Quicksort – Fine Tuning

CS 112 Spring 2020 – Apr 23

Sesh Venugopal

Fine Tuning Running Time

Since quicksort can sort an array in place (without using extra array space, unlike mergesort, which needs extra storage), and the average running time is `O(nlogn)`, it is widely used in practice

A core set of fine-tuning techniques have been developed to make quicksort run even better in real time

IMPORTANT: These techniques do not change the big O running times!

Fine Tuning Technique #1

Defend against worst case time for sorted input

A good sorting algorithm should do minimal work if the input is already in sorted order

But quicksort's behavior is the worst for sorted inputs, O(n^2) running time

This first fine-tuning technique works to correct this aberration, by picking the median of the first, middle, and last items as the pivot (instead of always picking the first):

[2, 5, 3, 15, 23, 25, 81, 16, 10]

median(2,23,10) = 10

Swap first with median

[10, 5, 3, 15, 23, 25, 81, 16, 2]

Then call *split*

[8, 5, 3, 15, 1, 7, 25, 81, 16, 10]

median(8,1,10) = 8

No need to swap with first, since 8 is first

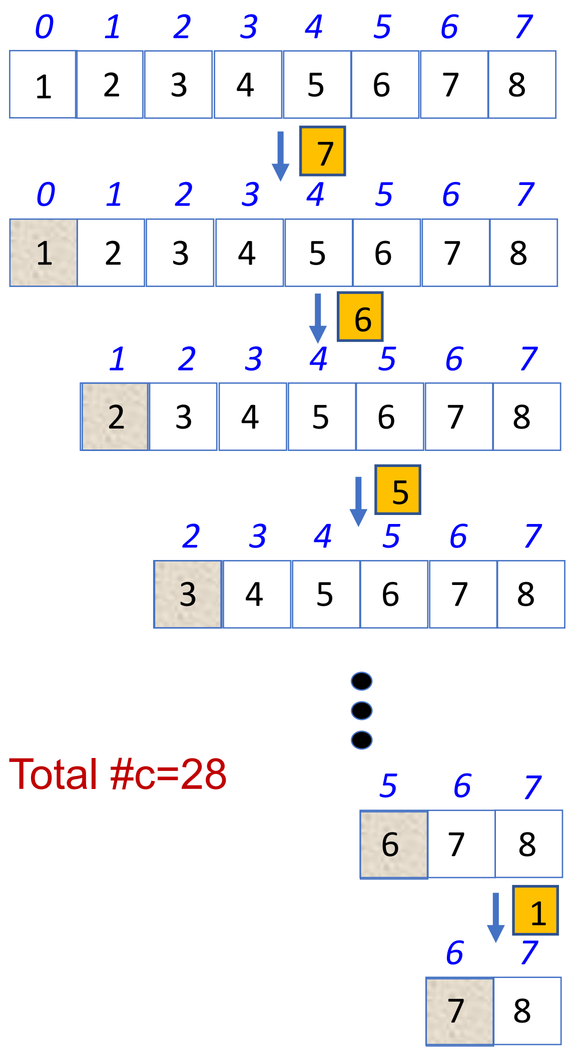
[8, 5, 3, 15, 1, 7, 25, 81, 16, 10]

Then call *split*

Alternatively, median(8,7,10)=8
(since there is an even number of items)

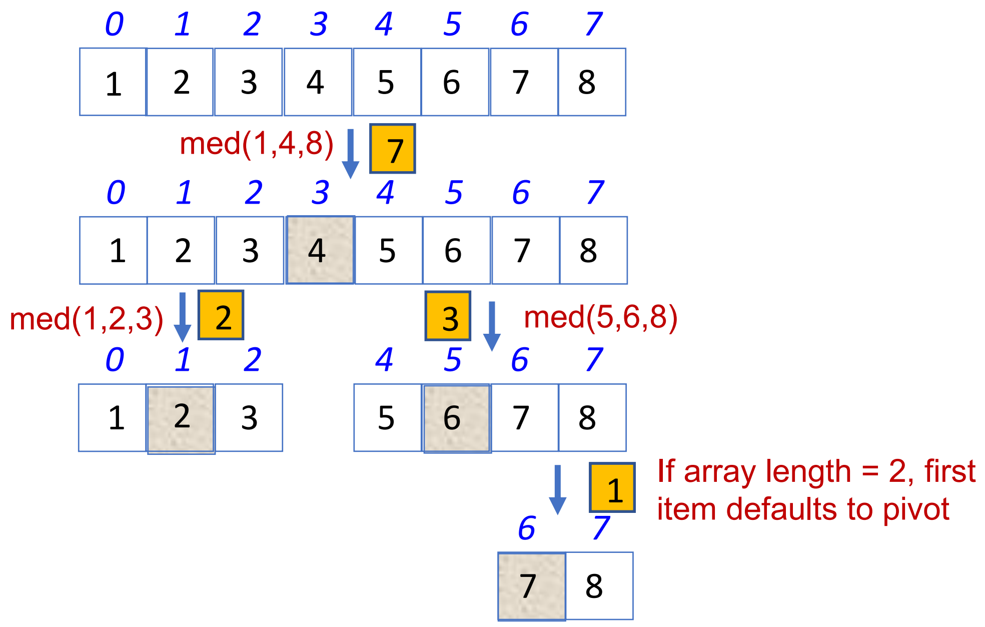
Since median is brought to the first position, the split algorithm doesn't change – it still works with the first item as the pivot

