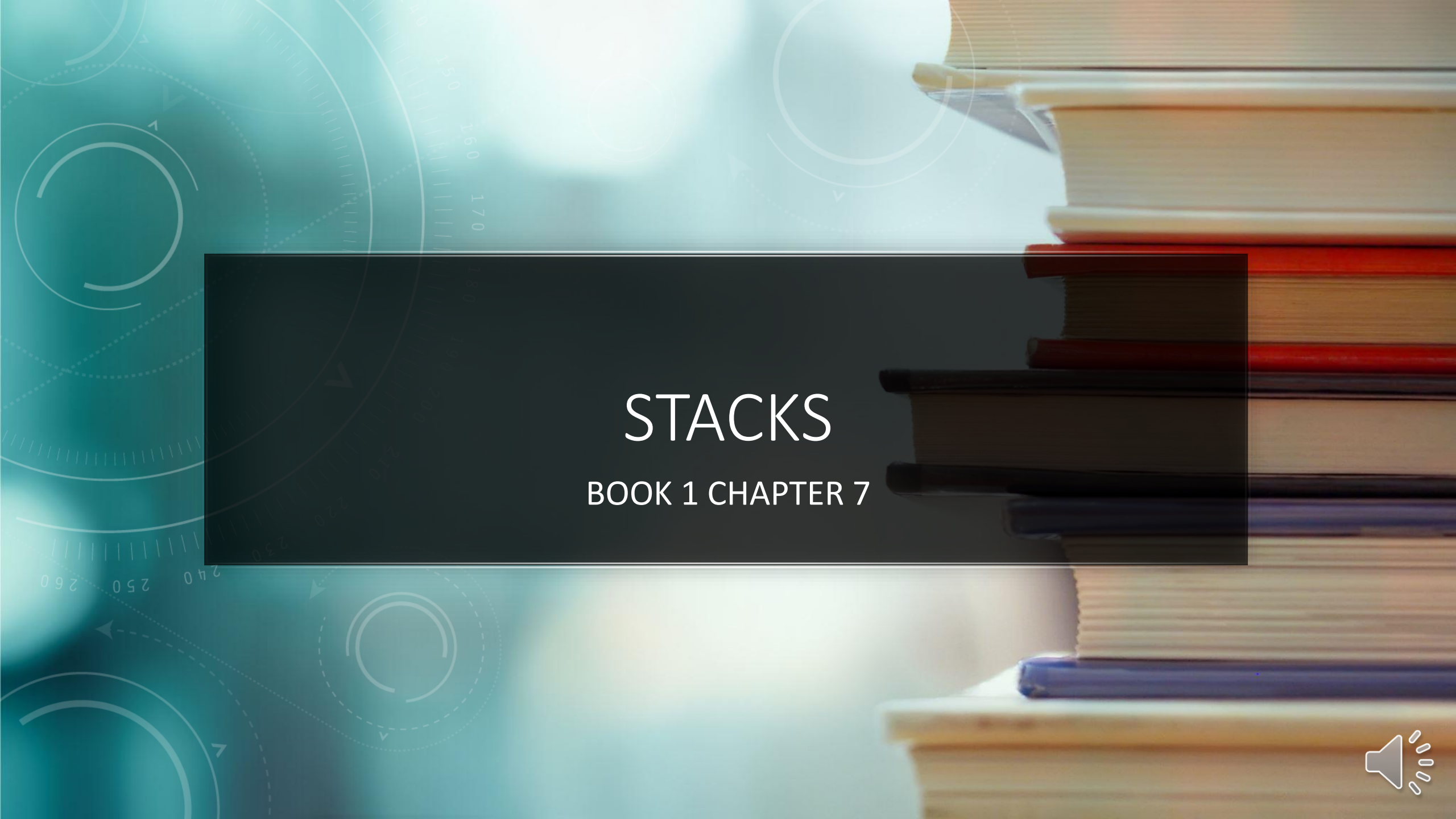
CS-218 DATA STRUCTURES AND ALGORITHMS

LECTURE 16

BY UROOJ AINUDDIN

STACKS

BOOK 1 CHAPTER 7



IN THE LAST LECTURE…

We were introduced to the quicksort algorithm.

We traced through the quicksort algorithm.

