Diego Escobedo April 7, 2020

6.046 Problem Set Exam

Collaborators: *None*

Problem 2

(A) Given a bridging set, we are interested in finding the vertex in A that has the magic edge, in addition to the edge in $B = V \backslash A$ that has the magic edge. Because the two sets are bridging, we know there is exactly one vertex in each that connects them. To find the correct vertex in A we will arbitrarily divide it into A_1 and A_2. Then, we will query Alyssa, asking her if A_1 and $V \backslash A_1$ are a bridging set.

Obviously, given that we previously knew there was a single magic vertex in A, there is a 50% chance that this arbitrary subset also contains the magic vertex. Given that we knew the second magic vertex was in $V \backslash A$, we know it is guaranteed to not be in either A_1 or A_2. So, when we query Alyssa, we will gain a lot of information about which vertex in A was the magic one. If she answers Yes for A_1, then we know the vertex is in A_1 and we therefore eliminated half of the options. If she answers no, then we know the vertex is in A_2 and we also eliminated half the options. Therefore, when looking for the magic vertex in A, our recurrence looks like $T(k) = T(\frac{k}{2}) + O(1)$, which is fairly obvious corresponds to an $O(logk)$ run time until we get to a single element (one of the two magic vertices). In the worst case, A will have $O(n)$ elements, so the runtime to find this element will be $O(logn)$.

We can follow an identical but parallel argument as to how to find the same vertex in $V \backslash A$, which would also yield the same runtime of $O(logn)$. If we add up the costs of these two operations, we get a total runtime of $O(logn)$, which meets our bound.

(B) First, let us think of dividing our big set A into A_1 and A_2, randomly. Each one of these is of size $\frac{n}{2}$, which means each individual element x_i has a 50% chance of being in A_1 and the same chance of being in A_2. Notice that our magic vertices (call them u and v) at this point have no special properties, so they are also both independently equally likely to end up in either category. Let us consider these two edges and the different scenarios. Consider the set A_1. There is a 50% chance that it gets u, and a 50% chance that it gets v. Therefore, there is a 25% chance it gets both, and a 25% chance it gets 0. In this case, it wouldn't be a bridging set. However, there is also a 50% chance it is a bridging set right off the bat.

Seeing this argument, it is easy to see that if we divide A into two subsets. in expectation, since we have a 50% chance of success in finding a bridging set, we will find a bridging set in two tries. This is an example of a Las Vegas algorithm with expected runtime of $O(1)$.

Problem 3

`Algorithm:` Create a direct access array X of size m with integer keys from 0 to m, with all the elements starting off at 0. Iterate through A. For each value in A, change the value of X to 1. Now, start doing a two finger algorithm from X. Assume we have pointer a at position 0 and pointer b at position 1. If X[b] is 0, move b forward 1 spot. Else, calculate c=2b-a. Index into X[c] and check whether the value is 1. If it is, increase the counter of pheryples. Otherwise, continue. b will move along X, while a will remain in place. Once 2b-a > m, we exit this loop, advance a to 1 position forward, and reset b to be 1 position after a. Repeat this process until a has gone through every element in X.

`Runtime:` Building a direct access array takes O(m). Iterating through A takes O(n) time, and for each element, we incur an O(1) cost to access the array at the given position and change it to 1, giving us O(n) time. Our two-finger algorithm runs in $O(m^2)$ time. Therefore, the total cost is $O(m + n + m^2) = O(n + m^2)$, which is asymptotically less than $(n + m)^2$, which means that our algorithm runs in $o((n + m)^2)$.

`Correctness:` The key to this problem lies in the fact that all the elements are unique. Therefore, by inserting them into a direct access array, we are in effect 'sorting' them in O(n + m) time, and also gaining the valuable set interface where we can perform a series of operations in favorable runtimes. The algorithm is correct because we are checking all pairs of numbers, and for each pair, checking whether there is a third number that can make a pheryple. The trick ies that we can find that third number in constant time, which is convenient for runtime purposes. However, at its core, we are ultimately still just checking every possible combination.

