Homework Assignment #4

Due: Noon, 26th Nov, 2018
Submit solutions via eClass

CMPUT 204
Department of Computing Science
University of Alberta

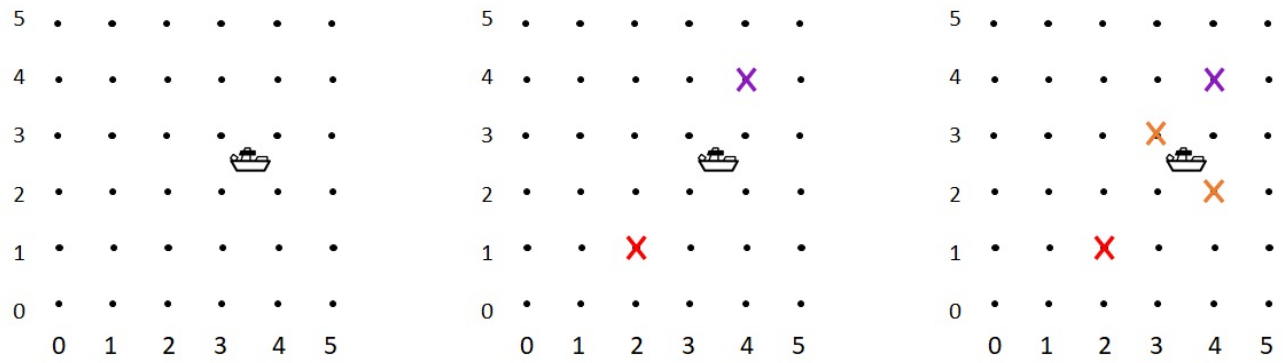
Note: All logs are base-2 unless stated otherwise.
This HW is composed of 5 problems, each worth 25 points. Should you answer all 5 problems, your grade will be composed of the **best** 4 **out of** 5.

Problem 1. (25 pts) A Version of "Battleships"

Alice and Beth are playing the following variation on the classic boardgame "Battleships." The game board is composed of a grid $\{0,1,2...,n\}\times\{0,1,2,...,n\}$ in which Beth has hidden inside one of the squared a battleship. In the given example (Figure 1a) where $n=5$, the board is set such that the ship is inside the square $[3,4]\times[2,3]$. Unfortunately, Alice is left with a single bomb, so Alice must find the square containing this battleship before she can use her only bomb to sink the battleship.

To that end, Alice has an arsenal of special "sonar"-bomb. A sonar-bomb is dropped to a single grid-point (i,j) and forces Beth to tell Alice the direction to the square where the battleship is placed *in relation to the point* (i,j): NE (above and right), NW (above and left), SW (below and left) or SE (below and right). So, to continue our example, in Figure 1b we illustrate the case that Alice drops a sonar on the point $(2,1)$ (the red X) then is gets as a reply 'NE' (since the battleship is found inside the subgrid $\{2,3,4,5\}\times\{1,2,3,4,5\}$); and should Alice drop a sonar on the point $(4,4)$ (the purple X) then the reply Alice gets is 'SW' (since the battleship is found inside the subgrid $\{0,1,2,3,4\}\times\{0,1,2,3,4\}$).

To continue on our example, Alice eventually (in Figure 1c) drops a sonar on the points covered with an orange X: $(3,3)$ (to which the reply is 'SE') and $(4,2)$ (to which the reply is 'NW'), and so now Alice knows for certainty the bomb is inside the $[3,4]\times[2,3]$ square, and she wins the game.



(a) (b) (c)

Figure 1: An Example of This Version of Battleships.

(a) (15 pts) Help Alice win the game. Devise an algorithm for Alice to minimize the number of sonar bombs she uses until she is certain which square holds Beth's battleship.

Answer. Here is an algorithm in which the number of sonar-bombs used is $s=\lceil\log_4(n^2)\rceil=\lceil 2\log_4(n)\rceil=\lceil 2\log_4(2)\log_2(n)\rceil=\lceil 2\cdot\frac{1}{2}\log_2(n)\rceil=\lceil\log(n)\rceil$. We call it `2dBinarySearch` and its first invocation is with $x_p=0, x_r=n; y_p=0, y_r=n$.

