Practical work Complexity & Algorithms

During the practical lessons of the course Complexity & Algorithms pairs of students will carry out their practical work. Each pair will weekly discuss their progress with the supervising teacher during the scheduled practical lessons.

The practical work consists of three practical assignments. These assignments must be carried out and completed with a delivery of the self developed code and a complete report. The report includes a description of the analysis, the design, the implementation and the achieved results, including a justification of all choices made.

The supervising teacher will judge the software and the reporting. This leads to three marks for the assignments, that must all be >=5. To the utmost one 5 for one of the assignments is allowed, and then the mark for the practical work will be the average of the 3 marks for the assignments. Normally this mark will be awarded to both students in case of a pair, but in special cases there might be an exception.

The assignments all have a deadline (hand-in time), namely at the end of week 2, 5 and 8 respectively. When timely delivered, feedback will follow as soon as possible, including a (provisional) mark. After week 8 the mark for the practical work will be determined as soon as possible, sometimes only after the students will be asked to discuss their delivered work.

If the practical work has not sufficiently been completed, the student may do re-work during the next quartile. Each practical assignment that did not meet the minimum requirements, must be re-worked. This can be done by improving the assignment, or by doing a substitute assignment. By the end of week 8 of that quartile all re-work assignments must have been deleivered. Maybe students will be asked to come by for a discussion of theit work, before the re-work mark will be determined.
When the re-work is still unsufficient all rights regarding any marks on separate assignments will expire.

Assignment 1: Random permutation algorithms

We want to generate a random permutation of the first N integers. Such permutations are often very handy in simulations.
For instance, [5,2,3,0,4,1] and [2,1,4,5,3,0] are legal permutations (for N=6), but [1,0,2,4,2,5] is not because it contains twice a 2 and no 3.
We assume that we have access to a random generator by a methode *randInt(i,j)* that will produce random integer numbers between *i* and *j*.
For the creation of a random permutation of the numbers *0,…,N-1* we have the following three algorithms:

1. One by one fill in the elements *a[0]* till *a[N-1]* of the array *a*. To fill in element *a[i]*, generate a random number just until you find one that is not equal to *a[0]* till *a[i-1]*.
2. Analogous to algorithm 1, but now maintain an extra array *used*. When a random number *r* is put into array *a*, mark *used[r]* as *true*. So when trying to fill in *a[i]* we can immediately see whether the generated random number has already been used or not (instead of looping through all spots *a[0]* till *a[i-1]*).
3. Execute the next steps for all values of i between 0 and N-1: put an i in the array at position i, so *a[i] = i*, and swap the content of *a[i]* immediately with a randomly choosen already filled in position in the array, so *swap(a[i], a[randInt(0,i)])*.