Problem 4

(A) The chance that someone needs help is a direct function of the number of people that have been seated before them. We declare X_i to be an indicator random variable (1 if student i needs help and 0 if they don't), with i ranging from 0 (for the first student, when 0 students have been seated) to $n-1$ (for the nth student, when $n-1$ students have been seated). We know that X_i takes on the value 1 with $p = \frac{(i)^2}{m^2}$. Thus, we can use linearity of expectation to find the expected value for the sum of the X_i's, X. The reason that the chance they need help is $\frac{(i)^2}{m^2}$ is because there are i out of the m seats taken up, and they only need help if they are placed in the i out of the m seats available twice. So, we can state:

$$
\begin{aligned}
X &= X_1 + X_2 + ... + X_n \\
E[X] &= E[X_1 + X_2 + ... + X_n] \\
E[X] &= E[X_1] + E[X_2] + ... + E[X_n] && \text{By linearity of expectation} \\
E[X] &= \sum_{i=0}^{n-1} \frac{(i)^2}{m^2} && \text{By Bernoulli RV expectation + equation above} \\
E[X] &= \frac{1}{m^2} \sum_{i=0}^{n-1} i^2 \\
E[X] &\leq \frac{1}{m^2} \frac{n^3}{3} && \text{By the hint}
\end{aligned}
$$

Thus, we can see that the expected number of students who will need help is at most $\frac{n^3}{3m^2}$.

(B) If we plug in $2n$ for m, we can see that we get that $E[X] = \frac{n}{12}$. By the form of the problem, it is a reasonable guess to assume that we are supposed to use Chernoff bounds. We assume we can use them because they are all mutually independent of one another. Each rv is concerned with what is happening at its current timestep only. In other words, the fact that someone was helped or not previously has no bearing on how long it will take this person to sit, that is simply a function of how many people came before. To use the chernoff bounds to find the probability that more than a third of the students will have to be helped, we need to find the β that satisfies $(1+\beta) * \frac{n}{12} = \frac{1}{3}$, which gives us $\beta = 3$. Using the correct Chernoff bound (case 2), we get that the probability that more than a third of students will need help is at most $e^{-\frac{n}{12}}$. In other words, $Pr[X > \frac{n}{3}] < e^{-\frac{n}{12}}$.

Problem 5

(A) Our potential function will be (iii), $\Phi(e_0, e_1) = (k_1 + k_2)e_0 + k_2e_1$.
(B) To prove that the amortized cost of taking a bath is $k_0 + k_1 + k_2$, we will do a proof by cases, and make sure that in each case the amortized cost $\hat{c_i} = c_i + \Delta\Phi(e_0, e_1)$ is equal to $k_0 + k_1 + k_2$. Our 3 cases are:

- **Neither tank runs out of water:** In this case, the actual cost is k_0 because we are drawing a single bath. The $\Phi_{\text{initial}}(e_0, e_1) = (k_1 + k_2)e_0 + k_2e_1$, and the $\Phi_{\text{final}}(e_0, e_1) = (k_1 + k_2)(e_0 + 1) + k_2e_1$. So, we get:

$$\begin{aligned}\Delta\Phi(e_0, e_1) &= \Phi_{\text{final}}(e_0, e_1) - \Phi_{\text{initial}}(e_0, e_1)\\ &= [(k_1 + k_2)(e_0 + 1) + k_2e_1] - [(k_1 + k_2)e_0 + k_2e_1]\\ &= (k_1 + k_2)(e_0 + 1) - k_1e_0 - k_2e_0\\ &= k_1e_0 + k_2e_0 + k_1 + k_2 - k_1e_0 - k_2e_0\\ &= k_1 + k_2\end{aligned}$$

  So, we can see that $\hat{c_i} = c_i + \Delta\Phi(e_0, e_1) = \mathbf{k_0 + k_1 + k_2}$, which is what we wanted for the amortized cost.

- **Tank 0 runs out of water:** In this case, the actual cost is k_1n+k_0 because we are first filling up the n (aka $e_0 = n$) units of water in Tank 0 with some of the units Tank 1 has, and then drawing a single bath. The $\Phi_{\text{initial}}(e_0, e_1) = (k_1+k_2)n+k_2e_1 = k_1n+k_2n+k_2e_1$, and the $\Phi_{\text{final}}(e_0, e_1) = (k_1 + k_2)(1) + k_2(e_1 + n) = k_1 + k_2 + k_2e_1 + k_2n$. So, we get:

$$\begin{aligned}\Delta\Phi(e_0, e_1) &= \Phi_{\text{final}}(e_0, e_1) - \Phi_{\text{initial}}(e_0, e_1)\\ &= [k_1 + k_2 + k_2e_1 + k_2n] - [k_1n + k_2n + k_2e_1]\\ &= k_1 + k_2 + k_2e_1 + k_2n - k_1n - k_2n - k_2e_1\\ &= k_1 + k_2 + -k_1n\end{aligned}$$

  So, we can see that $\hat{c_i} = c_i + \Delta\Phi(e_0, e_1) = \mathbf{k_0 + k_1 + k_2}$, which is better than what we wanted for the amortized cost.