Sorted Input, no fine tuning



Total #c=28

Sorted Input, pivot is median



Total #c=13

Fine Tuning Technique #2

Defend against recursion overhead

The recursive nature of quicksort results in running time overhead due to recursive method calls

This is not a big issue when the subarrays being sorted recursively are large enough, since the added overhead of a recursive call can be overlooked relative to the computational time to sort large arrays

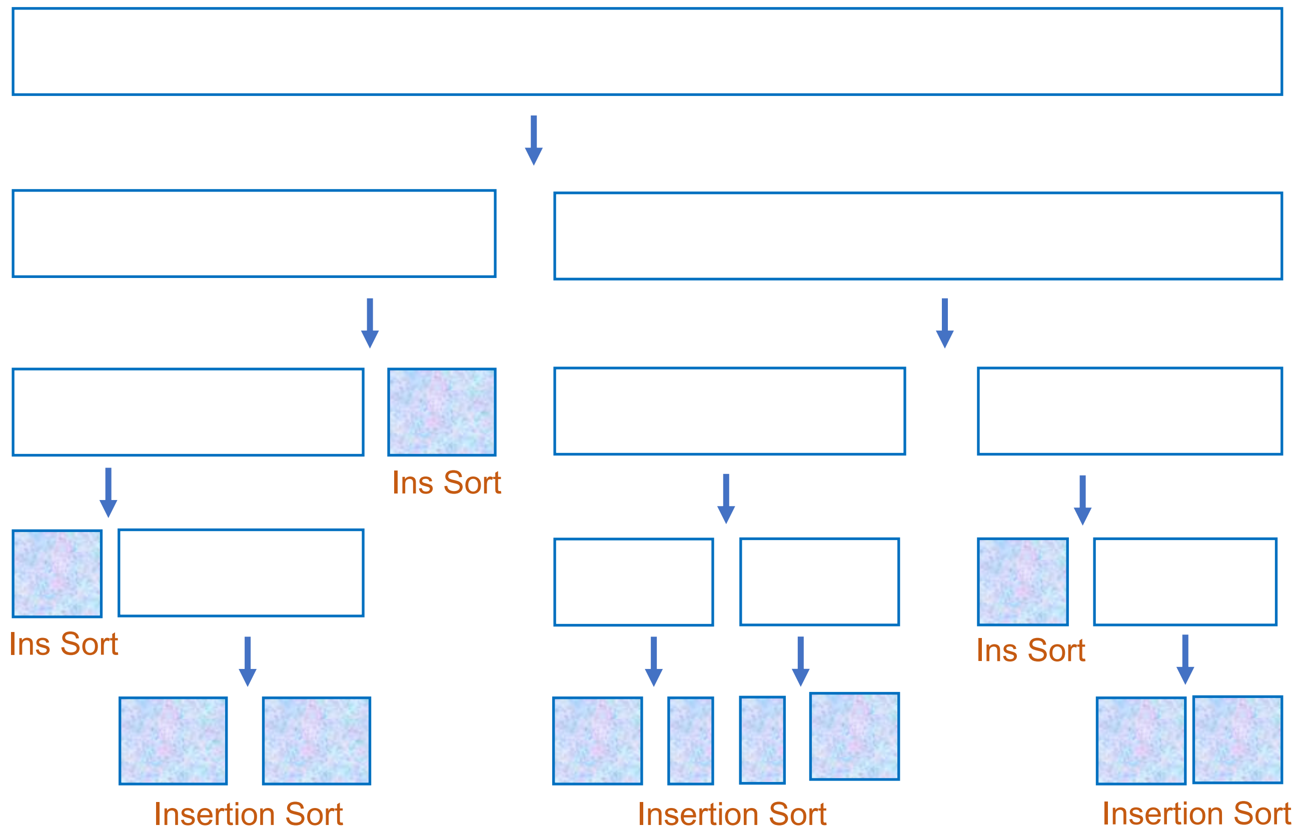
Trouble happens when the subarray being sorted gets to be too small to afford recursive calls