`procedure 2dBinarySearch(`x_p, x_r, y_p, y_r`)`
`if (`$x_r = x_p+1$ `and` $y_r = y_p+1$`) then`
  `return the square` $[x_p,x_r]\times[y_p,y_r]$ ** Or: "drop the bomb in this square"
$mid_x \leftarrow \lfloor\frac{x_p+x_r}{2}\rfloor$
$mid_y \leftarrow \lfloor\frac{y_p+y_r}{2}\rfloor$
`drop sonar bomb on` (mid_x, mid_y) `and get` $answer$
`if (`$answer$ `=NE or` $answer$ `=SE) then`
  $newx_p \leftarrow mid_x$
  $newx_r \leftarrow x_r$
`else`
  $newx_p \leftarrow x_p$

$newx_r \leftarrow mid_x$
`if (`$answer$ `=NE or` $answer$ `=NW) then`
$newy_p \leftarrow mid_y$
$newy_r \leftarrow y_r$
`else`
$newy_p \leftarrow y_p$
$newy_r \leftarrow mid_y$
`return 2dBinarySearch(`$newx_p, newx_r, newy_p, newy_r$`)`

The correctness of this algorithm follows from the following 2 observations:

1. If we invoke `2dBinarySearch` and the battleship is found inside the grid $\{x_p, x_p+1, ..., x_r\} \times \{y_p, y_p+1, ..., y_r\}$, then the battleship is found in the subgrid on which we make the next call to `2dBinarySearch`.

2. If we invoke `2dBinarySearch` on the grid $\{x_p, x_p+1, ..., x_r\} \times \{y_p, y_p+1, ..., y_r\}$, then the size of the subgrid in the following iteration is at most $\lceil \frac{x_r-x_p+1}{2} \rceil \times \lceil \frac{y_r-y_p+1}{2} \rceil$. In particular, for any axis with ≥ 2 points, this axis is cut in half and we get the we recurse on a subgrid where this axis is strictly smaller.

Both are due to the nature of our recursion — recursing on the right subgrid (the one to hold the battleship) of where each axis is (roughly) cut in half. To complete this to an inductive proof is straight-forward.

Therefore, if we begin with a board containing $n \times n$-squares, then in the next iteration the board size it at most $\lceil \frac{n+1}{2} \rceil$. So, if $T(n)$ represents the WC number of sonar bombs we use on a $n \times n$-board, then we get the recurrence relation:

$$T(n) = 1 + T(\lceil \tfrac{n+1}{2} \rceil)$$

We can of course approximate this via $T(n) = 1+T(n/2)$ because for every n which is a power of 2 this is the recurrence relation we get. And so, denoting $s = \lceil \log(n) \rceil$ we get $n \geq 2^s$ so: $T(n) \leq T(2^s) = s = \lceil \log(n) \rceil$.

(b) (10 pts) Prove that your algorithm is optimal. That is, if the algorithm that you designed uses in the worst-case s sonar-bombs, then show that any algorithm that is guaranteed to find the square holding the battleship must also use s sonar-bombs in the worst-case.

Answer. We claim that any *deterministic* algorithm that finds the one square holding the battleship must use $s = \log_4(n^2)$ sonar bombs. ASOC that there exists some algorithm A which in the worst-case make $< s$ sonar-bomb queries. Observe how each sonar bomb query gets one of 4 possible replies. That means that across all inputs, the algorithm may view $< 4^s = n^2$ possible sequences of query-replies. However, there are n^2 possible places where the battleship may be placed, and so, by the pigeonhole principle there exists two squares $[a, a+1] \times [b, b+1]$ and $[c, c+1] \times [d, d+1]$ which result in the same sequence of replies. This means that the algorithm gets the same replies on both of these squares and therefore must drop the bomb in a place which isn't one of these two different squares. Hence the algorithm errs on (at least) one of these two inputs. ■

Note: Ideally, your upper- and lower-bounds will match *exactly* (or up to ± 1). If you can't get them to match exactly, have your bounds be asymptotically the same.

Problem 2. (25 pts) A random graph $G(n,p)$ is generated using the following algorithm. First, we fix n nodes which we denote as $v_1, v_2, ..., v_n$. Then, for every two nodes (v_i, v_j) we toss an independent coin that comes out 'heads' with probability $= p$ — and if the coin comes up 'heads' we draw the edge $\{v_i, v_j\}$; o/w we put no edge between v_i and v_j. In other words, for every pair of nodes v_i, v_j we put a (non-directed) edge between them with probability p and this is done independently for each possible pair of nodes.

