Subject: Homework 5 solutions
From: Harold Connamacher <hsc21@case.edu>
To: Huixian Yang <hxy298@case.edu>,Sheng Guan <sxg967@case.edu>,Aimee Chau <axc693@case.edu>,Brendan Dowling <bld43@case.edu>,Colby Saxton <cas264@case.edu>,Jian Liu <jxl2184@case.edu>,Seohyun Jung <sxj393@case.edu>
Date Sent: Fri, 6 Dec 2019 22:27:00 -0500
Date Received: Fri, 6 Dec 2019 19:27:19 -0800 (PST)

Dear TAs,

Here are solutions to homework 5. The rubric is the same as before.

Problem 1:
The problem is NP-hard, and we can do a reduction from Subset Sum.

We are given a list of numbers v_1,\ldots, v_k and a target number T. For each number v_i, create a sell order with price 3, quantity v_i secret price 2, and secret min quantity v_i.
Create a buy order with price 1, quantity T, secrete price 2, and secret min quantity T.

There exists a sale if and only if there is a set of numbers that sum to T. Suppose there is a sale. Since the buy price is lower than the sell price, the sale has to be at the secret price. Since the buy order is for T and the min quantity is T, we must have a set of sell orders whose secret min's sum to exactly T. Therefore, there is a set of numbers that sums to T.

Suppose we have a set of numbers that sum to exactly T. If we choose those orders, we have a set of sell orders whose secrete minimums sum to exactly T, and that matches the buy order that is requesting a minimum of T. Therefore, we have a sale.

The runtime of the reduction is linear time since we are just turning numbers into sales of the same binary value with a little extra constant sized headers.

Problem 2:
Let X[a,b,k] be the best exchange rate of converting from currency a to currency b using intermediate currencies 1,...,k currencies.

Initially X[a,a,0] = 1 and X[a,b,0] = r(a,b) for all a,b.
Either we use currency k as an intermediate currency in the arbitrage or we do not.

X[a,b,k] = max{X[a,b,k-1], X[a,k,k-1]\times X[k,b,k-1]}.

The maximum "cycle" is the largest value X[a,a,n] for all a. To find the cycle, we just run the recurrence backwards. However, this may not be a true cycle because the algorithm does not prevent a vertex from repeating, but let us ignore that for this quiz. (Suppose there is a vertex v that repeats twice, then there must exist a v with T[v,v,.k] > 1, and so we have found a cycle that is a currency arbitrage > 1, and we can make that trade instead.)

The table is n^3 and it takes O(1) to calculate each value for a worst case running time of O(n^3).

The proof is: assume X[a,c,k-1] stores the best we can do trading from a to c using currencies 1 to k-1 for all a and c.
To convert a to b using intermediate currencies 1,..,k, we either use currency, in which case we must convert a to k using currencies 1,...,k-1, and then convert k to b using currencies 1,...,k-1, or we do not use k, in which case we use currencies 1,...,k-1 to convert currency a to b. Because

we maximize a product my maximizing each operand, we want the largest currency exchange rate, and by the induction hypothesis, X[a,c,k-1] stores the maximal exchange value for all a and c using currencies 1,...,k-1.

Problem 3:
THe problem is NP-hard, and we will do a reduction from Independent Set. We are given an instance of Independent Set: an gradh G with n vertices and m edges and a target number K. Create n jobs, one job for each vertex. Divide the day into m distinct time intervals, one for each edge. If a vertex v is incident to edge e, then job v has to do processing during time interval e. Let k = K, and we ask if there are k jobs we can schedule on one processor.

Prove we can schedule k jobs on the same processor if and only if G has an independent set of size K.

Assume we can find k jobs to schedule on one processor. Then there are no overlapping time intervals for any of the k jobs. That means the vertex for a job does not share an edge with any other of the k jobs, and since K = k we have an independent set of size K.

Assume we can find an independent set of size K. No pair of vertices in the independent set share an edge. Therefore none of the jobs associated with the chosen vertices share the same time interval. So we can schedule all the jobs, and there will be no overlap. Since k=K, we are able the schedule k jobs on the same processor.

Problem 4:
Algorithm,: Find all SCC's of the network. For each edge in a SCC, set its capacity to infinity. For each edge not in a SCC, set its capacity to m_e. Run F-F max flow. If there exists an edge e not in a SCC with the flow on e is less than m_e, then we need to push more flow. Run BFS on the network to find an s->t path that uses that edge e, and run BFS from s on the residual graph to find a min cut. There is some edge e* in the min cut that is on the s->t path that includes e. Increase the capacity of that edge by 1. Push more flow or find another min cut, and repeat until every non SCC edge has flow >= its capacity. For any edge in a SCC, if the SCC receives flow, then cycle the flow in the SCC until all edges in the SCC achieve their minimum capacity. (Minor point: if there is no flow to the SCC, then the incoming and outgoing edges all have 0 minimum capacity. We can then use DFS to find one incoming and outgoing edge and sets its minimum capacity to 1. If the SCC is not on any s->t path, then all the edges of it must have 0 minimum capacity or it is impossible to find such a flow.)