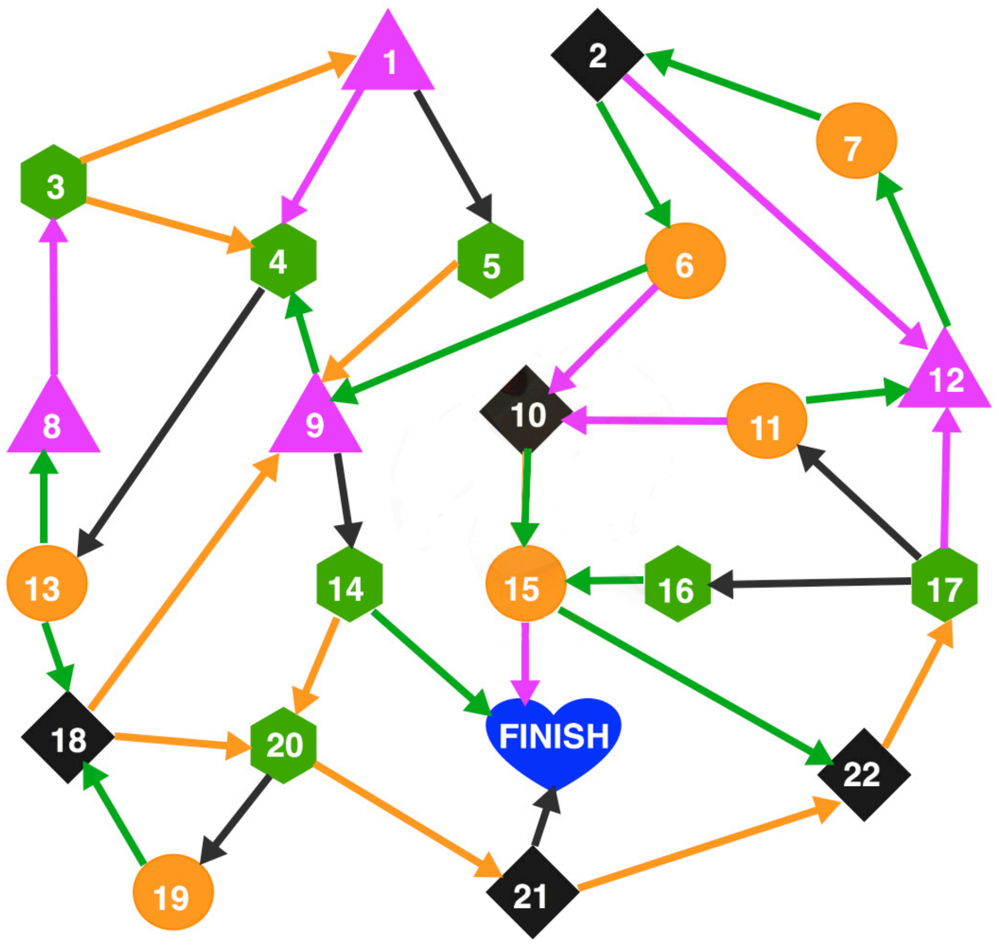
Explain in your documentation whether each of these algorithms produces legal permutations and whether the probability for each of these permutations is equal.
Give an as good as possible big-Oh estimation of the expected running time of each algorithm and explain how you came to that.
Write separate programs to execute each algorithm 10 times for each of the following values of N, and measure the average running time in each of the next situations:
Algorithm 1 with N = 5.000, 10.000, 20.000, 50.000, 100.000;
Algorithm 2 with N = 100.000, 500.000, 1.000.000, 5.000.000, 10.000.000;
Algorithm 3 with N = 5.000.000, 10.000.000, 20.000.000, 40.000.000, 80.000.000.
Present your measurements in a graph in your report and give a critical consideration on your measurements and the expected running times.

Assignment 2: Replacement Selection

When sorting external files it is important to achieve each possible optimization of an algorithm. One of the possibilities is to optimize the starting situation by providing as long as possible sorted parts (runs) in the to be sorted file.
One way to lenghten the initial runs in a file te verbeteren has been discussed in the lectures, i.e. how this can be realised using a dynamic heap with dead space.
Design and implement a class RSHeap, an efficient data structure for such a special kind of heap, including the corresponding methods.
Then write a program, based upon the above description, that will read from an (input)file with N random numbers, using an RSHeap with heapsize H, and producing the lengthened runs on an (output)file.
Choose several different combinations of values for N and H. Use a suitable testset when testing your algorithm, such that all special situations are covered.
Without using the replacement selection algorithm, an input file with N random numbers and a heapsize H would give you about N/H runs with an average length H.
Thanks to this algorithm the number of runs will be halved and the run length will be twice as long by average. Compare these expectations with your actual results.

Assignment 3: Pawn puzzle maze

In the puzzle maze below, in the starting situation two pawns will be placed at positions 1 and 2. The goal of the puzzle is, to get one of the pawns to the FINISH.
The pawns may be moved: a pawn can be moved along an arrow in the same color as the position of the other pawn. For instance: from the starting situation the pawn on position 2 may be moved to position 12. The same pawn may be moved several times.



Design and implement a program that will solve this puzzle.

Hints:
First design and implement a data structure for representing the puzzle maze. Then develop an algorithm to find the solution to the puzzle. Remember, that at any time the state of the puzzle can be described by for instance the two positions where the pawns are situated. From such a state a number of moves is possible. By making a move a new state will be created. A solution then is a sequence of states with a final state: one of the pawns is at FINISH.
Check the found solution(s) by hand!