CSE 321 HW3 REPORT

1-) If we have $2n$ boxes starting from $0\ to\ 2n-1$, If we change index 1 box with index $2n-2$, first and last pair will be in required order and we end up with $2n-4$ boxes in the middle that not ordered with required order. We reduce problem size with 4 boxes. We can derive this recurrence relation from code and above explanation:

$$T(2n) = T(2n-4) + 1$$

- 1 is for swapping, and $2n-4$ represent next problem size
- We have two initial condition one is if we have only black and white box we don't need to swap anything, boxes are ordered. If we have 4 boxes, this means we need to do only one swap. This conditions are

$$M(2) = 0$$
$$M(4) = 1$$

- If we substitute this equation:

$$T(2n) = T(2n-4) + 1$$
$$T(2n) = T(2n-8) + 2$$
$$T(2n) = T(2n-12) + 3$$

After k step

$$T(2n) = T(2n-4k) + k$$

For $2n-4k = 2$, k will be $\frac{n-1}{2}$ and equation will be

$$T(2n) = T(2) + \frac{n-1}{2}$$
$$T(2n) = \frac{n-1}{2}$$

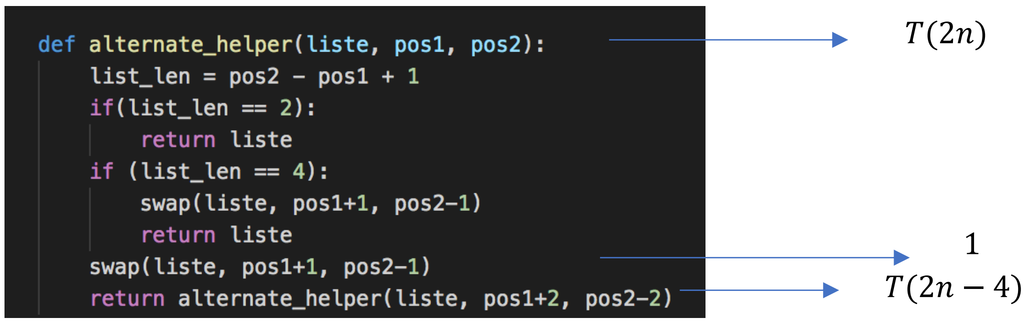
$$T(2n)\ \epsilon\ \theta(n)$$

For $2n-4k = 4$, k will be $\frac{n-1}{2}$ and equation will be

$$T(2n) = T(4) + \frac{n-1}{2}$$
$$T(2n) = 1 + \frac{n-2}{2}$$

$$T(2n)\ \epsilon\ \theta(n)$$

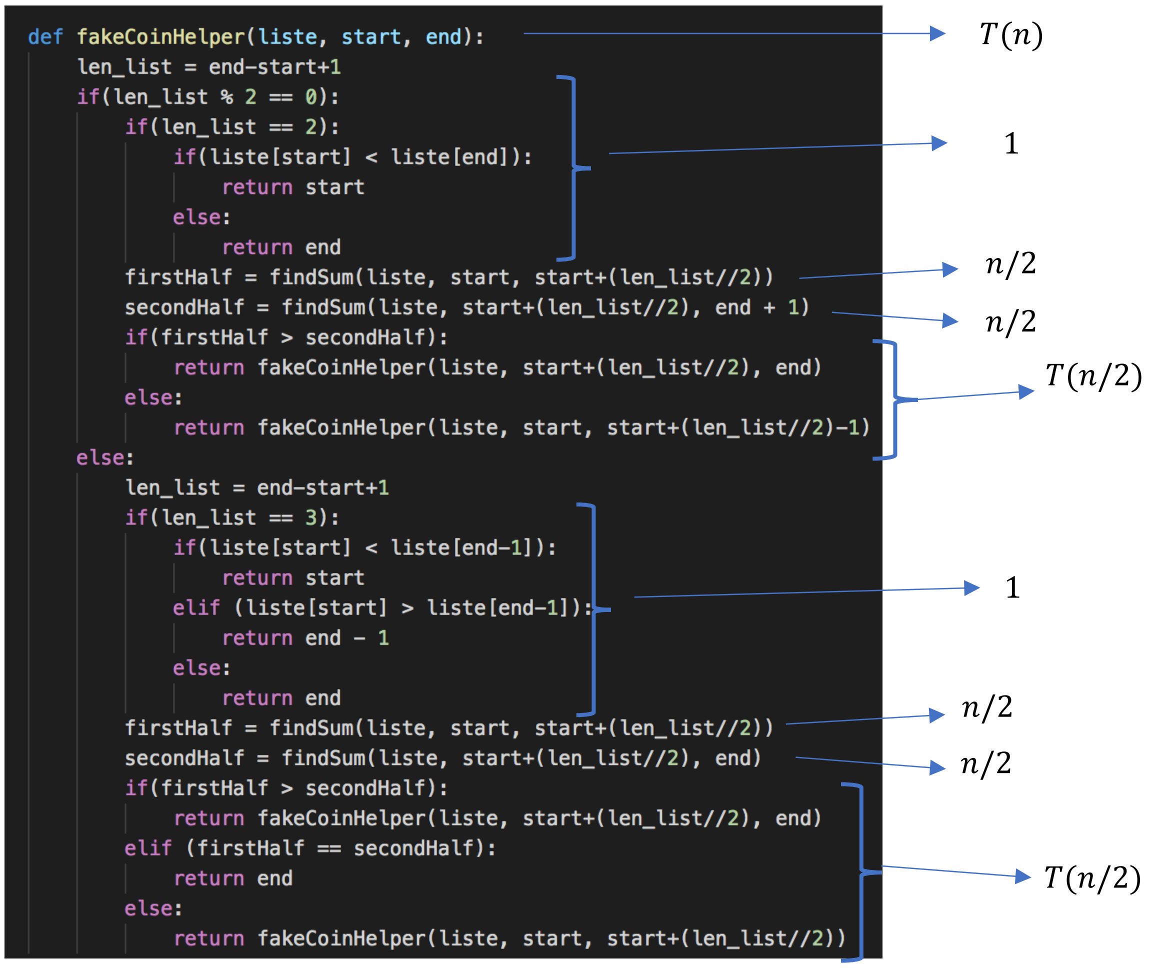
All cases will be $T(2n)\ \epsilon\ \theta(n)$, we will do all swappings for all size boxes.



```python
def alternate_helper(liste, pos1, pos2):
    list_len = pos2 - pos1 + 1
    if(list_len == 2):
        return liste
    if (list_len == 4):
        swap(liste, pos1+1, pos2-1)
        return liste
    swap(liste, pos1+1, pos2-1)
    return alternate_helper(liste, pos1+2, pos2-2)
```

In code i used an helper to give initial condition. I write a swap function for swapping boxes.

2-) In Fake Coin Problem, I asssumed fake is lighter than real one. I divided coins into two half. If we have even number coin, we can equally divide coins but if we have have odd number of coin, right half will have 1 additional coin. For even number coins, I sum two half and compared their weights. If left half heavier than right half this means we need to look right half for fake coin. Otherwise we need to look at other half for fake coin. For odd number of coins, I ignored last indexed coin. After that I do same process since we have even numbers of coin currently. Again, If one of them heavier this means we need to look other half. But for odd number of coins, if two half has equal weight, this means last indexed coin is fake coin. If not, we have even number of coins from one the halfs. We can do same process with even coins.



