Processes	**LABORATORY**	**FOUR**
OBJECTIVES		
1. Unix Time 2. Programming with Processes		

Unix Time and Y2K Problem

In Unix or Linux, the *time* is simply a *counter* (of data type **`time_t`**) that counts the number of seconds from a specific date, namely, **`1 January 1970, 00:00:00`** and a value of 86399 means 1 January 1970, 23:59:59. The integer counter is of size 32 bits. In each day, there are 24 hours, i.e. 86400 seconds. Each year is about 86400 * 365.2422 = 31556926 seconds. The size of a 32-bit integer can count up to 2147483647, i.e. a little more than 68 years. So the Unix time counter will overflow by year 2038. Remember the **Y2K** problem that people are using two digits (bytes) for the year instead of four digits to save storage and by the year 2000, the date will wrap around to 1900. This **2038 problem** is the **Unix Y2K bug** or **Unix Millennium bug.** Many Unix programs would not function correctly by 19 January 2038 or when computing a date beyond that day. How many of you have heard of this **Unix Y2K bug**?

Programmers are already advised to write their programs with checking for the invalid entry for the date data type **`time_t`** to avoid this **2038 problem** (**Y2K38 problem**) on Unix and Linux. Newer architectures based on 64-bit integers would no longer suffer from this problem by replacing the **`time_t`** data type. However, people need time to switch to new architectures and need to watch out for legacy codes and programs that rely on this 32-bit time representation. Could you *estimate* ***when*** will an overflow to this new 64-bit time data type occur? Remember that we still need to handle the **Y10K problem** in another 8000 years, after solving the **Y2K problem**.

Try to study **lab4A.c** for the calendar logic, and figure out how this program can be executed. You should also try to learn the skill of *table-lookup* for fast result determination. This is a very useful programming skill. As a *simple exercise*, you could *extend* the checking logic for leap years for year 1900, 2000 and 2100. When the year ends in "00", it must be divisible by 400 to qualify as a leap year. So years 1900 and 2100 are ***not*** leap years, but year 2000 is a leap year.

```
// lab 4A
// try to find out how this calendar-related program can be executed
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  char *name[] = {"Jan", "Feb", "Mar", "Apr", "May", "June",
                  "July", "Aug", "Sept", "Oct", "Nov", "Dec"};
  int yy, mm, dd, numDay;
  int i;

  yy = atoi(argv[1]); numDay = atoi(argv[2]); // atoi returns an integer
  if (numDay <= 0 || numDay > 366 || (numDay == 366 && yy % 4 != 0)) {
     printf("Sorry: wrong number of days\n");
     exit(1);
  }
  if (yy % 4 == 0) month[1] = 29;
  for (i = 0; i < 12; i++) {
      if (numDay <= month[i]) break;
      numDay = numDay - month[i];
  }
  mm = i; dd = numDay;

  printf("Date is %d %s %4d\n", dd, name[mm], yy);
}
```

Programming with Processes

In Unix/Linux the way to create a process is to use the **`fork()`** system call. It will create a child process as an *exact copy* of the parent, including the value of the program counter. However, once **`fork()`** is executed, the two copies of parent and child will be *separated* and the variables are ***not shared***. Note that the action of copying only occurs on demand but logically, we could assume that it occurs when **`fork()`** is completed. To get the id of a process, we could use the system call **`getpid()`**. The **`sleep()`** system call allows a process to stop for the specific number of seconds. The **`exit()`** system call allows the process to terminate successfully.

Since the two copies of parent and child are separated after **`fork()`**, how could a child know who is the parent? A child can get the id of the parent by **`getppid()`**. Try to *observe the variable values* produced by parent and child processes and *draw a picture to illustrate* the changes in values in **lab4B.c**. Note that there is no guarantee that the parent after **`fork()`** is executed before or after the child. The actual execution order depends on the CPU scheduling mechanism. Vary the **`sleep()`** parameters and observe the outputs. You should be able to generate different outputs by properly changing the waiting time.

```
// lab 4B
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int  pid, myid, parentid, cid;
  int  val;

  val = 1;
  myid = getpid();
  printf("My id is %d, value is %d\n",myid,val);
  pid = fork();
  if (pid < 0) { // error occurred
     printf("Fork Failed\n");
     exit(1);
  } else if (pid == 0) { // child process
     myid = getpid(); parentid = getppid();
     printf("Child: My id is %d, my parent pid is %d\n",myid,parentid);
     printf("Child: Pid is %d, value is %d\n",pid,val);
     val = 12;
     printf("Child: My id is %d, value is %d\n",myid,val);
     sleep(4);
     printf("Child: My id is %d, value is %d\n",myid,val);
     val = 13;
     printf("Child: My id is %d, value is %d\n",myid,val);
     sleep(4);
     printf("Child: My id is %d, value is %d\n",myid,val);
     printf("Child: Child %d completed\n",myid);
     exit(0);
  } else { // parent process
     printf("Parent: My child id is %d\n",pid);
     printf("Parent: Pid is %d, value is %d\n",pid,val);
     val = 2;
     printf("Parent: My id is %d, value is %d\n",myid,val);
     sleep(4);
     printf("Parent: My id is %d, value is %d\n",myid,val);
     val = 3;
     printf("Parent: My id is %d, value is %d\n",myid,val);
     sleep(4);
     printf("Parent: My id is %d, value is %d\n",myid,val);
     cid = wait(NULL);
     printf("Parent: Child %d collected\n",cid);
     exit(0);
  }
}
```

Again, since the child is an exact copy of the parent, all values stored in the variables before **`fork()`** are exactly the same. All values stored in the argument list and all other variables are directly *inherited*

by the child. This serves as a simple mechanism of passing information from parent to child, in case the information is known before the child creation.

