Midterm Assignment

Name : Ryan Christopher
Student ID : U09543360

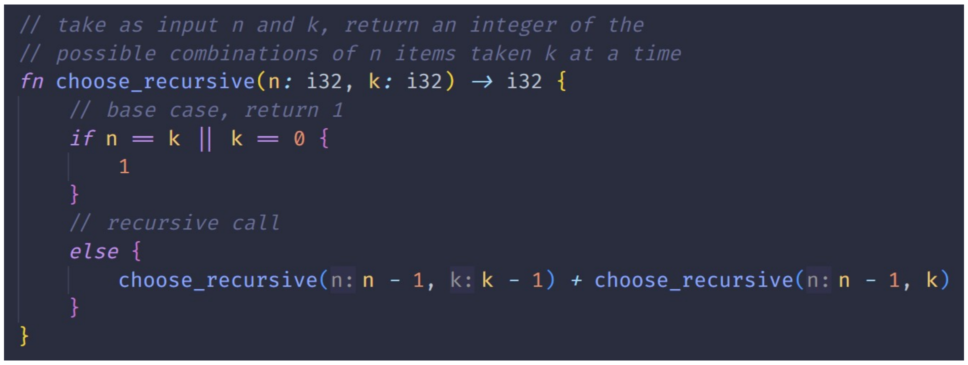
Problem 1) Implement the following algorithm:

Let C(n, k) be the number of combinations of n items taken k at a time.

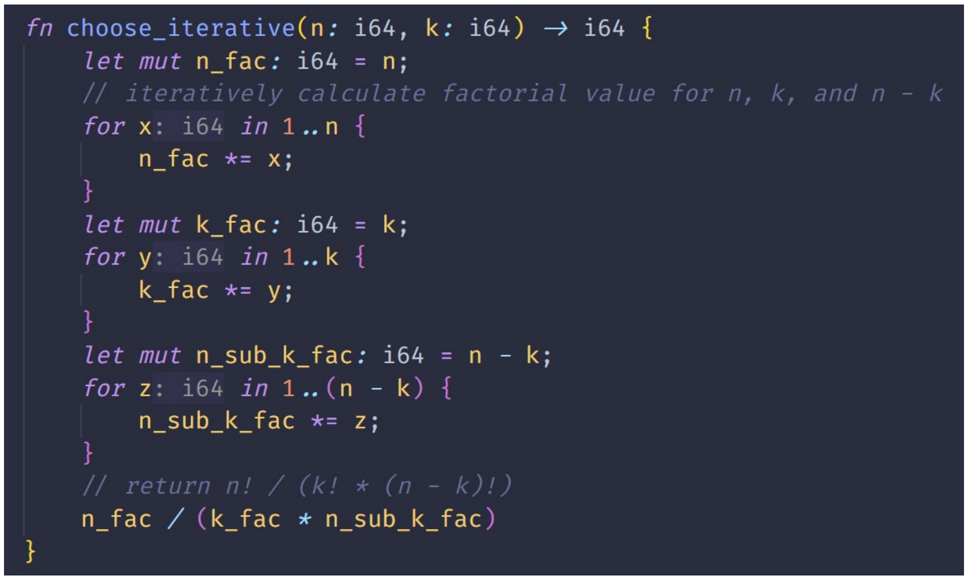
C(n, k) = 1 if n = k or k = 0
else
C(n, k) = C(n – 1, k – 1) + C(n – 1, k)

Give a tight asymptotic bound on the worst-case running time by solving the recurrence. Verify your answer by running the code for n = 6, 10, 16, and k = n / 2. Is a recursive approach the best one?