(a) (10 pts) What is the expected number of edges in a random graph $G(n,p)$?
Answer. For any pair of vertices $\{v_i, v_j\}$ let $X_{i,j}$ be a Bernoulli random variable indicating whether we have placed an edge between node v_i and node v_j. By definition:

$$\text{for any distinct } i,j, \quad \mathbf{E}[X_{i,j}] = \Pr[\text{an edge connects } v_i, v_j] = p$$

Therefore:

$$\mathbf{E}[\#\text{edges}] = \mathbf{E}[\sum_{\text{pairs of nodes } i,j} X_{i,j}] = \sum_{\text{pairs of nodes } i,j} \mathbf{E}[X_{i,j}] = \binom{n}{2} p \quad \left(= \frac{pn(n-1)}{2}\right)$$

(b) (15 pts) What is the expected number of triangles in a random graph $G(n,p)$?
A triangle: a triplet of distinct nodes $\{v_i, v_j, v_k\}$ such that all 3 edges $(\{v_i, v_j\}, \{v_j, v_k\}, \{v_i, v_k\})$ are present in the graph.
Answer. For any triplet of distinct vertices $\{v_i, v_j, v_k\}$ let $X_{i,j,k}$ be a Bernoulli random variable indicating whether there exists a triangle on v_i, v_j, v_k. The only way such a triangle is formed is by placing in G all 3 edge connecting v_i with v_j, v_i with v_k and v_j with v_k. Due to independence, the probability of all 3 edges being present is p^3. So, by definition:

$$\text{for any distinct } i,j,k \quad \mathbf{E}[X_{i,j,k}] = \Pr[v_i, v_j, v_k \text{ form a triangle}] = p^3$$

Therefore:

$$\mathbf{E}[\#\text{triangles}] = \mathbf{E}[\sum_{\text{triplets of nodes } i,j,k} X_{i,j,k}] = \sum_{\text{triplets of nodes } i,j,k} \mathbf{E}[X_{i,j,k}] = \binom{n}{3} p^3 \quad \left(= \frac{p^3 n(n-1)(n-2)}{6}\right)$$

For example, if $n = 5$, $p = \frac{1}{2}$ and we happened to pick the graph shown in Figure 2, then we have a graph with 7 edges and 2 triangles ($\{v_2, v_4, v_5\}$ and $\{v_2, v_3, v_4\}$).

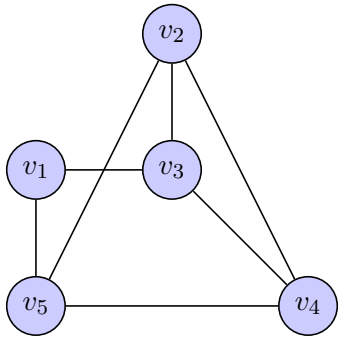


Figure 2

For example, if $n = 3$ and $p = \frac{1}{3}$, then all possible graphs over $\binom{3}{2} = 3$ edges that $G(3, \frac{1}{3})$ might generate are given in Figure 3 along their corresponding probabilities. Thus

$$\mathbf{E}[\#\text{edges}] = 0 \times \left(\frac{2}{3}\right)^3 + 1 \times 3 \times \left(\frac{2}{3}\right)^2 \times \frac{1}{3} + 2 \times 3 \times \frac{2}{3} \times \left(\frac{1}{3}\right)^2 + 3 \times \left(\frac{1}{3}\right)^3 = 1$$

$$\mathbf{E}[\#\text{triangles}] = 0 \times \left[\left(\frac{2}{3}\right)^3 + 3 \times \left(\frac{2}{3}\right)^2 \times \frac{1}{3} + 3 \times \frac{2}{3} \times \left(\frac{1}{3}\right)^2 \right] + 1 \times \left(\frac{1}{3}\right)^3 = \frac{1}{27}$$