Here is a program to compute the GPA of subjects of different levels, still based on the old GPA system. Would there be any difference if we create the child process in the first statement, i.e. move **`fork()`** to the *beginning*? In other words, move the line **`pid = fork();`** to the first line, before **`num_subj = (argc-1)/2;`** and **`printf("There are %d subjects\n",num_subj);`** Try to answer the question *before* you actually change the program and run it to see the actual outcome.

```
// lab 4C
#include <stdio.h>
#include <stdlib.h>

float computeGP(char grade[])
{
  float gp;

  switch (grade[0]) {
     case 'A': gp = 4.0; break;
     case 'B': gp = 3.0; break;
     case 'C': gp = 2.0; break;
     case 'D': gp = 1.0; break;
     case 'F': gp = 0.0; break;
     default: printf("Wrong grade %s\n",grade);
          return -1.0; // use a negative number to indicate an error
  }
  if (grade[1] == '+') gp = gp + 0.3;
  if (grade[1] == '-') gp = gp - 0.3;
  return gp;
}

int main(int argc, char *argv[])
{
  float gp, sum_gp = 0.0;
  int pid, cid;
  int num_subj, sub_code, count;
  int i;

  num_subj = (argc-1)/2;
  printf("There are %d subjects\n",num_subj);

  pid = fork();
  if (pid < 0) { // error occurred
     printf("Fork Failed\n");
     exit(1);
  } else if (pid == 0) { // child process: handle level 3 and level 4 subjects
     count = 0;
     for (i = 1; i <= num_subj; i++) {
          sub_code = atoi(argv[i*2-1]); // convert into integer
          if (sub_code >= 3000) {
             gp = computeGP(argv[i*2]);
             if (gp >= 0) { count++; sum_gp += gp; }
          }
     }
     printf("Child: ");
     printf("Your GPA for %d level 3/4 subjects is%5.2f\n",count,sum_gp/count);
     exit(0);
  } else { // parent process: handle level 1 and level 2 subjects
     count = 0;
     for (i = 1; i <= num_subj; i++) {
          sub_code = atoi(argv[i*2-1]); // convert into integer
          if (sub_code < 3000) {
             gp = computeGP(argv[i*2]);
             if (gp >= 0) { count++; sum_gp += gp; }
          }
     }
     printf("Parent: ");
     printf("Your GPA for %d level 1/2 subjects is%5.2f\n",count,sum_gp/count);
     cid = wait(NULL);
     printf("Parent: Child %d collected\n",cid);
     exit(0);
  }
}
```

You could modify the program with the technique in **lab4A.c** to print out English words rather than just numbers (assuming an upper bound on the number of subjects). A simple table lookup is often more effective than using a long list of **`switch/case`** statements. You could further compute the *weighted GPA* once you separate them into levels, i.e. level 1 and 2 subjects have a weight of 2 and level 3 and 4 subjects have a weight of 3. Do you find a bug there in **lab4C.c**?

```
lab4C 1011 B 3121 C+ 2011 C+ 4122 B- 3911 B+
There are five subjects
Child: Your GPA for three level 3/4 subjects is 2.77
Parent: Your GPA for two level 1/2 subjects is 2.65
Parent: Child 13579 collected
```

A simple way for the child process to return some brief information (just a byte from 0 to 255) back to the parent is to make use of the **`exit()`** system call, riding on the Unix/Linux mechanism of exit status reporting. By convention, any user or system program would return `0` if the program is executed successfully and non-zero otherwise. However, as a C programmer, it may be convenient to make use of this exit status for a child to return some other agreed-upon value back to the parent. Note that this is more a workaround approach to get return values, than a formal communication mechanism between parent and child.

An example for the parent to get back what the child would be returning via the **`exit()`** system call is shown. Instead of passing in **`NULL`** as the argument, you pass in *a pointer to an integer* to the **`wait()`** system call. Then the integer will contain the exit status in its *least significant byte*. Since **apollo** or **apollo2** is a little Endian machine in terms of hardware, you can extract this return value by dividing it by 256 and taking the quotient. That is not a good approach, since the program is *not portable* to a big Endian machine. A better way is to use the macro **`WEXITSTATUS()`** to extract the exit status as a byte, which works regardless of the machine. Try changing the exit status value in the child process and see how the changed value is returned from the child to the parent.

```
// lab 4D
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int  pid, myid, parentid, cid;
  int  ret, cret;

  myid = getpid();
  pid = fork();
  if (pid < 0) { // error occurred
     printf("Fork Failed\n");
     exit(1);
  } else if (pid == 0) { // child process
     myid = getpid();
     parentid = getppid();
     printf("Child: My id is %d, my parent pid is %d\n",myid,parentid);
     // do some calculation here
     ret = 3;
     exit(ret);
  } else { // parent process
     printf("Parent: My child id is %d\n",pid);
     cid = wait(&cret);
     printf("Parent: Child %d collected\n",cid);
     printf("Parent: Child returning value %d vs %d\n",cret,WEXITSTATUS(cret));
     exit(0);
  }
}
```

Orphan Processes (Optional)

