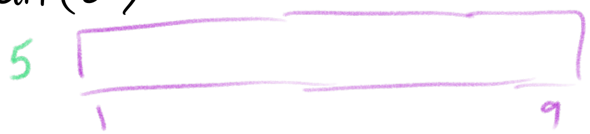
05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14	2	11	9	7
0	1	2	3	4	5	6	7	8	9

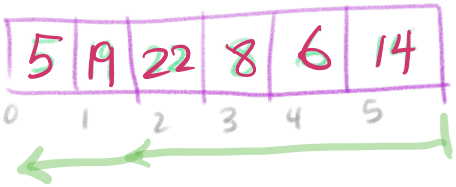
SortTail(0)



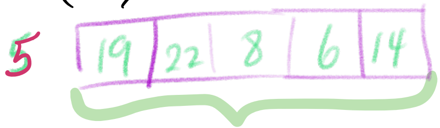
05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?



SortTail(0)



05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

22	8	6	14

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

22	8	6	14

SortTail(2)

22

8	6	14

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

22	8	6	14

SortTail(2)

22

8	6	14

SortTail(3)

8

6	14

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

22	8	6	14

SortTail(2)

22

8	6	14

SortTail(3)

8

6	14

SortTail(4)

6

14

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

22	8	6	14

SortTail(2)

22

8	6	14

SortTail(3)

8

6	14

SortTail(4)

6

14

SortTail(5)

14

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

22	8	6	14

SortTail(2)

22

8	6	14

SortTail(3)

8

6	14

SortTail(4)

6

14

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

22	8	6	14

SortTail(2)

22

8	6	14

SortTail(3)

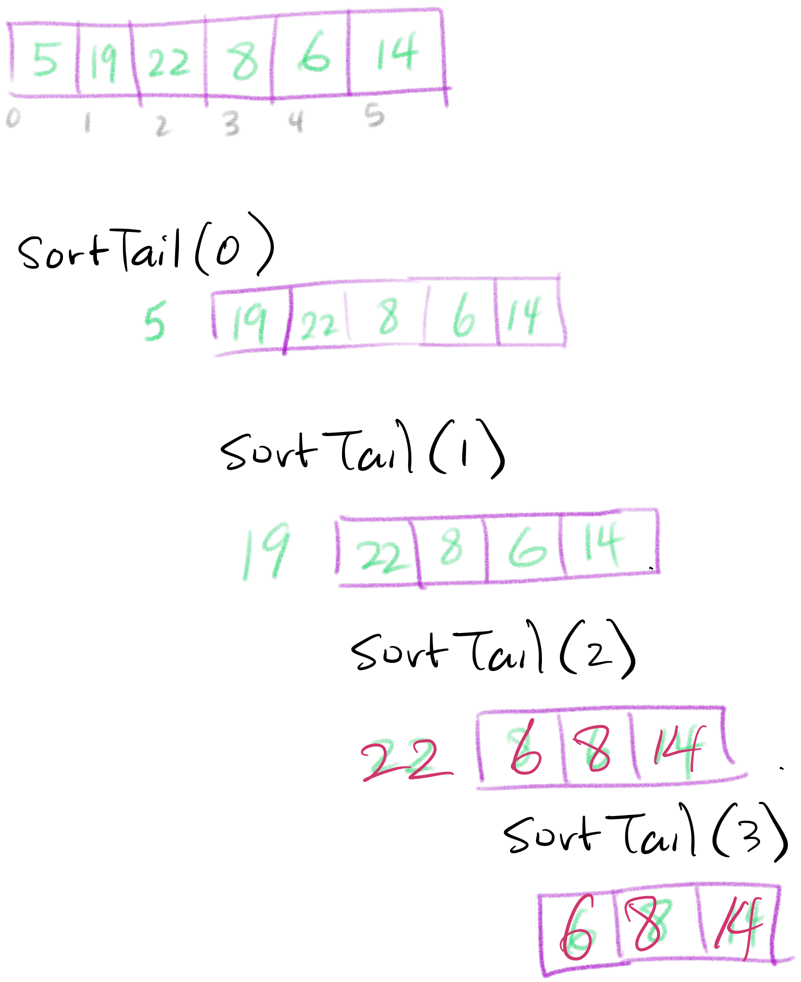
8

6	14

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?



05.2 . Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

22	8	6	14

SortTail(2)

22

8	6	14

SortTail(3)

6	8	14

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	19	22	8	6	14
0	1	2	3	4	5

SortTail(0)

5

19	22	8	6	14

SortTail(1)

19

6	8	14	22

05.2. Week 5 Lec 2 October 10, 2024

Recursion

Why does it work?

5	6	8	14	19	22
0	1	2	3	4	5

SortTail(0)

5

6	8	14	19	22

We could have done this work using a loop within a loop:

```
for all i from 4 downto 0 :
    // sort the tail end of the array
    // from i to 4
    if  i == 4   return
    else
        swap the ith element to the right
        in the sorted array arr[i+1 ... 4]
        until it is in its rightful place.
end for
```

The above-given description of how the program works to achieve its goal is called pseudocode and is used to present the algorithm we have elected to implement.

We have decided to use Insertion Sort Algorithm to solve the Sorting problem.

There are other algorithms to solve Sorting; different algorithms can have different behaviours (take more time, more or less space) and can even have slightly different results, if the requirements for solving the problem leave room for them.

Eg. Sort these records by their key value.

Key	
6	Mary
4	Bob
3	Abdullah
4	Miriam

The above-given description of how the program works to achieve its goal is called pseudocode and is used to present the algorithm we have elected to implement.

We have decided to use Insertion Sort Algorithm to solve the Sorting problem.

There are other algorithms to solve Sorting; different algorithms can have different behaviours (take more time, more or less space) and can even have slightly different results, if the requirements for solving the problem leave room for them.