- **Both tanks run our of water:** In this case, the actual cost is $k_2mn + k_1n + k_0$ because we are first filling up the mn (aka $e_1 = mn$) units of water in Tank 1 with some of the units Tank 2 has, then filling up the n (aka $e_0 = n$) units of water in Tank 0 with some of the units Tank 1 has, and then drawing a single bath. The $\Phi_{\text{initial}}(e_0, e_1) = (k_1 + k_2)n + k_2mn = k_1n + k_2n + k_2mn$, and the $\Phi_{\text{final}}(e_0, e_1) = (k_1 + k_2)(1) + k_2(n) = k_1 + k_2 + k_2n$. So, we get:

$$\begin{aligned}\Delta\Phi(e_0, e_1) &= \Phi_{\text{final}}(e_0, e_1) - \Phi_{\text{initial}}(e_0, e_1)\\ &= [k_1 + k_2 + k_2n] - [k_1n + k_2n + k_2mn]\\ &= k_1 + k_2 - k_1n - k_2mn\end{aligned}$$

So, we can see that $\hat{c_i} = c_i + \Delta\Phi(e_0, e_1) = \mathbf{k_0} + \mathbf{k_1} + \mathbf{k_2}$, which is better than what we wanted for the amortized cost.

Overall, since the worst case of all the amortized operations is $\mathbf{k_0} + \mathbf{k_1} + \mathbf{k_2}$, we can say that overall this is the amortized cost of giving a bath.

Problem 6

(A) We are asked to provide $P(\text{incorrect})$. So, we could also say that we are looking for $P(\text{incorrect}) = 1 - P(\text{correct}) \leq 1 - e^{-1}$. For this to be true, we want to prove that $P(\text{correct}) \geq e^{-1}$. We can calculate:

$$\begin{aligned} P(\text{correct}) = & P(\text{no duplicates}) * P(\text{correct}|\text{no duplicates}) \\ & + P(1 \text{ duplicate}) * P(\text{correct}|1 \text{ duplicate}) \\ & + ... \\ & + P(\left\lfloor \frac{m}{2} \right\rfloor \text{ duplicates}) * P(\text{correct}|\left\lfloor \frac{m}{2} \right\rfloor \text{ duplicates}) \end{aligned}$$

However, we can lower bound all of that by something useful, thanks to a few key observations. First of all, notice that there cannot be false positives. If there are no duplicated, we will never return duplicated. So, the very first term can basically be 'eliminated'. Further, we can consider the worst, most adversarial case possible. This would be the case with a single duplicate, where the duplicate elements would be the first and the last. This would give our hashing function a maximal chance of 'overriding' the first duplicate, leaving us unable to detect the duplicate. Therefore, we can 'collapse' all the other cases into this one, and say that

$$P(\text{correct}) \geq P(\text{correct}|1 \text{ worst-case duplicate})$$

In the case of the worst case duplicate, we can imagine that the first duplicate gets hashed to one of the m slots. Thus, a failure would happen if one of the other numbers in the sequence fell into the same square, with probability $\frac{1}{m}$. Thus, the probability of 'succeeding' (aka not prematurely colliding with the first duplicate) is $(1 - \frac{1}{m})$. We need this to be true for the $m - 1$ elements, which means that $P(\text{correct}|1 \text{ worst-case duplicate}) = (1 - \frac{1}{m})^{m-1} \geq e^{-1}$. Thus, we go back to our equation, and can now state

$$P(\text{correct}) \geq e^{-1}$$

which as we said in the beginning is enough to prove that the probability that the above algorithm returns an incorrect answer is at most $1 - e^{-1}$.