In Unix, if a parent process terminates before a child process terminates, the child process would become an orphan will be adopted by a new parent. Orphan processes will be adopted by process **init** with pid 1, which becomes the *new parent*. This approach is also used by Linux. Run the following program in *background* (**`lab4E o &`**) on **apollo2** and on a **Mac**. Type **`ps –lf`** and you will see that the orphan child process has been ***adopted*** by the process called **init** with pid **1** (look at the **PPID** column of information). This process with pid 1 is the **systemd** (**system daemon**) process on CentOS Linux. If a child terminates before a parent, it would become a zombie until waited or collected by its parent via **`wait()`** system call before the parent terminates. Run the program in *background* again (**`lab4E z &`**). Type **`ps –lf`** and you will see a *zombie process*. A zombie process is identified by a "**Z**" state (on the "**S**" or "**STAT**" column). Alternatively, you may use multiple windows to run and check, without putting the processes to the background.

```
// lab 4E : o means orphan and z means zombie
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  int pid;

  pid = fork();
  if (pid < 0) { // an error occurred
     printf("Fork Failed\n");
     exit(1);
  } else if (pid == 0) { // child process
     if (argv[1][0] == 'o')
        sleep(30); // child still executing when parent terminates
     printf("Child completed\n");
     exit(0);
  } else { // parent process
     // parent does not wait for child
     if (argv[1][0] == 'z')
         sleep(30); // child completes before parent
     printf("Parent completed\n");
     exit(0);
  }
}
```

Laboratory Exercise

Making use of **lab4C.c** for child process generation, you can create a program that will *generate **separate** child processes* to help dividing up a lot of processing tasks to be handled by each individual child. Effective division of labors among many processes actually forms the basis of ***cloud computing*** and ***big data processing***, making use of the computing power of multiple processors.

One common application to make use of abundant computing power is to perform *simulation study*. For simple questions concerning probabilistic results, mathematical analysis could be performed, e.g. the probability of getting 12 with two dice is 1/36 and the probability of getting 7 is 1/6. It is still possible to compute the probability of getting 40 with 10 dice. Likewise, one can compute the unconditional probability of playing for a 2/2 split in bridge with an analytic solution, but the conditional probability when some of the cards are known will deviate from the analytic unconditional probability. One way to estimate this probability is to *simulate*. In other words, generate many hands, say N, under the existing constraints and count the number of hands, h, satisfying the required condition. The required probability will then be h/N. Simple simulation could be applied to study queuing systems, e.g. performance of FCFS versus Priority scheduling algorithms under CPU scheduling. More complex computer simulation could be applied to study the infection model in epidemic study, typhoon trajectory prediction in meteorology, stock price change in financial technology, or pilot training in flight simulation.

In this exercise, you are to estimate the probability of teams facing each other in the knockout stage of the UEFA Champions League via simulation. Due to a possibly buggy drawing this year, there was a controversy on whether there should be a partial re-drawing or a complete re-drawing, after an illegal fixture was drawn, i.e. Manchester United was drawn against Villarreal, both belonging to the same group. You are not answering this question, but instead would like to study the probability of getting rivals facing each other. You might be surprised about the actual probabilities.

In the UEFA Champions League, there are 16 teams coming out of 8 groups, with 8 teams being the first-place team and 8 teams being the second-place team. Each first-place team will play a second-place team randomly. However, there are some illegal cases. In particular, the two teams in a match-up *cannot come from the same group*. Furthermore, the two teams *cannot come from the same country*. You are to randomly draw up the fixture for the 16 teams and decide whether your fixture is legal or not. Here are the groups and the teams. It can be seen that both Manchester United and Villarreal are from Group F.

A	B	C	D	E	F	G	H
ManCity	Liverpool	Ajax	RealMadrid	Bayern	ManUnited	Lille	Juventus
PSG	Atletico	Sporting	InterMilan	Benfica	Villarreal	Salzburg	Chelsea

Here is the information about the country of the teams. **Austria**: Salzburg; **England**: Chelsea, Liverpool, ManCity, ManUnited; **France**: Lille, PSG; **Germany**: Bayern; **Italy**: InterMilan, Juventus; **Netherlands**: Ajax; **Portugal**: Benfica, Sporting; **Spain**: Atletico, RealMadrid, Villarreal.

Here is an example of a legal fixture: Salzburg vs Bayern; Sporting vs ManCity; Benfica vs Ajax; Chelsea vs Lille; Atletico vs ManUnited; Villarreal vs Juventus; InterMilan vs Liverpool; PSG vs RealMadrid.

Here is an example of an illegal fixture: Salzburg vs Bayern; Sporting vs ManCity; Benfica vs Ajax; Chelsea vs Lille; Villarreal vs ManUnited; Atletico vs Juventus; InterMilan vs Liverpool; PSG vs RealMadrid.

Here is another example of an illegal fixture: Salzburg vs Bayern; Sporting vs ManCity; Benfica vs Ajax; Chelsea vs ManUnited; Atletico vs Lille; Villarreal vs Juventus; InterMilan vs Liverpool; PSG vs RealMadrid.

Here is yet another example of an illegal fixture: Salzburg vs Bayern; Sporting vs ManCity; Benfica vs Ajax; Juventus vs Lille; Atletico vs ManUnited; Villarreal vs Chelsea; InterMilan vs Liverpool; PSG vs RealMadrid.

You might think that it is not too hard to generate a legal fixture. However, given the 16 teams in this 2021-22 Champions League season, the probability of drawing up a legal fixture is simply less than

12%, when first-place teams are randomly drawn against second-place teams. In other words, if you repeat 100000 times, you would have less than 12000 legal fixtures. Larger number of attempts would lead to more accurate estimates.