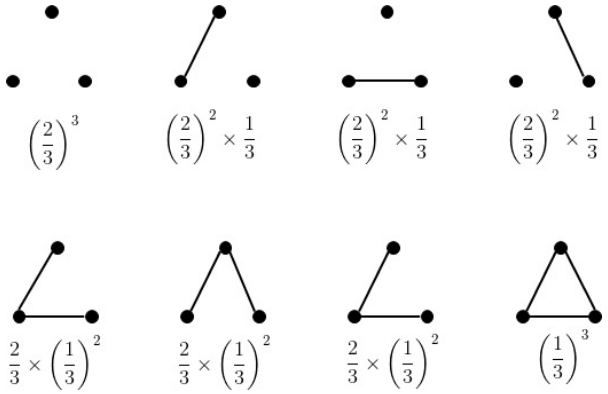


Figure 3

Problem 3. (25 pts) An irresponsible employee at the electronic warehouse of UofA has taken n different batteries and their corresponding n different lightbulbs out of their cases and left them unmarked, and it is up to match them up together. As the batteries and the lightbulbs are unmarked, you cannot see the voltage on each battery and cannot see the required voltage to light each lightbulb. Instead, you are handed an electronic circuit to which you connect on the one end a battery and on the other end a lightbulb, and then one of the following three can happen:

- If the battery you placed happens to be the right battery for the lightbulb you placed, then the lightbulb is turned on and light comes out.
- If the battery you placed happens to have a voltage that is too small for the lightbulb you placed, then the lightbulb remains dark.
- If the battery you placed happens to have a voltage that is too great for the lightbulb you placed, then the lightbulb over heats — but luckily the safety mechanism in the circuits kicks into action and a loud switch is flipped, disconnecting the circuit before the lightbulb burns out.

Design an algorithm that matches each of the n lightbulbs to the correct one of the n batteries that turns it on, based only on this given circuit and the three possible outcomes you may observe (i.e. {no light, light, saftey-switch flipped}) while using the given circuit only $O(n\log(n))$ many times.

Hint: QuickSort.

Answer. The idea is very similar to random QuickSort. Suppose we label all batteries as $b_1, b_2, ..., b_n$ such that for each i the voltage of battery b_i is $\leq$ the voltage of battery b_{i+1}. In fact, for the ease of notation, we use b_i to denote also the voltage of battery b_i. Similarly, we label the lightbulb $l_1, l_2, ..., l_n$ suc that the voltage required to turn l_i on is $\leq$ the voltage required to turn l_{i+1} on.

First note that if $n = 0$ our work is trivially done; if $n = 1$ we can match the one lightbuld to the one battery.

Otherwise, if $n > 1$, we pick a lightbulb l_i u.a.r and put it on one end of the given circuit, and then match it to all batteries . This allows us to partition the n batteries into 3 sets:

- B_l = all batteries that left l_i dark. Those have voltage $\leq b_i$.
- The singleton $\{b_i\}$ which is the only battery that turns l_i on.
- B_r = all batteries that overheat l_i. Those have voltage $> b_i$.

We remove l_i from the circuit, place b_i on the battery-end of the circuit and try all $n-1$ lightbulbs which aren't l_i. These help up to partition the set of lightbulbs into 3:

- L_l = all lightbulbs that overheat using b_i. This means that these lightbulbs require voltage $< b_i$ to be turned on.
- The singleton $\{l_i\}$ which matches b_i.
- L_r = all lightbulbs that remain dark using b_i. This means that these lightbulbs require voltage $> b_i$ to be turned on.

Now, we recurse our algorithm twice: once on (B_l, L_l) and once on (B_r, L_r). Note that by definition (B_l, L_l) contains all $i-1$ pairs that use voltage $< b_i$; and (B_r, L_r) contains all $n-i$ pairs that use voltage $> b_i$. Therefore, there exists a matching of the batteries in B_l to the lightbulbs in L_l which we find using the first recursive all; and there exists a matching of the batteries in B_r to the lightbulbs in L_r which we find using the second recursive call. This resolves the correctness of our algorithm.

[Note: for simplicity we assume the middle subset is a singleton in both cases. If not (there are multiple batteries and multiple lightbulbs of the same voltage) — we just match them arbitrarily.]

As for the expected number of times in which we use the circuit on an in put size n (n batteries and n lightbulbs), which we denote as $T(n)$. First, the circuit is used $n+n-1=2n-1$ many times, and in addition – the number of times it is used in both recursive calls. Should we pick lightbulb l_i than the two recursions are over inputs of size $i-1$ and $n-i$ respectively. Since lightbulb l_i is chosen w.p. $=\frac{1}{n}$ then the recurrence relation we get is:

$$T(n)=\sum_{i=1}^{n}\Pr[\text{we picked lightbulb } l_i]\Big(T(i-1)+T(n-i)+2n-1\Big)=\sum_{i=1}^{n}\frac{1}{n}\Big(T(i-1)+T(n-i)+2n-1\Big)$$
$$=2n-1+\sum_{i=1}^{n}\frac{T(i-1)+T(n-i)}{n}=2n-1+\frac{2}{n}\sum_{j=0}^{n-1}T(j)$$

Using the fact that $T(0)=T(1)=0$, we prove by induction that for every $n\geq 1$ we have $T(n)\leq 10n\log(n)$. The base is case is simple: $T(1)=0=50\cdot 1\cdot\log(1)$.

