Timing Assignment

Analysis

Worst Case

```
1  int kthLargest (int* array,  int N, int k)
2  {
3      int low = 0;              // O(1)
4      int high = N-1;           // O(1)
5
6      while( low < high )       // cond: O(1)
7      {
8        int start = low+1;      // O(1)
9        int stop = high;        // O(1)
10       while (start < stop)
11       {
12         if (array[start] > array[low])  // cond: O(1)
13           ++start;            // O(1)
14         else if (array[stop] <= array[low]) // cond: O(1)
15           --stop;             // O(1)
16         else
17         {
18           std::swap (array[start], array[stop]); // O(1)
19         }
20       }
21       if (array[start] < array[low])             // cond: O(1)
22         --start;              // O(1)
23       std::swap (array[low], array[start]);      // O(1)
24       if (start == k)
25         low = high = k;       // O(1)
26       else if (start > k)
27       {
28         high = start - 1;     // O(1)
29       }
30       else
31       {
32         low = start+1;        // O(1)
33       }
34     }
35     return array[low];        // O(1)
```

```
36 }
```

Evaluate condition block from lines 14-19 and then 12-19:

$$\begin{aligned} t_{if} &= O(1) + max(O(1), O(1)) \\ &= O(1 + 1) \\ &= O(1) \end{aligned}$$

```
1          ...
2          if (array[start] > array[low])  // cond: O(1)
3            ++start;            // O(1)
4          else if (array[stop] <= array[low]) // cond: O(1) total: O(1)
5            --stop;             // O(1)
6          else
7          {
8            std::swap (array[start], array[stop]); // O(1)
9          }
10         ...
```

Collapse

```
1          ...
2          if (array[start] > array[low])  // cond: O(1)
3            ++start;            // O(1)
4          else
5            // O(1)
6          ...
```

Evaluate

$$\begin{aligned} t_{if} &= O(1) + max(O(1), O(1)) \\ &= O(1 + 1) \\ &= O(1) \end{aligned}$$

Collapse

```
1  int kthLargest (int* array,  int N, int k)
2  {
3      int low = 0;              // O(1)
4      int high = N-1;           // O(1)
5
6      while( low < high )       // cond: O(1)
7      {
8        int start = low+1;      // init: O(1)
```

```
9       int stop = high;        // init: O(1)
10      while (start < stop)    // cond: O(1) body: O(1) #: distance(start,
stop)
11      {
12        if (array[start] > array[low])      // cond: O(1) total: O(1)
13      }
14      if (array[start] < array[low])        // cond: O(1)
15        --start;              // O(1)
16      std::swap (array[low], array[start]); // O(1)
17      if (start == k)
18        low = high = k;       // O(1)
19      else if (start > k)
20      {
21        high = start - 1;     // O(1)
22      }
23      else
24      {
25        low = start+1;        // O(1)
26      }
27    }
28    return array[low];        // O(1)
29  }
```

Evaluate `while` loop from lines 10-13

$$
\begin{aligned}
t_{while} &= O(1) + \sum_{i=start}^{stop} (O(1) + O(1) + O(1)) \\
&= O(1) + O(distance(start, stop) * (O(1) + O(1) + O(1)) \\
&= O(1) + O(distance(start, stop)) \\
&= O(distance(start, stop)
\end{aligned}
$$

```
1   int kthLargest (int* array,  int N, int k)
2   {
3       int low = 0;              // O(1)
4       int high = N-1;           // O(1)
5
6       while( low < high )       // cond: O(1)
7       {
8         int start = low+1;      // init: O(1)
9         int stop = high;        // init: O(1)
10        while (start < stop)    // cond: O(1) body: O(1) #: distance(start,
stop) total: O(distance(start, stop)
11
12        if (array[start] < array[low])        // cond: O(1)
```

```
13          --start;                // O(1)
14        std::swap (array[low], array[start]); // O(1)
15        if (start == k)
16          low = high = k;         // O(1)
17        else if (start > k)
18        {
19          high = start - 1;       // O(1)
20        }
21        else
22        {
23          low = start+1;          // O(1)
24        }
25      }
26      return array[low];          // O(1)
```

