B. Analysis

Analysis of worst case time complexity:

The main for loop runs N times because there are N numbers in the start list. The "find smallest" part is a loop, too, and it runs N times, then N-1 times, then N-2 times, and so forth down to 2 and then 1.

\[
\text{total steps} = 1 + 2 + 3 + 4 + \ldots + (N-2) + (N-1) + N
\]

Do the Gauss method to prove this is \( \frac{N(N+1)}{2} \) which is \( \frac{N^2 + N}{2} \) which is \( \frac{N^2 + N}{2} \)

C. Swapping

Given: \( x = 2; y = 7 \)

<table>
<thead>
<tr>
<th>Classic method</th>
<th>Addition method</th>
<th>XOR method</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{temp} = x )</td>
<td>( x = x + y )</td>
<td>( x = x \oplus y )</td>
</tr>
<tr>
<td>( x = y )</td>
<td>( y = x - y )</td>
<td>( y = x \oplus y )</td>
</tr>
<tr>
<td>( y = \text{temp} )</td>
<td>( x = x - y )</td>
<td>( x = x \oplus y )</td>
</tr>
</tbody>
</table>

\( \oplus \) stands for Exclusive OR and is a bitwise operation:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a\oplus b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Apply this rule to every column of bits:

\[
\begin{align*}
    x & = 101 \\
    y & = 110 \\
    \oplus & = 011 \\
    \oplus & = 110 \\
    \oplus & = 001 \\
    \oplus & = 101 \\
    \oplus & = 110
\end{align*}
\]

Python adds a notational shortcut:

\[
    x, y = y, x
\]

What is happening here is that \( x, y \) creates a tuple, same as \( (y, x) \)

Then you can unpack a tuple by giving as many variables as it has parts:

\[
\begin{align*}
\text{tup} = ("Mark", 59, "cats", False) \\
\text{name, age, pets, is_jesuit} = \text{tup}
\end{align*}
\]

\[
\begin{align*}
\text{name} &= \text{tup}[0] \\
\text{age} &= \text{tup}[1] \\
\text{pets} &= \text{tup}[2] \\
\text{is_jesuit} &= \text{tup}[3]
\end{align*}
\]

It unpacks each part into the corresponding variable, so \( x, y, x \) will be same as:

\[
\begin{align*}
\text{tup} &= (y, x) \\
\text{x} &= \text{tup}[0] \\
\text{y} &= \text{tup}[1]
\end{align*}
\]
1. def insertionSort(numbers):
2.     i=0
3.     j=0
4.     temp=0
5.     for i in range(1,len(numbers)):
6.         j = i
7.         while j > 0 and numbers[j] < numbers[j - 1]:
8.             numbers[j], numbers[j-1] = numbers[j-1], numbers[j]
9.             j -= 1
10.    
11.    values = [56, 27, 13, 6, 99, 18]
12.    insertionSort(values)
13.    print(values)
numbers

numbers[i], numbers[k] = numbers[k], numbers[i]
partition \( (\text{numbers}, i, k) \):

\[
\text{midpoint} = i + \frac{(k - i}){2} \quad \text{(integer division, throw away anything after the decimal)}
\]

\[
\text{pivot} = \text{numbers}[\text{midpoint}]
\]

\[
\text{lower} = i
\]

\[
\text{higher} = k
\]

keep doing the following:

\[
\text{while } \text{lower} < \text{pivot:}
\]

add 1 to lower

\[
\text{while pivot < numbers[higher]:}
\]

subtract 1 from higher

if lower is greater than higher
then you’re done, so return higher

otherwise
swap the values at \( \text{numbers}[\text{lower}] \) with \( \text{numbers}[\text{higher}] \)
add 1 to lower
subtract 1 from higher

You can only get out of this loop when lower and higher cross

The essential idea is to move all the values less than the pivot to the left of it (the lower indexes of the array) and all the values greater than the pivot to the right of it (the upper indexes.)