The file main.rs contains the code for this problem, where the functions "choose_recursive" and "choose_iterative" implement the algorithm given.



```rust
// take as input n and k, return an integer of the
// possible combinations of n items taken k at a time
fn choose_recursive(n: i32, k: i32) -> i32 {
    // base case, return 1
    if n == k || k == 0 {
        1
    }
    // recursive call
    else {
        choose_recursive(n: n - 1, k: k - 1) + choose_recursive(n: n - 1, k)
    }
}
```

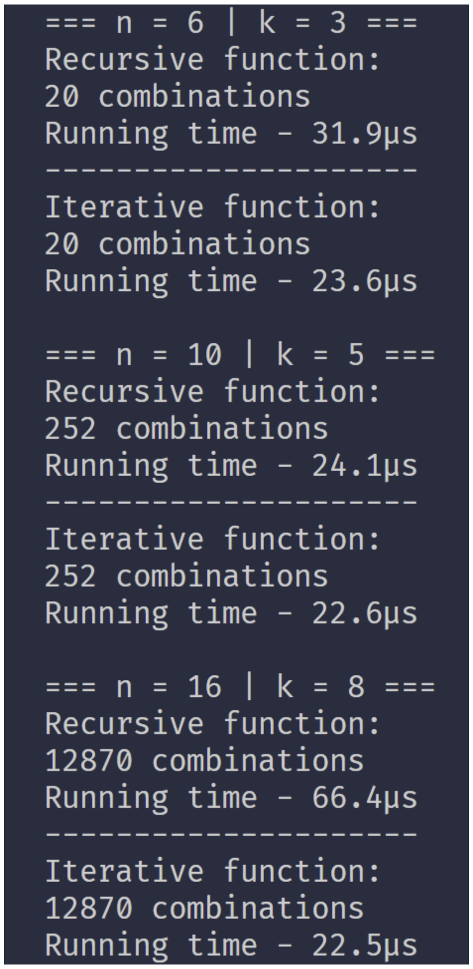


```rust
fn choose_iterative(n: i64, k: i64) -> i64 {
    let mut n_fac: i64 = n;
    // iteratively calculate factorial value for n, k, and n - k
    for x: i64 in 1..n {
        n_fac *= x;
    }
    let mut k_fac: i64 = k;
    for y: i64 in 1..k {
        k_fac *= y;
    }
    let mut n_sub_k_fac: i64 = n - k;
    for z: i64 in 1..(n - k) {
        n_sub_k_fac *= z;
    }
    // return n! / (k! * (n - k)!)
    n_fac / (k_fac * n_sub_k_fac)
}
```

The algorithm is bound by a worst-case running time of $\Theta(n! / (n - k)!)$ as the function must be recursively called that many times in order to calculate the number of possible combinations. While initially written with a recursive approach, the function “choose_iterative” iteratively calculates the factorial values and uses the algorithm’s tight asymptotic bound, $n! / (n - k)!$, to return the answer.

The two functions perform similarly at values $n = 6$ and 10, however once n becomes larger at 16 the iterative approach begins to run faster than the recursive approach.

The calculated combinations and running times for both functions with $n = 6, 10, 16$, and $k = n / 2$ are included in the following screenshot:



```
=== n = 6 | k = 3 ===
Recursive function:
20 combinations
Running time - 31.9µs
---------------------
Iterative function:
20 combinations
Running time - 23.6µs

=== n = 10 | k = 5 ===
Recursive function:
252 combinations
Running time - 24.1µs
---------------------
Iterative function:
252 combinations
Running time - 22.6µs

=== n = 16 | k = 8 ===
Recursive function:
12870 combinations
Running time - 66.4µs
---------------------
Iterative function:
12870 combinations
Running time - 22.5µs
```

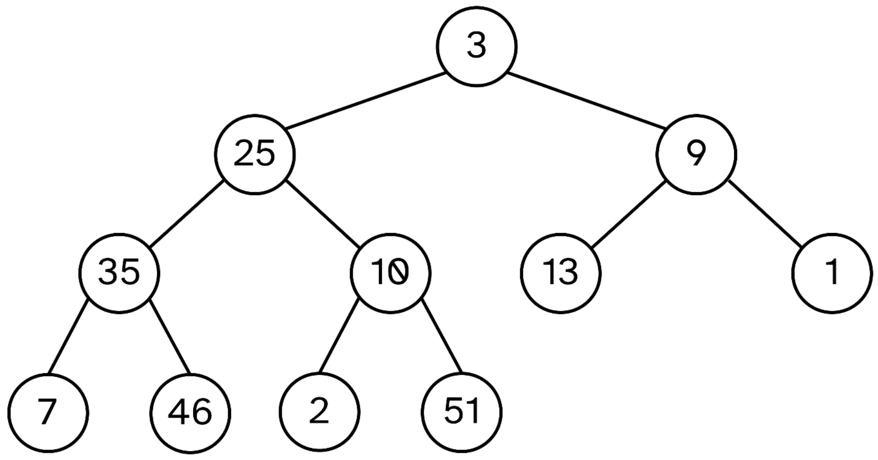
Problem 2) Assume the complete binary tree numbering scheme used by heapsort and apply the heapsort algorithm to the following key sequence:

(3, 25, 9, 35, 10, 13, 1, 7, 46, 2, 51)

The first element index is equal to 0.

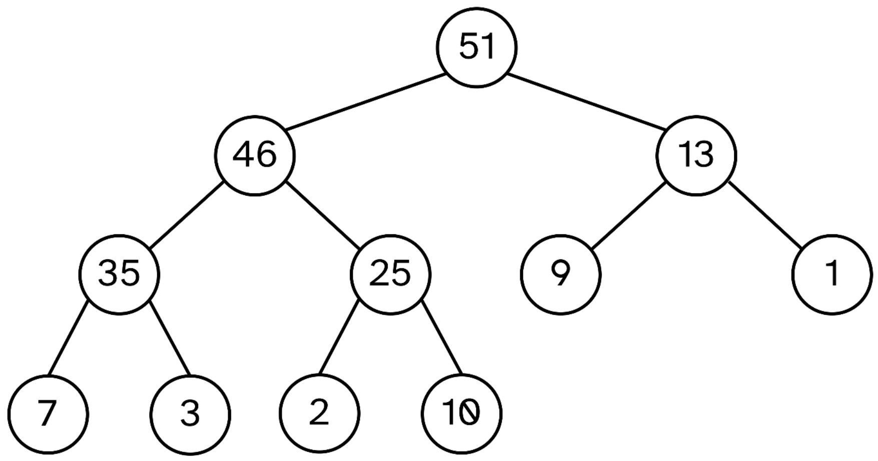
a) What value is in location 5 of the initial heap?

The initial tree before the heap is created:



	0	1	2	3	4	5	6	7	8	9	10
array:	3	25	9	35	10	13	1	7	46	2	51

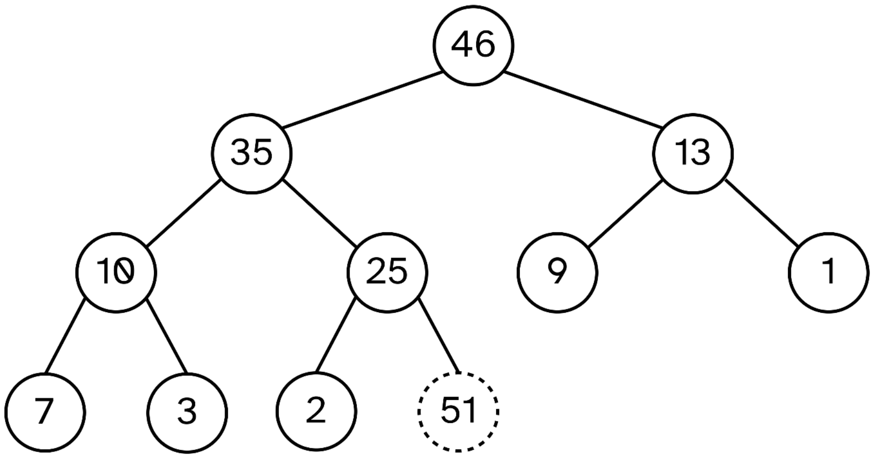
Once the initial heap is created, the value 9 is located at location 5:



	0	1	2	3	4	5	6	7	8	9	10
array:	51	46	13	35	25	9	1	7	3	2	10

b) After a single deletion (of the element at the heap root) and tree restructuring, what value is in location 5 of the new heap?

Once there is a single deletion and tree restructuring, 9 remains the value at location 5 of the new heap.

