Task 2.1.1

```
def allocate(shops, revenue, d, current_idx = None, prev_idx = None):

    if current_idx == None:
        return "Error"

    if not prev_idx:
        dist = d
    else:
        dist = shops[prev_idx] - shops[current_idx]

    if current_idx == 0:
        if dist < d:
            return 0, []
        return revenue[current_idx], [shops[current_idx]]

    including_current_revenue = 0
    including_current_shops = []
    excluding_current_revenue, excluding_current_shops = allocate(shops, revenue, d, current_idx-

    if dist >= d:
        including_current_revenue, including_current_shops = allocate(shops, revenue, d, current_
        including_current_revenue += revenue[current_idx]
        including_current_shops.append(shops[current_idx])

    if including_current_revenue > excluding_current_revenue:
        return including_current_revenue, including_current_shops
    return excluding_current_revenue, excluding_current_shops
```

Task 2.1.2

Time complexity for the worst case is:

T(n) = 2*T(n-1)* + *C1* = *4*T(n-2) + C1 + 2*C2* = *2^k*T(n-k) + C, then for k = n:

T(n) = 2^n*T(0) + C = θ(2^n)

Task 2.1.3

Multiple traversing through the elemnts of X[ ] when we do or do not take them into final array makes them redundant and hence brings up the overlapping of subproblems.

Task 2.1.4

```
def allocate(shops, revenue, d, current_idx = None, prev_idx = None):

    if M[current_idx][prev_idx] != None:
        return M[current_idx][prev_idx][0], M[current_idx][prev_idx][1]

    if current_idx == None:
        return "Error"

    if not prev_idx:
        dist = d
    else:
        dist = shops[prev_idx] - shops[current_idx]

    if current_idx == 0:
        if dist < d:
            M[current_idx][prev_idx] = (0, [])
            return 0, []
        M[current_idx][prev_idx] = (revenue[current_idx], [shops[current_idx]])
        return revenue[current_idx], [shops[current_idx]]

    including_current_revenue = 0
    including_current_shops = []
    excluding_current_revenue, excluding_current_shops = allocate(shops, revenue, d, current_idx-

    if dist >= d:
        including_current_revenue, including_current_shops = allocate(shops, revenue, d, current_
        including_current_revenue += revenue[current_idx]
        including_current_shops.append(shops[current_idx])

    if including_current_revenue > excluding_current_revenue:
        M[current_idx][prev_idx] = (including_current_revenue, including_current_shops)
        return including_current_revenue, including_current_shops
    M[current_idx][prev_idx] = (excluding_current_revenue, excluding_current_shops)
    return excluding_current_revenue, excluding_current_shops

N = 5
M = [[None for i in range(N)] for i in range(N)]
```

Task 2.1.5

Worst time complexuty is if we fill whole the matrix M[ ][ ] without repetitions and hence is O(n^2)

Task 2.2.1

Insertion sort constructs the output array by popping each element and inserting it into output array that is sorted on each iteration. Can be optimized by putting sorted sequence in head of the array being sorted to minimize space consumtion.

Task 2.2.2

```
def isort():
    for i in range(n):
        j = i
        while j>0 and x[j-1]>x[j]:
            x[j], x[j-1] = x[j-1], x[j]
            j-=1
```

Perfomance

Row	Best	Worst	k-sorted array
1	1	1	1
2	n	n	n
3	n+1	∑(i) = n(n+1)/2+1	nk+1
4	n	n(n+1)/2	nk
Overall	3n+2 = θ(n)	n^2+2n+2 = θ(n^2)	n(2k+1)+2 = θ(nk)

Hence: Worst and vest case running times of insertion sort are θ(n^2) and θ(n) respectively.

Task 2.2.3

In case of k-sorted array, insertion sort requires θ(nk) comparisons in the nested loop (check code above) and hence, in case of relatively small k comparing to n, overall complexity is θ(nk)~θ(n).

Therefore, insertion sort **is fast**, relatively to its worst case when applied to a k-sorted array.