Collapsing loop:

```
1   int kthLargest (int* array,  int N, int k)
2   {
3       int low = 0;                // O(1)
4       int high = N-1;             // O(1)
5
6       while( low < high )
7       {
8         int start = low+1;        // init: O(1)
9         int stop = high;          // init: O(1)
10        // O(distance(start, stop)
11
12        if (array[start] < array[low])        // cond: O(1)
13          --start;                // O(1)
14        std::swap (array[low], array[start]); // O(1)
15        if (start == k)
16          low = high = k;         // O(1)
17        else if (start > k)
18        {
19          high = start - 1;       // O(1)
20        }
21        else
22        {
23          low = start+1;          // O(1)
24        }
25      }
26      return array[low];          // O(1)
```

Evaluate condition block from lines 12-13:

$$
\begin{aligned}
t_{if} &= O(1) + max(O(1), O(1)) \\
&= O(1 + 1) \\
&= O(1)
\end{aligned}
$$

And collapsing

```
1  int kthLargest (int* array,  int N, int k)
2  {
3      int low = 0;                // O(1)
4      int high = N-1;             // O(1)
5
6      while( low < high )
7      {
8        int start = low+1;        // init: O(1)
9        int stop = high;          // init: O(1)
10       // O(distance(start, stop)
11
12       if (array[start] < array[low])        // cond: O(1) total: O(1)
13
14       std::swap (array[low], array[start]); // O(1)
15       if (start == k)
16         low = high = k;         // O(1)
17       else if (start > k)       // cond: O(1)
18       {
19         high = start - 1;       // O(1)
20       }
21       else
22       {
23         low = start+1;          // O(1)
24       }
25     }
26     return array[low];          // O(1)
```

Evaluate condition block from lines 17-24 and then 15-24:

$$
\begin{aligned}
t_{if} &= O(1) + max(O(1), O(1)) \\
&= O(1) + O(1) \\
&= O(1)
\end{aligned}
$$

```
...
      if (start == k)
        low = high = k;        // O(1)
      else if (start > k)      // cond: O(1) total: O(1)
      {
        high = start - 1;      // O(1)
      }
      else
      {
        low = start+1;         // O(1)
      }
...
```

Collapse 17-24

```
...
      if (start == k)
        low = high = k;        // O(1)
      else
        // O(1)
...
```

Evaluate

$$\begin{aligned} t_{if} &= O(1) + max(O(1), O(1)) \\ &= O(1) + O(1) \\ &= O(1) \end{aligned}$$

Collapse

```
int kthLargest (int* array,  int N, int k)
{
    int low = 0;               // O(1)
    int high = N-1;            // O(1)

    while( low < high )
    {
      int start = low+1;       // init: O(1)
      int stop = high;         // init: O(1)
      // O(distance(start, stop)

      if (array[start] < array[low])        // total: O(1)
        --start;               // O(1)
```

```
14        std::swap (array[low], array[start]); // O(1)
15        if (start == k)          // total: O(1)
16      }
17      return array[low];         // O(1)
```

Sum `while` body and replace start/stop on line 10 with low+1 and high respectively.

$$\begin{aligned} t_{body} &= O(1+1+distance(low+1, high)+1+1+1) \\ &= O(distance(low, high)) \end{aligned}$$

```
1  int kthLargest (int* array,  int N, int k)
2  {
3      int low = 0;               // init: O(1)
4      int high = N-1;            // init: O(1)
5      while( low < high )
6      // body: O(distance(low, high))
7      return array[low];         // O(1)
```