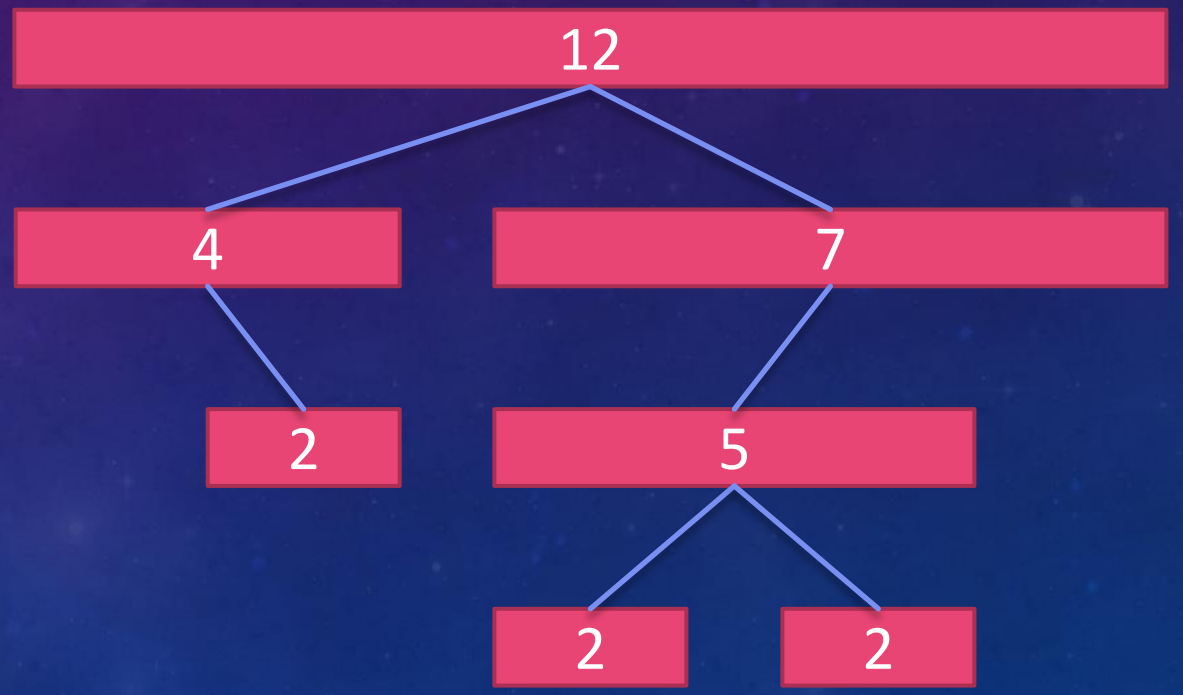
We investigated the time complexity of the algorithm in best and worst cases.

MEMORY REQUIREMENTS OF QUICKSORT

- The quicksort algorithm is an **in-place** or **in-situ** algorithm.
- It delivers the sorted sequence in the same data structure that holds the input sequence.
- The algorithm uses a stack to hold upper and lower bounds of the subsequences that remain to be sorted.
- Every time the pivot sits in its correct position, the subsequence of values lower than the pivot is stored in the stack, and sorting commences in the subsequence of values greater than the pivot.
- The maximum number of elements pushed onto the stack is determined by the number of nodes in the longest branch of the tree formed by the algorithm.