When a subarray gets sufficiently small, quicksort is NOT used to sort it. Instead, insertion sort is used

Yes, the average case of insertion sort is O(n^2) compared to quicksort's O(nlogn), but at small array sizes, the asymptotic big O times no longer apply – insertion sort will be comparable or even faster than quicksort

Fine Tuning Technique #2

When subarray gets too small, switch to insertion sort



Exactly what ”too small” is depends on the machine. Around 50-75 is a good starting number, which can be tweaked with some experimenting

Fine Tuning Technique #3

Defend against excessive recursion stack space usage

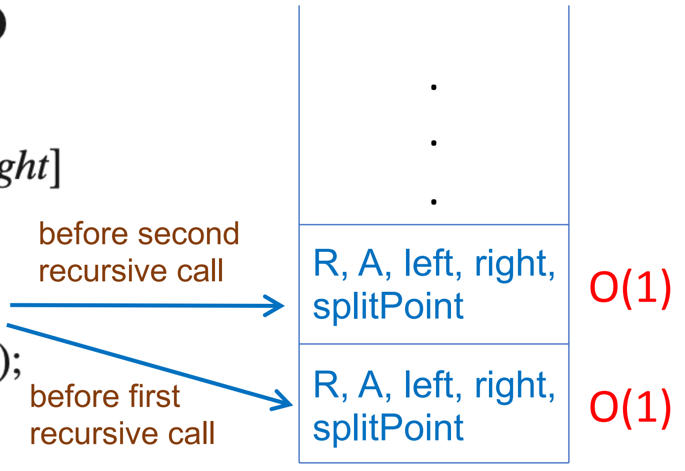
Every time a recursive call is made, the execution engine needs to push the calling method's state info on the activation stack

Activation stack info includes the address of the statement to which the recursive call must return, and the values of the local variables at the time of the call

Algorithm quicksort(A, left, right)

```
input: subarray A[left ... right]
output: sorted subarray A[left ... right]

    splitPoint ← split(A, left, right)
L → quicksort(A, left, splitPoint − 1);
R → quicksort(A, splitPoint + 1, right);
```



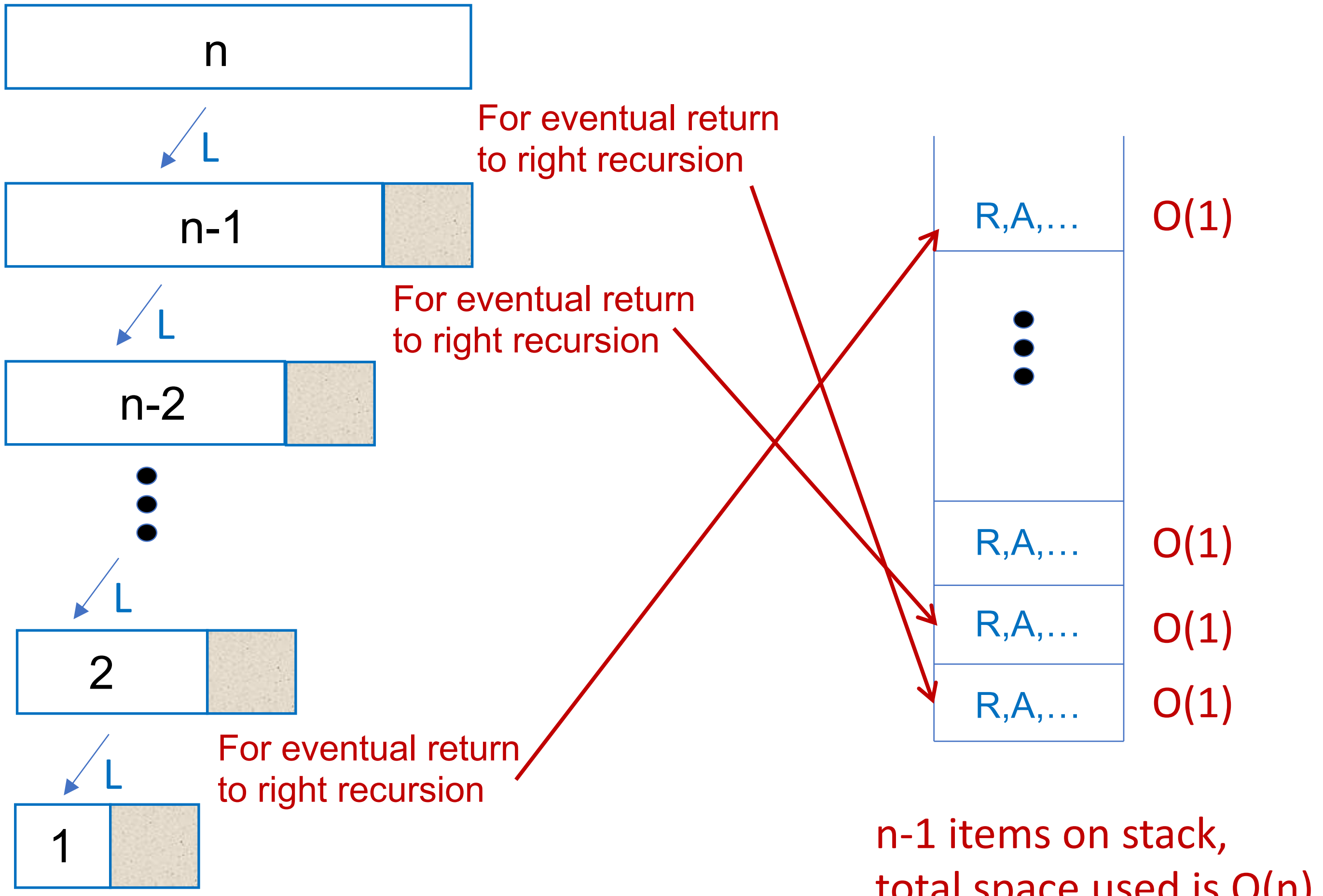
before second recursive call
before first recursive call
.
.
.
R, A, left, right, splitPoint
O(1)
R, A, left, right, splitPoint
O(1)