But we want to do this without wasting memory. We want to reuse the same array, which is important if the numbers array is extremely large.

quicksort\( (\text{number}, i, k) \):

\[
j = 0
\]

return if \( i >= k \)

\[
j = \text{partition}(\text{numbers}, i, k)
\]

\[
\text{quicksort}(\text{numbers}, i, j)
\]

\[
\text{quicksort}(\text{numbers}, j+1, k)
\]

quicksort doesn’t return anything because it sorts its list of numbers in place.
(Quicksort trace) partition function: i 0  k 8

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>89</td>
<td>13</td>
<td>65</td>
<td>50</td>
<td>15</td>
<td>66</td>
<td>72</td>
<td>24</td>
<td>89</td>
<td>67</td>
</tr>
</tbody>
</table>

lower | midpoint | pivot | higher
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>50</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

partition function: i ________  k ________

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

lower | midpoint | pivot | higher
def partition(numbers):
    newlist_lower = []
    newlist_higher = []
    midpoint = len(numbers) // 2
    pivot = numbers[midpoint]

    for i in range(len(numbers)):
        if numbers[i] < pivot:
            newlist_lower.append(numbers[i])

        if numbers[i] > pivot:
            newlist_higher.append(numbers[i])

    return (newlist_lower, pivot, newlist_higher)

def quicksort(numbers):
    if len(numbers) < 2:
        return numbers
    (lower, pivot, higher) = partition(numbers)
    lower = quicksort(lower)
    higher = quicksort(higher)
    return lower + [pivot] + higher

#mylist = [30, 10, 20]
mylist = [56, 10, 20, 99, 13, 87, 40, 72, 66, 5, 80]
(x, pivot, y) = partition(mylist)
print("lower=",x)
print("higher=",y)
newlist = quicksort(mylist)
print (newlist)
1. def insertionSort(numbers):
2.     i=0
3.     j=0
4.     temp=0
5.     for i in range(1,len(numbers)):
6.         j = i
7.         while j > 0 and numbers[j] < numbers[j - 1]:
8.             numbers[j], numbers[j-1] = numbers[j-1], numbers[j]
9.             j -= 1
10. 
11.     values = [56,27,13,6,99,18]
12.     insertionSort(values)
13.     print(values)
1. def binarySearch(numbers, key):
2.     assert type(numbers) is list, "1st arg must be a list"
3.     mid = 0; low = 0; high = 0
4.     high = len(numbers)-1
5.     while high >= low:
6.         mid = (high + low) // 2
7.         if numbers[mid] < key:
8.             low = mid + 1
9.         elif numbers[mid] > key:
10.            high = mid - 1
11.        else:
12.            return mid
13.     return -1  # not found
14. 
15. def main():
16.     numbers = [2, 4, 7, 9, 10, 11, 32, 45, 87, 99, 100,
17.        105, 112, 126, 129, 130, 131, 132, 435,
18.        499, 501, 516, 525, 572, 583, 594, 600]
19.     key = int(input("Enter a value: "))
20.     keyIndex = binarySearch(numbers, key)
21.     if keyIndex == -1:
22.         print (key,"was not found")
23.     else:
24.         print ("found", key, "at index",keyIndex)
25. main()
1. def binarySearch(numbers, key):
2.     assert type(numbers) is list, "1st arg must be a list"
3.     mid = 0; low = 0; high = 0
4.     high = len(numbers) - 1
5.     while high >= low:
6.         mid = (high + low) // 2
7.         if numbers[mid] < key:
8.             low = mid + 1
9.         elif numbers[mid] > key:
10.            high = mid - 1
11.        else:
12.            return mid
13.     return -1 #not found
14. def main():
15.     numbers = [2, 4, 7, 9, 10, 11, 32, 45, 87, 99, 100, 105, 112, 126, 129, 130, 131, 132, 143, 499, 501, 516, 525, 572, 583, 594, 600]
16.     key = int(input("Enter a value: "))
17.     keyIndex = binarySearch(numbers, key)
18.     if keyIndex == -1:
19.         print (key,"was not found")
20.     else:
21.         print ("found", key, "at index", keyIndex)
22.     main()