```python
def fakeCoinHelper(liste, start, end):
    len_list = end-start+1
    if(len_list % 2 == 0):
        if(len_list == 2):
            if(liste[start] < liste[end]):
                return start
            else:
                return end
        firstHalf = findSum(liste, start, start+(len_list//2))
        secondHalf = findSum(liste, start+(len_list//2), end + 1)
        if(firstHalf > secondHalf):
            return fakeCoinHelper(liste, start+(len_list//2), end)
        else:
            return fakeCoinHelper(liste, start, start+(len_list//2)-1)
    else:
        len_list = end-start+1
        if(len_list == 3):
            if(liste[start] < liste[end-1]):
                return start
            elif (liste[start] > liste[end-1]):
                return end - 1
            else:
                return end
        firstHalf = findSum(liste, start, start+(len_list//2))
        secondHalf = findSum(liste, start+(len_list//2), end)
        if(firstHalf > secondHalf):
            return fakeCoinHelper(liste, start+(len_list//2), end)
        elif (firstHalf == secondHalf):
            return end
        else:
            return fakeCoinHelper(liste, start, start+(len_list//2))
```

For even numbers, in all cases we need to divide coins until we reach size 2, and in this simle case we can compare two coin, we can decide which is fake.
For even number of coins we have this relation from above.

$$T(n) = T\left(\frac{n}{2}\right) + \frac{n}{2} + \frac{n}{2} + 1$$
$$T(n) = T\left(\frac{n}{2}\right) + n + 1$$

From master theorem $a = 1,\ b = 2\ and\ f(n) = n + 1$

$$f(n) = \theta(n^1)$$
$$1 > \log_2 1$$
$$T(n) = \theta(n)$$

For odd number of coins, there is a case that last coin can be fake coin. If this is the case we don't need recursive steps and complexity will be:

$$T(n) = 1 + \frac{n}{2} + \frac{n}{2} + 1$$
$$T(n) = n + 2$$
$$T(n) = \theta(n)$$

If this is not the case, we need to recursive steps and this is the same process as in even number of coins.