Proof: Consider a "minimal" S-T cut, (no edges from T to S), since the flow moves from S -> T and since there are no reverse edges, any minimal flow must be at least as large as the sum of the capacities of the S -> T edges across the cut. Therefore the flow must be at least as large as the largest minimal S-T cut.

Let e be an edge not in a SCC with flow less than m_e. Consider the s->t path that includes edge e, and let e* be the edge of the min cut that is part of this s->t path. Edge e is not part of any min cut since the flow on e is less than m_e. Therefore there is a larger S-T minimal cut that includes edge e. This minimal cut cannot include e* since there are no back edges in the cut. Thus there exists a larger minimal cut that does not include edge e* so we can freely increase the capacity of e* without affecting the larger minimal cut.

The resulting flow is the smallest flow that achieves flow on every edge. In the preceding round of the algorithm, the min cut found was not the largest minimal cut, and above we showed that the flow must be at least as large as the largest minimal cut. Since each round increases the flow by 1, we have a flow equal to the largest minimal cut.

Runtime: O(m) to find the strongly connected components, then O(f* m) where f* is the capacity of the flow. To compute the flow, at worst we have to do O(m) for one flow augment, then O(m) to find the min cut(s), and repeat until we get to f*.

Problem 5:
The solution uses dynamic programming.
Let T[a,b] store the minimum cost of cutting a board of length l_b-l_a that has b-a-1 marks at lengths l_b - l_k for k from a+1 to b.
T[a,a+1] = 0 for all a from 0 to n-1, because there are no marks to cut.
T[a,b] = min_{k from (a+1) to (b-1)} {T[a,k] + T[k,b] + c + k(l_b - l_a)}

The total minimum cost of cutting the board is in T[0,n]. The first cut should be at the kth mark where k is such that T[0,n] = T[0,k] + T[k,n] + c + k l_n. And we repeat for each piece. The optimal place to cut a board that runs from the a-th mark to the b-th mark (of the original board) is at mark k where k is such that T[a,b] = T[a,k] + T[k,b] + c + k(l_b - l_a).

The run time is O(n^3) because the table is nxn (only half the entries used) and we need O(n) to calculate each entry.

Proof. Assume T[a,k] and T[k,b] store the optimal cost of cutting a board of length l_k-l_a at marks a+1 to k-1 and cutting a board of length l_b - l_k at marks k+1 to b-1 for all k from a+1 to b-1. Consider a board of length l_b-l_a with marks at lengths l_b-l_k from k from a+1 to b-1. The first cut can be at any of these marks. Since the total cost is a simple summation of the cost of cutting at each mark, to get the minimum cost we need to optimal the cost of cutting the board after this cut, no matter where we make that first cut. By the induction hypothesis T[a,k] and T[k,b] store the minimum costs for each piece resulting from this cut. Therefore T[a,b] stores the minimum cost of cutting this board.

Problem 6:
(The problem is NP-hard. That is not required, but the proof is a reduction from Subset Sum.)

Here is my approximation algorithm: Sort the packages so w_1 >= w_2 >= ... >= w_n. Consider the packages in order, and place package i into the current truck if it fits. If it does not, send the truck and place package i into a new truck.

Proof. First we show that every truck except the last one ships with more than M/2 weight in it. Suppose a truck ships with less than M/2 in it, and package k was the one we tried to fit and failed. Then package k has weight larger than M/2 which means package k-1 has more than M/2 weight. But package k-1 was on the truck that shipped. That contradicts the assumption that the truck had less than M/2 weight.

The last truck may ship with less than M/2 weight, but the combined weight of the last two trucks is greater than M. If not, then we could have fit the packages in the last truck into the truck before it.

Thus, the average shipping weight of all the trucks is larger than M/2. Since the optimal solution uses K trucks, the total amount of weight shipped is at most KM, and we are using no more than KM / (M / 2) = 2K trucks.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

Subject: Final breakout session notes and quiz solution
From: Harold Connamacher <hsc21@case.edu>
To: Aimee Chau <axc693@case.edu>,Seohyun Jung <sxj393@case.edu>,Huixian Yang <hxy298@case.edu>,Jian Liu <jxl2184@case.edu>,Colby Saxton <cas264@case.edu>,Brendan Dowling <bld43@case.edu>,Sheng Guan <sxg967@case.edu>
Date Sent: Sun, 1 Dec 2019 16:29:46 -0500
Date Received: Sun, 1 Dec 2019 13:30:06 -0800 (PST)
Attachments: breakout12.pdf

Dear TAs,

Here is the solution to the last quiz, and the breakout notes are attached. A quick reminder that the quiz is different this time.

A key idea is that cycles can have an unlimited amount of "flow" in them by taking a single flow and cycling it as many times as needed. For example, consider a strongly connected component. You can send 1 flow into it, and then that flow can cycle around inside the SCC as many times as needed until the min capacity of each edge is met. Since only one unit of flow entered the SCC, only one unit of flow can leave the SCC. (Think of the SCC as a "lake" in the middle of a river where the lake can be as deep as needed, but does not affect the river flow.)

