NANYANG
TECHNOLOGICAL
UNIVERSITY

CE2101/ CZ2101: Algorithm Design and Analysis

Quicksort

Ke Yiping, Kelly

Learning Objectives

At the end of this lecture, students should be able to:

- Explain how “Divide and Conquer” approach is used in Quicksort
- Explain the pseudo code of Quicksort
- Manually execute Quicksort on an example input array
- Analyse time complexities of Quicksort in the best, average and worst cases

Quicksort

- Fastest general purpose in-memory sorting algorithm in the average case
- Implemented in Unix as qsort() which can be called in a program (see 'man qsort' for details)
- Main steps
  - Select one element in array as pivot
  - Partition list into two sublists with respect to pivot such that all elements in left sublist are less than pivot; all elements in right sublist are greater than or equal to pivot
  - Recursively partition until input list has one or zero element
- No merging is required because the pivot found during partitioning is already at its final position

↳ Since we do it recursively, every element should have a chance to be pivot.

Quicksort (Pseudo Code)

Quicksort (Pseudo Code)

```
void quicksort(int n, int m)
{
   int pivot_pos;
   if (n >= m)
       return;
   pivot_pos = partition(n, m);
   quicksort(n, pivot_pos - 1);
   quicksort(pivot_pos + 1, m);
}
```

n → start index

m → end index

(Do all the dirty work).

→此函数作用:找到n,m中的pivot position. 小于该pivot的移左,大于的移右. 最后返回pivot position.

n,m → 范围

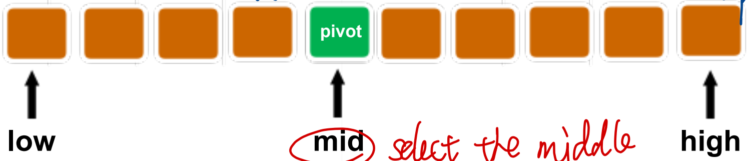
excludes pivot itself

NANYANG
TECHNOLOGICAL
UNIVERSITY

Partition Routine in Quicksort

Partition Routine in Quicksort

use this version !!! (select middle as pivot)



select the middle as pivot.

```
int partition(int low, int high)
{
  int i, last_small, pivot;
  int mid = (low+high)/2;
  swap(low, mid);
  pivot = slot[low];
  last_small = low;
```

traverse index

define boundry

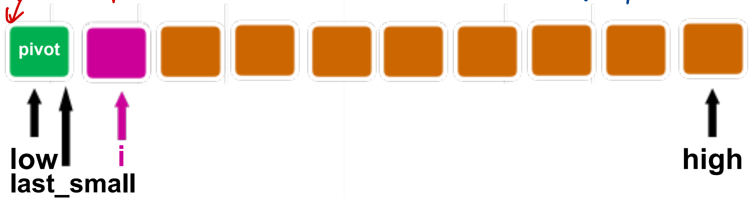
key value of pivot (not index).

point to the last index of the list containg "smallgup"

# move the pivot to begining of the list. (swap content, not index)

Partition Routine in Quicksort

after swap, pivot at first. (Don't need to check for pivot later on).



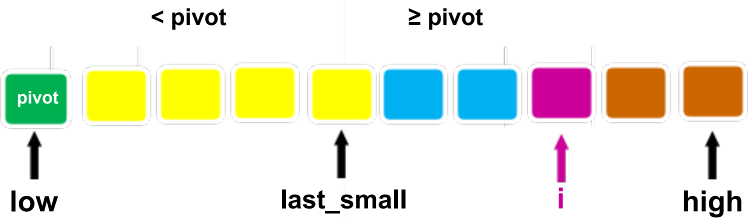
Initial State

※ Let's look at a general case to see if the code is correct.

```
int partition(int low, int high)
{.......
        for (i = low+1; i <= high; i++)
          if (slot[i] < pivot)
                swap(++last_small, i);
     swap(low, last_small);
       return last_small;
 }
```

