Programming Design, Spring 2014
Homework 6

Instructor: Ling-Chieh Kung
Department of Information Management
National Taiwan University

Submission. To submit your work, please upload the following two files to the online grading system at http://lckung.im.ntu.edu.tw/PD/.

1. A .pdf for Problems 1 to 3 ***AND*** the algorithm part of Problem 4.
2. Your .cpp file for Problem 4.

Each student must submit her/his individual work. No hard copy. No late submission. The due time of this homework is ***8:00am, April 7, 2014***. Please answer in either English or Chinese.

Problem 0

(0 point) Please read Sections 7.1–7.5 and 7.7–7.9 of the textbook.[1] In any case, I strongly suggest you to read the textbook thoroughly before you start to do this homework. For structures and type definitions, please read the slides.

Problem 1

(20 points; 4 points each) When you roll a dice,[2] the probabilities for you to see all integers within 1 and 6 are identical: They are all $\frac{1}{6}$. Mathematically, if we denote the outcome of rolling a dice as X, we have

$$\Pr(X = i) = \frac{1}{6} \quad \forall i = 1, ..., 6.$$

We say that X is a *random variable*: It is a variables (whose value may change), and its value is random.

What if we roll two dices? If X_1 and X_2 are the outcomes of rolling two dices, what will happen to $\bar{X}_2 = \frac{X_1+X_2}{2}$, their average? Of course we know $\bar{X}_2$ is still within 1 and 6, just like X_1 or X_2, but now not all possible values of $\bar{X}_2$ have the same probability. It is not very hard to calculate all the probabilities, as summarized in the table below:

i	1	1.5	2	2.5	3	3.5	4	4.5	5	5.5	6
$\Pr(\bar{X}_2 = i)$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

If we define $\bar{X}_3 = \sum_{j=1}^{3} X_j$, where X_j is the outcome of rolling the ith dice, we may still calculate $\Pr(\bar{X}_3 = i)$ for all possible i. Even before we do this tedious calculation, we know one thing: A value that is closer to 3.5 will has a higher probability than another value that is farther from 3.5. If we define $\bar{X}_n = \sum_{j=1}^{n} X_j$, where X_j is the outcome of rolling the ith dice, it is intuitive the centering effect will become stronger when n becomes larger.

Nevertheless, even though the above intuition seems to be correct, it is just an intuition. To scientifically verify it, there are at least two ways. The first way is, given any n, to calculate the probabilities for all possible values and compare the results of various values of n. If we do not want to do all those calculations, we may do experiments instead. In this case, writing a program will be much easier than rolling dices again again and again. That is why we write the following function:[3]

[1] The textbook is *C++ How to Program: Late Objects Version* by Deitel and Deitel, seventh edition.
[2] In this problem, we assume all the dices are fair.
[3] The library `<cstdlib>` is required for the functions `srand()` and `rand()`.

```
void diceAvg(int diceCount, int trialCount, double avg[], int seed)
{
  srand(seed);

  for(int i = 0; i < trialCount; i++)
  {
    int sum = 0;
    for(int j = 0; j < diceCount; j++)
      sum += rand() % 6 + 1;
    avg[i] = static_cast<double>(sum) / diceCount;
  }
}
```

The function does `trialCount` times of experiments. In each experiment, `diceCount` dices are rolled and the average is recorded in an element of array `avg`. The last parameter `seed` is the seed for generating random numbers.

(a) When you invoke this function, how many local variables are created in the memory before the first statement of the function is executed? What are they for? Clearly explain what happens for memory allocation.

(b) In the function, before an average is calculated, an explicit casting from `int` to `double` is performed on the variable `sum`. While this is of course required, an alternative way is to declare `sum` as a `double` variable. Explain why this alternative way is not as good as the current one.