Task 2.2.4

```
class Data:
    code, start_date, end_date, sponsor, description = 0,0,0,0,''
    def __init__(self, A):
        self.code = A[0]
        self.start_date = A[1]
        self.end_date = A[2]
        self.sponsor = A[3]
        if len(A) == 5:
            description = A[4]
    def get_key(self):
        code = self.code
        key = 0
        n = len(code)
        for i in range(n):
            key += ord(code[i])**(n-i)
        return key

def sort(x):
    n = len(x)
    for i in range(n):
        j = i
        while j>0 and (x[j-1]).get_key()>(x[j]).get_key():
            x[j], x[j-1] = x[j-1], x[j]
            j-=1
    return x
```

Task 2.2.5

```
def belongs (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return True
        elif A[mid] > x:
            return belongs(A, From, mid-1, x)
        else:
            return belongs(A, mid + 1, To, x)
    else:
        return False


def belongs(A, From, To, x):
    while From <= To:
```

```
        mid = From + (To - From) // 2;
        if A[mid] == x:
            return True
        elif A[mid] < x:
            From = mid + 1
        else:
            To = mid - 1
    return False


def belongs (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return True
        elif A[mid] > x:
            return belongs(A, From, mid-1, x)
        else:
            return belongs(A, mid + 1, To, x)
    else:
        return False


def belongs (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return True
        elif A[mid] > x:
            return belongs(A, From, mid-1, x)
        else:
            return belongs(A, mid + 1, To, x)
    else:
        return False
```

Task 2.2.6

```
null = None

def search (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return mid
        elif A[mid] > x:
            return search(A, From, mid-1, x)
        else:
```

```
            return search(A, mid + 1, To, x)
    else:
        return null
```

Task 2.2.7

T(n) = T(n/2) + c =>

=> second case of Master Theorem:

T = θ(log(n))

Task 2.2.8

```
def search (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return mid
        elif A[mid] > x:
            return search(A, From, mid-1, x)
        else:
            return search(A, mid + 1, To, x)
    else:
        return From

def insertion_sort(A):
    n = len(A)
    for i in range(n):
        idx = search(A, 0, i, A[i])
        elem = A[i]
        idx_1 = i
        while idx_1 > idx:
            A[idx_1] = A[idx_1-1]
            idx_1 -= 1
        A[idx] = elem
```

a) Such a use of insertion sort decreases amount of comparisons during running.

b) The implementation is shown above.

c) IT DOES NOT CHANGE TIME COMPLEXITY. Although, overall there's no difference in time complexity due to the need to insert n elements into sorted array which takes Theta(n^2), where n is the length of

array, the use of search() decreases amount of comparisons making insertion sort more effective for some tasks.

Task 2.2.9

```
def bubble_sort(A):
    n = len(A)
    for i in range(0, n):
        for j in range(i, n):
            if A[i] > A[j]:
                elem = A[i]
                A[i] = A[j]
                A[j] = elem
    return A

def bubble_sort_parallelized(A):
    n = len(A)

    #thread one
    A_left = bubble_sort(A[0: (n-1)//2])

    #thread two
    A_right = bubble_sort(A[(n-1)//2: n])

    new_A = [0]*n
    idx_l, idx_r = 0, 0
    len_l = len(A_left)
    len_r = len(A_right)

    for i in range(n):
        if len_l == idx_l:
            new_A[i] = A_right[idx_r]
            idx_r +=1
            continue
        if len_r == idx_r:
            new_A[i] = A_left[idx_l]
            idx_l +=1
            continue
        if A_left[idx_l] < A_right[idx_r]:
            new_A[i] = A_left[idx_l]
            idx_l += 1
        else:
            new_A[i] = A_right[idx_r]
            idx_r += 1
```

a) "Is Bubble Sort difficult to parallelize?" It is not difficult to parallelize Bubble Sort.

b) "Why?" Because it is not that different from standart Bubble Sort.

c) Code above demonstrates the main algorithm. Lines under comments "thread one" and "thread two" denote independent function calls so they can be run simultaneously and hence parallelized.

Task 2.1.1

```
def allocate(shops, revenue, d, current_idx = None, prev_idx = None):

    if current_idx == None:
        return "Error"

    if not prev_idx:
        dist = d
    else:
        dist = shops[prev_idx] - shops[current_idx]

    if current_idx == 0:
        if dist < d:
            return 0, []
        return revenue[current_idx], [shops[current_idx]]

    including_current_revenue = 0
    including_current_shops = []
    excluding_current_revenue, excluding_current_shops = allocate(shops, revenue, d, current_idx-

    if dist >= d:
        including_current_revenue, including_current_shops = allocate(shops, revenue, d, current_
        including_current_revenue += revenue[current_idx]
        including_current_shops.append(shops[current_idx])

    if including_current_revenue > excluding_current_revenue:
        return including_current_revenue, including_current_shops
    return excluding_current_revenue, excluding_current_shops
```

Task 2.1.2

Time complexity for the worst case is:

T(n) = 2*T(n-1)* + *C1* = *4*T(n-2) + C1 + 2*C2* = *2^k*T(n-k) + C, then for k = n:

T(n) = 2^n*T(0) + C = θ(2^n)

Task 2.1.3

Multiple traversing through the elemnts of X[ ] when we do or do not take them into final array makes them redundant and hence brings up the overlapping of subproblems.