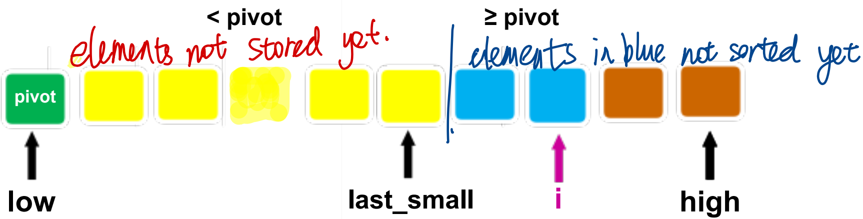
Partition Routine in Quicksort

General Case.



```
int partition(int low, int high)
{.......
        for (i = low+1; i <= high; i++)
          if (slot[i] < pivot)
                swap(++last_small, i);
     swap(low, last_small);
        return last_small;
 }
```

Partition Routine in Quicksort

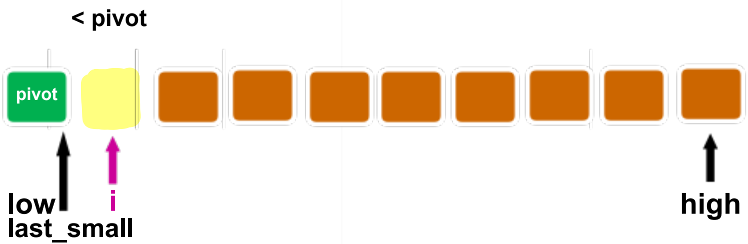


```
int partition(int low, int high)
{.......
        for (i = low+1; i <= high; i++)
          if (slot[i] < pivot)
               swap(++last_small, i);
       swap(low, last_small);
          return last_small;
 }
```

→ This make Quicksort unstable.

Might change the initial order of

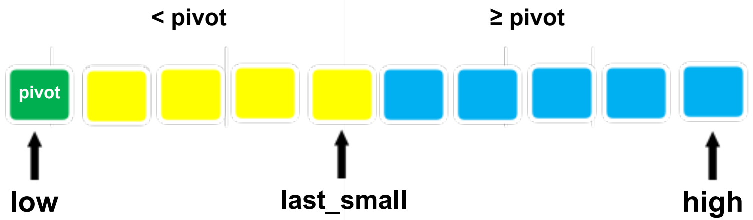
Partition Routine in Quicksort



```
int partition(int low, int high)
{.......
        for (i = low+1; i <= high; i++)
          if (slot[i] < pivot)
                swap(++last_small, i);
      swap(low, last_small);
         return last_small;
 }
```

如果第一个元素 <pivot.
进行一次 dummy swap. (可接受)

Partition Routine in Quicksort



```
int partition(int low, int high)
{…….
      for (i = low+1; i <= high; i++)
        if (slot[i] < pivot)
               swap(++last_small, i);
    swap(low, last_small);
       return last_small;
 }
```

Note:

- Loop terminates when *i* reaches high;
- swap **pivot** from position low to position last_small, to obtain the final position of pivot element.

Quicksort (Example)

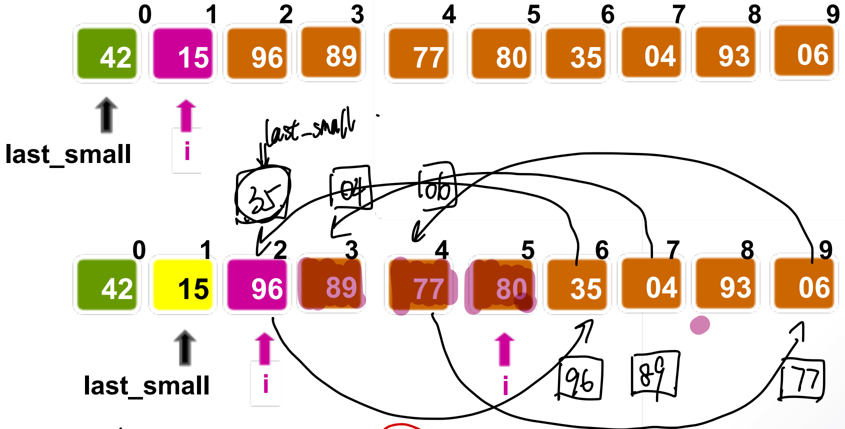
Quicksort (Example)