Eg. Sort these records by their key value.

Key	
3	Abdullah
4	Miriam
4	Bob
6	Mary

An algorithm to find a key in a sorted list

The problem (generally) to find a given value is called the search problem.

Specifically, our problem is to find a given value in a sorted array.

2	7	11	15	19	26	53	100
0	1	2	3	4	5	6	7 = sz-1

```
int locate ( int key); // prototype for locate
// returns the index i into the global array
// arr such that arr[i] == key.
// If it does not exist in the array,
// returns sz, where sz == the array size
```

One algorithm that solves the problem is Linear Search

ay

2	7	11	15	19	26	53	100
0	1	2	3	4	5	6	7 = sz-1

```
locate ( val )          // arr is sorted.
{
     for (int j = 0; j < sz; j++)
     {
            if ( arr[j] == val )
            {     return j;
            }
            else if (arr[j] > val)
            {     return sz;
     }      }
     return j;
}
```

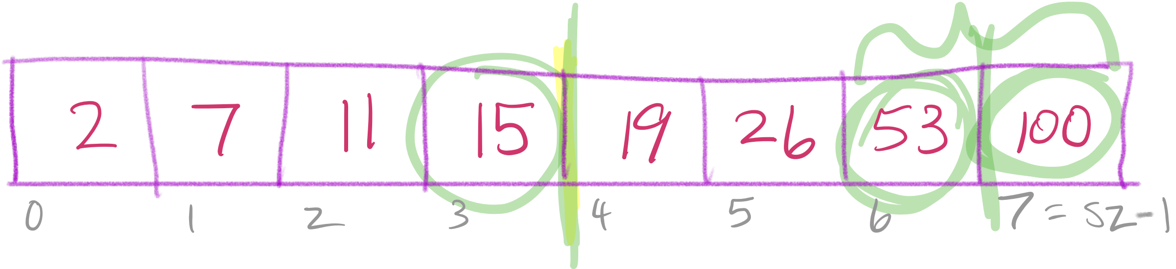
What is the expected number of comparisons to find the correct key, if it is in the array?

half the array size (for sorted arrays)

How many comparisons if it is NOT in the array?

half the array size

Can we do better (ie be more efficient)?



0	1	2	3	4	5	6	7 = sz-1
2	7	11	15	19	26	53	100

pseudo code:

```
start with the full range = $\overset{\ell}{\overset{\shortparallel}{0}} \,..\, \overset{r}{\overset{\shortparallel}{sz-1}}$

$m = \left\lfloor \frac{\ell + r}{2} \right\rfloor$ is the middle

compare val with arr[m]:
    - if val < arr[m]:
        look in arr[ℓ ... m-1]
    - if val > arr[m]:
        look in arr[m+1 .. r]
    - if val == arr[m]: return
```

What is the expected number of comparisons to find the correct key, if it is in the array?

How many comparisons if it is NOT in the array?

Can we do better (ie be more efficient)?

2	7	11	15	19	26	53	100
0	1	2	3	4	5	6	7 = sz-1

pseudo code:

```
start with the full range = $\overset{\ell}{\overset{\shortparallel}{0}} \,..\, \overset{r}{\overset{\shortparallel}{sz-1}}$
$m = \left\lfloor \frac{\ell + r}{2} \right\rfloor$ is the middle
compare val with arr[m]:
    - if val < arr[m]:

    - if val > arr[m]:

    - if val == arr[m]:
```

how do you detect when to stop?

This algorithm is called Binary Search and it runs much faster than Linear Search.

- if you have 1,000,000 keys to search, the number of comparisons you may have to execute is :

	Key found (expected)	Key found (worst case)	Key not Found
Linear Search	500K	1M	Expect 500K
Binary search			

This algorithm is called Binary Search and it runs much faster than Linear Search.

- if you have 1,000,000 keys to search, the number of comparisons you may have to execute is :

	Key found (expected)	Key found (worst case)	Key not Found
Linear Search	500K	1M	1M
Binary search			120

This algorithm is called Binary Search and it runs much faster than Linear Search.

- if you have 1,000,000 keys to search, the number of comparisons you may have to execute is:

	Key found (expected)	Key found (worst case)	Key not Found
Linear Search	500K	1M	1M
Binary search	$119 - \varepsilon$	120	120

Why is it 120?

Because that is how many times you have to divide a range in half till you get down to a range of just 1 element, if you started with 1,000,000.

$\log_2 1000000$ is ≈ 120

Binary search

```
int binarySearch( int arr[], int low, int high, int val)

    if ( high>=low)

        int mid = (low + high)/ 2

    if  (arr[mid]==val)

        return mid

    if ( arr[mid] < val )

        return binarySearch (arr, mid+1, high,
                                          val)

  return binarySearch(arr, low, mid-1, val)
```

Binary search

```
int binarySearch( int arr[ ], int low, int high, int val)
{ // pre-condition: low and high are in range 0.. sz-1 where
                                          // array is sz elements.
    if ( high>=low)
    {
        int mid = (low + high)/2 ;

    }
    if (arr[mid]==val)
    {
        return mid;
    }
    if ( arr[mid] < val )
    {
        return binarySearch( arr, mid+1 ,high, val);
    }

    return binarySearch( arr, low, mid-1, val );
}
```

How to pass in arrays as parameters:

$$\text{void function (float arr[ ], } \underbrace{\text{int size}} \text{)}$$

you can also calculate the number of elements:

$$\underbrace{\text{sizeof (arr)}}_{\text{\# bytes in arr}} \; / \; \underbrace{\text{sizeof (float)}}_{\text{\# bytes in a float}}$$

Note: can't use arr.size() for a C-type array.
(can use it for list containers like vectors — but we haven't covered those yet.)