# COL 100 Major Exam

November 22, 2018. 8:00 am - 10:00 am

**Name:**

**Entry Number:**

**Group:**

**Notes:**

- Total number of questions: 8. Max Marks: 40
- All answers should be written on the question paper itself.
- You can use any of the *Standard* or *Stanford* library functions while answering the questions.
- The last two sheets in the question paper are meant for rough work. If you run out of space, you can answer questions in this rough space. However, please clearly mention in the answer space for the appropriate question that we should look at the rough space for its answer.
- We will collect the question paper (including the last two sheets meant for rough work). We will not be collecting back any other rough sheets.

1. Write an iterative function (i.e. using loops and **no** recursion) checkOrdered(vector<int> vec) which inputs a vector of integers and checks whether the numbers in the vector are arranged in sorted (ascending) order. The function should return true if the vector is sorted (e.g., for "5, 9, 13") and should return false if the vector is not sorted (e.g., for "9, 5, 13"). You should also take care of duplicates appropriately (e.g., "5, 9, 9, 13" is sorted in ascending order but "5, 9, 13, 9" is not sorted in ascending order). Your program should run in time complexity $O(n)$. [5 points]

```
bool checkOrdered(vector<int> vec)
                or
bool checkOrdered(Vector<int> vec)
{
  for(int i=1; i< vec.size(); i++)
  {
    if(vec[i-1] > vec[i])  // can use vec.at(i) or vec.get(i) also
      return false;
  }
  return true;
}
```

2. Suppose we are given a file containing the entry numbers and grades of students in a class. As an example, consider the following file:

```
2018cs188934  D
2018ee189345  B
2018ce127678  A-
2018ch528145  B
2018me235609  B
2018tt123581  A
```

Write a program to read the grades from the file and print the number of students for each grade sorted by the number of students. For example, for the file given above, the output should be:

```
B 3
A 1
A- 1
D 1
B- 0
C 0
C- 0
E 0
F 0
```

Notice that "B 3" is printed first because it has the maximum number of students (3). Similarly, "D 1" is printed before "B- 0". If two grades have the same number of students, the higher grade is printed first (e.g., A is printed before A- and D). You can assume that "A" comes before "A-" in the lexicographic order (and similarly for other letter grades). The domain of the grade values is all the grades supported by the IITD system : A, A-, B , B-, C, C-, D, E, F. [5 points]

```
int main(){
  string filename;
  cin >> filename;
  ifstream infile;
  infile.open(filename, std::ifstream::in);
  map<string,int> grade_cnt = { {"A", 0}, {"A-", 0},{"B", 0},
```

```
  {"B-", 0}, {"C", 0},{"C-", 0}, {"D", 0}, {"E", 0},{"F", 0} };

  string entryno;
  string grade;
  while (infile >> entryno >> grade)
    grade_cnt[grade]++;
  infile.close();

  map<int, set<string>> grade_sorted;
  for (pair<string, int> grd_freq : grade_cnt) {
    string grd = grd_freq.first;
    int grd_cnt = grd_freq.second;
    if (!grade_sorted.count(grd_cnt)) {
      grade_sorted[grd_cnt] = set<string>();
    }
    grade_sorted[grd_cnt].insert(grd); // add(grd)
  }

for (map<int, set<string>>::reverse_iterator it=grade_sorted.rbegin();
it!=grade_sorted.rend(); ++it)  {
    int gfreq = it->first;
    set<string> grds = it->second;
    for (string grd : grds) {
      cout << grd << " " << gfreq << endl;   } } } }
```

**Some of the relevant functions from Stanford library:**

```
promptUserForFile(infile, "Input file?");
grade_cnt.put(grade, grade_cnt[grade]+1)

for (string grd : grade_cnt) {
  int grd_cnt = grade_cnt[grd] or grade_cnt.get(grd)

containsKey(grd_cnt)
```