Algorithm, part 1: Find all SCC's of the network. For each edge in a SCC, set its capacity to infinity. For each edge not in a SCC, set its capacity to m_e. Run F-F max flow.   If there exists an edge e not in a SCC with the flow on e is less than m_e, then we need to push more flow. Run BFS on the network to find an s->t path that uses that edge e, and run BFS from s on the residual graph to find a min cut. There is some edge e* in the max cut that is on the s->t path that includes e. Increase the capacity of that edge by 1. Push more flow or find another min cut, and repeat until every non SCC edge has flow >= its capacity. For any edge in a SCC, if the SCC receives flow, then cycle the flow in the SCC until all edges in the SCC achieve their minimum capacity. (Minor point: if there is no flow to the SCC, then the incoming and outgoing edges all have 0 minimum capacity. We can then use DFS to find one incoming and outgoing edge and sets its minimum capacity to 1. If the SCC is not on any s->t path, then all the edges of it must have 0 minimum capacity or it is impossible to find such a flow.)

Proof: Consider a "minimal" S-T cut, (no edges from T to S), since the flow moves from S -> T and since there are no reverse edges, any minimal flow must be at least as large as the sum of the capacities of the S -> T edges across the cut.   Therefore the flow must be at least as large as the largest minimal S-T cut.

Let e be an edge not in a SCC with flow less than m_e. Consider the s->t path that includes edge e, and let e* be the edge of the min cut that is part of this s->t path. Edge e is not part of any min cut since the flow on e is less than m_e. Therefore there is a larger S-T minimal cut that includes edge e. This minimal cut cannot include e* since there are no back edges in the cut. Thus there exists a larger minimal cut that does not include edge e* so we can freely increase the capacity of e* without affecting the larger minimal cut.

The resulting flow is the smallest flow that achieves flow on every edge. In the preceding round of the algorithm, the min cut found was not the largest minimal cut, and above we showed that the flow must be at least as large as the largest minimal cut. Since each round increases the flow by 1, we have a flow equal to the largest minimal cut.

Runtime: O(m) to find the strongly connected components, then O(f* m) where f* is the capacity of the flow to compute the flow (at worst we have to do O(m) for one flow augment, then O(m) to find the min cut(s), and repeat until we get to f*).

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, December 2, 2019

1. The quiz is different this time. Tell the students they have 15 minutes to work with each other to solve the quiz. They are not use computers or phones or consult their notes. They also are not to write anything on the quiz papers. They can write on boards, other paper, etc. But, they will have to put these away when taking the quiz.

2. Now they put everything away and write the quiz for 15 minutes.

3. Here is the final problem to give them:

   A fraternity has a lot of empty drink bottles from all the parties. You, as a new pledge, are given a task. Which algorithm task should you use, and which is NP-hard?

   (a) Place the bottles in a giant circle where adjacent bottles are brands of drink that were served at the same party.

   (b) Fill a bin that can hold up to 10 pounds with bottles to maximize the amount of money back on the deposit. The bottles are different weights and have different amounts of deposit.

   (c) Assign the bottles to different fraternity members to take to recycling such that the bottles are divided as evenly as possible and a member is only assigned bottles of a brand that that member brought to some party.

   (d) Assume the bottles all have twist caps, but they have slightly different sizes. They removed the labels from all the bottles, throw the caps in a big pile, and ask you to correctly recap every bottle.

4. (a) is NP-hard. Take a graph that you want to find a Hamilton Cycle on. Make every vertex its own brand of drink, and have each edge represent a party where both brands were served.

5. (b) is dynamic programming. This can be directly reduced to knapsack.

6. (c) is flow. Link every bottle to a member that brought that brand. Give capacity n/k to each edge from a member to t (where n is the # of bottles and k is the # of fraternity brothers). If that flow is not size n (all bottles assigned), increase the capacity of the brother $\rightarrow t$ edges by 1 until a flow of size n is achievable. Since there are at most nk edges, and we have to run the flow at most n times, we get $O(n^3k)$.

7. (d) is divide and conquer. Take a cap and divide all the bottles into 3 groups: fits the cap, the cap is too small and the cap is too large. Then take a bottle that matches this cap and divide all the caps equivalently. Then you can recurse on each half to get expected time $O(n \log n)$.

Subject: Quiz solution and breakout notes
From: Harold Connamacher <hsc21@case.edu>
To: Seohyun Jung <sxj393@case.edu>,Huixian Yang <hxy298@case.edu>,Brendan Dowling <bld43@case.edu>,Colby Saxton <cas264@case.edu>,Aimee Chau <axc693@case.edu>,Sheng Guan <sxg967@case.edu>,Jian Liu <jxl2184@case.edu>
Date Sent: Sun, 24 Nov 2019 13:58:18 -0500
Date Received: Sun, 24 Nov 2019 10:58:38 -0800 (PST)
Attachments: breakout11.pdf

Dear TAs,

The breakout notes are attached. I am going to follow Seohyun's idea and have a bunch of problems for the students to try to figure out which algorithm to use. However, these problems are not that simple. So, I want them to figure out what the algorithm theys should use, but you should not try to give the full proofs because you will not have time. They should be more motivated for this exercise since it is just like what the final will be.