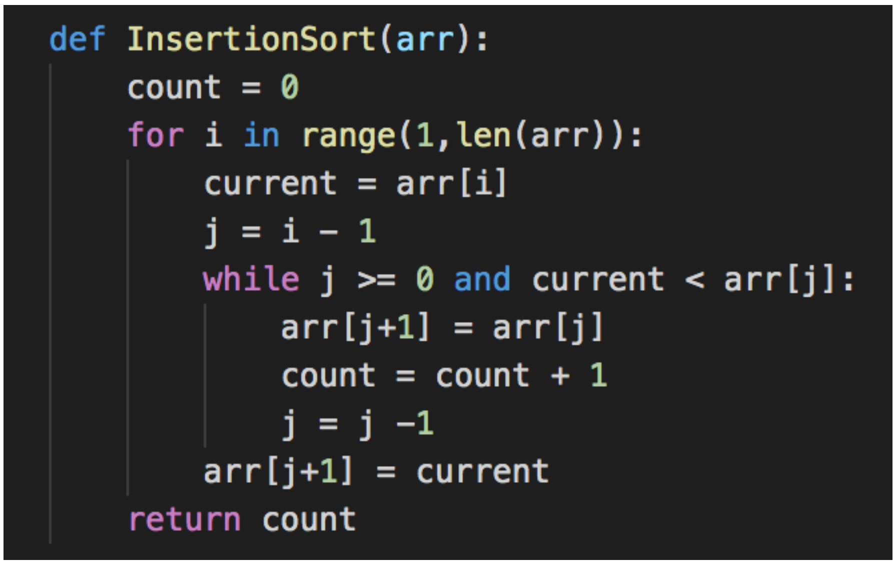
$$T(n) = T\left(\frac{n}{2}\right) + \frac{n}{2} + \frac{n}{2} + 1$$
$$T(n) = T\left(\frac{n}{2}\right) + n + 1$$

And from master theorem we will again end up with

$$T(n) = \theta(n)$$

So in all cases we have $\theta(n)$ complexity.

3-) For the insertion sort



```python
def InsertionSort(arr):
    count = 0
    for i in range(1,len(arr)):
        current = arr[i]
        j = i - 1
        while j >= 0 and current < arr[j]:
            arr[j+1] = arr[j]
            count = count + 1
            j = j -1
        arr[j+1] = current
    return count
```

The outer loop will always execute $\theta(n)$ times but inner loop execution depends on items on array. If the array sorted, there will be no execution of inner loop and no swapping.
Analysis of average case of insertion start with number of basic operations.
Let T_i is the number of basic operation at step i, $1 \leq i \leq n-1$
and $T = T_1 + T_2 + T_3 \ldots \ldots \ldots \ldots + T_n$ and average case will be equal to the

$$A(n) = E[T] = E[T = T_1 + T_2 + T_3 \ldots \ldots \ldots \ldots + T_n]$$

We can find $E[T_i]$ like that

$$E[T_i] = \sum_{j=1}^{i} j.p(T_i = j) \quad where\ p(T_i = j)\ probability\ of\ j\ comparison\ in\ step\ i$$

- 1 comparison will occur if $x = L[i] > L[i-1] \Longrightarrow \frac{1}{i+1}$
- 2 comparison will occur if $L[i-2] < x < L[i-1] \Longrightarrow \frac{1}{i+1}$
- 3 comparison will occur if $L[i-3] < x < L[i-2] \Longrightarrow \frac{1}{i+1}$
- .
- .
- .
- i comparison will occur if $L[1] < x < L[2] \Longrightarrow \frac{1}{i+1}$
- i comparison will occur if $x < L[1] \Longrightarrow \frac{1}{i+1}$

There are i+1 cases because i+1 intervals that x can fall in.

$$p(T_i = j) = \begin{cases} \dfrac{1}{i+1} & if\ 1 \leq j \leq i-1 \\ \dfrac{2}{i+1} & if\ j = i \end{cases}$$