0	1	2	3	4	5	6	7	8	9	10	11
44	33	11	55	77	90	40	60	99	22	88	66

TREE FOR OUR TRACE OF QUICKSORT

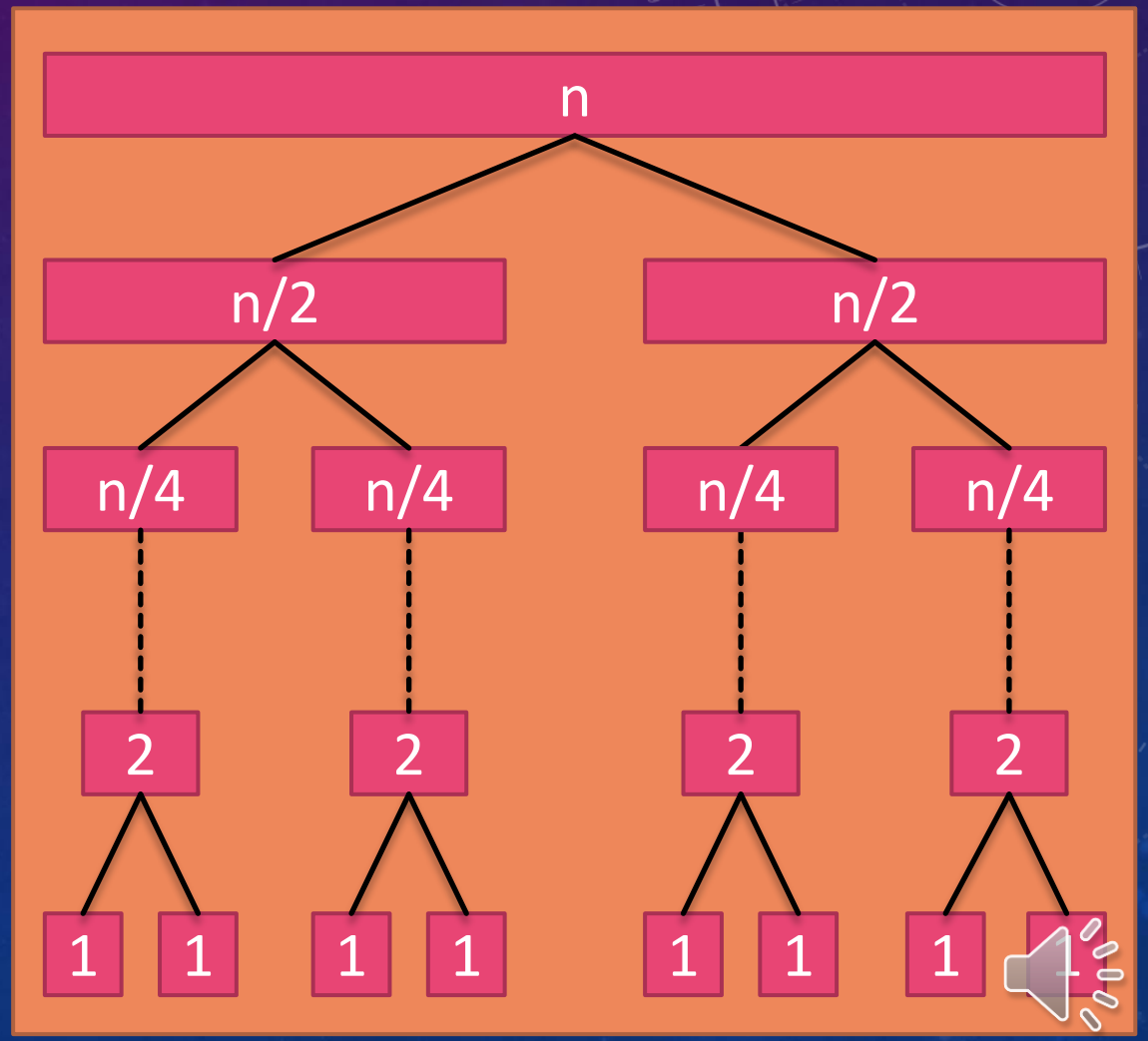


Maximum size of the stack = 3

BEST CASE MEMORY REQUIREMENTS OF QUICKSORT

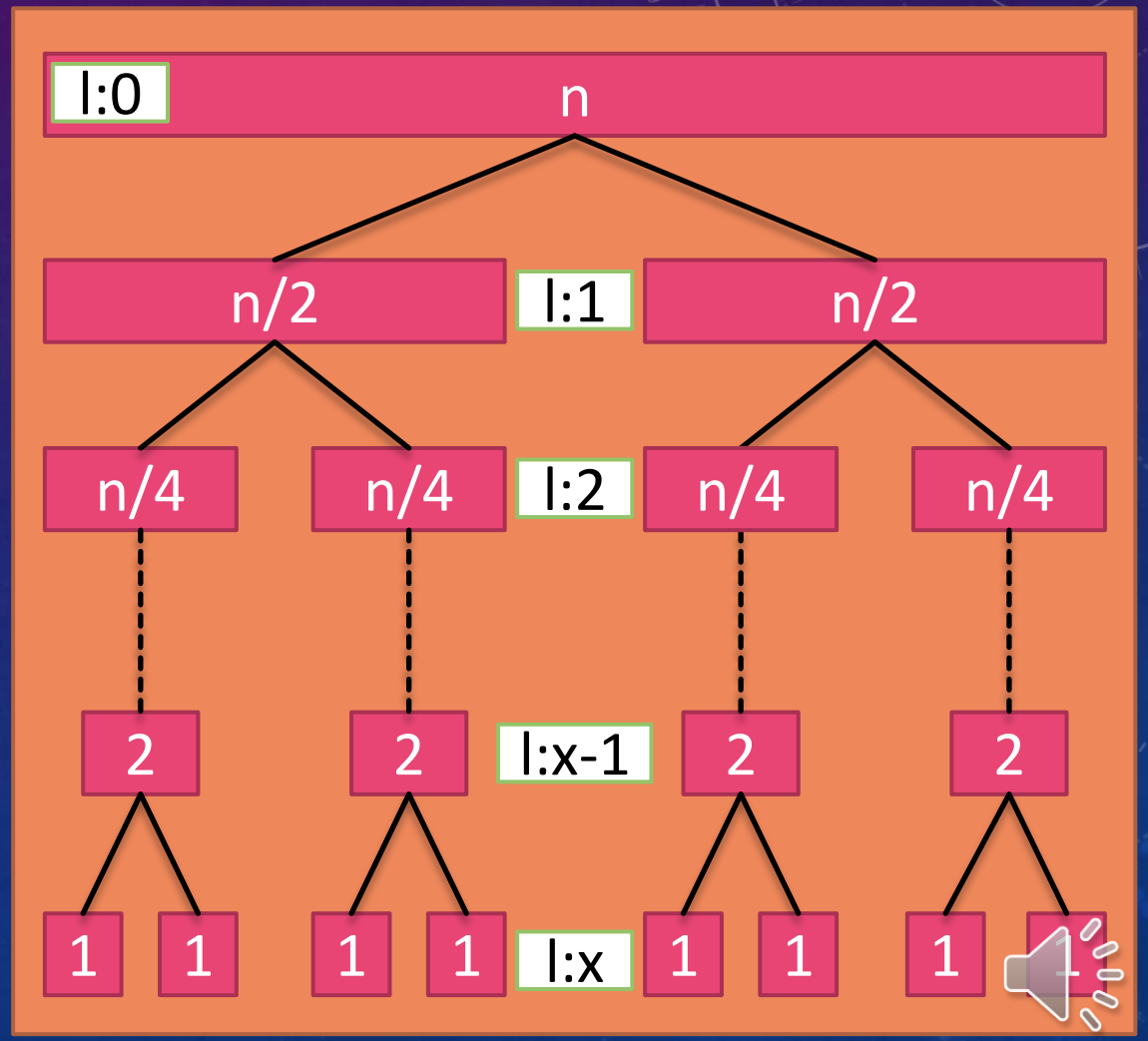
time

- In terms of ~~memory usage~~, the best case of quicksort happens when the correct position of the pivot is in the middle of the n-element sequence in every pass.
- In every pass, subsequences of nearly equal length break off on both sides.
- The resulting tree has branches of nearly equal length.
- For every subsequence with more than two elements, we push a subsequence onto the stack.



BEST CASE MEMORY REQUIREMENTS OF QUICKSORT

- In the tree of quicksort,
  - There are $n/2^0$ elements in level 0.
  - There are $n/2^1$ elements in sequences of level 1.
  - There are $n/2^2$ elements in sequences of level 2.
- Continuing this trend, let the last level be x.
  - There are $n/2^{x-1}$elements in sequences of level $x-1$.
  - There is $n/2^x$ element in sequences of level x.



BEST CASE MEMORY REQUIREMENTS OF QUICKSORT

- We know that the sequences of level x have one element each.