Here is a quiz solution:
Let X[a,b,k] be the best exchange rate of converting from currency a to currency b using intermediate currencies 1,...,k currencies.

Initially X[a,a,0] = 1 and X[a,b,0] = r(a,b) for all a,b.
Either we use currency k as an intermediate currency in the arbitrage or we do not.

X[a,b,k] = max{X[a,b,k-1], X[a,k,k-1]\times X[k,b,k-1]}.

The maximum "cycle" is the largest value X[a,a,n] for all a. To find the cycle, we just run the recurrence backwards. However, this may not be a true cycle because the algorithm does not prevent a vertex from repeating, but let us ignore that for this quiz. (Suppose there is a vertex v that repeats twice, then there must exist a v with T[v,v,.k] > 1, and so we have found a cycle that is a currency arbitrage > 1, and we can make that trade instead.)

The table is n^3 and it takes O(1) to calculate each value for a worst case running time of O(n^3).

The proof is: assume X[a,c,k-1] stores the best we can do trading from a to c using currencies 1 to k-1 for all a and c.
To convert a to b using intermediate currencies 1,..,k, we either use currency, in which case we must convert a to k using currencies 1,...,k-1, and then convert k to b using currencies 1,...,k-1, or we do not use k, in which case we use currencies 1,...,k-1 to convert currency a to b. Because we maximize a product my maximizing each operand, we want the largest currency exchange rate, and by the induction hypothesis, X[a,c,k-1] stores the maximal exchange value for all a and c using currencies 1,...,k-1.

(A second possible solution sets X[a,b,k] being the maximum exchange rate of a to b using at most k currency exchanges. In this case, the recurrence is X[a,b,k] = max {X[a,b,k-1], max_{all c} {X[a,c,k-1] x r(c,b)}}, and the running time is O(n^4).)

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, November 25, 2019

1. As usual, begin by handing out the quiz and collect it quiz at 3:40.

2. Place the students into groups and give them these problems: We have n items, and each item has a value. Item i has value v_i. Suppose we have two people and we want to divide the items up as follows:

   (a) Each person gets as close to the same amount of total value.

   (b) One person gets the $n/4$ of items that have the most value and the other gets $3n/4$ of items with least value.

   (c) No person ends up with more that twice the total value than the other person.

   (d) Each person gets to choose what items they want, and both people end up with the same number of desired items.

   Can they figure out the correct technique/algorithm for each? Do their answers change if instead of dividing between 2 people, we divide between m?

3. See if students can figure out the technique for each, figure out the actual algorithm for each, and have an idea how to prove each. Maybe you can ask the students to present their algorithm to the class.

4. **You will not have enough time to formally prove each, so don't go into those details.** I will provide the proofs below so that can understand the algorithms, and if students ask specific questions on how the proof might look, you can sketch it for them.

5. (a) is dymanic programming if no individual item has a lot of value. The recurrence is $T[i, k]$ = true if it is possible to get exactly k value from items 1 through i. The choices are we either include item i or we do not.

   $$T[i, k] = T[i-1, k] \vee T[i-1, k-v_i]$$

   Let $V = \sum_{i=1}^{n} v_i$ The solution is to look at $T[n, j]$ where j goes from $V/2$ to 1, and stop at the first value where $T[n.j]$ is true. The proof is straighforward because we either add the item in our group or we do not. The IH gives tells us whether, if we add the item, we can choose from the remaining items to get the needed total value. The running time is $O(nV)$. This is weakly polynomial so if V is not large compared to n it is fine.

   Otherwise, this problem is NP-hard via a reduction from Subset Sum. First we turn our problem into a decision problem: Can we divide amoung 2 people such that the difference in value between the two people receive is K'? Start with a Subset Sum problem: n values $v_1, \ldots, v_k$ and goal K. If $K < V/2$, we add a bogus item that has value $V - 2K$, and if $K > V/2$ add a bogus item with total value $2K - N$. Set $K' = 0$, and divide the items as evenly as possible, if we can divide them with $K' = 0$ difference between the two groups, and if $K < V/2$, then take the group that received the bogus item. the rest of its items sums to exactly K, as we solved Subset Sum. The same argument holds for when $K > N/2$.

   What happens if we have m people instead of 2? For the NP-hardness proof, nothing! Since we are now reducing to this problem, we can set $m = 2$ and show that with $m = 2$ is NP-hard, so it will still be NP-hard for arbitrary m. For the dynamic programming solution the recurrence gets a lot bigger: $T[i, k_1, \ldots, k_m]$ = true if it is possible to partition items $1, \ldots, i$ into m sets such that set j has value k_j, and

   $$T[i, k_1, \ldots, k_m] = T[i-1, k_1 - v_i, k_2, \ldots, k_m] \vee T[i-1, k_1, k_2 - v_i, \ldots, k_m] \vee \cdots \vee T[i-1, k_1, k_2, \ldots, k_m - v_i]$$

   So the runtime is now $O(nV^{m-1})$, and it is no longer a weakly polynomial algorithm.

