COMP 3804 — Solutions Assignment 2

Question 1: Write your name and student number.

Solution: Aitana Bonmatí, 14

Question 2: Since you all miss COMP 2804 so much, let's start with a basic probability question. First some notation: If $x_1, x_2, \ldots, x_m$ are real numbers, then their product is written as

$$\prod_{i=1}^{m} x_i = x_1 \cdot x_2 \cdot x_3 \cdots x_m.$$

Let $(S, \Pr)$ be a probability space, let $m \geq 2$ be an integer, and let $B_1, B_2, \ldots, B_m$ be a sequence of events in this space. In this question, you will prove that

$$\Pr(B_1 \cap B_2 \cap \cdots \cap B_m) = \Pr(B_1) \cdot \prod_{i=2}^{m} \Pr(B_i \mid B_1 \cap B_2 \cap \cdots \cap B_{i-1}). \tag{1}$$

Question 2.1: Using the definition of conditional probability, prove that (1) holds when $m = 2$.
Question 2.2: Using the definition of conditional probability, prove that (1) holds when $m = 3$.
Question 2.3: Using induction and the definition of conditional probability, prove that (1) holds for every integer $m \geq 2$.

Solution: Recall that

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)}.$$

For $m = 2$, (1) becomes

$$\Pr(B_1 \cap B_2) = \Pr(B_1) \cdot \Pr(B_2 \mid B_1).$$

Starting with the right-hand side, we get

$$\begin{aligned} \Pr(B_1) \cdot \Pr(B_2 \mid B_1) &= \Pr(B_1) \cdot \frac{\Pr(B_1 \cap B_2)}{\Pr(B_1)} \\ &= \Pr(B_1 \cap B_2). \end{aligned}$$

For $m = 3$, (1) becomes

$$\Pr(B_1 \cap B_2 \cap B_3) = \Pr(B_1) \cdot \Pr(B_2 \mid B_1) \cdot \Pr(B_3 \mid B_1 \cap B_2).$$

Starting with the right-hand side, we get

$$\begin{aligned} &\Pr(B_1) \cdot \Pr(B_2 \mid B_1) \cdot \Pr(B_3 \mid B_1 \cap B_2) \\ &= \Pr(B_1) \cdot \frac{\Pr(B_1 \cap B_2)}{\Pr(B_1)} \cdot \frac{\Pr(B_1 \cap B_2 \cap B_3)}{\Pr(B_1 \cap B_2)} \\ &= \Pr(B_1 \cap B_2 \cap B_3). \end{aligned}$$

From these two cases, you should see what is going on. We now do the induction proof. We have seen above that (1) is true for $m = 2$. Let $m \geq 3$ and assume that (1) is true for $m - 1$, i.e., we assume that

$$\Pr(B_1 \cap B_2 \cap \cdots \cap B_{m-1}) = \Pr(B_1) \cdot \prod_{i=2}^{m-1} \Pr(B_i \mid B_1 \cap B_2 \cap \cdots \cap B_{i-1}).$$

We note that

$$\Pr(B_1 \cap B_2 \cap \cdots \cap B_m) = \Pr(B_1 \cap B_2 \cap \cdots \cap B_{m-1}) \cdot \Pr(B_m \mid B_1 \cap B_2 \cap \cdots \cap B_{m-1}).$$

By using the induction hypothesis and some algebra, we get

$$\begin{aligned}
&\Pr(B_1 \cap B_2 \cap \cdots \cap B_m) \\
&= \Pr(B_1 \cap B_2 \cap \cdots \cap B_{m-1}) \cdot \Pr(B_m \mid B_1 \cap B_2 \cap \cdots \cap B_{m-1}) \\
&= \Pr(B_1) \cdot \left(\prod_{i=2}^{m-1} \Pr(B_i \mid B_1 \cap B_2 \cap \cdots \cap B_{i-1}) \right) \cdot \Pr(B_m \mid B_1 \cap B_2 \cap \cdots \cap B_{m-1}) \\
&= \Pr(B_1) \cdot \prod_{i=2}^{m} \Pr(B_i \mid B_1 \cap B_2 \cap \cdots \cap B_{i-1}).
\end{aligned}$$