	0	1	2	3	4	5	6	7	8	9
Start	77	15	96	89	42	80	35	04	93	06
					↑ pivot					
Swap	77	15	96	89	42	80	35	04	93	06

Partition the elements …

Quicksort (Example)

Partitioning…



0	1	2	3	4	5	6	7	8	9
42	15	96	89	77	80	35	04	93	06
last_small	i								

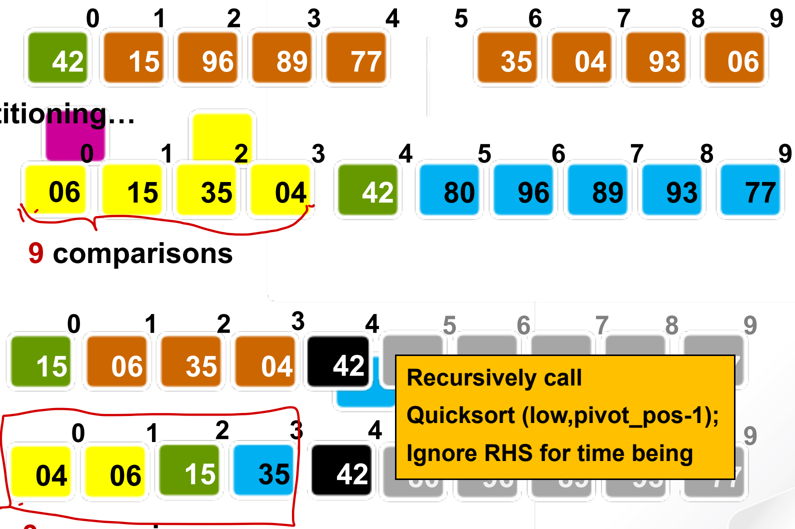
0	1	2	3	4	5	6	7	8	9
42	15	96	89	77	80	35	04	93	06
	last_small	i			i				

最终: 06 15 35 04 42 80 96 89 93 77

pivot

We have done 9 comparisons.

Quicksort (Example)



Step 1:

0	1	2	3	4	5	6	7	8	9
42	15	96	89	77		35	04	93	06

After partitioning…

0	1	2	3	4	5	6	7	8	9
06	15	35	04	42	80	96	89	93	77

9 comparisons

Step 2:

Swap

0	1	2	3	4	5	6	7	8	9
15	06	35	04	42					

Insert

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42					

3 comparisons

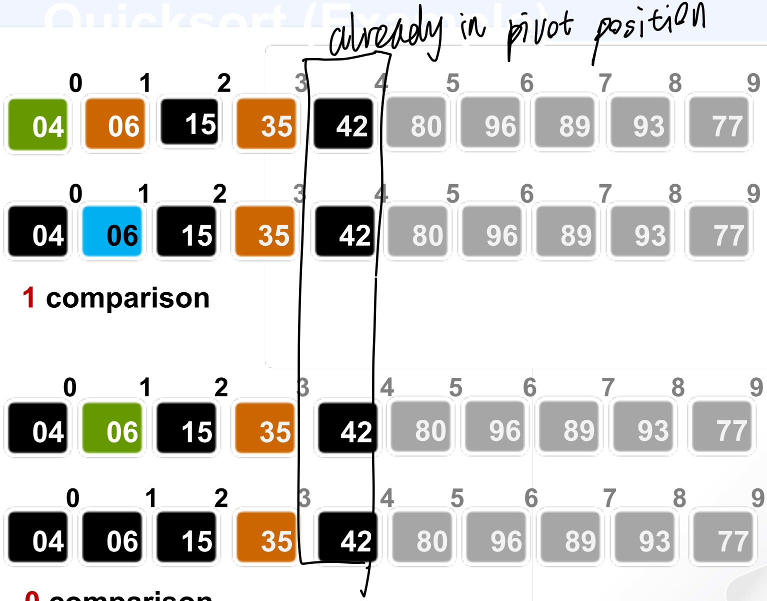
Recursively call

Quicksort (low,pivot_pos-1);