6. (b) is Divide and Conquer. If the students come up with a sort, it is $O(n \log n)$, and we can do better. The hint might help them remember some divide and purge ideas. The algorithm set $K = n/4$ and try to find the K largest items. Take a random item, and sort the set of items into those with smaller value

and those with larger than or equal value. If the set with larger value has more than K items, repeat on this set again looking forthe K largest value items. If it has less than K, recurse on the set with lesser value, but now look for $K-$ (size of set with greater value) and add the result of the recursive call to the other set. The worse case time is $O(n^2)$ but the average case time is $T(n) = T(3n/4) + O(n)$ which is $O(n)$. The proof is again straightfoward because we assume the algorithm correctly finds the largest K values, for any $K < n$, on sets with fewer than n elements.

What happens if we have m people instead of 2? Where each person gets some fraction of items, on person gets a small number of the most valuable, the next gets a somewhat larger number of the next most valuable, and so on? We can just repeat this process and get $O(nm)$, or if $m > \log n$, just sort first and then divide so it is $O(n \log n)$.

7. (c) Is greedy, but the proof is trickier. Sort all the items from largest to smallest. Give the largest item to the first person, and the second person gets the next largest items until the second person has more value than the first. If the second person gets all the remaining items, and still has less value than the first person, we can do a quick check to see if the first person has more than twice the value than the second. If so, then a "fair" division is impossible. If not, we do a loop. Considering the remaining items from largest to smallest value, give the next item to the person with less total value.

   The standard greedy "swap" proof does not work well because the point is not to minimize the difference between the two people. We saw in (a) that this is NP-hard. So, instead we will use a loop invariant proof. The invariant is simply that after each iteration, no person has more than twice the value of the other. Before the first iteration, the second person has more value, but has at least 2 items. If the second person has more than twice the value, than one of these items must be worth more than the item the first person has. That is impossible. Now, assume we start the loop with no person having more than twice the value of the other. We give the next item to the person with less. If that person still has less, the invaraint holds. If that person has more than twice the first, then the item added must be worth more than any item the first person had, but that is impossible.

   The runtime is $O(n \log n)$ because we had to sort.

   What if we increase the number of people to m? Now we want the person with the most to have no more than twice the person with the least. The algorithm is roughly the same, but the division of the first items is a little trickier. We again go through the items from most to least, and give each person one item. Then we loop handing out items to the person with the least until the first person now has the least. Here we stop and see if the first person has more than twice the least. If so, then a fair division is impossible. Otherwise, we continue to give the person with the least value the next item, and the same loop invariant above holds.

8. (d) This is flow, but with only two people there is also a very simple greedy algorithm. For flow, have the source connect to each person with caoacity $k = \min\{d, n/2\}$ where d is the smallest number of items any one person chose. Have each person connect to an item they desire with capacity 1. Connect each item to a sink with capacity 1. Run flow. If we get a flow of $2k$ then each person sent one flow to k items they desire, and those are the items they get. If not? Then we reduce the capacity of each source to person vertex by 1 and now run flow to see if we can get $2(k-2)$. You can be more clever about reducing the capacities, but that does not significantly change the running time. This is $O(n^3)$ (number of edges is $O(n)$ and the max flow is $\leq n/2$, and we may need to run the flow algorithm up to $n/2$ times).

   Here is the simple greedy algorithm. Give the person who wants the least all the items they want. Call that person, person 1. Give person 2 all the items left over that they want (up to the number person 1 has). Until the two people have the same number of items, take an item from person 1 that person 2 wants and give it to person 2. This is $O(n)$.

   What happens when we have m people? The greedy algorithm no longer works, but the flow algorithm can be easily adjusted by adding another person vertex and edge from source to that vertex with capacity k. Now, we see of the total flow is mk.

Subject: Quiz solution and breakout notes
From: Harold Connamacher <hsc21@case.edu>
To: Brendan Dowling <bld43@case.edu>,Jian Liu <jxl2184@case.edu>,Huixian Yang <hxy298@case.edu>,Colby Saxton <cas264@case.edu>,Seohyun Jung <sxj393@case.edu>,Aimee Chau <axc693@case.edu>,Sheng Guan <sxg967@case.edu>
Date Sent: Sun, 17 Nov 2019 10:58:53 -0500
Date Received: Sun, 17 Nov 2019 07:59:12 -0800 (PST)
Attachments: breakout10.pdf

Dear TAs,

The breakout notes are attached, and here is the quiz solution. For grading the quiz solution, give partial credit if the student attempts a dynamic programming solution that is weakly polynomial. (Reasonable algorithm technique used.) I don't see what the recurrence relation is for that, but this feels close enough to subset sum, that there *might* be one. However, be very skeptical of any algorithms sine there are a lot of things that have to be handled such as the order quantity being larger than the secret min. And so a straightforward adaptation of knapsack/subset sum's algorithm will not work.