Question 3: In class, we have seen the following randomized selection algorithm:

Algorithm RSELECT(S, k):
Input: Sequence S of numbers, integer k with $1 \leq k \leq |S|$
Output: k-th smallest number in S
if $|S| = 1$
then return the only element in S
else $p =$ uniformly random element in S;
by scanning S and **making** $|S| - 1$ **comparisons**, divide it into
$S_< = \{x \in S : x < p\}$,
$S_= = \{x \in S : x = p\}$,
$S_> = \{x \in S : x > p\}$;
if $k \leq |S_<|$
then RSELECT$(S_<, k)$
else if $k \geq 1 + |S_<| + |S_=|$
then RSELECT$(S_>, k - |S_<| - |S_=|)$
else return p
endif
endif
endif

Let T be the random variable whose value is the **number of comparisons** made by this algorithm. With n denoting the length of the sequence S, we have shown that the expected value of T is $O(n)$.

A natural question to ask is if the value of T is $O(n)$ *with high probability*, i.e., does there exist a positive constant C, such that

$$\lim_{n\to\infty} \Pr(T \leq Cn) = 1?$$

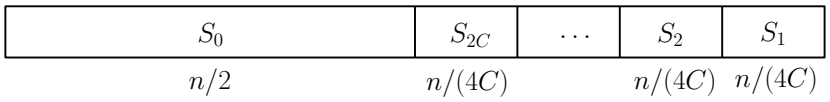
In this question, you will prove that the answer is "no".

In the rest of this question, we assume that the sequence S contains the numbers $1, 2, 3, \ldots, n$ in sorted order. We are going to run algorithm RSELECT$(S, 1)$**.**

Algorithm RSELECT$(S, 1)$ and its recursive calls choose pivots $p_1, p_2, p_3, \ldots$:

- p_1 is chosen uniformly at random in $\{1, 2, \ldots, n\}$.
- If $p_1 \neq 1$, then p_2 is chosen uniformly at random in $\{1, 2, \ldots, p_1 - 1\}$.
- If $p_2 \neq 1$, then p_3 is chosen uniformly at random in $\{1, 2, \ldots, p_2 - 1\}$.
- If $p_3 \neq 1$, then p_4 is chosen uniformly at random in $\{1, 2, \ldots, p_3 - 1\}$.
- Etcetera.

Let C be an arbitrary positive integer, and let n be a very large integer that is a multiple of $4C$. Divide the set $\{1, 2, \ldots, n\}$ into $2C + 1$ subsets, as indicated in the figure below. The subset S_0 has size $n/2$, whereas each subset S_i, for $1 \leq i \leq 2C$, has size $n/(4C)$.



Define the events

$$A = \text{"}T > Cn\text{"}$$

and, for each $i = 1, 2, \ldots, 2C$,

$$B_i = \text{"}p_i \in S_i\text{"}.$$

Question 3.1: Prove that

$$\Pr\left(B_1 \cap B_2 \cap \cdots \cap B_{2C}\right) \leq \Pr(A).$$

Question 3.2: Use Question 2 to prove that

$$\Pr(A) \geq \left(1/(4C)\right)^{2C}.$$

Question 3.3: Conclude that

$$\Pr(T \leq Cn) \leq 1 - \left(1/(4C)\right)^{2C}.$$

Solution: For **3.1**, it is sufficient to show that

$$B_1 \cap B_2 \cap \cdots \cap B_{2C} \subseteq A,$$

i.e., if the event $B_1 \cap B_2 \cap \cdots \cap B_{2C}$ occurs, then the event A also occurs.

Let us assume that the event $B_1 \cap B_2 \cap \cdots \cap B_{2C}$ occurs. We are going to show that A also occurs, i.e., $T > Cn$.

When the first pivot p_1 is chosen, each of the other $n-1$ numbers is compared to p_1. Thus, the number of comparisons made is $n-1$, which is more than $n/2$.