(c) Suppose the function is invoked by the following main function:

```
int main()
{
  double avg[1000] = {0};
  diceAvg(5, 1000, avg, 0);
  // run your codes here
  return 0;
}
```

Write some C++ codes to generate the frequency distribution for the values stored in `avg`. Divide the range $[1, 6]$ into ten classes: $[1, 1.5)$, $[1.5, 2)$, $[2, 2.5)$, ..., $[5, 5.5)$, $[5.5, 6]$ and count the number of occurrences for each class. Then print out your codes and the frequencies of all classes.

(d) Suppose the function is invoked by the following main function:

```
int main()
{
  int diceCount = 0;
  int trialCount = 0;
  cin >> diceCount >> trialCount;

  double* avg = new double[trialCount];

  diceAvg(diceCount, trialCount, avg, 0);

  return 0;
}
```

Explain why `avg` must be pointer. Though the program can generate averages correctly, explain what is (are) missing in this main function.

(e) Run the above program multiple times with the same `trialCount` but different `diceCount`. Report some frequency distributions with various values of `diceCount` to show that the degree of centering increases as `diceCount` increases. You will be graded based on how persuasive your data are.[4]

Problem 2

(10 points) Open the memory management tool in your computer (e.g., "Task Manager" in MS Windows) and then run the following program:

```
int main()
{
  for(int i = 0; i < 200000000; i++)
  {
    double* d = new double;
  }
  cout << "...";

  return 0;
}
```

(a) (5 points) How many bytes are wasted before the `cout` statement is executed? Explain why.

(b) (5 points) Print screen for a proof that a huge amount of system memory in your computer have been wasted.

(b) (0 points) Add `delete d;` in the loop and see what happens.

(c) (0 points) Increase the number of iterations to exhaust your system memory.[5]

Problem 3

(20 points; 4 points each) Suppose that you need to write a program with a lot of calculations regarding vectors with different dimensions. A structure that may represent a vector is

```
struct MyVector
{
  int n;  // the vector will be of dimension n
  int* m; // m[i] should be the ith element
};
```

The reason for making `m` a pointer rather than a static array is clear: We want to dynamically determine the dimension of the vector. Otherwise, for one-dimensional vectors we need a structure, for two-dimensional ones we need another structure, ..., and so on. Now the problem is, now to allocate memory spaces according to n?

One may write the following statements (say, in her main function)

```
MyVector a;
cin >> a.n;
a.m = new int[a.n];
for(int i = 0; i < a.n; i++)
  a.m[i] = 0;
```

[4] The theory behind this problem is called the central limit theorem. It says that the probability distribution of a sample mean will approach a normal distribution when the sample size becomes larger. This topic will be discussed in details in the course "Statistics (I)".

[5] The issue of memory management will be discussed in details in the course "Operating Systems".

to allocate a memory space to be pointed by `a.m` and add a `delete [] a.m;` statement somewhere to release these spaces. Please note that because `a.m` is an integer pointer, it records the address of a *dynamically allocated integer array*.

As the elements should be initialized to 0 once the memory space is allocated, those statements should be forced to be executed together. Therefore, let's group these statements into a function. While we have two choices (global-function and member-function implementations), here we choose to implement a member function by expanding our definition of `MyVector`:

```
struct MyVector
{
  int n;              // the vector will be of dimension n
  int* m;             // m[i] should be the ith element
  void init(int dim); // allocate a space to m
};

void MyVector::init(int dim)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = 0;
}
```

The main function may then be modified as

```
MyVector a;
int dimension = 0;
cin >> dimension;
a.init(dimension);
```

Still, a `delete` statement should be put somewhere to release the space.

(a) Implement the function `init()` as a global function. Then compare the two implementations for pros and cons. Which one do you think is more natural or intuitive? Why?