Task 2.1.4

```
def allocate(shops, revenue, d, current_idx = None, prev_idx = None):

    if M[current_idx][prev_idx] != None:
        return M[current_idx][prev_idx][0], M[current_idx][prev_idx][1]

    if current_idx == None:
        return "Error"

    if not prev_idx:
        dist = d
    else:
        dist = shops[prev_idx] - shops[current_idx]

    if current_idx == 0:
        if dist < d:
            M[current_idx][prev_idx] = (0, [])
            return 0, []
        M[current_idx][prev_idx] = (revenue[current_idx], [shops[current_idx]])
        return revenue[current_idx], [shops[current_idx]]

    including_current_revenue = 0
    including_current_shops = []
    excluding_current_revenue, excluding_current_shops = allocate(shops, revenue, d, current_idx-

    if dist >= d:
        including_current_revenue, including_current_shops = allocate(shops, revenue, d, current_
        including_current_revenue += revenue[current_idx]
        including_current_shops.append(shops[current_idx])

    if including_current_revenue > excluding_current_revenue:
        M[current_idx][prev_idx] = (including_current_revenue, including_current_shops)
        return including_current_revenue, including_current_shops
    M[current_idx][prev_idx] = (excluding_current_revenue, excluding_current_shops)
    return excluding_current_revenue, excluding_current_shops

N = 5
M = [[None for i in range(N)] for i in range(N)]
```

Task 2.1.5

Worst time complexuty is if we fill whole the matrix M[ ][ ] without repetitions and hence is O(n^2)

Task 2.2.1

Insertion sort constructs the output array by popping each element and inserting it into output array that is sorted on each iteration. Can be optimized by putting sorted sequence in head of the array being sorted to minimize space consumtion.

Task 2.2.2

```
def isort():
    for i in range(n):
        j = i
        while j>0 and x[j-1]>x[j]:
            x[j], x[j-1] = x[j-1], x[j]
            j-=1
```

Perfomance

Row	Best	Worst	k-sorted array
1	1	1	1
2	n	n	n
3	n+1	∑(i) = n(n+1)/2+1	nk+1
4	n	n(n+1)/2	nk
Overall	3n+2 = θ(n)	n^2+2n+2 = θ(n^2)	n(2k+1)+2 = θ(nk)

Hence: Worst and vest case running times of insertion sort are θ(n^2) and θ(n) respectively.

Task 2.2.3

In case of k-sorted array, insertion sort requires θ(nk) comparisons in the nested loop (check code above) and hence, in case of relatively small k comparing to n, overall complexity is θ(nk)~θ(n).

Therefore, insertion sort **is fast**, relatively to its worst case when applied to a k-sorted array.

Task 2.2.4

```
class Data:
    code, start_date, end_date, sponsor, description = 0,0,0,0,''
    def __init__(self, A):
        self.code = A[0]
        self.start_date = A[1]
        self.end_date = A[2]
        self.sponsor = A[3]
        if len(A) == 5:
            description = A[4]
    def get_key(self):
        code = self.code
        key = 0
        n = len(code)
        for i in range(n):
            key += ord(code[i])**(n-i)
        return key

def sort(x):
    n = len(x)
    for i in range(n):
        j = i
        while j>0 and (x[j-1]).get_key()>(x[j]).get_key():
            x[j], x[j-1] = x[j-1], x[j]
            j-=1
    return x
```

Task 2.2.5

```
def belongs (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return True
        elif A[mid] > x:
            return belongs(A, From, mid-1, x)
        else:
            return belongs(A, mid + 1, To, x)
    else:
        return False


def belongs(A, From, To, x):
    while From <= To:
```

```
        mid = From + (To - From) // 2;
        if A[mid] == x:
            return True
        elif A[mid] < x:
            From = mid + 1
        else:
            To = mid - 1
    return False


def belongs (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return True
        elif A[mid] > x:
            return belongs(A, From, mid-1, x)
        else:
            return belongs(A, mid + 1, To, x)
    else:
        return False


def belongs (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return True
        elif A[mid] > x:
            return belongs(A, From, mid-1, x)
        else:
            return belongs(A, mid + 1, To, x)
    else:
        return False
```

Task 2.2.6

```
null = None

def search (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return mid
        elif A[mid] > x:
            return search(A, From, mid-1, x)
        else:
```

```
            return search(A, mid + 1, To, x)
    else:
        return null
```

