# Lecture 10.3 : Quicksort (part 2)¶

## Partitioning walkthrough

- We want to partition the list `[9, 7, 5, 11, 12, 2, 14, 3, 10, 6]`.
- Initially:

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 9 | 7 | 5 | 11 | 12 | 2 | 14 | 3 | 10 | 6 |
| Index | q | | | | | | | | | |
| Index | j | | | | | | | | | |

- `9 > 6` : increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 9 | 7 | 5 | 11 | 12 | 2 | 14 | 3 | 10 | 6 |
| Index | q | | | | | | | | | |
| Index | | j | | | | | | | | |

- `7 > 6` : increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 9 | 7 | 5 | 11 | 12 | 2 | 14 | 3 | 10 | 6 |
| Index | q | | | | | | | | | |
| Index | | | j | | | | | | | |

- `5 < 6` : swap `A[j]` with `A[q]`, increment `q`, increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 5 | 7 | 9 | 11 | 12 | 2 | 14 | 3 | 10 | 6 |
| Index | | q | | | | | | | | |
| Index | | | | j | | | | | | |

- `11 > 6` : increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 5 | 7 | 9 | 11 | 12 | 2 | 14 | 3 | 10 | 6 |
| Index | | q | | | | | | | | |
| Index | | | | | j | | | | | |

- `12 > 6` : increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 5 | 7 | 9 | 11 | 12 | 2 | 14 | 3 | 10 | 6 |
| Index | | q | | | | | | | | |
| Index | | | | | | j | | | | |

- `2 < 6` : swap `A[j]` with `A[q]`, increment `q`, increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|

| Array | 5 | 2 | 9 | 11 | 12 | 7 | 14 | 3 | 10 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| Index | | | q | | | | | | | |
| Index | | | | | | | j | | | |

- `14 > 6` : increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 5 | 2 | 9 | 11 | 12 | 7 | 14 | 3 | 10 | 6 |
| Index | | | q | | | | | | | |
| Index | | | | | | | | j | | |

- `3 < 6` : swap `A[j]` with `A[q]`, increment `q`, increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 5 | 2 | 3 | 11 | 12 | 7 | 14 | 9 | 10 | 6 |
| Index | | | | q | | | | | | |
| Index | | | | | | | | | j | |

- `10 > 6` : increment `j`

| Index | p | | | | | | | | | r |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 5 | 2 | 3 | 11 | 12 | 7 | 14 | 9 | 10 | 6 |
| Index | | | | q | | | | | | |
| Index | | | | | | | | | | j |

- `j == r` : swap `A[r]` with `A[q]`

| Index | p | | | | | | | | r | |
|---|---|---|---|---|---|---|---|---|---|---|
| Array | 5 | 2 | 3 | 6 | 12 | 7 | 14 | 9 | 10 | 11 |
| Index | | | | q | | | | | | |
| Index | | | | | | | | | j | |

## Partitioning summary (pivot highlighted)

- Partitioning [9, 7, 5, 11, 12, 2, 14, 3, 10, 6] yields [5, 2, 3, **6**, 12, 7, 14, 9, 10, 11].
- Partitioning [5, 2, 3] yields [2, **3**, 5].
- [2] does not require partitioning as it contains a single element.
- [3] does not require partitioning as it contains a single element.
- Partitioning [12, 7, 14, 9, 10, 11] yields [7, 9, 10, **11**, 14, 12].
- Partitioning [7, 9, 10] yields [7, 9, **10**].
- Partitioning [7, 9] yields [7, **9**].
- [7] does not require partitioning as it contains a single element.
- Partitioning [14, 12] yields [**12**, 14].
- [14] does not require partitioning as it contains a single element.
- Since partitioning happens *in-place* we do not need to recombine sublists and the list is sorted.