Random numbers are the key to simulation study. They are generated via a *pseudo-random number generator*, which is able to generate seemingly "random" numbers good enough for most practical applications. Yet, it is able to repeat the same sequence of numbers generated given the same ***seed***. That is very useful in simulation study for cross-comparison and for debugging. The most common generator is the *linear congruential generator*, provided in standard C library. In this exercise, you are to make use of a pseudo-random number generator by controlling the seed, thus the outcome. The outcomes are *deterministic* and this would facilitate debugging. If you want the program to really randomly generate fixtures, set the seed to be the *current time* (which would differ each time the program is executed; see **`time_t`** above with the **`tv_usec`** field).

```
#include <stdlib.h>
  . . .
  int seed = 2022;
  . . .
  srand(seed); // do this only once and before using rand() to generate
  . . .
  while . . . {
      . . .
      r = rand() % 8 + 1; // generate a random number from 1 to 8
      . . .
  }
  . . .
```

In this exercise, there are ***n*** child processes. The parent is just like a *server* that accepts user request and passes it to a child process to do the evaluation. Remember that a child will know *everything* about the parent before **`fork()`** is executed, so there is no need for the parent to pass in any argument to the child (and this simplifies the program design). The child should be able *to extract its own part* of the input data knowing its position from the argument list or from any stored array before **`fork()`**. To facilitate program development and debugging, the program would be executed in different modes, as indicated by the ***first*** argument. Depending on the mode, other arguments should be interpreted accordingly. The modes to be provided include *debug mode* (**D**) and *estimation mode* (**E**). Note that if we are able to read the team information from a file, it will greatly simplify the way to test the program, without the need to get the information from the argument list. You can assume that there are no errors in the user input argument list. Do not forget to ***wait*** for the child to complete to avoid *zombie processes*.

In the ***debug mode***, there will be up to 48 additional arguments. The first 32 arguments represent the advancing teams and their countries, in the order from Group A to Group H. The remaining arguments represent the full or partial fixture to be tested, in the format of S_1 F_1 S_2 F_2, … S_m F_m, where S_i are second-place teams, and F_i first-place teams.

In the ***estimation mode***, there will be $1+2t+32+I$ additional arguments, where t is the number of tasks. The ***second*** argument to the program is the number of child processes, ***n***. This is followed by the t tasks. Each of the t tasks is specified by 2 arguments: number of fixtures to be generated and the seed for random number generator. The next 32 arguments contain the team and country information in the same format as in the debug mode. The final I optional arguments contain the information of teams of interest. Each task is performed by each child in a *round-robin* manner. If there are more tasks than the number of child processes, some children would take up more than one task. The simulations are only performed by child processes, and the parent is not doing any real work. Each child will finally estimate the probabilities that the teams of interest will be facing each other potential teams. Note that I may be zero, i.e., there is no team of interest and that part can be ignored. You can test the impact on the change of probabilities of potential match-up when advancing teams are slightly varied. For example, if Chelsea ended up being the top team in Group H, this would have affected the fates of other English teams (see the last example).

In this exercise, the children processes are producing results independently. A better solution is to let each child report the result back. Then the parent process will *combine* together the results computed by

the children. Remember that more attempts in the simulation will lead to more accurate estimations. That is the concept behind *cloud computing*, based on the famous ***map/reduce*** paradigm. Inputs/simulation requests are divided up and *mapped* to different child processes; the parent then gets back the partial results and consolidates them via *reduction*. That would require the ability that the child processes can report the computed probabilities back to the parent (i.e., ability of inter-process communication). These complicated arrangements are not required in this exercise.

Please provide ***appropriate comments*** and check to make sure that your program can be ***compiled*** and ***execute*** properly under the Linux (**apollo** or **apollo2**) environment before submission. Note that you have to let all children ***execute together concurrently*** and ***must not*** control the output order from different children. The final results produced by all the children may be in any *mixed order* and you *do not need* to do anything to change the output order. Just let the nature play the output game for you. It is quite likely that those tasks generating more fixtures would execute more slowly, but it also depends on the early termination of illegal fixtures determination in your implementation.

Sample executions:

```
fixture D ManCity England PSG France Liverpool England Atletico Spain Ajax Netherlands
Sporting Portugal RealMadrid Spain InterMilan Italy Bayern Germany Benfica Portugal
ManUnited England Villarreal Spain Lille France Salzburg Austria Juventus Italy
Chelsea England Salzburg Bayern Sporting ManCity Benfica Ajax Chelsea Lille Atletico
ManUnited Villarreal Juventus InterMilan Liverpool PSG RealMadrid

fixture D ManCity England PSG France Liverpool England Atletico Spain Ajax Netherlands
Sporting Portugal RealMadrid Spain InterMilan Italy Bayern Germany Benfica Portugal
ManUnited England Villarreal Spain Lille France Salzburg Austria Juventus Italy
Chelsea England Salzburg Bayern Sporting ManCity Benfica Ajax Chelsea Lille Villarreal
ManUnited Atletico Juventus InterMilan Liverpool PSG RealMadrid

fixture D ManCity England PSG France Liverpool England Atletico Spain Ajax Netherlands
Sporting Portugal RealMadrid Spain InterMilan Italy Bayern Germany Benfica Portugal
ManUnited England Villarreal Spain Lille France Salzburg Austria Juventus Italy
Chelsea England Salzburg Bayern Sporting ManCity Benfica Ajax Chelsea Lille Atletico
Juventus InterMilan Liverpool PSG RealMadrid

fixture E 3 10000 2022 20000 2023 ManCity England PSG France Liverpool England
Atletico Spain Ajax Netherlands Sporting Portugal RealMadrid Spain InterMilan Italy
Bayern Germany Benfica Portugal ManUnited England Villarreal Spain Lille France
Salzburg Austria Juventus Italy Chelsea England

fixture E 3 100000 2022 200000 2023 300000 2024 400000 2025 500000 2026 600000 2027
100000 2028 ManCity England PSG France Liverpool England Atletico Spain Ajax
Netherlands Sporting Portugal RealMadrid Spain InterMilan Italy Bayern Germany Benfica
Portugal ManUnited England Villarreal Spain Lille France Salzburg Austria Juventus
Italy Chelsea England

fixture E 3 1000000 2021 2000000 2022 1500000 2023 2500000 2024 ManCity England PSG
France Liverpool England Atletico Spain Ajax Netherlands Sporting Portugal RealMadrid
Spain InterMilan Italy Bayern Germany Benfica Portugal ManUnited England Villarreal
Spain Lille France Salzburg Austria Juventus Italy Chelsea England Chelsea

fixture E 3 1000000 2021 4000000 2022 ManCity England PSG France Liverpool England
Atletico Spain Ajax Netherlands Sporting Portugal RealMadrid Spain InterMilan Italy
Bayern Germany Benfica Portugal ManUnited England Villarreal Spain Lille France
Salzburg Austria Juventus Italy Chelsea England Chelsea ManCity Liverpool ManUnited

fixture E 4 1000000 2021 4000000 2022 ManCity England PSG France Liverpool England
Atletico Spain Ajax Netherlands Sporting Portugal RealMadrid Spain InterMilan Italy
Bayern Germany Benfica Portugal ManUnited England Villarreal Spain Lille France
Salzburg Austria Chelsea England Juventus Italy Chelsea ManCity Liverpool ManUnited
```