Since B_1 occurs, p_1 is in S_1. Thus, there is a recursive call, in which the second pivot p_2 is chosen. This second pivot is compared to all elements in S_0 (and to others as well). Thus, the number of comparisons is at least $n/2$.

Since B_2 occurs, p_2 is in S_2. Thus, there is a recursive call, in which the third pivot p_3 is chosen. This third pivot is compared to all elements in S_0 (and to others as well). Thus, the number of comparisons is at least $n/2$.

Etc., etc.

Since B_{2C-1} occurs, p_{2C-1} is in S_{2C-1}. Thus, there is a recursive call, in which the pivot p_{2C} is chosen. This $(2C)$-th pivot is compared to all elements in S_0 (and to others as well). Thus, the number of comparisons is at least $n/2$.

By adding up all these comparisons, we conclude that the value of T, i.e., the total number of comparisons, is more than

$$2C \cdot n/2 = Cn.$$

For **3.2**: Using **3.1**, it is sufficient to show that

$$\Pr\left(B_1 \cap B_2 \cap \cdots \cap B_{2C}\right) \geq \left(1/(4C)\right)^{2C}.$$

First note that

$$\Pr\left(B_1 \cap B_2 \cap \cdots \cap B_{2C}\right) \neq \Pr(B_1) \cdot \Pr(B_2) \cdots \Pr(B_{2C}),$$

because the events $B_1, B_2, \ldots, B_{2C}$ are not independent. That is why we are going to use Question 2.

Using Question 2, it is sufficient to show that

$$\Pr\left(B_1\right) \cdot \prod_{i=2}^{2C} \Pr\left(B_i \mid B_1 \cap B_2 \cap \cdots \cap B_{i-1}\right) \geq \left(1/(4C)\right)^{2C}.$$

- $\Pr(B_1)$ is the probability that the first pivot p_1 is in S_1. There are n choices for p_1, and $n/(4C)$ of these are in S_1. Thus,

$$\Pr(B_1) = \frac{n/(4C)}{n} = 1/(4C).$$

- To determine $\Pr(B_2 \mid B_1)$, we assume that event B_1 occurs, i.e., the first pivot p_1 is in S_1. There are $p_1 - 1 \leq n$ ways to choose the second pivot p_2. Note that all elements in S_2 are included in these. Among all possible choices for p_2, there are $n/(4C)$ ways to choose it from S_2. Thus,

$$\Pr(B_2 \mid B_1) = \frac{n/(4C)}{p_1 - 1} \geq \frac{n/(4C)}{n} = 1/(4C).$$

- In general, let i be such that $2 \leq i \leq 2C$. To determine $\Pr(B_i \mid B_1 \cap B_2 \cap \cdots \cap B_{i-1})$, we assume that all events $B_1, B_2, \ldots, B_{i-1}$ occur. There are $p_{i-1} - 1 \leq n$ ways to choose the pivot p_i. Note that all elements in S_i are included in these. Among all possible choices for p_i, there are $n/(4C)$ ways to choose it from S_i. Thus,

$$\Pr(B_i \mid B_1 \cap B_2 \cap \cdots \cap B_{i-1}) = \frac{n/(4C)}{p_{i-1} - 1} \geq \frac{n/(4C)}{n} = 1/(4C).$$

- Putting it al together, we get

$$\Pr(B_1) \cdot \prod_{i=2}^{2C} \Pr(B_i \mid B_1 \cap B_2 \cap \cdots \cap B_{i-1}) \geq (1/(4C))^{2C}.$$

3.3 is easy, because

$$\Pr(T \leq Cn) = 1 - \Pr(A) \leq 1 - (1/(4C))^{2C}.$$

Question 4: You are given a min-heap $A[1 \ldots n]$ and a variable *largest* that stores the largest number in this min-heap.

In class, we have seen algorithms INSERT(A, x) (which adds the number x to the min-heap and restores the heap property) and EXTRACTMIN(A) (which removes the smallest number from the heap and restores the heap property).