The problem is NP-hard, and we can do a reduction from Subset Sum.

We are given a list of numbers v_1,\ldots, v_k and a target number T. For each number v_i, create a sell order with price 3, quantity v_i secret price 2, and secret min quantity v_i. Create a buy order with price 1, quantity T, secrete price 2, and secret min quantity T.

There exists a sale if and only if there is a set of numbers that sum to T. Suppose there is a sale. Since the buy price is lower than the sell price, the sale has to be at the secret price. Since the buy order is for T and the min quantity is T, we must have a set of sell orders whose secret min's sum to exactly T. Therefore, there is a set of numbers that sums to T.

Suppose we have a set of numbers that sum to exactly T. If we choose those orders, we have a set of sell orders whose secrete minimums sum to exactly T, and that matches the buy order that is requesting a minimum of T. Therefore, we have a sale.

The runtime of the reduction is linear time since we are just turning numbers into sales of the same binary value with a little extra constant sized headers.

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, November 18, 2019

1. As usual, begin by handing out the quiz and collect it quiz at 3:40.

2. Place the students into groups and give them this problem: Many programs that you enter text on (messaging apps, a word processors) compare the words you enter to a dictionary. If the word is not found, the program assumes it is a typographical error and presents the user with a list of possible alternatives. The goal is to find words that are "close" to the word the user entered. One way to define "close" is with *edit distance.*

   The edit distance between two strings $s1$ and $s2$ is the minimum number of changes that you must make to $s1$ for it to become $s2$. There are three types of changes:

   - You can remove a character of $s1$.
   - You can add a character to $s1$.
   - You can replace a character of $s1$ with a different character.

   Design an algorithm that takes two strings, $s1$ and $s2$, $s1$ had n characters and $s2$ has m characters, and gives the (minimum) edit distance between them.

3. The first thing the students should ask is if it feels like an NP-hard problem or a polyomial time problem. One thing that should jump out at them is that we are minimizing a metric and we have choices. That suggests dynamic programming.

4. The next question is whether we want the minimum subproblem from each choice. To answer that, we have to think about what is the subproblem.

5. A useful hint is to think about the change needed to match that last character of $s1$ to the last character of $s2$.

6. They should hopefully start to realize that there are three choices, and depending on the choice, we have a different subproblem. That subproblem can be solved optimally, and we then do the last step to match the strings.

7. The next step is to use that insight to create the recurrence relation. What information do we need to know at each step? Which prefix of $s1$ we need to match to which prefix of $s2$.

8. Here is a possible solution and proof:

   Create a table $T[i,j]$ to store the minimum edit distance between the first i characters of $s1$ and the first j characters of $s2$.

   $T[i,j] = \min\{T[i-1,j-1] + c(i,j), T[i-1,j]+1, T[i,j-1]+1\}$ where $c(i,j)$ is 0 if $s1[i] = s2[j]$ and 1 if $s1[i] \neq s2[j]$. We initialize the table with $T[0,0] = 0$.

   Proof by induction. The base case is trivial.

   Assume $T[i-1,j-1]$ stores the minimum edit distance between the first $i-1$ characters of $s1$ and the first $j-1$ characters of $s2$.
   Assume $T[i,j-1]$ stores the minimum edit distance between the first i characters of $s1$ and the first $j-1$ characters of $s2$.
   Assume $T[i-1,j]$ stores the minimum edit distance between the first $i-1$ characters of $s1$ and the first j characters of $s2$.

   Consider $T[i,j]$. We have four choices for our "last" conversion to get $s1[1..i]$ to be equal to $s2[1..j]$

   - We match the ith character of $s1$ with the jth character of $s2$ (only if they are the same). In this case, we had to previously convert the $i-1$ characters of $s1$ to equal the $j-1$ characters of $s2$.

- We replace the ith character of $s1$ with the jth character of $s2$. In this case, we had to previously convert the $i-1$ characters of $s1$ to equal the $j-1$ characeters of $s2$.
- We remove the ith character of $s1$. In this case, we had to previously convert the $i-1$ characters of $s1$ to the j characters of $s2$.
- We add the jth character of $s2$ to $s1$. In this case, we had to previously conver the first i characters of $s1$ to the first $j-1$ characters of $s2$.

For each of these choices, it does not matter how we match the resulting prefixes because those choices do not affect our last choice. Therefore, to get the shortest edit distance, we should match them optimally. By the induction hypotheses, $T[i-1, j-1]$, $T[i, j-1]$, and $T[i-1, j]$ store the minimum edit distance for each of the possible prefix cases above. That completes the proof.

How do we get the solution? For this proboem, we only want the minimum edit distance between $s1$ and $s2$, and that will be the value stored in $T[n, m]$

The running time is $\Theta(nm)$ because we have to fill a $n \times m$ table, and it takes constant time for each entry.