Activation stack

L,R: memory address of respective statements
A: memory address of array
left, right, splitPoint: values of local variables

O(1) space for all of these

Recursion Stack Space Usage: Worst Case

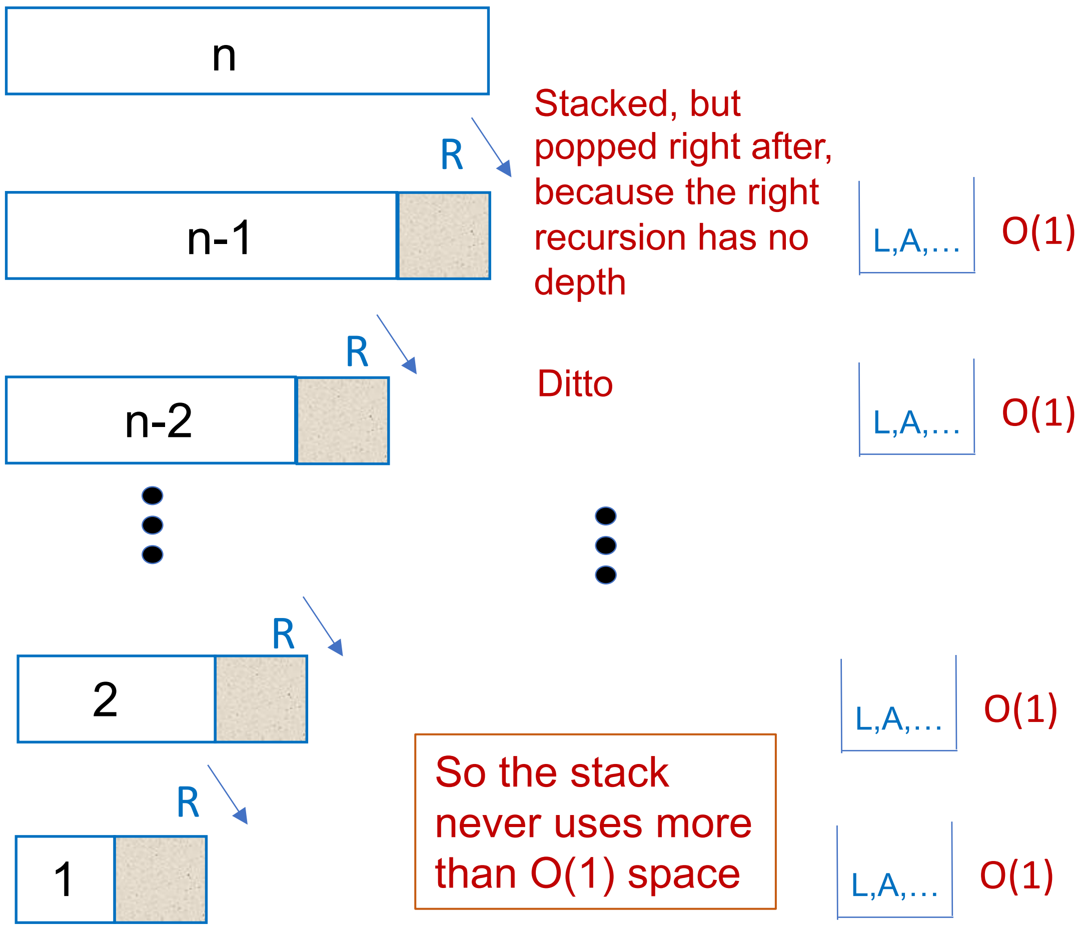

n
L
n-1
For eventual return to right recursion
L
n-2
For eventual return to right recursion
L
2
L
1
For eventual return to right recursion
R,A,…
O(1)
R,A,…
O(1)
R,A,…
O(1)
R,A,…
O(1)

n-1 items on stack,
total space used is O(n)

Recursion Stack Space Usage: Best Case

We could choose to do the right recursion first! (The left and right recursion calls are independent, so they can be done in either order.)

```
    splitPoint ← split(A,left,right)
R → quicksort(A,splitPoint+1,right)
L → quicksort(A,left,splitPoint-1)
```



n
R
n-1
Stacked, but popped right after, because the right recursion has no depth
L,A,…
O(1)
R
n-2
Ditto
L,A,…
O(1)
R
2
L,A,…
O(1)
R
1
L,A,…
O(1)

So the stack never uses more than O(1) space

Recursion Stack Space Usage: General Case

So what should we do in the general case to minimize stack space usage?

Answer: Always recurse on the smaller side first!

```
splitPoint ← split(A,left,right)

if ((splitPoint – left) < (right – splitPoint) then
    quicksort(A,left,splitPoint-1)