Explain, in a few sentences, how these two algorithms can be modified such that the value of *largest* is correctly maintained. The running times of the two modified algorithms must still be $O(\log n)$.

Solution:

- INSERT(A, x): The algorithm is the same as the one we have seen in class. At the end of this algorithm, we set $\mathit{largest} = \max(\mathit{largest}, x)$. The running time will be $O(\log n) + O(1) = O(\log n)$.

- EXTRACTMIN(A): The algorithm is the same as the one we have seen in class. Note that, if the heap is not empty afterwards, the value of *largest* does not change. If A is empty afterwards, then we set $\mathit{largest} = -\infty$. The running time will be $O(\log n) + O(1) = O(\log n)$.

Question 5: Let m be a large integer and consider m non-empty sorted lists $L_1, L_2, \ldots, L_m$. All numbers in these lists are integers. Let n be the total length of all these lists.

Describe an algorithm that computes, in $O(n \log m)$ time, two integers a and b, with $a \leq b$, such that

- each list L_i contains at least one number from the set $\{a, a+1, \ldots, b\}$ and
- the difference $b - a$ is minimum.

For example, if $m = 4$,

$$\begin{aligned} L_1 &= (2, 3, 4, 8, 10, 15) \\ L_2 &= (1, 5, 12) \\ L_3 &= (7, 8, 15, 16) \\ L_4 &= (3, 6), \end{aligned}$$

then the output can be $(a, b) = (4, 7)$ or $(a, b) = (5, 8)$.

As always, justify the correctness of your algorithm and explain why the running time is $O(n \log m)$.

Hint: Use a min-heap of size m and use Question 4. For the example, draw it like this, then stare at it until you "see" the algorithm:

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

Solution: After having stared at the figure long enough, you will have noticed that we want to compute the smallest interval that contains at least one element from each list. We will do this by "sliding" an interval $[a, b]$ from left to right and take care that this interval always contains at least one element from each list.

- For each $i = 1, 2, \ldots, m$, take the first element x_i in list L_i and remove it from L_i. Let H be the sequence $x_1, x_2, \ldots, x_m$. Let *smallest* be the smallest number in H, let *largest* be the largest number in H, and let $answer = largest - smallest$. Then the interval $[smallest, largest]$ contains at least one element from each list, and the length of this interval is *answer*.
- How do we "slide" this interval to the next one: We delete *smallest* from H. Let i be the index such that *smallest* was in L_i. Then we add the first element in L_i to H, and delete it from L_i. Now we recompute *smallest*, *largest*, and *answer*.
- We repeat this until one of the lists gets empty.
- What are the operations that we need?

  - We have to delete the smallest number in H.
  - We have to insert an element into H.
  - We have to keep track of the largest number in H.

- The first two operations suggest that we store H in a min-heap.
- For the third operation, we will use Question 4.

Here is a more formal description of the algorithm. We assume that every number "knows" the index of the list L_i it is part of.

Initialization:

- For $i = 1, 2, \ldots, m$, let x_i be the first element in L_i, and delete it from L_i.
- Build a min-heap H for the numbers $x_1, x_2, \ldots, x_m$.
- Set *smallest* = MIN(H).
- By scanning $x_1, x_2, \ldots, x_m$, compute the largest among these numbers and store it in the variable *largest*.
- Set *answer* = *largest* − *smallest*.

Repeat the following as long as all lists are non-empty:

- x = EXTRACTMIN(H); this updates *largest*, see Question 4.
- Let i be the index such that x was in L_i.
- Let x_i be the first element in L_i and remove it from L_i.
- INSERT(H, x_i); this updates *largest*, see Question 4.
- Set *smallest* = MIN(H).
- Set *answer* = min(*answer*, *largest* − *smallest*).

