COSC230 Assignment 1

This assignment consolidates material from *Practical 1* on "Programming in `C++`", and *Practical 2* on "Complexity Analysis". Please submit all of your assignment files through `Turing` (see the link Assignment Submission via `Turing`). Before you submit your assignment please check your code compiles on `Turing` and does what you expect it to.

Question 1: [*25 marks*]

Download *Assignment 1 code files* from myLearn. Enhance the class `Rational` (from *Practical 1*) by filling out the class interface provided below and implementing additional overloaded binary operators as listed. Also use the `C++` exception-handling feature to ensure a valid rational number is used when a `Rational` object is instantiated. Check your code against the `Rational` class test provided, and **only submit** your `Rational.cc` file.

```
class Rational {
    friend std::ostream& operator<<(std::ostream&, const Rational&);
public:
    Rational(): numerator(), denominator(1) {  }
    Rational(int, int);                          // Constructor now validates object
    void reduce();
    void set_number(int, int);
    Rational operator+(const Rational&) const;  // Addition
                                                 // Subtraction
                                                 // Multiplication
                                                 // Division
                                                 // Less-than
                                                 // Greater-than
                                                 // Equal-to
                                                 // Not-equal-to
private:
    int numerator;
    int denominator;
};
```

Question 2: [*25 marks*]

Download *Assignment 1 code files* from myLearn. Enhance the class `Vec` (from *Practical 1*) by allowing the array to grow dynamically. This can be achieved by implementing two extra member functions: `push_back` and `grow`, and maintaining one extra data member: `array_current_size`. If an array is full before appending a new element using `push_back`, then `push_back` should call the member function `grow`, which: (1) allocates a new array with twice the original space, (2) copies the existing array elements into the new array, (3) frees space from the old array, and (4) assigns new values to the data members. The default constructor should initialize `array_current_size` and `array_limit` to zero. Use the class interface given below, and check your code against the interactive testing file provided, and **only submit** your `Vec.cc` file. Remember to test your code for possible memory leaks using `valgrind`.

```
class Vec {
public:
    Vec();
    Vec(int);
    ~Vec();
    Vec(const Vec&);
    Vec& operator=(const Vec&);
    double& operator[](int);
    void push_back(const double&);                          // New member function
    void grow();                                            // New member function
    int get_current_size() { return array_current_size; }
    int get_limit() { return array_limit; }
private:
    int array_current_size;                                 // New data member
    int array_limit;
    double* pointer_to_data;
};
```

Question 3: [*25 marks*]

Consider the following `C++` code for a function that returns the kth-smallest integer from an array of n integers.

```
int selectkth(int a[], int k, int n)
{
    int i, j, mini, tmp;
    for (i = 0; i != k; i++)
    {
        mini = i;
        for (j = i+1; j != n; j++)
        {
            if (a[j] < a[mini])
                mini = j;
        }
        tmp = a[i];
        a[i] = a[mini];
        a[mini] = tmp;
    }
    return a[k-1];
}
```

(a) Give an asymptotic analysis of time efficiency for the best case where only the smallest integer is returned. Please show working.

(b) Give an asymptotic analysis of time efficiency for the worst case where the largest integer is returned. Please show working.

Please submit answers to Questions 3 and 4 in a single `pdf` file via `Turing`.

Question 4: [*25 marks*]

Use indicator random variables to perform an average-case analysis of `sequential search` where the key is always found exactly in the middle of the array (for simplicity, assume the array length n is always an odd number). Show all working, and make sure you can justify your solution.