Subject: Homework solution and rubric
From: Harold Connamacher <hsc21@case.edu>
To: Sheng Guan <sxg967@case.edu>,Jian Liu <jxl2184@case.edu>,Aimee Chau <axc693@case.edu>,Brendan Dowling <bld43@case.edu>,Seohyun Jung <sxj393@case.edu>,Huixian Yang <hxy298@case.edu>,Colby Saxton <cas264@case.edu>
Date Sent: Fri, 15 Nov 2019 18:37:51 -0500
Date Received: Fri, 15 Nov 2019 15:38:09 -0800 (PST)

For this homework, we are using both rubrics from the previous homeworks.

For each question where the student uses a reduction, the student's answer should have the following:
1. A conversion of one problem to another instead of directly trying to solve the original problem.
2. The reduction needs to go in the correct direction.
3. A correct reduction.
4. An "if-and-only-if" proof structure with reasonable rigor. For example, the proof must have clear assumptions.
5. The "if" direction is correct.
6. The "only-if" direction is correct.
7. A correct running time for their reduction (or reduction + algorithm) is given.
8. A good justification of their running time.

Here is the grade to assign:
8: Perfect. All 8 above achieved.
7: Very good. All 8 points attempted, and at least 6 of the 8 points achieved.
6: Good. Achieved at least 1, 2, 4 and either 5 or 6.
5: Acceptable. At least two of 1, 2, 4, 5 and 6.
4: Poor. Attempted points 1 and 4.
3: Poor-. There is some attempt at doing a reduction.
2: There some attempt at an algorithm but not a reduction.
0: There is nothing useful to grade.

For each question where the student is giving an algorithm directly, the student's answer should have the following:
1. The algorithm is precise and clear.
2. The algorithm used the correct technique (greedy/DP/divide and conquer) for the problem. It is not enough to say they are using a certain technique. It is very common for a student to say they are using DP, but the algorithm recurrence is actually a greedy algorithm.
3. The algorithm is correct.
4. They provide a proof that has reasonable rigor. For example, there are clear assumptions made. The biggest mistake is for students to just describe what their algorithms do and call that a "proof".
5. They use a proof appropriate for the algorithm type they are trying to write (not necessarily for the correct algorithm).
6. The proof is correct.
7. The correct runtime for their algorithm (not for the correct algorithm) is given.
8. A reasonable justification of the running time is given.

And here is the grade to assign:
8: Perfect. All 8 above achieved.
7: Very good. All 8 points attempted, and at least 6 of the 8 points achieved.
6: Good. Achieved at least 1, 2, 4 and 5.
5: Acceptable. At least two of 1, 2, 4, and 5.
4: Poor. Attempted points 1 and 4.
3: Poor-. Attempted either point 1 or 4.

2: There some attempt at an algorithm.
0: There is nothing useful to grade.

Problem 1:
Suppose that the tree of least cost paths from h to all other vertices matches an optimal tree for the first k vertices, in the order our algorithm added them, but then the optimal algorithm uses a different path to vertex k+1. Let x be the vertex immediately preceding vertex (k+1) on the optimal path and y be the vertex immediately preceding (k+1) on the greedy path. Either x < k+1 or x > k+1 (ordered by how they were added to V by greedy). If x < k+1, then by the induction hypothesis, L[x] = the optimal time we can reach O, and then greedy considered L[x] + f(x,k+1) and L[y] + f(y, k+1) and found that L[y] + f(y, k+1) <= L[x] + f(x, k+1). Thus we can change O to have y precede k+1 instead of x, and the time to k+1 can only decrease, and thus the time to any vertex after k+1 also can only decrease. Suppose x > k+1. Then there are two vertices x' and x'' with x' < k+1 and x'' > k+1 where the edge (x',x'') is on the optimal path to k+1. By the same argument as above, we know L[y]+f(y,k+1) <= L[x'] + f(x',x''). By the induction hypothesis, L[x'] is the optimal time to x', and so changing O so the optimal path to k+1 goes from y can only decrease the time to k+1, and thus also only decrease the time to any vertex reached from k+1.

Problem 2:
To prove the problem is NP-hard we do a reduction from Subset Sum.

Given a set of n numbers, {v_1, ..., v_n} and a desired sum S, create a transportation network as follows. For number v_i, create vertices s_i, x_i, y_i, t_i and edges (s_i, x_i), (s_i, y_i), (x_i, t_i), and (y_i, t_i). Each edge has cost and length equal to 1 except that (s_i, x_i) has length v_i + 1 and (s_i,y_i) has cost v_i + 1. We create the full network by having s = s_1, t_i = s_{i+1}, and t_n = t. We set C = 2n + S and we let L be 2n - S + (v_1 + ... + v_n) (any L that is at least this amount will work).

Suppose there exists a path from s to t with total cost C and total length L. Each time the path entered a vertex s_i, it had to choose between going to y_i or going to x_i. If it goes to y_i, we add v_i to our subset Y, and if it goes to x_i we do not. Every path from s to t is exactly 2n edges. Thus the sum of all v_i in Y is C - 2n = S, and we have a solution to Subset Sum.