Task 2.2.7

T(n) = T(n/2) + c =>

=> second case of Master Theorem:

T = θ(log(n))

Task 2.2.8

```
def search (A, From, To, x):
    if To >= From:
        mid = From + (To - From) // 2
        if A[mid] == x:
            return mid
        elif A[mid] > x:
            return search(A, From, mid-1, x)
        else:
            return search(A, mid + 1, To, x)
    else:
        return From

def insertion_sort(A):
    n = len(A)
    for i in range(n):
        idx = search(A, 0, i, A[i])
        elem = A[i]
        idx_1 = i
        while idx_1 > idx:
            A[idx_1] = A[idx_1-1]
            idx_1 -= 1
        A[idx] = elem
```

a) Such a use of insertion sort decreases amount of comparisons during running.

b) The implementation is shown above.

c) IT DOES NOT CHANGE TIME COMPLEXITY. Although, overall there's no difference in time complexity due to the need to insert n elements into sorted array which takes Theta(n^2), where n is the length of

array, the use of search() decreases amount of comparisons making insertion sort more effective for some tasks.

Task 2.2.9

```
def bubble_sort(A):
    n = len(A)
    for i in range(0, n):
        for j in range(i, n):
            if A[i] > A[j]:
                elem = A[i]
                A[i] = A[j]
                A[j] = elem
    return A

def bubble_sort_parallelized(A):
    n = len(A)

    #thread one
    A_left = bubble_sort(A[0: (n-1)//2])

    #thread two
    A_right = bubble_sort(A[(n-1)//2: n])

    new_A = [0]*n
    idx_l, idx_r = 0, 0
    len_l = len(A_left)
    len_r = len(A_right)

    for i in range(n):
        if len_l == idx_l:
            new_A[i] = A_right[idx_r]
            idx_r +=1
            continue
        if len_r == idx_r:
            new_A[i] = A_left[idx_l]
            idx_l +=1
            continue
        if A_left[idx_l] < A_right[idx_r]:
            new_A[i] = A_left[idx_l]
            idx_l += 1
        else:
            new_A[i] = A_right[idx_r]
            idx_r += 1
```

a) "Is Bubble Sort difficult to parallelize?" It is not difficult to parallelize Bubble Sort.

b) "Why?" Because it is not that different from standart Bubble Sort.

c) Code above demonstrates the main algorithm. Lines under comments "thread one" and "thread two" denote independent function calls so they can be run simultaneously and hence parallelized.

Task 2.4.

MyStruct is a self-balancing data structure with elements destributed between b1 and b2 of minBinaryHeap and maxBinaryHeap classes respectively so that none of them is ever greater for more than 2 than another. Then the median is either max element of b2 or min element of b1 or both. Now using default BinaryHeap interface, we get code beneath:

```
public class MyStruct {
    int median;
    minBinaryHeap b1;
    maxBinaryHeap b2;

    public MyStruct(int[] A) {
        b1 = new minBinaryHeap();
        b2 = new maxBinaryHeap();
        int size = A.length;
        this.median = size/2 + 1;
        for (int i = 0; i < size/2+1; i++) {
            b2.insert(A[i]);
        }
        for (int i = size/2+1; i < size; i++) {
            b1.insert(A[i]);
        }
    }
    void insert(int k) {            // Theta(log(n))
        if (k < b2.getMax()) {
            b2.insert(k);
            if (b2.size > b1.size + 1) {
                b1.insert(b2.getMax());
                b2.delete(b2.getMax());
            }
            this.median = b2.getMax();
        }
        else {
            b1.insert(k);
            if (b1.size > b2.size + 1) {
                b2.insert(b1.getMin());
                b1.delete(b1.getMin());
            }
            this.median = b1.getMin();
        }
    }
    int remove_median() {           // Theta(log(n))
        int med = this.median;

        if (b1.size == b2.size) {
            b1.delete(b1.getMin());
```

```
            this.median = b2.getMax();
        }
        if (b1.size > b2.size) {
            b1.delete(b1.getMin());
            this.median = b2.getMax();
        }
        if (b1.size < b2.size) {
            b2.delete(b2.getMax());
            this.median = b2.getMax();
        }
        return med;
    }
    int size() {                    // Theta(1)
        return b1.size + b2.size;
    }
    boolean isEmpty() {             // Theta(1)
        return b1.size + b2.size == 0;
    }
}
```

2.3.1.