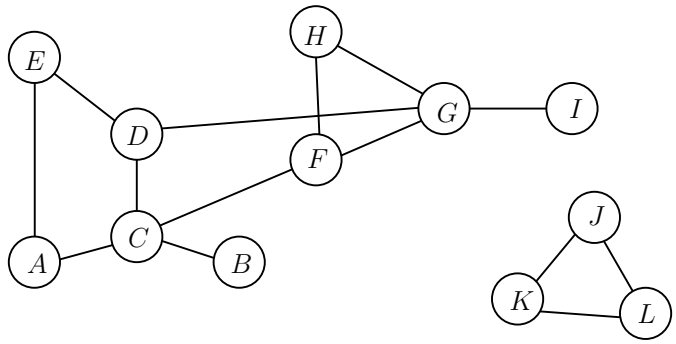
After the loop has terminated: Return *answer*.

Running time:

- The initialization takes $O(m)$ time.
- Note that, at any moment, the heap stores m numbers. Therefore, each operation that we perform on the heap takes $O(\log m)$ time.
- There are at most n EXTRACTMIN-operations, at most n INSERT-operations, and at most n MIN-operations.

- Thus, the loop takes time $O(n \log m)$.
- We conclude that the total running time is $O(m+n \log m)$, which is $O(n \log m)$, because $m \leq n$.

Question 6: Consider the following undirected graph:



Question 6.1: Draw the DFS-forest obtained by running algorithm DFS on this graph. Recall that algorithm DFS uses algorithm EXPLORE as a subroutine.

In the forest, draw each tree edge as a solid edge, and draw each back edge as a dotted edge.

Whenever there is a choice of vertices (see the two lines labeled (*)), pick the one that is alphabetically first.

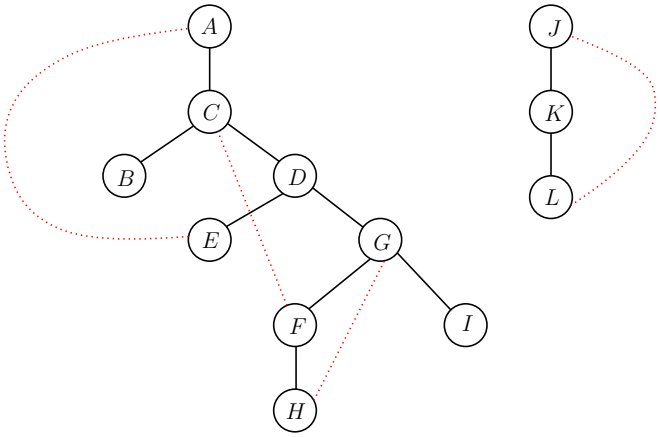
Question 6.2: Do the same, but now, whenever there is a choice of vertices (see the two lines labeled (*)), pick the one that is alphabetically last.

Algorithm DFS(G):
for each vertex u
do *visited*(u) = *false*
endfor;
$cc = 0$;
for each vertex v (*)
do if *visited*(v) = *false*
  then $cc = cc + 1$
    EXPLORE(v)
  endif
endfor

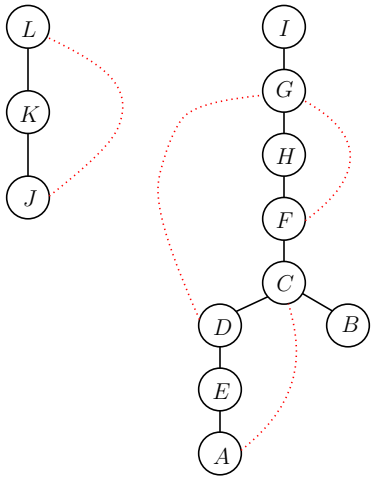
Algorithm EXPLORE(v):
visited(v) = *true*;
ccnumber(v) = cc;
for each edge $\{v, u\}$ (*)

do if *visited*$(u) = false$
then EXPLORE(u)
endif
endfor

Solution: We start with **(6.1)**. In case there is more than one choice, we pick the alphabetically smallest one. Thus, algorithm DFS(G) starts by calling EXPLORE(A). The vertices in each adjacency list are sorted in increasing order. Here is the resulting DFS-forest, which consists of two trees:



Next we do **(2.2)**. In case there is more than one choice, we pick the alphabetically largest one. Thus, algorithm DFS(G) starts by calling EXPLORE(L). The vertices in each adjacency list are sorted in decreasing order. Here is the resulting DFS-forest, which consists of two trees:



Question 7: Prove that an undirected graph $G = (V, E)$ is bipartite if and only if G does not contain any cycle having an odd number of edges.