(B) In this case, yes, H is universal. Let us recall the definition of a universal hash function. We say that for two keys k and k', H is universal if for a random choice of function, the chance of k and k' colliding is less than or equal to (in the case of mapping from n-1 to n-2) $\frac{1}{n-1}$. Looking at the hash function, we can see that an adversarial key could be $l = n-1$. This is because it always hashes to 0 across all functions, whereas other numbers can randomly span the range given that we are randomly choosing a. So, let us fix key k as $n-1$. Now, we can observe that there is only 1 other key that maps to 0. This key has the value $l = n-1-a$ for one of n-1 random a's. So, given that we choose $k = n-1$, if we choose k' equal to any number between 0 and n-2, there is only a $\frac{1}{n-1}$ chance that it collides. Since $k = n-1$ is an adversarial key, $\frac{1}{n-1}$ is an upper bound on the probability of a collision. So, we can say that for this hash family H,

$$\Pr_{h \in H}\{h(k) = h(k')\} \leq \frac{1}{n-1}$$

which for our hash family that takes inputs to $\{0, ...n-2\}$ means that it is universal.

(C) As before, we discussed that an adversarial input would be one where the first and last element were a duplicate, and all the other elements were not. So, let us analyze what the probability of success would be. Let us imagine that s_1 and s_{n-1} are a key $x = n-1$. Therefore, there are another $n-2$ keys that are different from x. However, our sequence is only $n-1$ long, and two of those already have $x = n-1$ as their value, so let us imagine that we randomly assign these $n-3$ items in the sequence distinct values from the $n-2$ keys that are different from x. We know that, for $x = n-1$ and x', their chance of colliding is exactly $\frac{1}{n-1}$ (this is the upper bound we proved above). So, since we have $n-3$ distinct keys to test, we can add all the probabilities that they collide together, which means the probability of one of our $n-3$ keys colliding with x is $\frac{n-3}{n-1}$. So, in other words, if we cleverly choose a sequence, our probability of failure is $\frac{n-3}{n-1}$. This means that our probability of success is $\frac{2}{n-1}$.

Problem 7

(A)

`Algorithm:` Consider each offer o. If $o * \gamma \geq$R, then automatically accept the offer. If we get to the last person and we have not yet accepted any offers, accept the last person's offer. x‘
`Runtime/Space Complexity:` We only need to store the value of R and γ, so constant space complexity. We only need to perform multiplication and comparison, which takes O(1) time per input.

`Correctness:` Obviously, if we accept an offer o from the sequence s.t. $o * \gamma \geq$R, we are correct. So, lets cover the other case, where the biggest offer that we should have accepted was not within a factor of γ of R.

In this case, this means that the optimal offer $o^\star$ was subject to $o^\star < \frac{R}{\gamma}$. Therefore, this means that the last offer o^- had to be within a γ factor of $o^\star$, which means the following must be true for out algorithm to be correct: $o^- * \gamma \geq o^\star$. Further, by the inequality above, it is also required that $o^- \geq \frac{R}{\gamma^2}$, which just means that $o^- \geq$ r. Since we are told that all the values are greater than or equal to r, which means the last offer can be anything and it will be within a factor of γ of the highest offer. This concludes our proof by cases.

(B)

`Algorithm:` For each offer o, accept it with probability

$$\int_r^o \frac{2o}{R^2 - r^2} do$$

Because there is a randomness in whether we accept it or not, when given the same set of inputs, it could generate different answers.

`Runtime/Space Complexity:` We are not given any runtimes or space complexities. However, calculating that probability is not that terrible and only requires squaring a number, which can obviously take some time if we are dealing with big ranges but should not be a problem.

`Correctness:` We want the expected value of our accepted offer r.v. T to be within a factor of $\frac{1}{2\lceil log_2 \gamma \rceil}$ of the highest offer. To be robust against the worst case, this means that we want:

$$E[T] = t^\star \geq \frac{R}{2\lceil log_2 \gamma \rceil}$$

So, let us first calculate the expected value of T. This is kind of tricky and annoying, but bear with me. The expectation for a continuous random variable like the value of the offer can be represented by

$$\int_r^R t * f_T(t) dt$$

where $f_T(t)$ is a function that represents the probability of this event happening. In our case, this can be expressed as the product of two things: the probability of the offer taking on that value times the probability of us taking the offer at that value. So, we can express:

$$\begin{aligned} f_T(t) &= f_O(t) * \int_r^t \frac{2t}{R^2 - r^2} dt * c \\ &= \left(\frac{1}{R - r}\right) * \left(\frac{2}{R^2 - r^2} * \int_r^t t dt\right) * \left(\frac{3(R + r)}{R + 2r}\right) \\ &= \frac{3 * (t^2 - r^2) * (R + r)}{(R^2 - r^2)(R - r)(R + 2r)} \end{aligned}$$

where $f_O(t)$ is the pdf for the r.v. representing the offer (continuous rv. from r to R), the integral is the probability we calculated above in the algorithm section, and $c = \frac{3(R+r)}{R+2r}$, which is a normalizing factor to make sure the integral is equal to 1 (I said that quick but hooooooly took a long time to calculate), a property that all pdf's must have. Excellent!