(b) What bad things may happen if the following statements are executed?

```
MyVector a;
int dimension = 0;
cin >> dimension;
a.init(dimension);
cin >> a.n; // this is always allowed by the compiler
// some statements
delete [] a.m;
```

(c) What bad things may happen if the following statements are executed?

```
MyVector a;
int dimension = 0, dimension2 = 0;
cin >> dimension;
a.init(dimension);
cin >> dimension2;
a.init(dimention2);
// some statements
delete [] a.m;
```

(d) What bad things may happen if the following statements are executed?

```
MyVector a;
int dimension = 0;
cin >> dimension;
a.init(dimension);
// some statements, and then the main function is finished
```

(e) Implement a member function that returns true if all vector elements are nonnegative and false otherwise.

Note. With functionalities provided by `struct`, the bad things in Parts (b), (c), and (d) may always happen if the programmer is not careful enough. To minimize these risks, C++ "strengthens" structures into classes and provide mechanisms to prevent the above errors. These mechanisms will be introduced in the following weeks.

Problem 4

(50 basic points and 20 bonus points) Imagine that you are working on your final project due tomorrow morning in a study room. There are still a lot of works to do, and you, as your team leader, must assign tasks to your teammates. You have four teammates who all have the same ability for performing all the tasks. You find that there are twenty tasks to complete, each requires a different amount of working time. Once you assign the tasks to teammates, they will all try their best. Suppose you have all agreed to leave the study room after all teammates finish their assigned tasks, then the question is: How to assign tasks so that you may go home as early as possible?

The imaginary story actually happens in factories. In the so called *minimum makespan for identical machines* problem, one decision maker faces exactly the same problem as in the above story. There are n jobs to be assigned to m machines. The *processing time* of job j is q_j, i.e., it takes q_j units of time to complete job j. Please note that the machines are identical, which means the processing time of a job is independent of the machine. It is also assumed that the processing time is independent of the jobs to be processed before or after it. To formally describe this problem, let

$$x_{ij} = \begin{cases} 1 & \text{if job } j \text{ is assigned to machine } i \\ 0 & \text{otherwise} \end{cases}, i = 1, ..., m, j = 1, ..., n,$$

then the *completion time* of machine i is $\sum_{j=1}^{n} q_j x_{ij}$. With this definition, the decision maker's problem is to minimize the *makespan* (the largest completion time)

$$\min \max_{i=1,...,m} \sum_{j=1}^{n} q_j x_{ij}$$

subject to the constraints $x_{ij} \in \{0, 1\}$ for all $i = 1, ..., m$ and $j = 1, ..., n$.

Let's consider the following example. Suppose you have three machines and ten jobs. For jobs 1, 2, ..., and 10, their processing times q_js are 2, 2, 2, 5, 5, 6, 6, 6, 8, and 9, respectively. The total processing time of the ten jobs is $\sum_{j=1}^{10} q_j = 51$. Obviously, no schedule can generate a makespan smaller than $\frac{51}{3} = 17$. However, it is hard to find such a schedule, if it exists. In fact, it is hard enough to answer whether such a schedule exists: Because a job cannot be divided and assigned to multiple machines, it is possible that such an ideal result is not attainable. One not-so-good schedule is to assign jobs 1 to 4 to machine 1, jobs 5 to 7 to machine 2, and jobs 8 to 10 to machine 3. The resulting complete times of the three machines are 11, 17, and 23, respectively, and thus the makespan is 23. One better schedule is to assign jobs 1, 4, 5, and 6 to machine 1, jobs 2, 7, and 9 to machine 2, and jobs 3, 8, and 10 to machine 3. The completion times of the three machines are 18, 16, and 17, respectively. The makespan 18 is very close to the best possible makespan! However, we have no idea whether this schedule is an optimal schedule.