Sample outputs:

```
Parent, pid 12344: debug mode
Child 1, pid 12345: Group A: ManCity (England) and PSG (France)
Child 1, pid 12345: Group B: Liverpool (England) and Atletico (Spain)
Child 1, pid 12345: Group C: Ajax (Netherlands) and Sporting (Portugal)
Child 1, pid 12345: Group D: RealMadrid (Spain) and InterMilan (Italy)
Child 1, pid 12345: Group E: Bayern (Germany) and Benfica (Portugal)
Child 1, pid 12345: Group F: ManUnited (England) and Villarreal (Spain)
```

```
Child 1, pid 12345: Group G: Lille (France) and Salzburg (Austria)
Child 1, pid 12345: Group H: Juventus (Italy) and Chelsea (England)
Child 1, pid 12345: Salzburg (Austria) vs Bayern (Germany)
Child 1, pid 12345: Sporting (Portugal) vs ManCity (England)
Child 1, pid 12345: Benfica (Portugal) vs Ajax (Netherlands)
Child 1, pid 12345: Chelsea (England) vs Lille (France)
Child 1, pid 12345: Atletico (Spain) vs ManUnited (England)
Child 1, pid 12345: Villarreal (Spain) vs Juventus (Italy)
Child 1, pid 12345: InterMilan (Italy) vs Liverpool (England)
Child 1, pid 12345: PSG (France) vs RealMadrid (Spain)
Child 1, pid 12345: Legal fixture
Parent, pid 12344: 1 child completed execution
```

```
Parent, pid 12348: debug mode
Child 1, pid 12349: Group A: ManCity (England) and PSG (France)
. . .
Child 1, pid 12349: Group H: Juventus (Italy) and Chelsea (England)
Child 1, pid 12349: Salzburg (Austria) vs Bayern (Germany)
Child 1, pid 12349: Sporting (Portugal) vs ManCity (England)
Child 1, pid 12349: Benfica (Portugal) vs Ajax (Netherlands)
Child 1, pid 12349: Chelsea (England) vs Lille (France)
Child 1, pid 12349: Villarreal (Spain) vs ManUnited (England)
Child 1, pid 12349: Atletico (Spain) vs Juventus (Italy)
Child 1, pid 12349: InterMilan (Italy) vs Liverpool (England)
Child 1, pid 12349: PSG (France) vs RealMadrid (Spain)
Child 1, pid 12349: Illegal fixture
Parent, pid 12348: 1 child completed execution
```

```
Parent, pid 12358: debug mode
Child 1, pid 12359: Group A: ManCity (England) and PSG (France)
. . .
Child 1, pid 12359: Group H: Juventus (Italy) and Chelsea (England)
Child 1, pid 12359: Salzburg (Austria) vs Bayern (Germany)
Child 1, pid 12359: Sporting (Portugal) vs ManCity (England)
Child 1, pid 12359: Benfica (Portugal) vs Ajax (Netherlands)
Child 1, pid 12359: Chelsea (England) vs Lille (France)
Child 1, pid 12359: Atletico (Spain) vs Juventus (Italy)
Child 1, pid 12359: InterMilan (Italy) vs Liverpool (England)
Child 1, pid 12359: PSG (France) vs RealMadrid (Spain)
Child 1, pid 12359: Legal fixture
Parent, pid 12358: 1 child completed execution
```

```
Parent, pid 23456: 3 children, 2 tasks, estimation mode
Child 3, pid 23459: No task assigned
Child 1, pid 23457: Group A: ManCity (England) and PSG (France)
. . .
Child 1, pid 23457: Group H: Juventus (Italy) and Chelsea (England)
Child 2, pid 23458: Group A: ManCity (England) and PSG (France)
. . .
Child 2, pid 23458: Group H: Juventus (Italy) and Chelsea (England)
Child 1, pid 23457: Seed 2022 with 1125 legal fixture out of 10000 (11.250%)
Child 2, pid 23458: Seed 2023 with 2300 legal fixture out of 20000 (11.500%)
Parent, pid 23456: 3 children completed execution
```

```
Parent, pid 23465: 3 children, 7 tasks, estimation mode
Child 1, pid 23466: Group A: ManCity (England) and PSG (France)
. . .
Child 1, pid 23466: Group H: Juventus (Italy) and Chelsea (England)
Child 2, pid 23467: Group A: ManCity (England) and PSG (France)
. . .
Child 2, pid 23467: Group H: Juventus (Italy) and Chelsea (England)
Child 3, pid 23468: Group A: ManCity (England) and PSG (France)
. . .
Child 3, pid 23468: Group H: Juventus (Italy) and Chelsea (England)
Child 1, pid 23466: Seed 2022 with 11564 legal fixture out of 100000 (11.564%)
Child 2, pid 23467: Seed 2023 with 23486 legal fixture out of 200000 (11.743%)
Child 1, pid 23466: Seed 2028 with 11674 legal fixture out of 100000 (11.674%)
Child 3, pid 23468: Seed 2024 with 35227 legal fixture out of 300000 (11.743%)
```

```
Child 1, pid 23466: Seed 2025 with 46995 legal fixture out of 400000 (11.749%)
Child 2, pid 23467: Seed 2026 with 58364 legal fixture out of 500000 (11.673%)
Child 3, pid 23468: Seed 2027 with 69927 legal fixture out of 600000 (11.655%)
Parent, pid 23465: 3 children completed execution
```

```
Parent, pid 23481: 3 children, 4 tasks, estimation mode
Child 1, pid 23482: Group A: ManCity (England) and PSG (France)