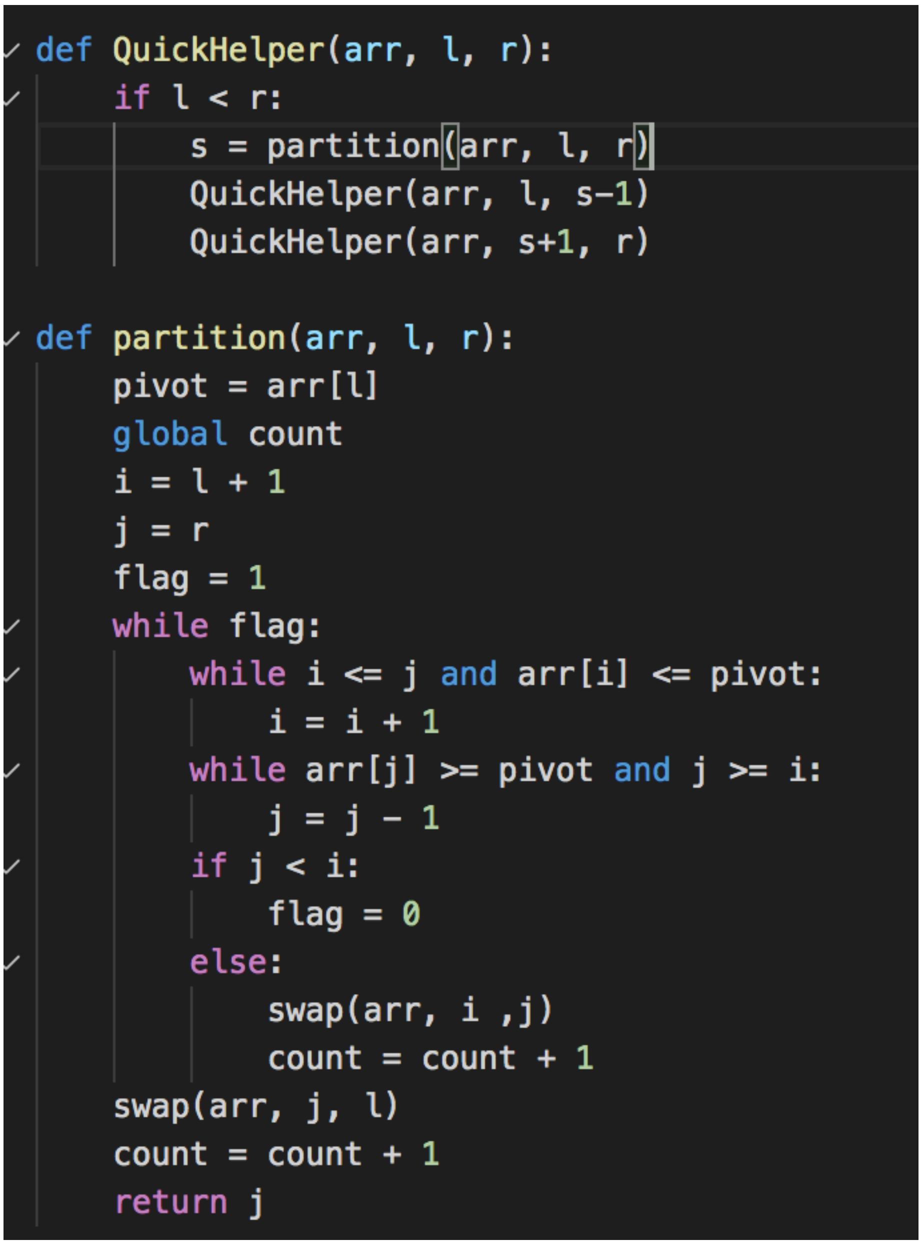
$$\mathrm{E}[T_i] = \left(\sum_{j=1}^{i-1} j.\frac{1}{i+1}\right) + \frac{2i}{i+1} = \frac{i(i+3)}{2(i+1)}$$

$$\mathrm{A(n)} = \mathrm{E[T]} = \sum_{i=1}^{n-1} E[T_i] = \sum_{i=1}^{n-1} (\frac{i}{2} + 1 - \frac{1}{i+1} = \frac{n(n-4)}{4} + (n-1) - H(n)$$