Solution: Assume that G is bipartite, i.e., the vertex set V of G can be partitioned into two sets L and R, such that each edge has one vertex in L and one vertex in R. We have to show that G does not contain any odd cycle. Assume there is an odd cycle

$$v_1, v_2, v_3, \ldots v_{2k+1}, v_1.$$

We may assume that v_1 is in L. Then v_2 is in R, v_3 is in L, v_4 is in R, $\ldots$, v_{2k-1} is in L, v_1 is in R. This is a contradiction.

Now we assume that G does not contain any odd cycle. We have to show that G is bipartite. We run algorithm DFS(G) and classify each edge as a tree edge or a back edge. Consider the tree defined by the tree edges. Using this tree, we partition the vertex set V into two subsets L and R: The vertices at the even levels are added to L, whereas the vertices at the odd levels are added to R. It is clear that for every tree edge, one vertex is in L and the other vertex is in R. Assume, by contradiction, that there is a back edge $\{u, v\}$ such that both u and v are in L (the case when they are both in R is symmetric). We may assume that v is in the subtree of u. Consider the following cycle in G:

- Start at u, follow tree edges to v, then follow the back edge to u.

Since both u and v are in L, this cycle has an odd number of edges, which is a contradiction.

Question 8: Since Taylor Swift and Travis Kelce miss each other very much, they decide to meet. Taylor and Travis live in a connected, undirected, non-bipartite graph $G = (V, E)$. Taylor lives at vertex s, whereas Travis lives at vertex k.

Taylor and Travis move in steps. In each step, Taylor must move from her current vertex to a neighboring vertex, and Travis must move from his current vertex to a neighboring vertex.

Prove that there exists a moving strategy such that Taylor and Travis meet each other at the same vertex.

Hint: While moving around in the graph, each of Taylor and Travis may visit the same vertex more than once, and may traverse the same edge more than once.

If the graph G consists of the single edge $\{s, k\}$, then they will never be at the same vertex. But in this case G is bipartite.

If the graph contains a path having 8 edges, Taylor lives at one end-vertex, and Travis lives at the other end-vertex, will they ever be at the same vertex?

Question 7 is useful.

Solution: To get some intuition, assume there is a path $(1, 2, 3, 4, 5, 6, 7)$ with 6 edges, Taylor lives at vertex 1 and BF (boyfriend) lives at vertex 7.

- In step 1, Taylor moves to 2, and BF moves to 6.
- In step 2, Taylor moves to 3, and BF moves to 5.
- In step 3, Taylor moves to 4, and BF moves to 4. Both Taylor and BF are happy!

This works as long as the number of edges on the path is even. In other words, if there is a path with an even number of edges between the vertices s and k, then there is a strategy such that Taylor and BF meet at the same vertex. In fact, this path may be a *walk*, i.e., a sequence of vertices such that (i) the first vertex is s, the last vertex is k, and there is an edge between any two consecutive vertices in this sequence. Note that in a walk, vertices may be visited more than once and edges may be traversed more than once.

If we can prove that there always is a walk with an even number of edges between the vertices s and k, then we have answered the question.

Since the graph G is not bipartite, we know from Question 7 that there is a cycle C in G with an odd number of edges. Let c be an arbitrary vertex on this cycle C. Since the graph G is connected, there is a path P from s to c, and there is a path Q from c to k.

- If the concatenation PQ (which is a path or a walk) has an even number of edges, then we are done.
- Assume that PQ has an odd number of edges. Consider the walk W that starts at s, follows P to c, then walks along the cycle C back to c, and then follows Q from c to k. Since C has an odd number of edges, the number of edges on W is "odd plus odd", which is even.