## Implementing partitioning

- The following code partitions a list:

```
def partition(A, p, r):

    # q and j start at p
    q = j = p

    # up to but not including pivot
    while j < r:

        # move values less than or equal to pivot and update q
        if A[j] <= A[r]:
            A[q], A[j] = A[j], A[q]
            q += 1

        j += 1

    # swap pivot with A[q]
    A[q], A[r] = A[r], A[q]

    # return pivot index
    return q
```

## Coding quicksort

- Remember, quicksort is a *recursive* sorting algorithm. It repeatedly calls *partition* until there is nothing left to partition. The code looks like this:

```
# recursively partition list until sorted
def quicksort(A, p, r):

    # return if nothing to sort (zero or one element)
    if r <= p:
        return

    # partition and sort left and right sublists
    q = partition(A, p, r)
    quicksort(A, p, q-1)
    quicksort(A, q+1, r)
```

## Timing quicksort

- In the example below we place 10,000 randomly generated integers in a list. We sort them by two methods: quicksort and selection sort and verify the results match:

```
>>> import random
>>> from quicksort_102 import quicksort
>>> from selectionsort_102 import selectionsort
>>> A = random.sample(range(-9999, 10000), 10000)
>>> B = A[:]
>>> quicksort(A, 0, len(A)-1)
>>> selectionsort(B)
>>> A == B
True
```

- If you run the above code you will find that quicksort is significantly faster than selection sort. We can use Python's `timeit` module to time our code. Let's use it to time quicksort:

---

```
import timeit

def main():

    init = """
import random
from quicksort_112 import quicksort
A = random.sample(range(-99999, 100000), 10000)
"""

    s = """
quicksort(A, 0, len(A)-1)
"""

    count = 5

    times = timeit.repeat(s, init, repeat=count, number=1)

    print('{:.2f}s'.format(sum(times)/count))
```

- The above code requires some explanation. The `timeit.repeat()` function requires four parameters:

    1. The first parameter is the command we wish to time. This is specified in `s` and is a call to our `quicksort()` function.
    2. The second parameter is any initialisation we wish to carry out before the execution of our command. Our initialisation code is specified in the `init` string. It imports required functions and initialises the list we wish to sort to some random contents. The time taken to carry out the initialisation is excluded from the results returned by `timeit`.
    3. The third parameter is the number of times we wish to repeat our experiment (including the initialisation step). We repeat the experiment five times. All five times are returned in a list. We print their average.
    4. The fourth parameter is the number of runs of the command we wish to carry out per experiment. Because our command requires reinitialisation between experiments we set this number to one.
- Running the above code shows it takes quicksort a fraction of a second (the result will vary from machine to machine) to sort the list of 10,000 integers:

```
$ python3 time_me.py
0.05s
```

- Now let's use `timeit` to time selection sort:

```
import timeit

def main():

    init = """
import random
from selectionsort_112 import selectionsort
A = random.sample(range(-99999, 100000), 10000)
"""

    s = """
selectionsort(A)
"""

    times = timeit.repeat(s, init, repeat=count, number=1)
```

```
    print('{:.2f}s'.format(sum(times)/count))
```

- Running the above code shows it takes selection sort more than eleven seconds (the result will vary from machine to machine) to sort a list of 10,000 integers:

```
$ python3 time_me.py
11.85s
```

- What effect would doubling the size of the sorting problem to 20,000 integers have on the running time of selection sort? The time complexity of selection sort is $O(n^2)$. That means its running time is proportional to the square of the number of items to be sorted. Since the running time of our original experiment is proportional to $n^2$, the running time of our second experiment is proportional to $(2n)^2 = 4n^2$. Thus we would predict the running time to quadruple to 47.4s.
- If we run the experiment we observe an average running time of:

```
$ python3 time_me.py
48.12s
```

- Thus the predicted running time and measured running time are reasonably close.
- What effect would doubling the size of the sorting problem have on the running time of quicksort? The time complexity of quicksort is $O(n \cdot lg(n))$. That means its running time is proportional to the product of the number of items to be sorted and the log (to the base 2) of the number of items to be sorted.
- If t1 is the running time of our first experiment (10,000 items to be sorted) and t2 is the running time of the second (20,000 items to be sorted) then:

$$\frac{t2}{t1} = \frac{20000 \cdot lg(20000)}{10000 \cdot lg(10000)} = 2.15$$

- This means our predicted running time for sorting 20,000 items is: $0.05 \cdot 2.15 = 0.107s$.
- If we run the experiment we observe an average running time of:

```
$ python3 time_me.py
0.11s
```

- Pretty close.