3. Given a string $s$ and a character $ch$, we would like to return the first index at which $ch$ appears in the string $s$. We would like to return $-1$ if $ch$ does not occur in the given string $s$. Write a recursive program (i.e., **no** loops) to achieve this funcionality. Your program should be as efficient as possible. What is the time complexity of your program? Can you do this task in $O(n)$? Note that copying a string of size $n$ takes $O(n)$ time. [5 points]

```
int findIndex(string s, char ch, int i=0) {
  if (i > s.length())
    return -1;
  else if (s[i] == ch)
    return i;
  return findIndex(s, ch, i+1);
}
```

```
Time complexity is O(n^2), where n is number of characters in the
string "s". The string "s" is passesd as an argument to the
recursive function and is copied to function call-stack
on each recursive call.

It can be done in O(n) time if we use a reference to
string s in the function argument.
```

4. Given two strings $s1$ and $s2$, we say that $s1$ is an anagram of $s2$, if $s1$ can be obtained by a permutation of characters in $s2$. For example, "stressed" is anagram for "desserts". Similarly, "dormitory" is an anagram for "dirtyroom". Write a function checkAnagram(string $s1$, string $s2$) which inputs two strings $s1$ and $s2$, and returns true if $s1$ is an anagram of $s2$, false otherwise. Your implementation should be as efficient as possible (more points for more efficient implementations). What is the Big-O time complexity of your implementation? [5 points]

```
bool checkAnagram(string s1, string s2)
{
  map<char,int> freqmap_s1;
  map<char,int> freqmap_s2;
  if(s1.length() != s2.length()) return false;

  for(int i=0; i< s1.length(); i++)
  {
    if(!freqmap_s1.count(s1[i])) freqmap_s1[s1[i]] = 0;
    freqmap_s1[s1[i]]++;
  }

  for(int i=0; i< s2.length(); i++)
  {
    if(!freqmap_s2.count(s2[i])) freqmap_s2[s2[i]] = 0;
    freqmap_s2[s2[i]]++;
  }

  for (pair<char, int> char_freq_s1 : freqmap_s1) {
    char char_s1 = char_freq_s1.first;
    int freq_s1 = char_freq_s1.second;
    if(!freqmap_s2.count(char_s1) || freqmap_s2[char_s1] != freq_s1)
      return false;
  }
  return true;
}
```

```
Time complexity is O(n)
```

5. Given a vector of integers, you have to print all the subsets of the set of integers in the vector. Your implementation should be recursive. Example: if input = $\{1, 2, 4\}$, then the output should be:

```
{}
{1}
{1,2}
{1,2,4}
{1,4}
{2}
{2,4}
{4}
```

Note that you can print the subsets in any order. [5 points]

```
void print_vec(vector<int> subset)
{
  cout << "{";
  int len = subset.size();
  if(len == 0) {cout << "}" << endl; return;}
  for(int i=0; i< len-1; i++)
    cout << subset[i] << ",";
  cout << subset[len-1] << "}" << endl;
}

void print_subsets(vector<int> V, vector<int> &subset, int idx=0)
{
  if(idx == V.size()) return;
  if(idx == 0) print_vec(subset);
  subset.push_back(V[idx]);
  print_vec(subset);
  print_subsets(V,subset, idx + 1);
  subset.pop_back();
  print_subsets(V,subset, idx + 1);
  return;
}
```

6. Given two strings s1 and s2, write a recursive function checkSubsequence(string const &s1, string const &s2) to check whether string s1 is a subsequence of s2 Recall that a string s1 is a sub-sequence of s2 if s1 can be obtained by deleting zero or more characters in s2. For example, "pat" is a substring of "painter" but "pat" is not a substring of "pale". Note that for s1 to be a subsequence of s2, the characters in s1 should occur in the same order as s2. e.g., "abc" is not a subsequence of "adcb". You should not use any loops, you should use only recursion and your implemented checkSubsequence function should make atmost O(n) recursive calls.[5 points]

```
bool checkSubsequence(string const &s1, string const &s2)
{
  if(s1.length() == 0) return true;
  if(s1.length() > s2.length()) return false;
  int idx = s2.find(s1[0]);
  if(idx == std::string::npos) return false;
  return checkSubsequence(s1.substr(1), s2.substr(idx+1));
}
```