Now we can simplify the integral for the probability must prove the following:

$$\begin{aligned}
E[T] &= \int_r^R t * \frac{3 * (t^2 - r^2) * (R + r)}{(R^2 - r^2)(R - r)(R + 2r)} dt \\
&= 3 \int_r^R \frac{t^3(R + r) - t(r^2 R - r^3)}{(R^2 - r^2)(R - r)(R + 2r)} dt \\
&= \frac{3}{(R^2 - r^2)(R - r)(R + 2r)} \left((R + r) \int_r^R t^3 dt - (r^2 R - 2r^3) \int_r^R t dt \right) \\
&= \frac{3}{(R^2 - r^2)(R - r)(R + 2r)} \left((R + r) \frac{R^4 - r^4}{4} - (r^2 R - 2r^3) \frac{R^2 - r^2}{2} \right) \\
&= \frac{3}{(R^2 - r^2)(R - r)(R + 2r)} \left(\frac{(R + r)(R^4 - r^4) - (r^2 R - 2r^3)(R^2 - r^2)}{4} \right) \\
&= \frac{3}{(R - r)(R + 2r)} \left(\frac{(R + r)(R^2 + r^2) - r^2(R - 2r)}{4} \right)
\end{aligned}$$

Now, lets check whether that is bigger than the original equation we had:

$$E[T] = \frac{3}{(R - r)(R + 2r)} \left(\frac{(R + r)(R^2 + r^2) - r^2(R - 2r)}{4} \right) \geq \frac{R}{2 \lceil log_2 \gamma \rceil}$$

$$6 \lceil log_2 \gamma \rceil \left((R + r)(R^2 + r^2) - r^2(R - 2r) \right) \geq 4R(R - r)(R + 2r)$$

It is not immediately obvious by looking at this expression, but this is always true! Zing!!!!! Therefore, this means that the expected value is within a factor of $\frac{1}{2\lceil log_2 \gamma \rceil}$ of the highest offer (which in the worst case is R).

Problem 8

(A) Frandly's task is to make the cost of producing a spanning tree the minimum possible. This is very easy, since our objective in building the roads is minimizing the cost. Therefore, her actions and our actions work in tandem. This is differently from Nesty, because even though he might assign some tree with really high weights, you might be able to find an alternative path that isn't super high. But, returning to Frendly's simple task, all she has to do is assign the n-1 lowest costs to a spanning tree. This can realistically be any spanning tree, and there are many different algorithms (such as BFs and DFS) that could tell us a spanning tree. Therefore, if we assign the lowest weight edges to any spanning tree, it will become a minimum cost spanning tree of cost $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. The rest of the roads can be assigned any way that is desired, since the builder will ignore these as he builds the MST.

(B) For this one, we can use a proof by induction. **Inductive hypothesis:** Nesty can assign the road costs so that the cheapest road network will have a cost of $\sum_{i=1}^{n-1}(\binom{i}{2}+1)$.
Base Case: n=1 vacuously true. For n = 2, there is a single edge weight, 1. There is only one way to assign it, between the two edges, so the minimum cost road assignment is 1. $\sum_{i=1}^{2-1}(\binom{i}{2}+1) = 1$. Correct.
Inductive Step: Assuming this is true for k nodes, let us prove that this implies it must be true for k+1. So, if this works for k, we can see that we must have assigned all those edges successfully for the cost to be the summation from the IH. If we add a new node to the system, that means there will be k new connections (from all the previous nodes in the network to the next one, since th graph is fully connected). Since the edge weights of the k-network ranged from $\{1, ..., \binom{k}{2}\}$, that means that the new edges will range from $\{\binom{k}{2}+1, ..., \binom{k}{2}+k = \binom{k+1}{2}\}$. Now, obviously, the minimum cost builder will be forced to take one of those edges, and they will choose the lowest one. So, the total cost will be $\sum_{i=1}^{k-1}(\binom{i}{2}+1) + \binom{k+1}{2}$, which is just equal to $\sum_{i=1}^{k}(\binom{i}{2}+1)$. Thus, we can see that the IH being true for k nodes implies it is true for k+1 nodes too. This concludes the proof by induction.