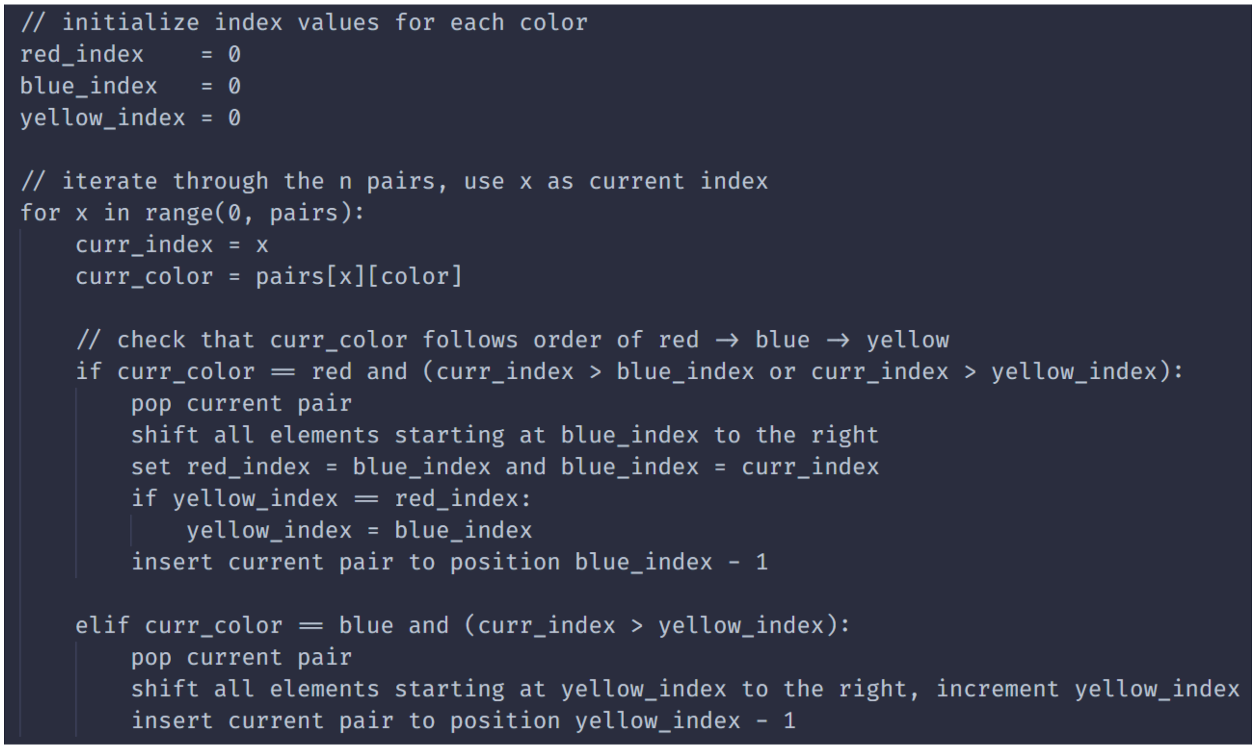


	0	1	2	3	4	5	6	7	8	9	10
array:	46	35	13	10	25	9	1	7	3	2	51

Problem 3) Assume that we are given *n* pairs of items as input, where the first item is a number and the second item is one of three colors (red, blue, or yellow). Further assume that the items are sorted by number. Give an O(*n*) algorithm to sort the items by color (and reds before all blues before all yellows) such that the numbers for identical colors stay sorted.

For example:

(1, blue), (3, red), (4, blue), (6, yellow), (9, red)
should become:
(3, red), (9, red), (1, blue), (4, blue), (6, yellow)



```
// initialize index values for each color
red_index    = 0
blue_index   = 0
yellow_index = 0

// iterate through the n pairs, use x as current index
for x in range(0, pairs):
    curr_index = x
    curr_color = pairs[x][color]

    // check that curr_color follows order of red → blue → yellow
    if curr_color == red and (curr_index > blue_index or curr_index > yellow_index):
        pop current pair
        shift all elements starting at blue_index to the right
        set red_index = blue_index and blue_index = curr_index
        if yellow_index == red_index:
            yellow_index = blue_index
        insert current pair to position blue_index - 1

    elif curr_color == blue and (curr_index > yellow_index):
        pop current pair
        shift all elements starting at yellow_index to the right, increment yellow_index
        insert current pair to position yellow_index - 1
```