Ignore RHS for time being

Quicksort (Example)

already in pivot position



Step 3:

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	80	96	89	93	77

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	80	96	89	93	77

1 comparison

Step 4:

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	80	96	89	93	77

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	80	96	89	93	77

0 comparison

Quicksort (Example)

Step 5:

0	1	2	3	4	5	6	7	8	9
	06	15	35	42	80	96	89	93	77

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	80	96	89	93	77

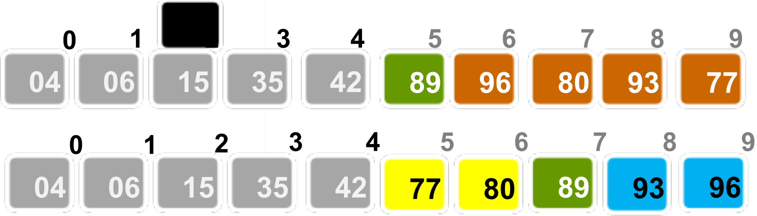
0 comparison

Sorting of LHS completed

Quicksort (Example)

Dealing with right half of the array:

Step 6:

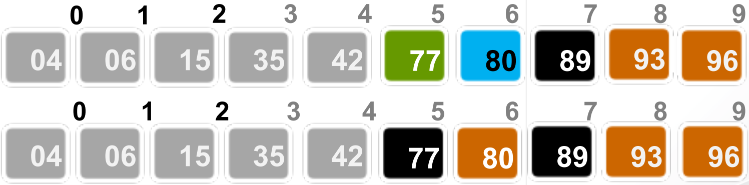


0	1	[redacted]	3	4	5	6	7	8	9
04	06	15	35	42	89	96	80	93	77

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

4 comparisons

Step 7:



0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

1 comparison

Quicksort (Example)

Dealing with right half of the array:

Quick Sort is not very efficient in smallsize.

Step 8:

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

0 comparison

Step 9:

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

1 comparison

Quicksort (Example)

Step 10:

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

0 comparison

Final outcome:

0	1	2	3	4	5	6	7	8	9
04	06	15	35	42	77	80	89	93	96

Comments on Quicksort

- **Which element of array should be pivot?** In this implementation, we take the middle element as pivot (other choices possible).
- Use quicksort(0, size – 1) to invoke quick sort; ‘size’ is the number of elements in array slot[ ].
- During partitioning, the middle element (pivot) is moved to the 1st position (i.e. slot[0]).
- A ‘for’ loop goes through the rest of array to split it into two portions.

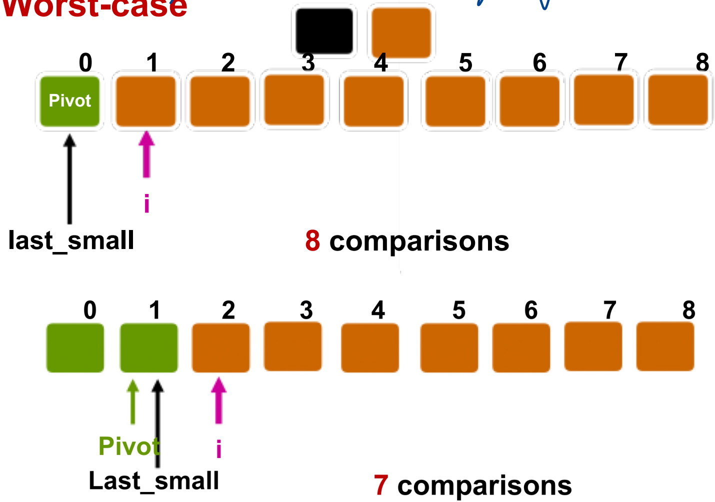
- Not efficient in sorting small dataset
- Usually used in hybrid mode
  - large size: QuickSort
  - small size: InsertionSort.