(C) Because this is a fully connected graph, it is going to help us find an intuition as to why this bound is the maximum. First, let us recall the MST property. We are given G = (V,E), and told that U is a proper subset of V and V $\neq \emptyset$. Then, if (u,v) is an edge of **lowest cost** with $u \in U$ and $v \in V \backslash U$, then there is a MST containing (u,v). This property, which was proved in lecture, is the key to this problem. Recall that our objective is to "build roads so that any pair of towns can reach each other and with a minimal total cost". This is, in other words, building an MST, but the trick is that there is an adversarial opponent assigning edge weights beforehand. But, the MST property gives us an idea of how to determine which edge actually makes it into the MST.

Imagine a singleton set U (aka a U is just a single element x). Given a graph with n vertices, this means that there are n-1 candidates to be the 'light edge' connecting x to the MST. Here is the key intuition. No matter what Nesty makes these weights, no matter how high they are, since they are all distinct, the optimal builder will 'eliminate' n-2 edges with higher weight. In other words, if we want to force the optimal builder to include a high cost edge, there has to be n-2 *even higher* cost distinct edges to force him to use that one. To put this more intuitively, if x is the last element we need to connect to form an MST, and since x is connected to every node in the MST, we are just going to take the lowest cost edge.

With the intuition from the paragraph above, we can begin to actually do some math. The actual problem is a little different from what is given above. For the first node we consider, we can give the builder n-1 edges, but for the second node, the figure will be n-2, then n-3, and so on because the edges between those subsequent nodes and the ones before them have already been assigned. So, we can think of how to give them to him in the worst adversarial case. Obviously, we would start by trying to give the builder many bad edges. We would first give them the worst n-1 edges, then following worst n-2, then the following worst n-3, and so on for the n different nodes.

So, let us think what edge weights the builder is using. Call the last edge weight, $\binom{n}{2}$, z. So, from the first bundle, the builder would choose to use $z-(n-1)+1$. From the second, they would choose $z-(n-1)-(n-2)+1$, form the third $z-(n-1)-(n-2)-(n-3)+1$, and so on and so forth. So, let us add this all up. That is going to be n z's , n-1 1's, n (n-1)s, n-1 (n-2)s, etc.. This will give us an equation $n*\binom{n}{2}+n-1-\sum_{i=1}^{n} i*(i-1)$. This simplifies to $n-1+\frac{n^2(n-1)}{2}+\frac{n(n+1)}{2}-\frac{2n^3+3n^2+n}{6}$. This further simplifies to $n-1+\frac{n^3-3n^2+2n}{6}$. Perfect. This is the simplest form of the absolute max the adversary can make the builder use.

Now, lets check what the form in the equation was and see if it matches:

$$\begin{aligned}
\text{cost} &= -\left(\binom{n}{2}+1\right)+\sum_{i=1}^{n}(\binom{i}{2}+1) \\
&= -\left(\binom{n}{2}+1\right)+\sum_{i=1}^{n}\binom{i}{2}+\sum_{i=1}^{n}1 \\
&= -\left(\binom{n}{2}+1-n\right)+\frac{1}{2}\left(\sum_{i=1}^{n}i^2-\sum_{i=1}^{n}i\right) \\
&= -\left(\binom{n}{2}+1-n\right)+\left(\frac{n^3-n}{6}\right) \\
&= \frac{6n}{6}-\left(\frac{3n^2-3n}{6}\right)+\left(\frac{n^3-n}{6}\right)-1 \\
&= \left(\frac{n^3-3n^2+8n}{6}\right)-1 \\
&= n-1+\left(\frac{n^3-3n^2+2n}{6}\right)
\end{aligned}$$

BAM!!!!! LETS GOOOOOOOOOOOOOOOOOOOOO. That was kind of long (maybe unnecessarily so), but you see the point. That formula we used for our induction proof above is the absolute maximum the adversary can force the builder to use.