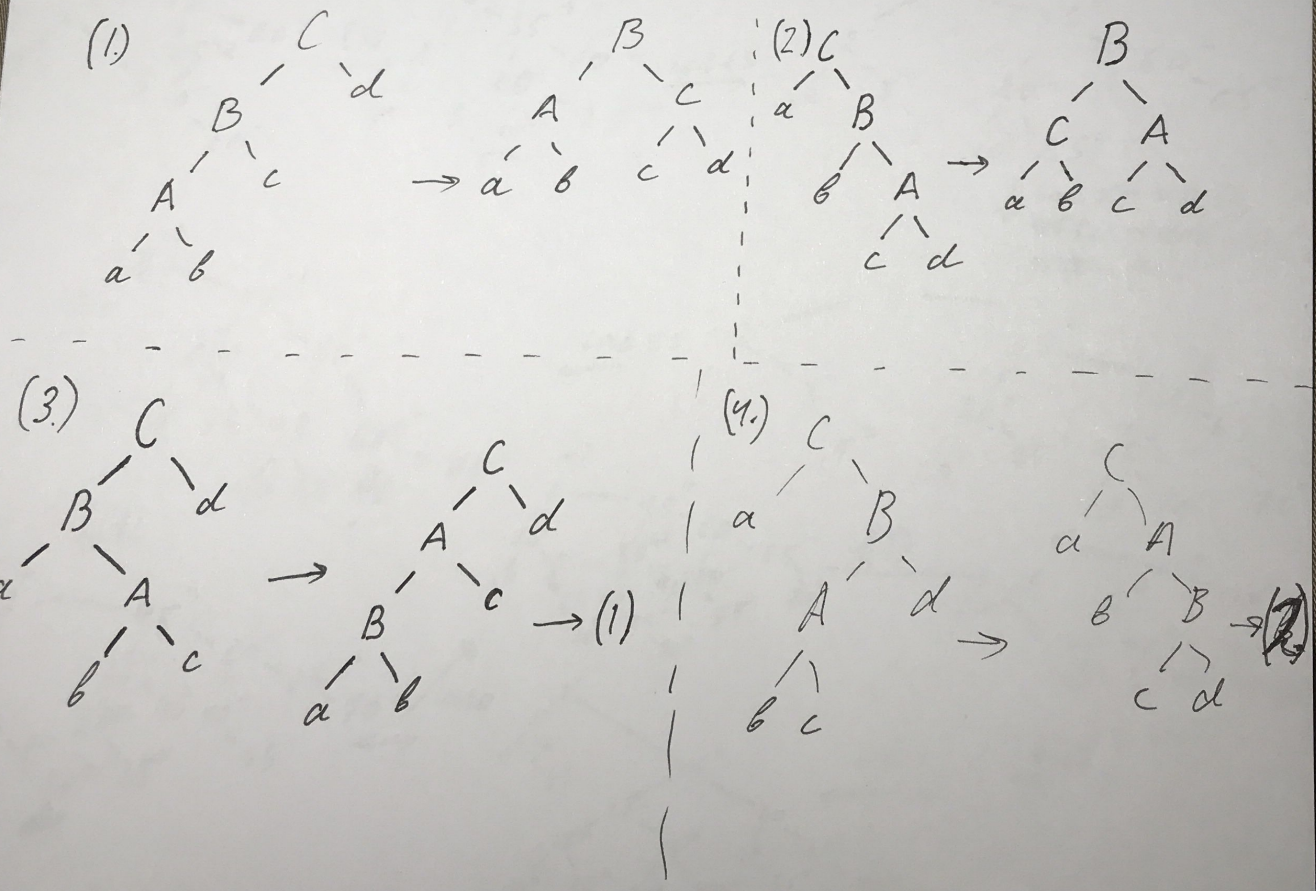
AVL is a self-balancing binary search tree, every node of the which satisfies that its subtrees' lengths differ by at most one, if in case it at some moment the property is not satisfied, rebalancing via rotations is done.

Task 2.3.2

First, insert the value treating the AVL as usual BST.

Then, denote first three nodes along the path to the value as C, B, A starting from the first node with unbalanced subtrees if any, if there's none, that's it.