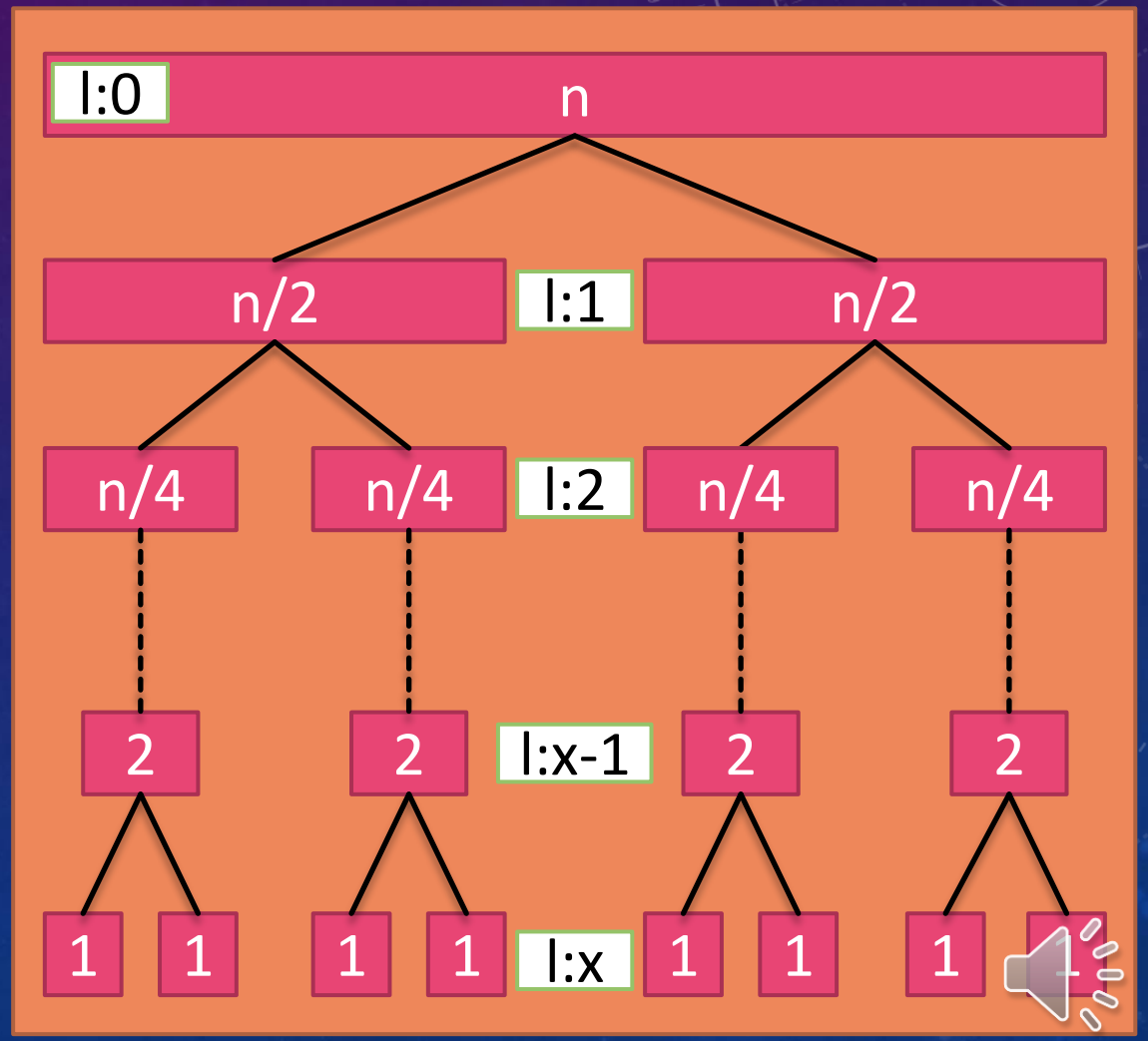
$O(\log n)$

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

$$2^x = n$$

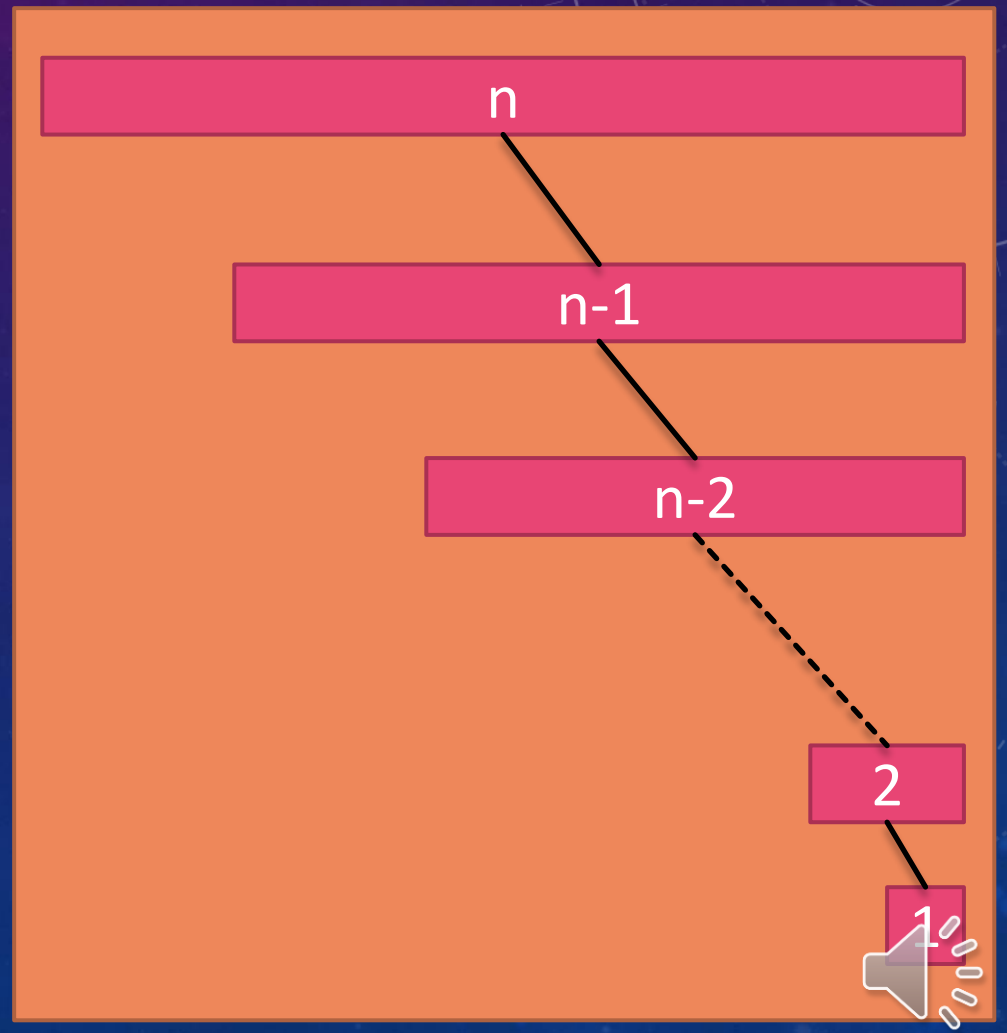
$$x = \log n$$

- We push a subsequence on stack in all but the last two levels.
- Number of levels from 0 to $x - 2$ is $x - 1$.
- The maximum length of the stack is $x - 1$ or $\log n - 1$.



WORST CASE MEMORY REQUIREMENTS OF QUICKSORT

- In terms of ~~memory usage~~ time, the worst case of quicksort happens when the sequence is already sorted in ascending or descending order. The correct position of the pivot is at the first or the last index of the sequence in every pass.
- The longest possible subsequence breaks off every time.
- The resulting tree has branches differing in length by 1.
- For every subsequence with more than two elements, we push a subsequence onto the stack.
- We assume the sequence is sorted in ascending order.