$$\boldsymbol{H(n) = \theta(logn)}$$

$$\boldsymbol{A(n) = \theta(n^2)}$$

Fort he QuickSort average case analysis



```
def QuickHelper(arr, l, r):
    if l < r:
        s = partition(arr, l, r)
        QuickHelper(arr, l, s-1)
        QuickHelper(arr, s+1, r)

def partition(arr, l, r):
    pivot = arr[l]
    global count
    i = l + 1
    j = r
    flag = 1
    while flag:
        while i <= j and arr[i] <= pivot:
            i = i + 1
        while arr[j] >= pivot and j >= i:
            j = j - 1
        if j < i:
            flag = 0
        else:
            swap(arr, i ,j)
            count = count + 1
    swap(arr, j, l)
    count = count + 1
    return j
```

$$T = T_1 + T_2$$

Number of operations in rearrange   Number of operations in recursive calls

$$A(n) = E[T] = E[T_1] + E[T_2]$$

$$E[T_1]\ will\ be \rightarrow\ r - l + 2 = n + 1$$

$$E[T_2] = \sum_x E\left[T_2 \mid \underbrace{\bar{x}}_{\substack{randrom\ variable \\ for\ pivot\ position}} = x\right] . \underbrace{p(\bar{x} = x)}_{\frac{1}{n}}$$

$$\mathrm{A(n)} = (n+1) + \sum_{i=1}^{n} (A[i-1] + A(n-i) . \frac{1}{n}$$

$$\underbrace{= (n+1) + \begin{bmatrix} A(0) + (n-1) \\ +A(1) + A(n-2) \\ +A(2) + A(n-3) \\ . \\ . \\ . \\ A(n-2) + A(1) \\ +A(n-1) + A(0) \end{bmatrix} . \frac{1}{n}}_{2[A(0)+A(1)+\cdots+A(n-1)]}$$

$$= (\mathrm{n}+1) + \frac{2}{n}[A(0) + A(1) + \cdots + A(n-1)]$$

$$\begin{array}{rl} & \mathrm{n.A(n) = n(n+1)} + 2[A(0) + A(1) + \cdots + A(n-1)] \\ - & (n-1).A(n-1) = n.(n-1) + 2[A(0) + A(1) + \cdots + A(n-2)] \\ \hline & n.A(n) - (n-1).A(n-1) = 2n + 2A(n-1) \end{array}$$

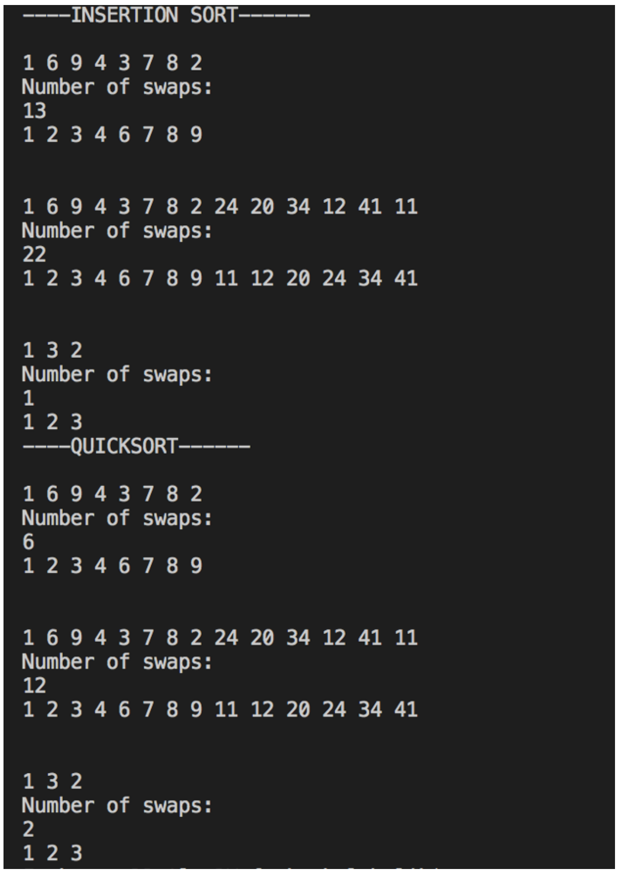
$$\frac{1}{n.(n+1)} . \left\{ \frac{A(n)}{n+1} - \frac{A(n-1)}{n} = \frac{2}{n+1} \right\} \Longrightarrow \frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1}$$

$$Change\ variable\ t(n) = \frac{A(n)}{n+1}\ and\ can\ be\ reduced\ to\ t(n) = t(n-1) + \frac{2}{n+1}$$