Evaluate `while` loop

$$\begin{aligned} t_{while} &= O(1) + O(1) + \sum_{i=0}^{N-1}(O(distance(0, N-1))) \\ &= O(1+1+N^2) \\ &= O(N^2) \end{aligned}$$

```
1  int kthLargest (int* array,  int N, int k)
2    total: O(N^2)
```

Worst case complexity of `kthLargest` $= O(N^2)$

Average Case

```
1  int kthLargest (int* array,  int N, int k)
2  {
3      int low = 0;               // O(1)
4      int high = N-1;            // O(1)
5
6      while( low < high )        // cond: O(1) #: N-1
7      {
8        int start = low+1;       // O(1)
9        int stop = high;         // O(1)
```

```
10        while (start < stop)
11        {
12          if (array[start] > array[low])  // cond: O(1)
13            ++start;            // O(1)
14          else if (array[stop] <= array[low]) // cond: O(1)
15            --stop;             // O(1)
16          else
17          {
18            std::swap (array[start], array[stop]); // O(1)
19          }
20        }
21        if (array[start] < array[low])             // cond: O(1)
22          --start;              // O(1)
23        std::swap (array[low], array[start]);      // O(1)
24        if (start == k)
25          low = high = k;       // O(1)
26        else if (start > k)
27        {
28          high = start - 1;     // O(1)
29        }
30        else
31        {
32          low = start+1;        // O(1)
33        }
34      }
35      return array[low];        // O(1)
36  }
```

Evaluate condition block from lines 14-19 and then 12-19:

Since both the `if` and `else` portions are $O(1)$ we don't need to figure the average here.

```
1           ...
2           if (array[start] > array[low])  // cond: O(1)
3             ++start;            // O(1)
4           else if (array[stop] <= array[low]) // cond: O(1) total: O(1)
5             --stop;             // O(1)
6           else
7           {
8             std::swap (array[start], array[stop]); // O(1)
9           }
10          ...
```

Collapse

```
        ...
        if (array[start] > array[low])  // cond: O(1)
          ++start;              // O(1)
        else
          // O(1)
        ...
```

Evaluate

Since both the `if` and `else` portions are $O(1)$ we don't need to figure the average here.

Collapse

```
int kthLargest (int* array,  int N, int k)
{
    int low = 0;                // O(1)
    int high = N-1;             // O(1)

    while( low < high )
    {
      int start = low+1;        // init: O(1)
      int stop = high;          // init: O(1)
      while (start < stop)      // cond: O(1) body: O(1)
      {
        if (array[start] > array[low])      // cond: O(1) total: O(1)
      }
      if (array[start] < array[low])        // cond: O(1)
        --start;                // O(1)
      std::swap (array[low], array[start]); // O(1)
      if (start == k)
        low = high = k;         // O(1)
      else if (start > k)
      {
        high = start - 1;       // O(1)
      }
      else
      {
        low = start+1;          // O(1)
      }
    }
    return array[low];          // O(1)
}
```

The inner while loop will still have to iterate over each item regardless of the data distribution just like it did for the worst case scenario, so:

$$\begin{aligned} t_{while} &= O(1) + \sum_{i=start}^{stop} (O(1) + O(1) + O(1)) \\ &= O(1) + O(distance(start, stop) * (O(1) + O(1) + O(1)) \\ &= O(1) + O(distance(start, stop)) \\ &= O(distance(start, stop) \end{aligned}$$

Important Note before collapsing:

- Assuming that an average array will be unsorted
- There will be on average an even mix of numbers that are greater than and less than the first number in the array
- Therefore, on average, `start` will finish the loop roughly halfway through the array.