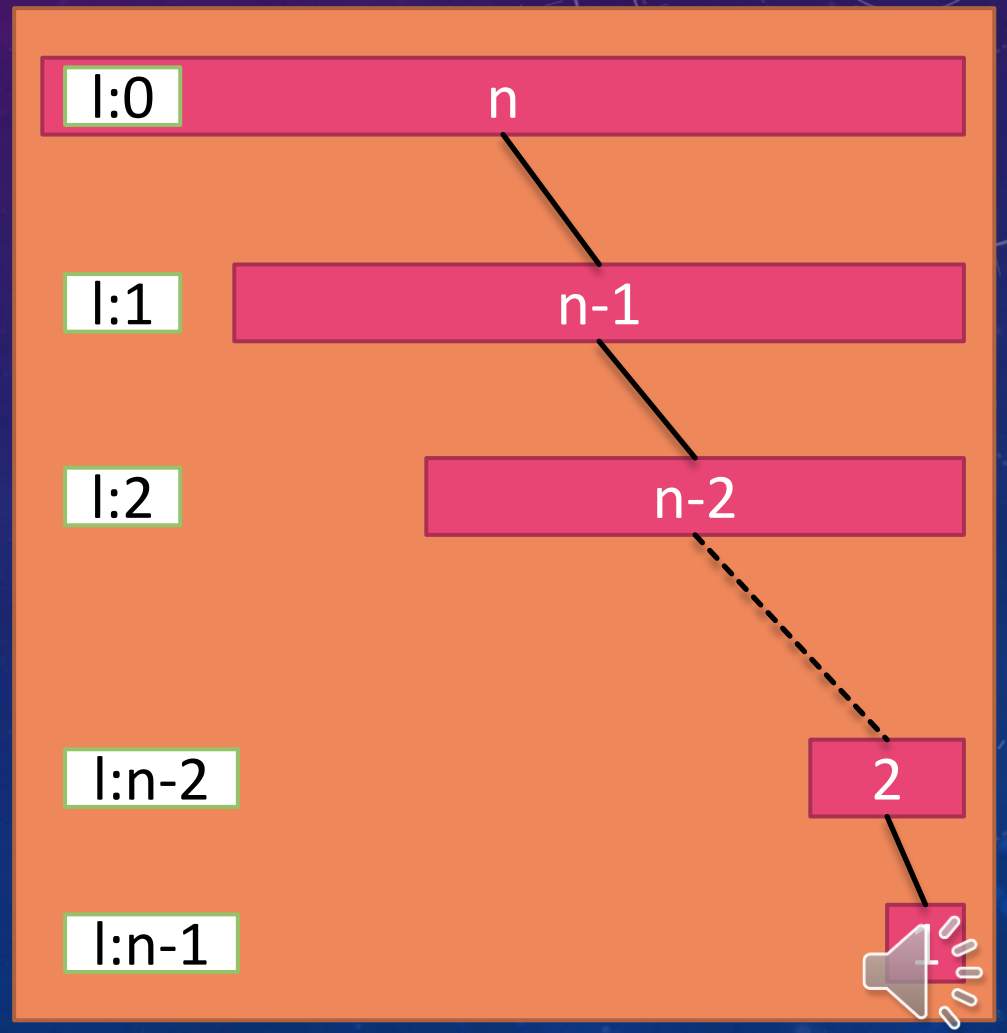


We push one sublist at the end of each pass and we pop it out at the beginningof the next pass, so the stack never grows. The space taken by the stack is O(1).

WORST CASE MEMORY REQUIREMENTS OF QUICKSORT

- In the tree of quicksort,
  - There are $n - 0$ elements in level 0.
  - There are $n - 1$ elements in level 1.
  - There are $n - 2$ elements in level 2.
- Continuing this pattern,
  - There are $n - 2$ elements in level $n - 2$.
  - There is $(n - 1)$ element in level $n - 1$.
- We push subsequent sublists to the stack in all but the last two levels.
- Number of levels from 0 to n is $n - 2$.
- The maximum length of the stack is $n - 2$.

O(n)



SUMMING IT ALL UP

- The worst case for quicksort is when the sequence is already sorted. It is considered the worst case, because only one subsequence breaks off after placing the pivot correctly, on only one side of the pivot, in every pass. The subsequence breaking off is only one element shorter than the previous.
- Time complexity of the worst case is $O(n^2)$.
- Space complexity of the worst case is $O(n)$. O(1)
- The best case for quicksort is when two nearly equal subsequences break off after placing the pivot correctly, one on each side of the pivot, in every pass. The subsequences breaking off are roughly half the length of the previous.
- Time complexity of the best case is $O(n \log n)$.
- Space complexity of the best case is $O(\log n)$.

OPTIMIZING QUICKSORT

If the sequence is presorted, picking the first element as the pivot will always result in the worst case.

We can optimize quicksort by avoiding the worst-case scenario.

We desire some way of getting close to the best-case scenario for quicksort.

This results in a running time that is asymptotically the same as that of the best case.

This can be achieved by a strategy in which the pivot does not sit in the first or last position in any pass.

SOLUTION #1: RANDOMIZING THE PIVOT INDEX

- In each pass, generate a random number between left and right, both included.
- Let the random number be x.
- Assume the element at x to be the pivot. Set $loc = x$.
- This ensures the same index is not used as position of the pivot every time.
- Running time of each pass now includes the running time of the algorithm for random number generation.

SOLUTION #2: MEDIAN-OF-THREE

- In each pass, generate three random numbers between left and right, both included.
- Let the random numbers be x, y and z.
- Assume the median of the elements at indices x, y and z to be the pivot. Set loc to the index containing the median.
- The median is the middle element in a sorted list. Since the pivot is median of three elements, we know that there is at least one element larger than the pivot and at least one element smaller than the pivot. This ensures that subsequences break off on both sides of the pivot.
- Running time of each pass now includes the running time of the algorithm for random number generation and time to sort the elements of indices x, y and z.