$$by\ backwar\ substituiton\ t(n) = \sum_{i=2}^{n} \frac{2}{i+1}$$

$$t(n) = 2\underbrace{H(n+1)}_{\substack{Harmonic \\ Series}} - 3$$

$$A(n) = 2(n+1).H(n+1) - 3(n+1) \in\ \theta(nlogn)$$

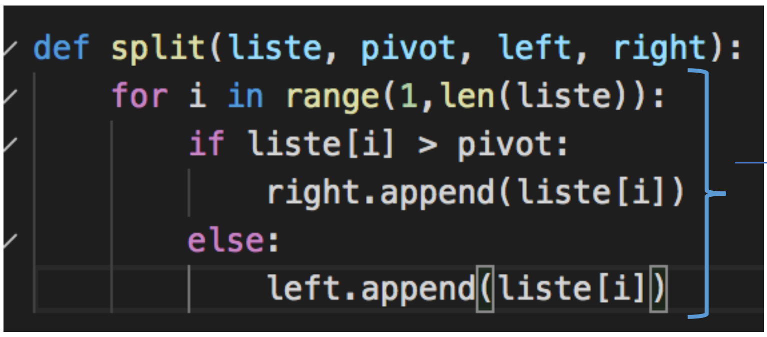


```
----INSERTION SORT------

1 6 9 4 3 7 8 2
Number of swaps:
13
1 2 3 4 6 7 8 9


1 6 9 4 3 7 8 2 24 20 34 12 41 11
Number of swaps:
22
1 2 3 4 6 7 8 9 11 12 20 24 34 41


1 3 2
Number of swaps:
1
1 2 3
----QUICKSORT------

1 6 9 4 3 7 8 2
Number of swaps:
6
1 2 3 4 6 7 8 9


1 6 9 4 3 7 8 2 24 20 34 12 41 11
Number of swaps:
12
1 2 3 4 6 7 8 9 11 12 20 24 34 41


1 3 2
Number of swaps:
2
1 2 3
```

For the small number size arrays, insertion sort is more stable and has less number of swaps. As seen in inputs, small sized arrays has less swaps for insertion sort but when the array size increased, number of swaps in insertion sort greater than quicksort. As expected, when we put size on complexity equations, quicksort has less number of swaps, but if we put this for small array like size 3, this equations not hold but complexity deals with large size inputs, so this totally normal.

4-) Finding median of list requires finding kth smallest element of list. To find median we can find middle smallest element for odd size list or we can find average of two middle element of even size array.

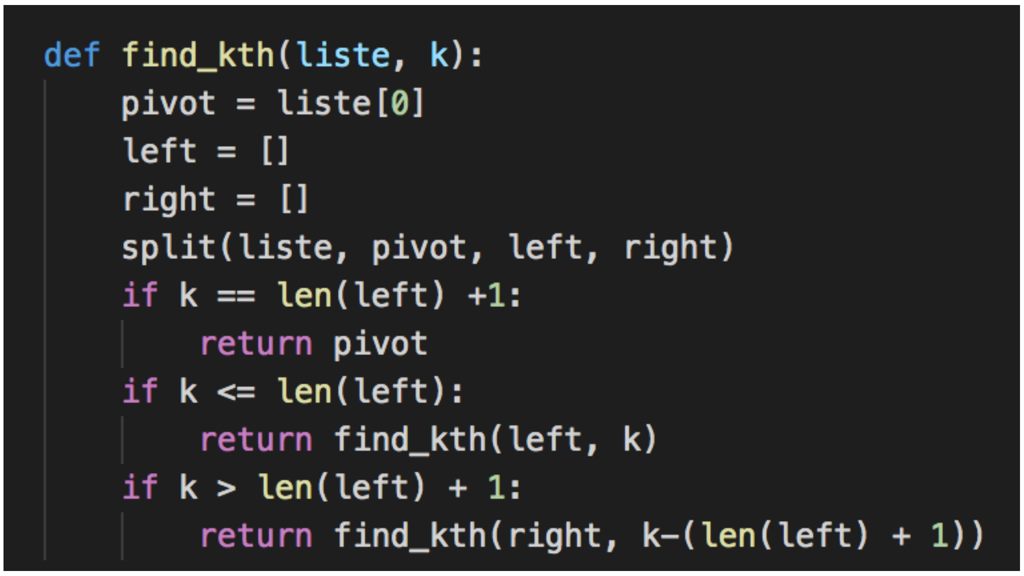
Process of finding requires splitting list into two half. Left half elements will be less than pivot and right half elements will be greater than pivot. We always choose pivot as first element of currently processed list. If left half size +1 is equal k, this means we find position for kth element if list is sorted and this is the kth smallest element. If k less than left half size this means that we need look left half, kth smallest element in left half. If kth smalles element is in right half(i.e k greater than left size + 1) we need to look right half and since we decrease problem size we need decrease k with proper index.



```
def split(liste, pivot, left, right):
    for i in range(1,len(liste)):
        if liste[i] > pivot:
            right.append(liste[i])
        else:
            left.append(liste[i])
```