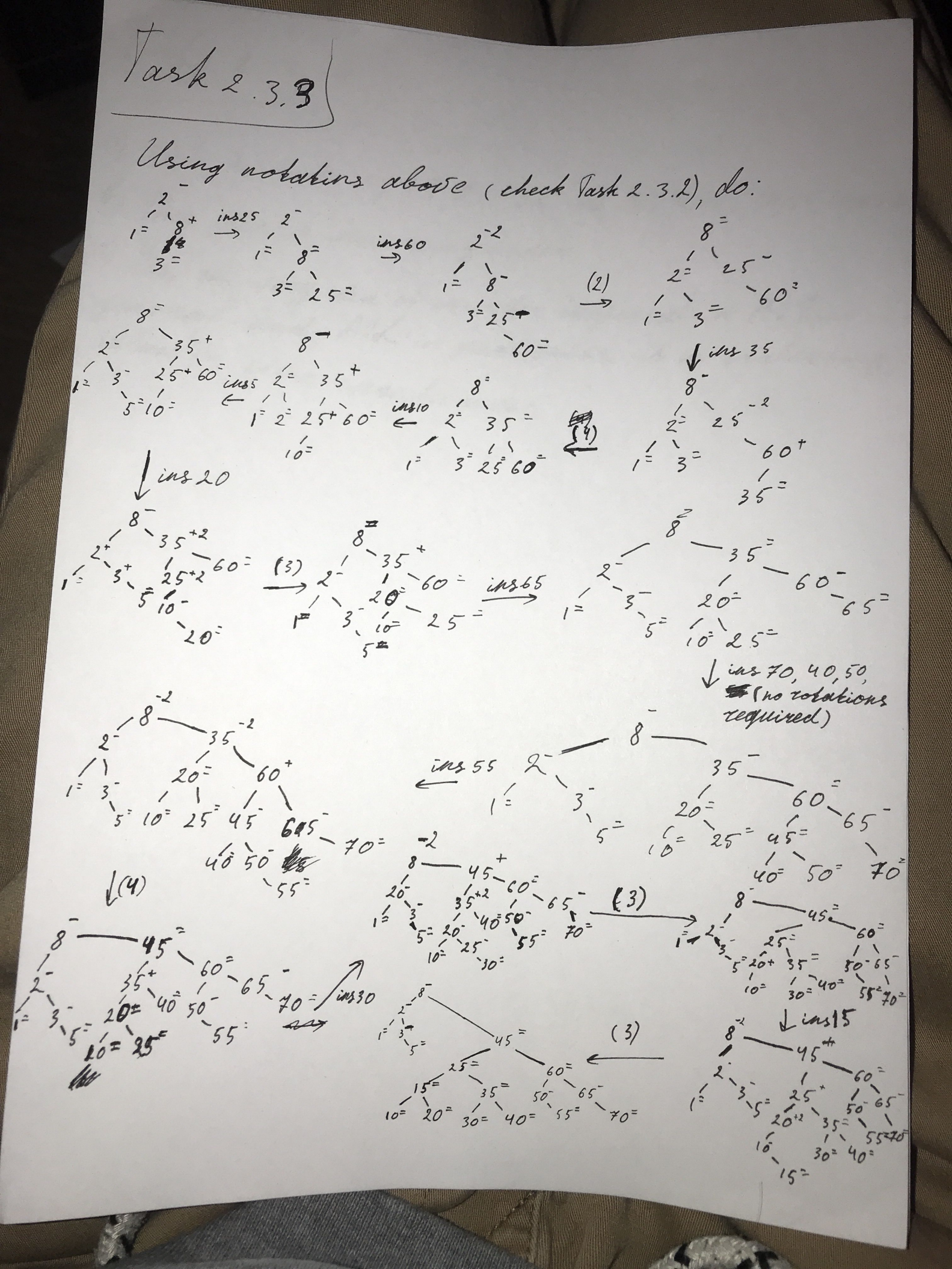
Finally, depending on the pattern of the unbalanced subtrees, do one of the following permutations:



* a, b, c, d - some subtrees.

Task 2.3.3

Using notations above (check Task 2.3.2), do:



