting algorithm

## Analysis of algorithms

- Another strategy is to count the total number of CPU instructions that may execute for the worst input (i.e., the input for which the algorithm is expected to behave very badly)
  - What is a CPU instruction?



## CPU instructions

d = a + b + c;	movl -4(%rbp), %eax
	movl -8(%rbp), %ebx
	movl -12(%rbp), %ecx
	mov %eax, %edx
	add %ebx, %edx
	add %ecx, %edx
	mov %edx, -16(%rbp)

A program statement is eventually converted to a sequence of CPU instructions that actually executes when you run an application.

## CPU instructions

```
d = a + b + c;      movl -4(%rbp), %eax
                    movl -8(%rbp), %ebx
                    movl -12(%rbp), %ecx
                    mov %eax, %edx
                    add %ebx, %edx
                    add %ecx, %edx
                    mov %edx, -16(%rbp)
```

Analyzing the assembly is hard because high-level operations like loops that take most of the time are not intuitive in assembly.

Assembly code is the actual code that is going to execute at runtime. However, it is hard to analyze the assembly code than a C program. Therefore, instead of counting the number of CPU instructions, our focus would be on counting the high-level operations.