Collapsing:

```
1  int kthLargest (int* array,  int N, int k)
2  {
3      int low = 0;                // O(1)
4      int high = N-1;             // O(1)
5
6      while( low < high )
7      {
8        int start = low+1;        // init: O(1)
9        int stop = high;          // init: O(1)
10       while (start < stop)      // cond: O(1) body: O(1) #: distance(start,
   stop) total: O(distance(start, stop))
11       if (array[start] < array[low])        // cond: O(1)
12         --start;                // O(1)
13       std::swap (array[low], array[start]); // O(1)
14       if (start == k)
15         low = high = k;         // O(1)
16       else if (start > k)
17       {
18         high = start - 1;       // O(1)
19       }
20       else
21       {
22         low = start+1;          // O(1)
23       }
24     }
25     return array[low];          // O(1)
26 }
```

The conditionals from 11-12, 16-23, and 14-23 still run in constant time as well, so for each of those:

$$\begin{aligned} t_{if} &= O(1) + max(O(1), O(1)) \\ &= O(1 + 1) \\ &= O(1) \end{aligned}$$

Another Note before collapsing:

- As stated previously, `start` on average will be at roughly the midpoint in the array when reaching these conditionals.
- This means that whether `start > k` or `start < k`, we will end up with roughly half of the elements in the array for the next iteration of the outer loop.

Collapsing:

```
1  int kthLargest (int* array,  int N, int k)
2  {
3      int low = 0;               // O(1)
4      int high = N-1;            // O(1)
5
6      while( low < high )
7      {
8        int start = low+1;       // init: O(1)
9        int stop = high;         // init: O(1)
10       // total: O(distance(start, stop))
11       // total: O(1)
12       std::swap (array[low], array[start]); // O(1)
13       // total: O(1)
14     }
15     return array[low];         // O(1)
16 }
```

For lines 7-14 we can collapse and replace the `start` and `stop` values of the inner while assesment with `low + 1` and `high`.

$$\begin{aligned} O(distance(start, stop)) &= O(distance(low + 1, high)) \\ &= O(distance(low, high)) \end{aligned}$$

```
int kthLargest (int* array,  int N, int k)
{
    int low = 0;              // O(1)
    int high = N-1;           // O(1)

    while( low < high ) // cond: O(1)
    // body: O(distance(low, high))
    return array[low];        // O(1)
}
```

As noted earlier, the array in each iteration on average will be half the size of the previous iteration. This leads to the conclusion that it will be logarithmic in nature. Since `low` and `high` are initialled as `0` and `N - 1` immediately before the loop, we can go ahead and make the substitutions in and declare that the loop will repeat `log N` times.

$$\begin{aligned} \#ofiterations &= O(log(distance(low, high))) \\ &= O(log(distance(0, N-1))) \\ &= O(log(distance(0, N))) \\ &= O(logN) \end{aligned}$$

Similarly we can substitute in `N` for the body of the loop as well:

$$\begin{aligned} body &= O(distance(low, high)) \\ &= O(distance(0, N-1)) \\ &= O(distance(0, N)) \\ &= O(N) \end{aligned}$$

```
int kthLargest (int* array,  int N, int k)
{
    int low = 0;              // O(1)
    int high = N-1;           // O(1)

    while( low < high ) // cond: O(1) body: O(N) #: log N total: N log N
    return array[low];        // O(1)
}
```

Summing up the remaining sequential statements:

$$\begin{aligned} total &= O(1 + 1 + NlogN + 1) \\ &= O(NlogN) \end{aligned}$$

Therefore, the average case complexity of kthLargest is O(N log N)

Conclusions

After performing time tests on the algorithm, the actual run times fell in between the worst case analysis and the best case analysis; however, it did tend to run much closer to the worst case. Because of this, I do feel that the actual performance does support the predicted complexity overall. To be sure I had properly bracketed the function, even though the worst case did have a slightly downward trending slope, I chose a slightly larger function of $N^2 logN$ to ensure that it's slope was a true downward slope ratio with time.

I do think that the average case should have been the one that ran closer to actual times. I feel part of the reason for the discrepancy is that the average case treats a random array as being balanced between lower and higher numbers than the first being evenly dispersed between both half of the array. While this may hold more or less true on average, I think that anything outside of perfectly balanced has a great effect on the runtime.