SOLUTION #3: MEDIAN OF FIRST LAST AND MIDDLE ELEMENTS

- In each pass, pick elements from indices left, right, and $\lfloor(\text{left} + \text{right})/2\rfloor$.
- Assume the median of these elements to be the pivot. Set loc to the index containing the median.
- The median is the middle element in a sorted list. Since the pivot is median of three elements, we know that there is at least one element larger than the pivot and at least one element smaller than the pivot. This ensures that subsequences break off on both sides of the pivot.
- Running time of each pass now includes the time to sort the first, last and middle elements of the sequence.

HOMEWORK

1. Run the quicksort algorithm for the given sequence of keys. Use solution #3 for optimization of quicksort.

   56,98,12,47,35,26,58,41

2. Change the quicksort code (sortwithstack.py) to implement solution #1 for optimization of quicksort.

RECURSION

BOOK 1 CHAPTER 10

THE RUN-TIME STACK

CALLING AND RETURNING FROM FUNCTIONS

TRANSFER OF CONTROL

- Each time a function is called, an **activation record** is automatically created in order to maintain information related to the function.
- One piece of information in the activation record is the **return address**. This is the location of the next instruction to be executed in the calling function.
- When the called function returns, the return address is obtained from the activation record and execution can resume from where it left off in the calling function.
- The activation records also include **storage space for local variables**.
- A variable created within a function is local to that function and is said to have local scope. Local variables are created when a function begins execution and are destroyed when the function terminates.

MANAGEMENT OF ACTIVATION RECORDS

An activation record is created for every function call.

The system must manage the collection of activation records and remember the order in which they were created.

Remembering the order of creation allows the system to return to the next statement in the calling function when a called function completes.

The system stores the activation records on a **run-time stack**.

The run time stack is a stack hidden from the programmer and managed by the operating system.

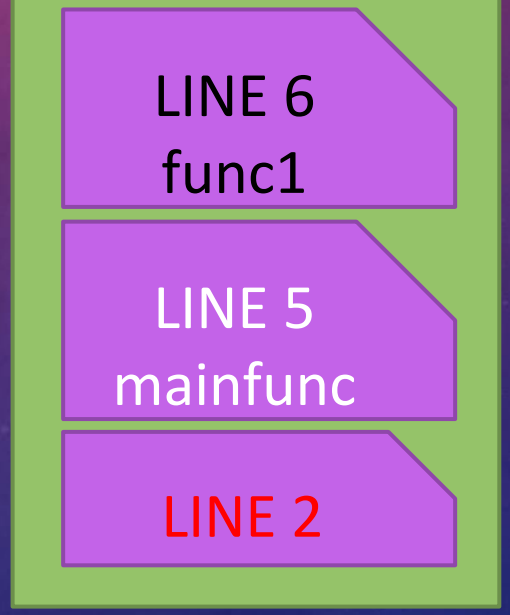
```
1.  def func1(x):
2.      print("Executing func1")
3.      y=x*4-6
4.      print("Passing x=",x,"and y=",y,"to func2")
5.      z=func2(x,y)
6.      print("Back in func1")
7.      return z
```

```
1.  def func2(a,b):
2.      print("Executing func2")
3.      c=(a+b)/2
4.      print("Passing c=", c, "to func1")
5.      return c
```

```
1.  def mainfunc():
2.      print("Executing mainfunc")
3.      m=int(input("Enter a number:"))
4.      n=func1(m)
5.      print("Back in mainfunc")
6.      print("The answer is",n)
7.      return
```

```
1.  mainfunc()
2.  print("DONE")
```

RUN-TIME STACK



OUTPUT

```
Executing mainfunc
Enter a number:4
Executing func1
Passing x= 4 and y= 10 to func2
Executing func2
Passing c= 7.0 to func1
Back in func1
Back in mainfunc
The answer is 7.0
DONE
```

SO WHAT DID WE LEARN TODAY?

We investigated the space complexity of quicksort in best and worst cases.

We looked at ways to optimize quicksort.

We were introduced to the run-time stack.

THINGS TO DO

Read the book!

Note your questions and put them up in the relevant online session.

Email suggestions on content or quality of this lecture at uroojain@neduet.edu.pk