. . .
Child 1, pid 23482: Group H: Juventus (Italy) and Chelsea (England)
Child 2, pid 23483: Group A: ManCity (England) and PSG (France)
. . .
Child 2, pid 23483: Group H: Juventus (Italy) and Chelsea (England)
Child 3, pid 23484: Group A: ManCity (England) and PSG (France)
. . .
Child 3, pid 23484: Group H: Juventus (Italy) and Chelsea (England)
Child 1, pid 23482: Seed 2021 with 117474 legal fixture out of 1000000 (11.747%)
Child 1, pid 23482: Statistics for Chelsea
Child 1, pid 23482: Chelsea vs RealMadrid - 37657 out of 117474 fixture (32.056%)
Child 1, pid 23482: Chelsea vs Lille - 29689 out of 117474 fixture (25.273%)
Child 1, pid 23482: Chelsea vs Bayern - 25150 out of 117474 fixture (21.409%)
Child 1, pid 23482: Chelsea vs Ajax - 24978 out of 117474 fixture (21.263%)
Child 3, pid 23484: Seed 2023 with 176957 legal fixture out of 1500000 (11.797%)
Child 3, pid 23484: Statistics for Chelsea
Child 3, pid 23484: Chelsea vs RealMadrid - 57088 out of 176957 fixture (32.261%)
Child 3, pid 23484: Chelsea vs Lille - 44693 out of 176957 fixture (25.256%)
Child 3, pid 23484: Chelsea vs Bayern - 37709 out of 176957 fixture (21.310%)
Child 3, pid 23484: Chelsea vs Ajax - 37467 out of 176957 fixture (21.173%)
Child 2, pid 23483: Seed 2022 with 234503 legal fixture out of 2000000 (11.725%)
Child 2, pid 23483: Statistics for Chelsea
Child 2, pid 23483: Chelsea vs RealMadrid - 75447 out of 234503 fixture (32.173%)
Child 2, pid 23483: Chelsea vs Lille - 59181 out of 234503 fixture (25.237%)
Child 2, pid 23483: Chelsea vs Bayern - 50097 out of 234503 fixture (21.363%)
Child 2, pid 23483: Chelsea vs Ajax - 49778 out of 234503 fixture (21.227%)
Child 1, pid 23482: Seed 2024 with 293932 legal fixture out of 2500000 (11.757%)
Child 1, pid 23482: Statistics for Chelsea
Child 1, pid 23482: Chelsea vs RealMadrid - 94031 out of 293932 fixture (31.991%)
Child 1, pid 23482: Chelsea vs Lille - 74734 out of 293932 fixture (25.426%)
Child 1, pid 23482: Chelsea vs Ajax - 62630 out of 293932 fixture (21.308%)
Child 1, pid 23482: Chelsea vs Bayern - 62537 out of 293932 fixture (21.276%)
Parent, pid 23481: 3 children completed execution
```

```
Parent, pid 23992: 3 children, 2 tasks, estimation mode
. . .
Child 3, pid 23995: No task assigned
Child 1, pid 23993: Seed 2021 with 117474 legal fixture out of 1000000 (11.747%)
Child 1, pid 23993: Statistics for Chelsea
Child 1, pid 23993: Chelsea vs RealMadrid - 37657 out of 117474 fixture (32.056%)
Child 1, pid 23993: Chelsea vs Lille - 29689 out of 117474 fixture (25.273%)
Child 1, pid 23993: Chelsea vs Bayern - 25150 out of 117474 fixture (21.409%)
Child 1, pid 23993: Chelsea vs Ajax - 24978 out of 117474 fixture (21.263%)
Child 1, pid 23993: Statistics for ManCity
Child 1, pid 23993: ManCity vs InterMilan - 21869 out of 117474 fixture (18.616%)
Child 1, pid 23993: ManCity vs Villarreal - 21746 out of 117474 fixture (18.511%)
Child 1, pid 23993: ManCity vs Atletico - 21706 out of 117474 fixture (18.477%)
Child 1, pid 23993: ManCity vs Sporting - 17528 out of 117474 fixture (14.921%)
Child 1, pid 23993: ManCity vs Benfica - 17313 out of 117474 fixture (14.738%)
Child 1, pid 23993: ManCity vs Salzburg - 17312 out of 117474 fixture (14.737%)
Child 1, pid 23993: Statistics for Liverpool
Child 1, pid 23993: Liverpool vs Villarreal - 21897 out of 117474 fixture (18.640%)
Child 1, pid 23993: Liverpool vs InterMilan - 21884 out of 117474 fixture (18.629%)
Child 1, pid 23993: Liverpool vs PSG - 20945 out of 117474 fixture (17.829%)
Child 1, pid 23993: Liverpool vs Salzburg - 17789 out of 117474 fixture (15.143%)
Child 1, pid 23993: Liverpool vs Sporting - 17574 out of 117474 fixture (14.960%)
Child 1, pid 23993: Liverpool vs Benfica - 17385 out of 117474 fixture (14.799%)
Child 1, pid 23993: Statistics for ManUnited
Child 1, pid 23993: ManUnited vs InterMilan - 21882 out of 117474 fixture (18.627%)
Child 1, pid 23993: ManUnited vs Atletico - 21821 out of 117474 fixture (18.575%)
Child 1, pid 23993: ManUnited vs PSG - 20800 out of 117474 fixture (17.706%)
Child 1, pid 23993: ManUnited vs Salzburg - 17982 out of 117474 fixture (15.307%)
```

```
Child 1, pid 23993: ManUnited vs Benfica - 17596 out of 117474 fixture (14.979%)
Child 1, pid 23993: ManUnited vs Sporting - 17393 out of 117474 fixture (14.806%)