Quicksort’s Performance

Quicksort's Performance

Worst-case

→ pivot is the smallest/largest element in the sublist

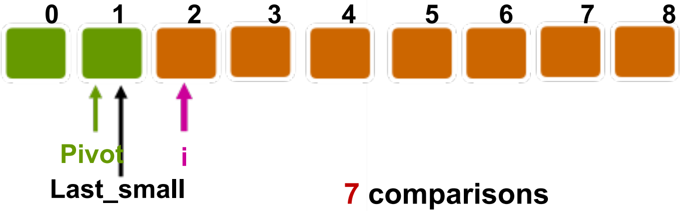


8 comparisons

7 comparisons

Quicksort's Performance

Worst-case



7 comparisons

Quicksort’s Performance

Worst case happens when the pivot does a bad job at splitting the array evenly, if pivot is the smallest or the largest key each time, then the total no. of key comparisons is O(*n*2).

$$\sum_{k=2}^{n}(k-1)=\sum_{k=1}^{n-1}k=\frac{n(n-1)}{2}$$

Quicksort's Performance

Best case happens when the pivot happens to divide the array into two sub-arrays of equal length, in every partitioning.

For simplicity, let's assume:

$n=2^k$

- $\cancel{n = 2k}$, i.e. $k = \lg n$.
- Each step, the pivot divides the array of length n into two sub-arrays each of length approximately $n/2$.

□□□□□ ... □ (n-1)

解: $T(n) = 2T(\frac{n}{2}) + cn$ → 为方便计算, n-1简化为n

$= 2\cdot[2T(\frac{n}{2^2}) + c\cdot\frac{n}{2}] + cn = 2^2T(\frac{n}{4}) + 2cn$

$= 4(2T(\frac{n}{8}) + c\cdot\frac{n}{4}) + 2cn = 8T(\frac{n}{8}) + 3cn$

$\vdots$

$= \boxed{2^kT(\frac{n}{2^k})} + kcn.$

$=0\cdot$

$\therefore T(n) = cn\cdot\lg n = n\lg n$

Recursive function:

n-1次 key comparison

$$\begin{cases} T(1) = 0 \\ T(n) = 2T(\frac{n}{2}) + c(n-1) \end{cases}$$

Quicksort's Performance

The recurrence equati[redacted]:

T(1) = 0,

T(n) = 2T(n/2) + cn, where c is a constant

T(n) = 2 (2T(n/4) + cn/2) + cn

= 22T(n/4) + 2cn

= 23T(n/8) + 3cn

…

= 2k T(n/2k) + kcn

= nT(1) + cnlgn = cnlgn

$\therefore \; T(n) = \Theta(n \lg n)$

Because n $\approx 2^k$, i.e. k = lg n, and T(1) = 0

Quicksort's Performance

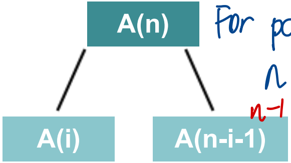
Average case: assume that the keys are distinct and that all permutations of the keys are equally likely.

k = no. of elements in the range of the array being sorted,

T(k), W(k)

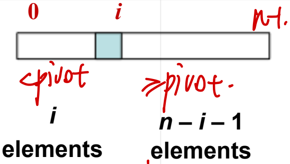
A(*k*) = no. of comparisons done for this range,

i = final position of the pivot, counting from 0,



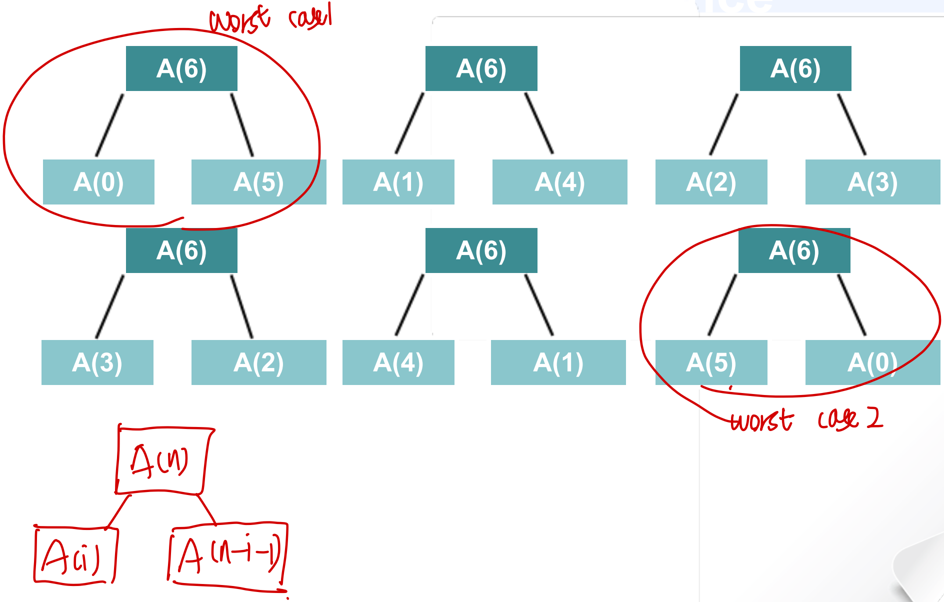
For partition function:

n element → n-1 comparisons.



(compare each value to pivot value).

Quicksort's Performance



Quicksort's Performance

Thus,

cost of partition for n=6.

$$A(6) = 5 + 1/6(\ \underline{A(0) + A(5)} + \underline{A(1) + A(4)} + \underline{A(2) + A(3)} + \ldots + \underline{A(5) + A(0)})$$

$$\to A(5) = 4 + 2[A(0)+A(4)] + 2[A(1)+A(3)] + A(2)+A(2)) \times \frac{1}{5}$$

$$A(0) = A(1) = 0$$

$$A(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}\left[A(i) + A(n-i-1)\right] = \Theta(n \lg n)$$

Proof is not required

Quicksort's Performance

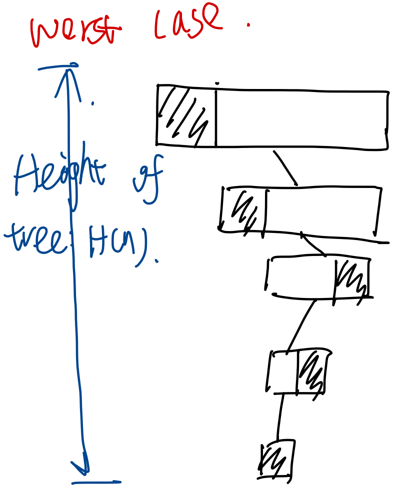
QuickSort usually used in hybrid-algo.

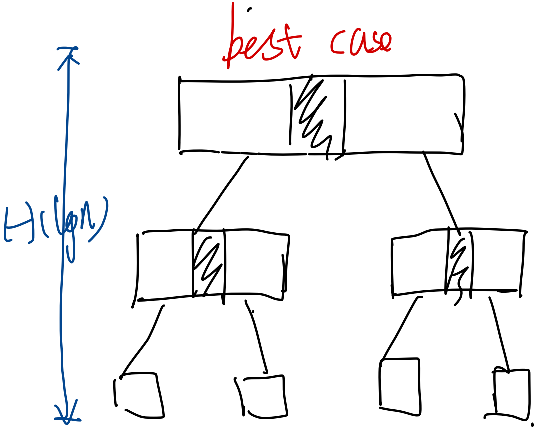
- Strengths:
  - Fast on average

    Constant subsumed by big-O notation. $\Theta(n\lg n)$. (Why unique?)
  - No merging required
  - Best case occurs when pivot always splits array into equal halves
- Weaknesses:
  - Poor performance when pivot does not split the array evenly
  - Quicksort also performs badly when the size of list to be sorted is small
  - If more work is done to select pivot carefully, the bad effects can be reduced

    pivot can be sorted more wisely

Recursive Tree





$H(n)\cdot O(n) = O(n^2).$

$H(\lg n)\cdot O(n) = O(n\lg n).$

Summary

- Quicksort uses the “Divide and Conquer” approach.
- Partition function splits an input list into two sub-lists by comparing all elements with the pivot:
  - Elements in the left sub-list are < pivot and
  - Elements in the right sub-list are ≥ pivot.
- Quicksort is called recursively on each sub-list.
- The worst-case time complexity of Quicksort is □(*n*2).
- The best-case and average-case time complexities of Quicksort are both □(*n*lg*n*).