Induction step: Fix $n\geq 2$. Assuming $T(j)\leq 50j\log(j)$ for any $1\leq j<n$ we have that

$$T(n)=2n-1+\frac{2}{n}\sum_{j=0}^{n-1}$$
$$\overset{\text{IH}}{\leq}2n-1+\frac{100}{n}\sum_{j=0}^{n-1}j\log(j)=2n-1+\frac{100}{n}\sum_{j=0}^{\lfloor n/2\rfloor}j\log(j)+\frac{100}{n}\sum_{j=\lfloor n/2\rfloor+1}^{n-1}j\log(j)$$
$$\leq 2n-1+\frac{100\log(n/2)}{n}\sum_{j=0}^{\lfloor n/2\rfloor}j+\frac{100\log(n)}{n}\sum_{j=\lfloor n/2\rfloor+1}^{n-1}j$$
$$=2n-1+\frac{100\log(n)-100}{n}\sum_{j=0}^{\lfloor n/2\rfloor}j+\frac{100\log(n)}{n}\sum_{j=\lfloor n/2\rfloor+1}^{n-1}j=2n-1+\frac{100\log(n)}{n}\sum_{j=0}^{n}j-\frac{100}{n}\sum_{j=0}^{\lfloor n/2\rfloor}j$$
$$=2n-1+\frac{100\log(n)}{n}\cdot\frac{(n-1)n}{2}-\frac{100}{n}\cdot\frac{\lfloor n/2\rfloor(\lfloor n/2\rfloor+1)}{2}\leq 2n-1+50(n-1)\log(n)-\frac{100}{n}\cdot\frac{(\frac{n}{2}-1)\frac{n}{2}}{2}$$
$$=2n-1+50(n-1)\log(n)-25(\frac{n}{2}-1)$$
$$=50n\log(n)-50\log(n)+2n-\frac{25}{2}n-1+25\overset{\log(n)\geq 1}{\leq}50n\log(n)+25-1-50\leq 50n\log(n)\qquad\blacksquare$$

Problem 4. (25 pts) Counting Bit-Flipping Operations:

(a) (20 pts) Normally our analyses gloss over code lines of the form "$i \leftarrow i+1$" as we consider incrementing an integer by 1 as a constant-time operation. But this is not necessarily true if i is a large integer whose binary representation involves m bits, and we decide to look at this operation at a very fine resolution — on the bit level.

For example, if we consider the integer $i = 2^{m-1}$ and we then increment it to $i+1$, the binary representation of i, which is $\boxed{0}\underbrace{\boxed{1}\boxed{1}\boxed{1}...\boxed{1}}_{m-1}$, changes to the binary representation of $i+1$, which is $\boxed{1}\underbrace{\boxed{0}\boxed{0}\boxed{0}...\boxed{0}}_{m-1}$. So incrementing i has flipped *all* m bits in the binary representation, and if we count each bit-flipping as an operation, then this took $O(m)$-time. This means that in the worst-case, incrementing an integer n costs $O(\log(n))$.

Prove that even if we look at each bit-flipping as a single operation, the *amortized* cost of n incrementation operations of a counter that starts at 0 is still $O(1)$. In other words, in the following code, the amortized cost of each operation is still $O(1)$:

```
i ← 0
while (i < n) do
    i ← i + 1
```

Hint: Assign a cost to each increment-operation based on the number of bits it flipped; for each possible cost c bound the number of operations with cost at least c.

Answer. We call an incrementation request as j-heavy if it invokes $\geq j$ bit-flipping operations. We argue that any two consecutive j-heavy operations are 2^{j-1}-apart. The reason is that in order to have an incrementation instruction for which we flip $\geq j$ bits, then it must be the case we flip bit 0, bit 1, bit 2,..., bit $j-2$, and also bit $j-1$. Moreover, the first $j-2$ bits must be flipped from 1 to 0 because the first bit flipped from 0 to 1 stops the process of bit-flipping (there's no need to change the following bits). It follows that such an incrementation must flow x number of incrementation which set the first $j-2$ bits to 1. I.e., strictly in between two consecutive j-heavy instructions we must have that we incremented the counter at least $1+2+4+8+...+2^{j-2} = 2^{j-1}-1$ many times. And so, in any interval of $2^{j-1} = 2^{j-1}-1+1$ instructions there could be at most one j-heavy instruction.