This algorithm performs the sort in *O*(n) time as there is only 1 pass through the list of items. On each element, if there is a red after a blue or blue after a yellow, the algorithm references the first blue or yellow element and sets them forward to create a "gap" for the current element that is out of place to be popped and inserted into the correct order. Because we can assume the items are sorted by number, as long as we have the index of the first of each element, we can place out of order reds and blues by subtracting 1 from the blue_index or yellow_index values to get the correct color group and position.

Problem 4) Insert the keys

<13, 19, 35, 71, 31, 6, 23, 4, 98, 101>

into a hash table of size *m* = 13 using linear hashing.

Here, $h(k, i) = ((k \bmod m) + i) \bmod m$, $i = 0, 1, 2, \ldots$

How many times do you increment *i* to resolve collisions?

m			
0	13		
1			
2			
3			
4	4		
5	31		
6	19	71	6
7	71	6	98
8	6	98	
9	35	98	
10	23	98	101
11	98	101	
12	101		

Sequence of inserts:

```
13  → 0
19  → 6
35  → 9
71  → 6, 7
31  → 5
6   → 6, 7, 8
23  → 10
4   → 4
98  → 7, 8, 9, 10, 11
101 → 10, 11, 12
```

Total times i incremented to resolve collisions: **9**

Does double hashing with *h*2(*k*, *i*) = (*k mod* 7 + *i*) help to minimize number of increments?

Yes, double hashing reduces the number of increments from 9 to 5.

m	
0	13
1	71 6
2	101
3	
4	4
5	31
6	19 71 6
7	98
8	
9	35 71
10	23 101
11	6
12	

Sequence of inserts:

```
13  → 0
19  → 6
35  → 9
71  → 6, 9, 1
31  → 5
6   → 6, 1, 11
23  → 10
4   → 4
98  → 7
101 → 10, 2
```

Total times i incremented to resolve collisions: **5**

Problem 5) Describe an efficient algorithm that, given *n* random integers in the range of 1 to *k*, preprocesses the input and then answers any query about how many of the *n* integers fall into the range [a..b] in O(1) time.

To achieve this function, the *n* random integers must first be sorted. Take the following list of numbers as an example:

```
[8, 9, 4, 5, 2, 5, 2, 8, 10, 1, 2, 7, 3, 10]
```

Once the algorithm sorts the numbers, they will look like so:

```
[1, 2, 2, 2, 3, 4, 5, 5, 7, 8, 8, 9, 10, 10]
```

The algorithm will then create a dictionary with keys ranging from `0` to *k* and the value for each key initialized to `0`. It will then iterate through the sorted array with current_value set to the first element of the sorted list, and each time a different value is found, set the key : value pair in the dictionary equal to the index of the new value. At the end of the array, set the last key : value pair to the length of the sorted array. Once done, the dictionary will look like the following:

```
counts = {‘0’: 0, ‘1’: 1, ‘2’: 4, ‘3’: 5, ‘4’: 6, ‘5’: 8,
          ‘6’: 8, ‘7’: 9, ‘8’: 11, ‘9’: 12, ‘10’: 14}
```

Now, the range from [a..b] for any two integers in the list can be found when we can look up the total count up to each value for every value in the list. For example, lets suppose our range query were [2..7]. In the sorted list, the range would be the following values:

```
[2, 2, 2, 3, 4, 5, 5, 7]
```

There are 8 integers in total that fall between 2 and 7. To calculate this, we would look at the value in the dictionary with the key a – 1, in this case counts[‘1’] which would be 1, and the value with the key b, counts[‘7’] would would be the value 9. Then the two values will be subtracted, 9 – 1, to determine that 8 integers are within the specified range. The correct number of integers between the range [2..7] have been answered in *O*(1) time as a value is able to be retrieved by the corresponding key in constant time within a dictionary.