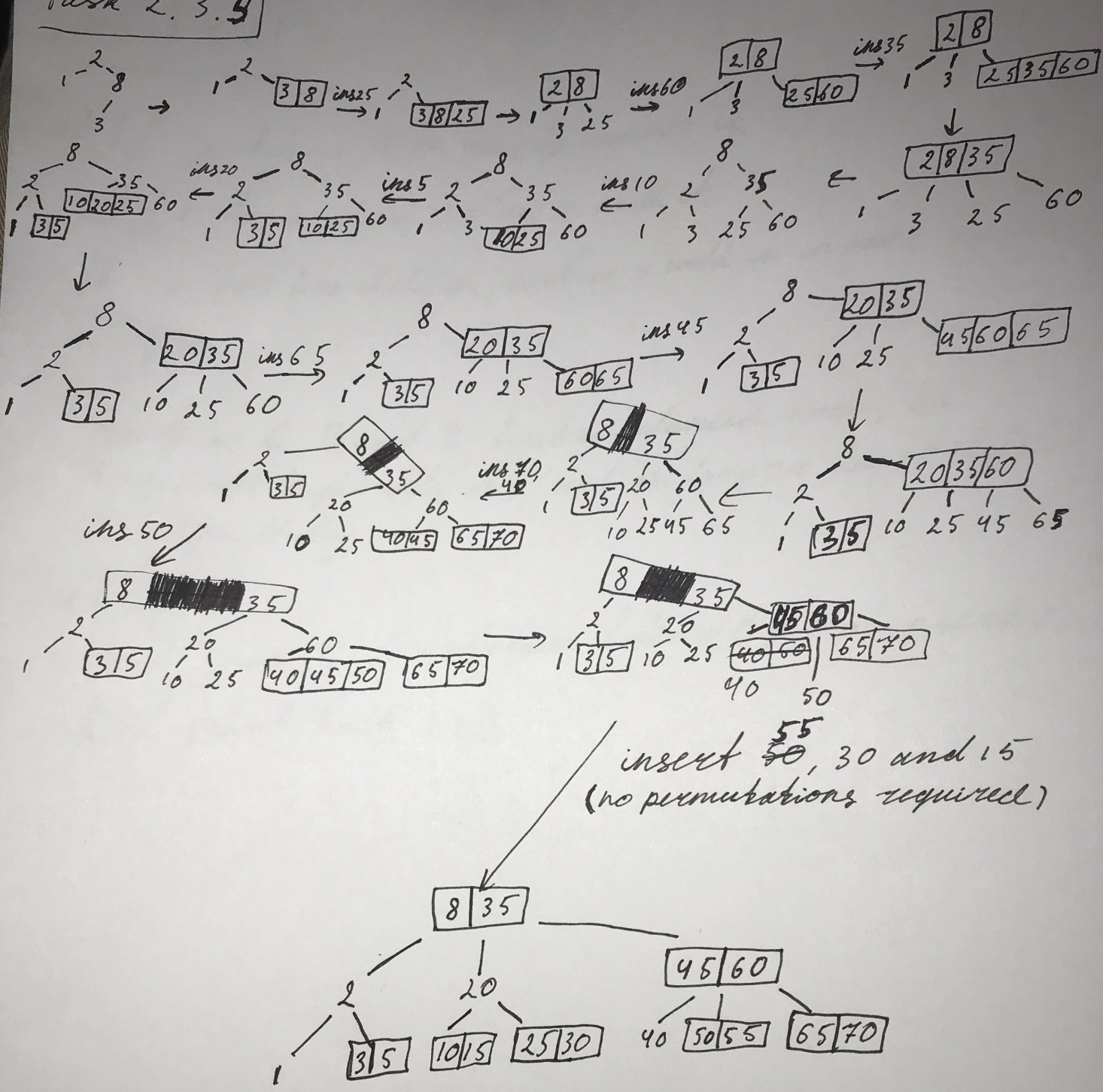
Task 2.3.4

a.) 1, 2, 3, 5, 8, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70.

b.) They are in increasing order.

c.) Yes, the very idea of in-order traversal in BST in general and AVL in particular is in traversing values in some specific order.

Task 2.3.5


ins25
ins60
ins35
ins20
ins5
ins10
ins65
ins45
ins70
ins50
insert 55, 30 and 15
(no permutations required)

Task 2.3.6

First: If x has no child, simply remove it.

If x has a child, replace x with it.

If x has two children, replace x with its in-order successor.

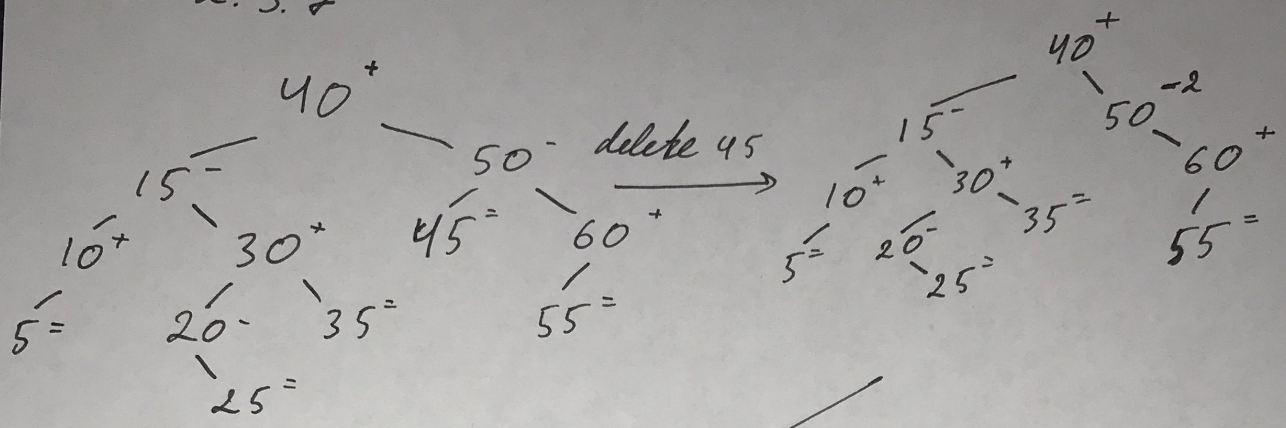
Second:

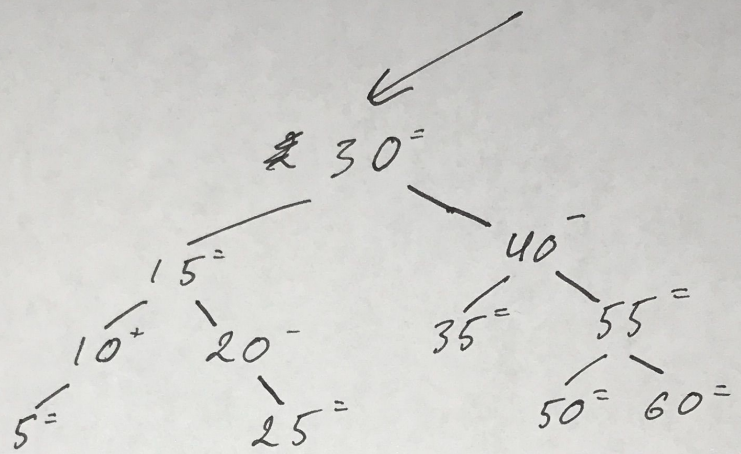
Denote as C, B and A first unbalanced node, the larger ~~height~~ height child of C and the longer height child of B respectfully.

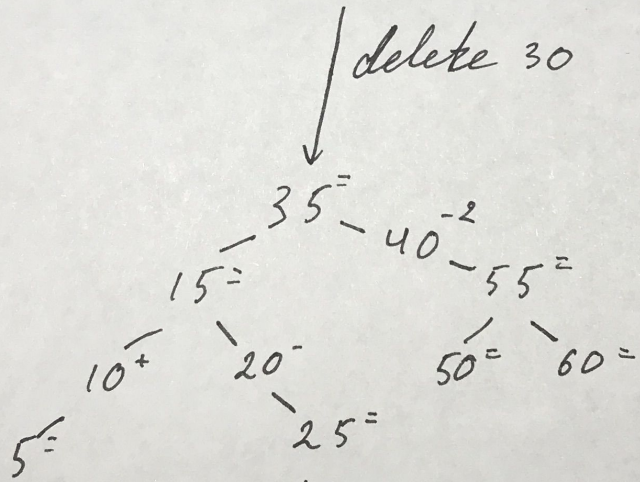
Third:

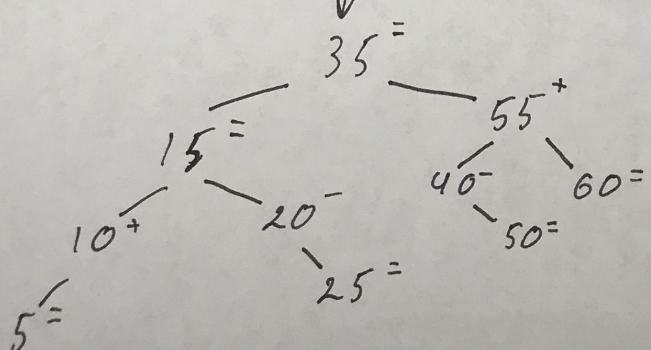
With new CBA-notation follow logic of insert for AVL (check Task 2.3.2).

Task 2.3.7


40+
15-
50-
10+
30+
45=
60+
5=
20-
35=
55=
25=
delete 45
40+
15-
50-2
10+
30+
60+
5=
20-
35=
55=
25=


30=
15=
40-
10+
20-
35=
55=
5=
25=
50=
60=


delete 30
35=
40-2
15=
55=
10+
20-
50=
60=
5=
25=


35=
15=
55+
10+
20-
40-
60=
50=
25=
5=