Child 2, pid 23994: Seed 2022 with 469391 legal fixture out of 4000000 (11.735%)
Child 2, pid 23994: Statistics for Chelsea
Child 2, pid 23994: Chelsea vs RealMadrid - 151201 out of 469391 fixture (32.212%)
Child 2, pid 23994: Chelsea vs Lille - 118308 out of 469391 fixture (25.205%)
Child 2, pid 23994: Chelsea vs Bayern - 100323 out of 469391 fixture (21.373%)
Child 2, pid 23994: Chelsea vs Ajax - 99559 out of 469391 fixture (21.210%)
Child 2, pid 23994: Statistics for ManCity
Child 2, pid 23994: ManCity vs InterMilan - 87129 out of 469391 fixture (18.562%)
Child 2, pid 23994: ManCity vs Villarreal - 86997 out of 469391 fixture (18.534%)
Child 2, pid 23994: ManCity vs Atletico - 86965 out of 469391 fixture (18.527%)
Child 2, pid 23994: ManCity vs Salzburg - 70031 out of 469391 fixture (14.920%)
Child 2, pid 23994: ManCity vs Benfica - 69310 out of 469391 fixture (14.766%)
Child 2, pid 23994: ManCity vs Sporting - 68959 out of 469391 fixture (14.691%)
Child 2, pid 23994: Statistics for Liverpool
Child 2, pid 23994: Liverpool vs Villarreal - 87171 out of 469391 fixture (18.571%)
Child 2, pid 23994: Liverpool vs InterMilan - 87154 out of 469391 fixture (18.567%)
Child 2, pid 23994: Liverpool vs PSG - 83949 out of 469391 fixture (17.885%)
Child 2, pid 23994: Liverpool vs Salzburg - 71309 out of 469391 fixture (15.192%)
Child 2, pid 23994: Liverpool vs Sporting - 70142 out of 469391 fixture (14.943%)
Child 2, pid 23994: Liverpool vs Benfica - 69666 out of 469391 fixture (14.842%)
Child 2, pid 23994: Statistics for ManUnited
Child 2, pid 23994: ManUnited vs InterMilan - 87370 out of 469391 fixture (18.613%)
Child 2, pid 23994: ManUnited vs Atletico - 87157 out of 469391 fixture (18.568%)
Child 2, pid 23994: ManUnited vs PSG - 83627 out of 469391 fixture (17.816%)
Child 2, pid 23994: ManUnited vs Salzburg - 71627 out of 469391 fixture (15.260%)
Child 2, pid 23994: ManUnited vs Benfica - 69946 out of 469391 fixture (14.901%)
Child 2, pid 23994: ManUnited vs Sporting - 69664 out of 469391 fixture (14.841%)
Parent, pid 23992: 3 children completed execution
```

```
Parent, pid 24123: 4 children, 2 tasks, estimation mode
. . .
Child 4, pid 24127: No task assigned
Child 3, pid 24126: No task assigned
Child 1, pid 24124: Seed 2021 with 226285 legal fixture out of 1000000 (22.629%)
Child 1, pid 24124: Statistics for Chelsea
Child 1, pid 24124: Chelsea vs Villarreal - 36430 out of 226285 fixture (16.099%)
Child 1, pid 24124: Chelsea vs Atletico - 36195 out of 226285 fixture (15.995%)
Child 1, pid 24124: Chelsea vs PSG - 34362 out of 226285 fixture (15.185%)
Child 1, pid 24124: Chelsea vs InterMilan - 30901 out of 226285 fixture (13.656%)
Child 1, pid 24124: Chelsea vs Salzburg - 30253 out of 226285 fixture (13.369%)
Child 1, pid 24124: Chelsea vs Benfica - 29129 out of 226285 fixture (12.873%)
Child 1, pid 24124: Chelsea vs Sporting - 29015 out of 226285 fixture (12.822%)
Child 1, pid 24124: Statistics for ManCity
Child 1, pid 24124: ManCity vs Atletico - 37398 out of 226285 fixture (16.527%)
Child 1, pid 24124: ManCity vs Villarreal - 37172 out of 226285 fixture (16.427%)
Child 1, pid 24124: ManCity vs InterMilan - 31849 out of 226285 fixture (14.075%)
Child 1, pid 24124: ManCity vs Juventus - 30176 out of 226285 fixture (13.335%)
Child 1, pid 24124: ManCity vs Salzburg - 30029 out of 226285 fixture (13.270%)
Child 1, pid 24124: ManCity vs Sporting - 29899 out of 226285 fixture (13.213%)
Child 1, pid 24124: ManCity vs Benfica - 29762 out of 226285 fixture (13.152%)
Child 1, pid 24124: Statistics for Liverpool
Child 1, pid 24124: Liverpool vs Villarreal - 37322 out of 226285 fixture (16.493%)
Child 1, pid 24124: Liverpool vs PSG - 35746 out of 226285 fixture (15.797%)
Child 1, pid 24124: Liverpool vs InterMilan - 32028 out of 226285 fixture (14.154%)
Child 1, pid 24124: Liverpool vs Salzburg - 30688 out of 226285 fixture (13.562%)
Child 1, pid 24124: Liverpool vs Sporting - 30397 out of 226285 fixture (13.433%)
Child 1, pid 24124: Liverpool vs Benfica - 30094 out of 226285 fixture (13.299%)
Child 1, pid 24124: Liverpool vs Juventus - 30010 out of 226285 fixture (13.262%)
Child 1, pid 24124: Statistics for ManUnited
Child 1, pid 24124: ManUnited vs Atletico - 37063 out of 226285 fixture (16.379%)
Child 1, pid 24124: ManUnited vs PSG - 35420 out of 226285 fixture (15.653%)
Child 1, pid 24124: ManUnited vs InterMilan - 32140 out of 226285 fixture (14.203%)