The minimum makespan for identical machines problem has been studied extensively in the past. It has been shown that the problem is *NP-hard*. Roughly speaking, most researchers believe that as long as a problem instance is large enough (having sufficiently many jobs and machines), no algorithm guarantees to find an optimal schedule in a reasonable time. Therefore, all we want will be to find a good schedule – as good as possible.[6]

In this problem, you will try to design and implement your own algorithm for finding a schedule to assign jobs to machines. You may use any algorithm, including any existing algorithm you may find online. Your program will be graded according to the performance of your algorithm, i.e., the smaller the makespan your schedule has, the higher grades you get. However, make sure that your algorithm is fast enough (and does not exceed the time limit of PDOGS). For example, though completely enumerating all possible schedules allows you to find an optimal schedule, it is too time-consuming and will not be able to solve some large problem instances given in the input in time.[7]

Input/output formats

The input will consist of 20 lines of positive integers. In each line, there are $n+2$ integers $(m, n, q_1, q_2, ..., q_n)$, where two consecutive integers are separated by a white space. The first integer m is the number of machines (labeled as 1, 2, ..., m), the second integer n is the number of jobs (labeled as 1, 2, ..., n), and q_i is the processing times of job i. It is given that $n \leq 10000$, $2 \leq m \leq 100$, $m < n$, and $q_i \leq 10000$ for $i = 1, ..., n$.

For each problem instance contained in a line, your program should output a schedule by assigning jobs to machines. A schedule is represented as $(J_1, J_2, ..., J_m)$, where J_i is the set of jobs assigned to machine i. For each set J_i, output the job numbers in an ascending order (i.e., from small to large). Between sets, output a star sign * for separation. Two consecutive symbols (including job numbers and star signs) should be separated by a white space.

As an example, suppose we have a line

```
3 10 2 2 2 5 5 6 6 6 8 9
```

as the input, a valid output can be

```
1 4 5 6 * 2 7 9 * 3 8 10
```

with, as always, a new line character appended at the end. Please note that within each set, job numbers must be output in an ascending order. Moreover, each job must be assigned to exactly one machine.

Grading criteria

As we mentioned, most researchers believe that there is no efficient algorithm for finding an optimal schedule for this program. Therefore, we do not encourage you to try to find one. To get all the 50 basic points, you only need to write a program that can generate a feasible schedule that is no worse than one found by a very weak algorithm. To win the 20 bonus points, try to do as well as you can. The detailed grading rules are below:

- 30 basic points and 20 bonus points will be based on the performance. PDGOS will compile your program, feed testing data into your program, verify whether you generate feasible solutions, and evaluate your performances for feasible solutions. For each line of input which represents a problem instance, we associate two numbers z_1 and z_2 to it, where z_1 is the outcome of a bad algorithm and z_2 is that of a better algorithm:

    1. z_1 is the makespan of the following naive schedule: Assigning job i to machine $\text{mod}(i-1, m)+1$, where $\text{mod}(a, b)$ is the remainder of a divided by b.

[6] The issue of NP-hardness will be discussed in details in the course "Algorithms".
[7] The issue of optimization will be discussed in details in the course "Operations Research".

2. z_2 is the makespan of a schedule generated by an algorithm that we will not tell you until the deadline. However, $z_2 \leq z_1$ is guaranteed.

If your output schedule is not feasible (e.g., if a job is assigned to two or no machine), you get 0 point. Suppose your output is feasible and the makespan of your schedule is z, the grades you get for this instance will be

$$\begin{cases} 1.5 & \text{if } z \in (z_1, \infty) \\ 1.5 + \dfrac{z_1 - z}{z_1 - z_2} & \text{if } z \in [z_2, z_1] \\ 2.5 & \text{if } z \in (0, z_2) \end{cases}$$

In other words, you get all basic points if you are able to implement exactly the naive algorithm that generates z_1. If you still have some time after it, we encourage you to try to do better to win some bonus points. As long as your algorithm beats ours (which is just an okay one), you get all the bonus points.

- 10 points will be based on how you write your program, including the logic and format. Please try to write a robust, efficient, and easy-to-read program.
- 10 points will be based on how you explain your algorithm. Please describe your algorithm in the PDF file for steps and the design rationale. It is preferred to use both words and pseudocodes. Please try to describe your algorithm in a precise and easy-to-understand way.