Having proved that any two consecutive j-heavy operations are 2^{j-1}-apart, it follows that among the overall n incrementation operations there are at most $\frac{n}{2^{j-1}}$ j-heavy operations. And so the total cost of all n incrementation calls is

$$\sum_{j=1}^{\log(n+1)} j \cdot \frac{n}{2^{j-1}} = \sum_{j=1}^{\log(n+1)} j \cdot \frac{n}{\frac{1}{2}\cdot 2^j} = 2n \cdot \sum_{j=1}^{\log(n+1)} \frac{j}{2^j} \leq 2n \cdot \sum_{j=1}^{\infty} \frac{j}{2^j} \overset{(*)}{\leq} 12n$$

where the reason for the inequality marked by $(*)$ is given in our analysis of building a heap in linear time.

Therefore the amortized cost of incrementing is $\leq \frac{12n}{n} = 12 = O(1)$.

(b) (5 pts) Suppose that in addition to incrementing i we allow decrementing i as well (setting $i \leftarrow i-1$). Show that there exists a sequences of n incrementation and decrementation instructions, starting with $i = 0$, whose amortized cost is $\Omega(\log(n))$.

Answer. The sequence of instructions begins with the counter $i = 0$ and increases it to, say, $n/8$ through $n/8$ incrementation instructions. Just like in the previous article, the amortized cost of each incrementation is $O(1)$, and so these initial $n/8$ instructions cost altogether $O(n)$.

Now $i = n/8$ and if n is a power of 2, say $n = 2^k$ then $n/8 = 2^{k-3}$, which means i's binary representation is a single one followed by a sequence of $k-3$ zeros. This means that $i-1 = 2^{k-3}-1$ is represented by a single zero followed by $k-3$ ones.

$$i = 1\underbrace{000\ldots0}_{k-3} \qquad \text{and so} \qquad i-1 = 0\underbrace{111\ldots1}_{k-3}$$

So now, the remaining $\frac{7n}{8}$ instructions oscillate between `increment(`i`)` and `decrement(`i`)`, and each instruction thus flip (at least) $k-3$ bit. All in all, we get a sequence of increment/decrement instructions that costs $\frac{n}{8} \cdot 1 + \frac{7n}{8} \cdot (k-3)$. For large enough n, where $k = \log(n) \geq 6$ we have that $k - 3 \geq \frac{1}{2}k = \frac{1}{2}\log(n)$ hence this sequence of instructions has cost $\geq \frac{7n}{8} \cdot \frac{\log(n)}{2} = \frac{7}{16} n \log(n) \in \Omega(n \log(n))$; making the amortized cost per instruction to be $\Omega(\log(n))$.

Problem 5. (25 pts) A tournament is a n-node *directed* graph such that between any two vertices u_i and u_j there is either a directed edge from u_i to u_j or a directed edge from u_j to u_i, but not both directed edges. In other words, a tournament is formed by taking the (undirected) complete graph over n nodes and assigning each edge a direction arbitrarily.
(As the name 'tournament' suggests, imagine you have n tennis players and all possible pairs of player have a match between them, and so for player i and player j you have a directed edge $\langle i, j\rangle$ if i losses to j in the match they had.)

(a) (15 pts) Prove (by induction on n) that in any n-node tournament one can order all vertices from first to last — $v_1, v_2, ..., v_n$ — such that in this order we get a directed path of length $n-1$ starting at v_1 and ending at v_n. (I.e. for every i we have that there's a directed edge from v_i to v_{i+1}.)

FYI, such a path is called a *Hamiltonian path* in a directed graph.

Answer. We prove that in any n-node tournament has an ordering of the vertices as $v_1, v_2, ..., v_n$ such that for every i we have a directed edge $\langle v_i, v_{i+1}\rangle$ by induction on n.

The base case is for $n = 1$ (where the claim trivially holds), or when $n = 2$ and so the tournament is either the single edge $\langle u_1, u_2\rangle$ (in which this is the ordering of the 2 nodes) or the single edge $\langle u_2, u_1\rangle$ (in which case the required ordering is the inverse of the two nodes).