Child 1, pid 24124: ManUnited vs Salzburg - 31022 out of 226285 fixture (13.709%)
Child 1, pid 24124: ManUnited vs Benfica - 30510 out of 226285 fixture (13.483%)
Child 1, pid 24124: ManUnited vs Juventus - 30385 out of 226285 fixture (13.428%)
Child 1, pid 24124: ManUnited vs Sporting - 29745 out of 226285 fixture (13.145%)
Child 2, pid 24125: Seed 2022 with 903032 legal fixture out of 4000000 (22.576%)
```

```
Child 2, pid 24125: Statistics for Chelsea
Child 2, pid 24125: Chelsea vs Villarreal - 144477 out of 903032 fixture (15.999%)
Child 2, pid 24125: Chelsea vs Atletico - 143748 out of 903032 fixture (15.918%)
Child 2, pid 24125: Chelsea vs PSG - 137427 out of 903032 fixture (15.218%)
Child 2, pid 24125: Chelsea vs InterMilan - 124103 out of 903032 fixture (13.743%)
Child 2, pid 24125: Chelsea vs Salzburg - 121169 out of 903032 fixture (13.418%)
Child 2, pid 24125: Chelsea vs Sporting - 116066 out of 903032 fixture (12.853%)
Child 2, pid 24125: Chelsea vs Benfica - 116042 out of 903032 fixture (12.850%)
Child 2, pid 24125: Statistics for ManCity
Child 2, pid 24125: ManCity vs Villarreal - 148433 out of 903032 fixture (16.437%)
Child 2, pid 24125: ManCity vs Atletico - 148419 out of 903032 fixture (16.436%)
Child 2, pid 24125: ManCity vs InterMilan - 127487 out of 903032 fixture (14.118%)
Child 2, pid 24125: ManCity vs Salzburg - 121127 out of 903032 fixture (13.413%)
Child 2, pid 24125: ManCity vs Juventus - 119512 out of 903032 fixture (13.235%)
Child 2, pid 24125: ManCity vs Benfica - 119308 out of 903032 fixture (13.212%)
Child 2, pid 24125: ManCity vs Sporting - 118746 out of 903032 fixture (13.150%)
Child 2, pid 24125: Statistics for Liverpool
Child 2, pid 24125: Liverpool vs Villarreal - 147791 out of 903032 fixture (16.366%)
Child 2, pid 24125: Liverpool vs PSG - 142373 out of 903032 fixture (15.766%)
Child 2, pid 24125: Liverpool vs InterMilan - 127721 out of 903032 fixture (14.144%)
Child 2, pid 24125: Liverpool vs Salzburg - 123349 out of 903032 fixture (13.659%)
Child 2, pid 24125: Liverpool vs Sporting - 121208 out of 903032 fixture (13.422%)
Child 2, pid 24125: Liverpool vs Benfica - 120327 out of 903032 fixture (13.325%)
Child 2, pid 24125: Liverpool vs Juventus - 120263 out of 903032 fixture (13.318%)
Child 2, pid 24125: Statistics for ManUnited
Child 2, pid 24125: ManUnited vs Atletico - 148417 out of 903032 fixture (16.435%)
Child 2, pid 24125: ManUnited vs PSG - 142087 out of 903032 fixture (15.734%)
Child 2, pid 24125: ManUnited vs InterMilan - 127580 out of 903032 fixture (14.128%)
Child 2, pid 24125: ManUnited vs Salzburg - 123798 out of 903032 fixture (13.709%)
Child 2, pid 24125: ManUnited vs Juventus - 120715 out of 903032 fixture (13.368%)
Child 2, pid 24125: ManUnited vs Benfica - 120596 out of 903032 fixture (13.355%)
Child 2, pid 24125: ManUnited vs Sporting - 119839 out of 903032 fixture (13.271%)
Parent, pid 24123: 4 children completed execution
```

Level 1 requirement: the child processes are created properly and can display the correct team information.
Level 2 requirement: the child processes can determine correctly the legality of the given full or partial fixture in debug mode.
Level 3 requirement: the child processes can generate the fixtures and compute correctly the probability of legal fixtures in estimation mode, when there is no team of interest.
Level 4 requirement: the child processes can generate the fixtures and compute correctly all probabilities of legal fixtures and for teams of interest, with correct task allocation to child processes.
Bonus level: being able to support extra features. Examples: determining the legality of a partial fixture that is destined to be illegal (see the third example above, which is considered legal based on the given partial fixture, but the inclusion of the remaining match-up *Villarreal vs ManUnited* will make it illegal); estimating the probability of some rival match-up, e.g. *PSG vs Barcelona* (the team does not make it this year) *vs Bayern*; estimating the probability of some interesting match-up, e.g. both French teams playing against both Italian teams, all Spanish teams facing English teams, etc. You need to provide some explanation on your extra features.

Name your program **drawing.c** and **submit it via BlackBoard on or before 4 March 2022**.