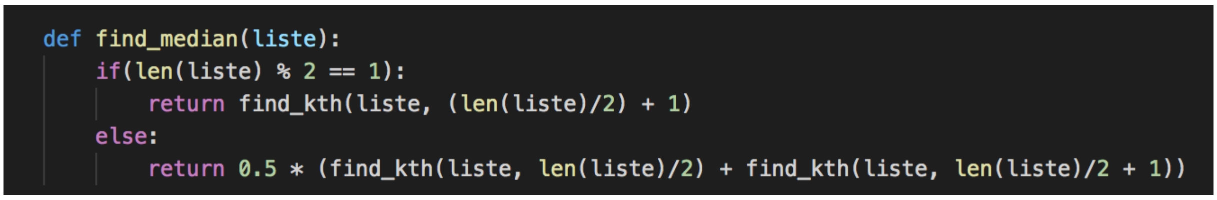
$\xrightarrow{\quad} \theta(n)$

Splitting requires $\theta(n)$ complexity.



```python
def find_kth(liste, k):
    pivot = liste[0]
    left = []
    right = []
    split(liste, pivot, left, right)
    if k == len(left) +1:
        return pivot
    if k <= len(left):
        return find_kth(left, k)
    if k > len(left) + 1:
        return find_kth(right, k-(len(left) + 1))
```

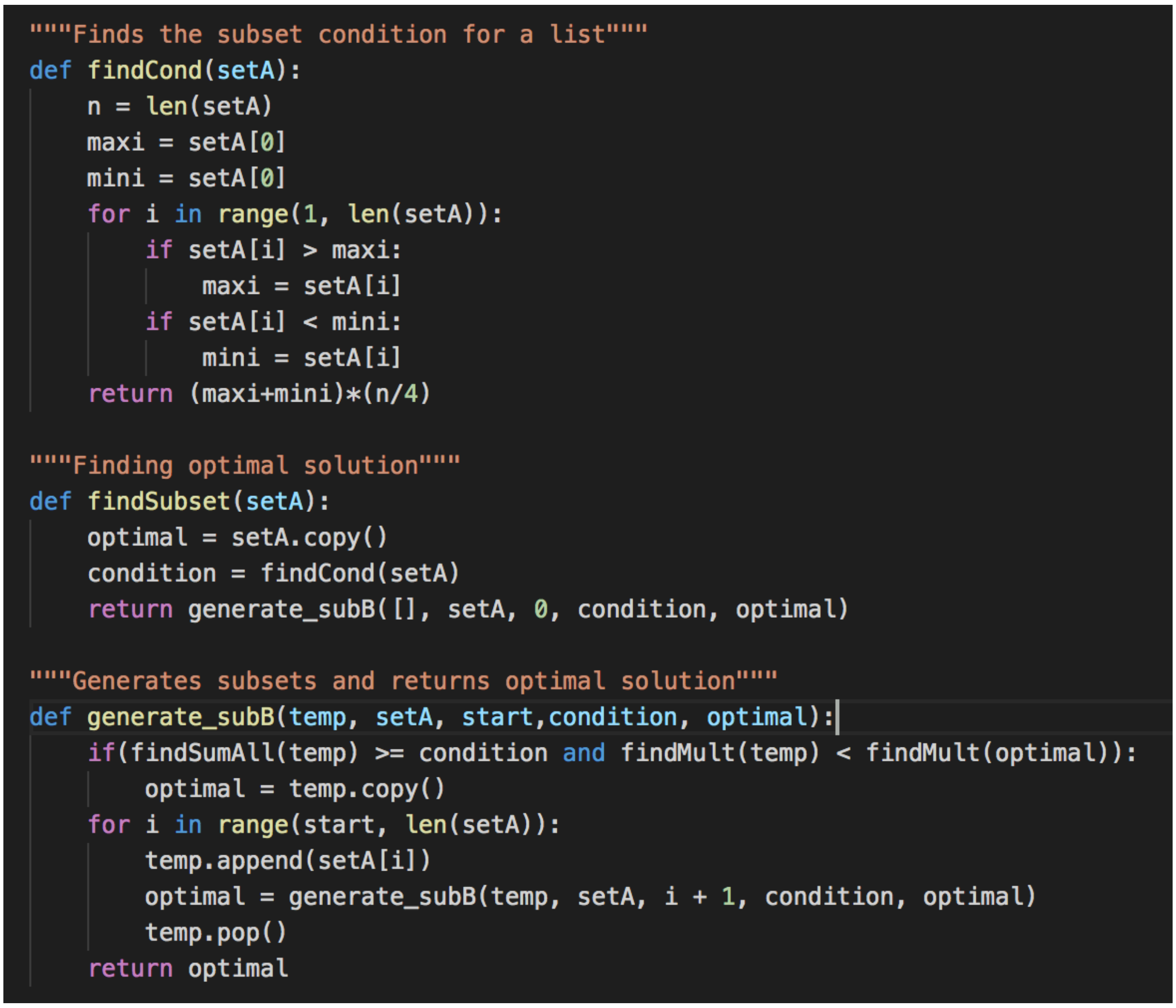
Finding kth smallest element, in best case we can find correct place for pivot and this is the our kth smalles element. This is only $\theta(n)$. For other cases, in worst case partion can split array into two halves and one of them can be empty. In this case this is going to $O(n^2)$ but for average this will be $O(n)$. In best and average case this will be $O(n)$, but for worst case this will be $O(n^2)$.



```python
def find_median(liste):
    if(len(liste) % 2 == 1):
        return find_kth(liste, (len(liste)/2) + 1)
    else:
        return 0.5 * (find_kth(liste, len(liste)/2) + find_kth(liste, len(liste)/2 + 1))
```