Induction step. Fix $n \geq 3$. Assuming the claim holds for any tournament of size $n-1$ we show it also holds for a tournament of size n. Take a tournament over n nodes, look at the first $n-1$ of them and by induction take the path $v_1, ...v_{n-1}$ for which $\langle v_i, v_{i+1}\rangle$ exists in the graph for each i. Consider now the following three options:

- If there's an edge $\langle u_n, v_1\rangle$ then we have found the required n-node path: $u_n, v_1, ..., v_{n-1}$.
- If there's an edge $\langle v_{n-1}, u_n\rangle$ then again we have found the require path: $v_1, v_2, ..., v_{n-1}, u_n$.
- Otherwise, let's color each node on $v_1, ..., v_{n-1}$ black if there's an edge $\langle v_i, u_n\rangle$ or white if there's an edge $\langle u_n, v_i\rangle$. Because this is a tournament exactly one of the two edges must be present and so each node gets a color. Moreover, by assumption, v_1 is colored black and v_{n-1} is colored white. Therefore, iterating over the path, we begin with a black node and conclude with a white node. Hence, there must be an intermediate node v_j colored black where the following node v_{j+1} is colored white. Hence, in the graph we have the two edges $\langle v_j, u_n\rangle$ and $\langle u_n, v_{j+1}\rangle$. And voila, we have found the required n-node path: $v_1, v_2, ..., v_j, u_n, v_{j+1}, ..., v_{n-1}$. ■

Hint: the induction step, where we add a new node u_n to a tournament of size $n-1$ has two easy cases. One easy case is where there's an edge from u_n to v_1, and the other easy case is where there's an edge from v_{n-1} to u_n. What happens if neither hold?

(b) (10 pts) Convert your proof into a recursive algorithm that takes as an input such a tournament G in the adjacency-matrix model, and returns such a $n-1$-edge path. What is the runtime of your algorithm? Does it remind you of any sorting algorithm?

Answer. Our algorithm mimics `InsertionSort`. It holds all nodes in an array A which we first assume to hold $u_1, u_2, .., u_n$ (the array holding the nodes of G originally). All it does is to find the highest index i for a node for which there is an edge $\langle A[i], u_n\rangle$. If i is the last node, this is the second easy case; otherwise $i+1$ is such that the edge $\langle u_n, A[i+1]\rangle$ exists in the tournament so by putting u_n between $A[i]$ and $A[i+1]$ we follow on the third case. Lastly, if for all i we have found only the edge $\langle u_n, A[i]\rangle$ then we put u_n as the first node (the first of the three cases above).

`procedure PathInTournament(`$G = (V, E), n, A$`)` ** A is the array holding the vertices in the required order.
`for (`$j \leftarrow 2$ `upto` n`) do`
    $key \leftarrow u_j$
    $i \leftarrow j - 1$

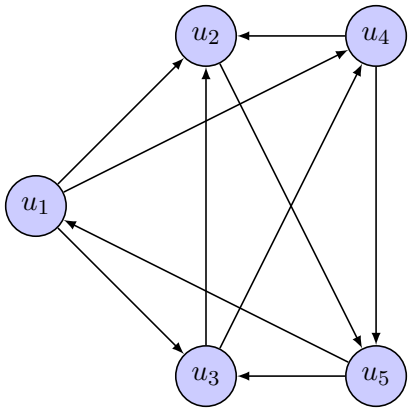


Figure 4: An example of a 5-node tournament, where:
– one possible path is $v_1 = u_1, v_2 = u_4, v_3 = u_5, v_4 = u_3, v_5 = u_2$;
– another possible path is $v_1 = u_3, v_2 = u_4, v_3 = u_5, v_4 = u_1, v_5 = u_2$.;
– and yet another possible path is $v_1 = u_4, v_2 = u_2, v_3 = u_5, v_4 = u_1, v_5 = u_3$ (and there are others too).

```
while (i > 0 and ⟨key, A[i]⟩ ∈ E(G)) do
    A[i + 1] ← A[i]
    i ← i − 1
A[i + 1] ← key
```

Much like insertion sort, in the worst-case we are forced for each node u_j to traverse all edges from u_j to u_i for all $i < j$. This makes the WC-runtime $O(n^2)$.