7. Write a function that inputs a number $n$ ($n >= 1$) and returns a grid with numbers 1 to $n^2$ arranged in a diagonal fashion. For example, for $n = 3$, your output should be:

```
1 3 6
2 5 8
4 7 9
```

In other words, you start from top diagonal (going left to right), and then go all the way down to the bottom diagonal. e.g., for $n = 3$, you first start with the first diagonal row moving right and upwards with one element (1), then the second diagonal row moving right and upwards with two elements (2 and 3), then the third diagonal row moving right and updwards with three elements (4, 5 and 6), then the fourth diagonal row moving right and upwards with two elements (7 and 8), and finally the fifth diagonal row moving right and upwards with one element (9). As another example, for $n = 4$, your output should be:

```
1  3  6 10
2  5  9 13
4  8 12 15
7 11 14 16
```

You can either return the grid using the Grid ADT discussed in class, or a Vector of Vectors. Your function should look something like the following:

```
Grid<int> makeGrid(int n)
{
  //your code goes here
}
or
Vector<Vector<int>> makeGrid(int n)
{
  //your code goes here
}
```

[5 points]

```
int min(int a, int b){
  return (a < b)? a: b; }

int max(int a, int b){
  return (a > b)? a: b; }

Grid<int> makeGrid(int n)
or
vector<vector<int>> makeGrid(int n)
{
  int val=1;
  int row =0, col =0;
  vector<vector<int>> vec(n, vector<int>(n));
  or
  Grid<int> vec(n, n);
  for(int sum_row_col=0; sum_row_col<(2*n-1); sum_row_col++)
  {
    row = min(n-1, sum_row_col);
    col = max(0, sum_row_col- n+1);
    int count = min(sum_row_col+1, n-col);
    for(int j=0; j< count; j++)
    {
      vec[row-j][col+j] = val;
      val++;
    }
  }
  return vec;
}
```

8. Given two stacks of numbers $s1$ and $s2$, write a function separateEvenOdd(stack<int> &s1, stack<int> &s2) which puts all the odd numbers on $s1$ and all the even numbers on $s2$. You can assume that initially all the numbers are in $s1$ (and $s2$ is empty). You are not allowed to make use of any additional stacks or any other data structures such as vectors or queues. It is okay to use a constant number of temporary variables. Hint1: Can you think of an $O(n^2)$ algorithm? Hint2: Initially, you can transfer all the elements at top of $s1$ which are even to $s2$. Then, at each step, you can transfer one even element from $s1$ to $s2$ at a time and transferring this one element may involve multiple sub-steps.[5 points]

```
void separateEvenOdd(stack<int> &s1, stack<int> &s2)
{
  while(!s1.empty() && (s1.top()%2 == 0))
  {
    s2.push(s1.top());
    s1.pop();
  }
  int even_temp = 1;
  while(true){
    while(!s1.empty() && (s1.top()%2 == 1)){
      s2.push(s1.top());
      s1.pop();
    }
    if(!s1.empty()) {
      even_temp = s1.top();
      s1.pop();
    }
    else even_temp = 1;
    while(!s2.empty() && (s2.top()%2 == 1)){
      s1.push(s2.top());
      s2.pop();
    }
    if(even_temp %2 == 0) s2.push(even_temp);
    else break;
  }
}
```

**Recursive algorithm:**

```
void separateEvenOdd(stack<int> &s1, stack<int> &s2)
{
  int odd_tmp;
  if(!s1.empty() && (s1.top()%2 == 0))
  {
    s2.push(s1.top());
    s1.pop();
    separateEvenOdd(s1, s2);
  }
  else if(!s1.empty() && (s1.top()%2 == 1))
  {
    odd_tmp = s1.top();
    s1.pop();
    separateEvenOdd(s1, s2);
    s1.push(odd_tmp);
  }
  else
    return;
}
```