For finding median we can use finding kth smallest element and this element will be middle element for odd size list. Finding median of even size list we first find left middle and then right middle. Average of this medians will give us median of list. For odd size list finding median will be $O(n)$ for best and and average case and $O(n^2)$ for the worst case. For even size list finding median will be $O(n) + O(n) = O(n)$ fort he best and average case and $O(n^2) + O(n^2) = O(n^2)$ for the worst case.

5-)



```python
"""Finds the subset condition for a list"""
def findCond(setA):
    n = len(setA)
    maxi = setA[0]
    mini = setA[0]
    for i in range(1, len(setA)):
        if setA[i] > maxi:
            maxi = setA[i]
        if setA[i] < mini:
            mini = setA[i]
    return (maxi+mini)*(n/4)

"""Finding optimal solution"""
def findSubset(setA):
    optimal = setA.copy()
    condition = findCond(setA)
    return generate_subB([], setA, 0, condition, optimal)

"""Generates subsets and returns optimal solution"""
def generate_subB(temp, setA, start,condition, optimal):
    if(findSumAll(temp) >= condition and findMult(temp) < findMult(optimal)):
        optimal = temp.copy()
    for i in range(start, len(setA)):
        temp.append(setA[i])
        optimal = generate_subB(temp, setA, i + 1, condition, optimal)
        temp.pop()
    return optimal
```

In this problem, I created a function that finds all subsets. Generate_SubB function takes an empty set as an initial set and also takes all set. Our problem is finding optimal, so I assumed optimal is whole set as initial. After executing function, it checks currently processed subset is optimal for given condition. If it is, it makes current subset new optimal subset. In main functionality of function, finds subset and give it to the next call. In a subset for every element there is two choice. Element is either in subset or not. Also to cover optimal solution, i returned finded solution and stored that.

In complexity analysis of that function findSumAll and findMult's cost is $\theta(n)$.
For loop decreases problem size with $T(n-i)$ and do this for $n-1$ times. From there we can find this formula.

$$T(n) = \sum_{i=1}^{n-1} T(n-i) + n + 1$$

Let’s find this summation

$$m(n) = \sum_{i=1}^{n-1} m(n-i) = m(n-1) + m(n-2) + \cdots. + m(1)$$

Our initial condition $T(1) = 2$ and $T(0) = 1$

$$m(n-1) = \sum_{i=1}^{n-2} T(n-i) = m(n-2) + m(n-3) + \cdots. + m(1)$$

We end up with:

$$m(n) = 2m(n-1)$$

And this is equal to the $O(2^n)$

From there

$$T(n) = O(2^n) + n + 1$$

$$T(n) = O(2^n)$$