    quicksort(A,splitPoint+1,right)
else
    quicksort(A,splitPoint+1,right)
    quicksort(A,left,splitPoint-1)
endif
```

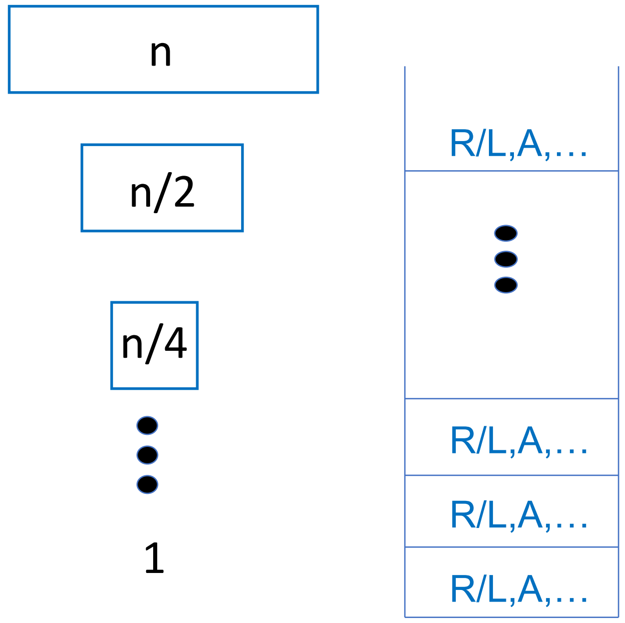
Recursion Stack Space Usage: General Case

For an array of size n, the smaller side's size can be at most (worst case) n/2-1 (let's assume n/2 for sake of analysis)

If we recurse on the smaller side first, in the next round the smaller side would be n/4, then n/8 ….

And each time we recurse, the address to which we return for the other side recursion is stacked (either for left side or right side, whichever is bigger)

We would hit bottom in log n steps, and log n would be the height of the stack – no path in the recursion tree can be more than log n deep



So the stack never uses more than O(log n) space

Quicksort with all these fine-tuning techniques

See Resources -> Apr 23 -> Quicksort.java for the implementation that includes

(1) picking median as pivot,
(2) using insertion sort when array gets too small, i.e. falls below a threshold (threshold is a parameter sent to quicksort)
(3) recursing on the smaller side first

(You will not be required to code any aspect of quicksort, this is just for your understanding of how everything comes together.)