Suppose we have a subset of {v_1, ..., v_n} that sums to S. Then we choose a path in the network. Each time we get to s_i for some element v_i, if v_i is in the subset, we go to y_i, and if it is not, we go to x_i. The result is a path where C = 2n + S and L = 2n - S + (v_1 + ... + v_n). We have a solution to the path problem.

Problem 3:
Sort the edges of G by bandwidth so that b(e_1) >= b(e_2) >= .... Consider the edges in this order and add the edges to the graph until a and b are in the same component. This algorithm costs O(m log m) to sort the m edges, and we will do a union-find data structure to keep track of the components. This is O(log n) (or O(log* n)) per edge addition. Thus the total running time is O(m log m) = O(m log n) and the answer is the bandwidth of the last edge added.

Suppose there is an optimal path O between a and b that has greater bandwidth than what we get from the algorithm. Let x be this bandwidth. This is a contradiction because the algorithm considered the edges in decreasing bandwidth, and when it added all the edges with bandwidth x or greater in G, a and b were not connected.

Problem 4:
A simple reduction to Dijkstra.
For each edge, if the edge is compromised, give it weight 1 and if not compromised give it

weight 0. Use Dijkstra to find the shortest path from s to t. The run time of Dijkstra is O(m log n) and the reduction is O(m) to give O(m log n).

The path uses K compromised edges if and only if the cost of the shortest s - t path in the new graph is K. Assume the path has K compromised edges, then it used K edges of weight 1 and the rest of weight 0. Assume the shortest path is length K, then it used K edges of weight 1, and these are the K compromised edges.

Problem 5:
Create a single source and connect to all the supply nodes and a single sink and connect the distribution nodes to it. Give every edge infinite capacity and every node (except the source and sink) capacity 1. I showed them the conversion of node capacities to normal flow on Wednesday, so they can do this conversion so now we have a graph where the "node" edges are the only edges with capacity 1 and the rest of the edges have capacity infinite. Run FF and find the minimum cut by doing a DFS/BFS from the source on the residual graph. Since the maximum flow is n (assume n vertices and m edges), the run time is O(nm).

Assume we have a flow of size K. By the max flow/min cut theorem, there are K edges we can cut, and they must be "vertex" edges. So if we remove those vertices from the original graph, we separate the source nodes from the distribution nodes. (Technically we separate the source from the sink, but the cut can't happen at the source or sink, and so it is either an internal node or a supply/distribution node, but that will still separate the remaining supply nodes from the distribution nodes).

Assume we can separate the graph by removing K nodes. Then in the modified graph, we can also separate the graph by removing those K "vertex" edges. Since each "vertex" edge has capacity 1, there must be a cut of size K.

Problem 6:
(For this problem, the proof points need to cover both parts of the proof.)
Let a and b be at distance > n/2, but assume removing no single vertex separates a from b. Therefore, there exists two vertex disjoint paths from a to b. Both paths cannot be > n/2 length, so this contradicts the claim that the distance from a to b is > n/2.

The flow solution is to give every vertex capacity 1 and every edge capacity infinity and to find the minimum cut. The proof is similar to problem 5. Since the max flow is 1, this is an O(m) algorithm where m is the number of edges.

Here is a DFS solution. Run DFS from a. Let v be a vertex on the DFS path from a to b (and neither a nor b). Removing v separates a from b if and only if there is no path from a to b that avoids v. All non-DFS tree edges can only go from a vertex to a direct ancester of that vertex in the tree. So, any a->b path that avoids v must use an edge from the DFS path between a and v to a vertex that is either on the DFS path between v and b or in the subtree rooted at v. Label the vertices from a to b on the DFS where a = 1, the child of a = 2, and so on, until you get to b, and then the DFS tree below b can be labelled arbitrarily. Loop through the labelled vertices in decreasing order starting with the highest labelled vertex. For each vertex w, let shortcut[w] = the lowest label vertex w can reach either directly or going through a vertex with a higher label than w.

```
    directlyreach[w]  = min {a} where (w,a) is an edge
    indirectlyreach[w] = min {shortcut[a]} where (w,a) is an edge and w < a
    canreach[w] = min {directlyreach[w], indirectlyreach[w]}
```

Claim: v separates a from b if and only if for the vertex v+1, canreach[v+1] = v.
Proof: Assume v separates a from b, then the child of v on the path to v cannot be able to reach a vertex above v, so it must have it's "canreach" value = v. Suppose canreach[v+1] = v. Then for suppose there is a path a -> x -> y -> b where x < v and y > v. Then canreach[v+1] <=

canreach[y] < v.

The DFS is O(m), labelling the path is O(n), and calculating canreach considers each edge at most twice, and traversing the path to find a vertex with v = canreach[v+1] is O(n), so the total runtime is O(n).

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

Subject: Breakout notes and quiz solution
From: Harold Connamacher <hsc21@case.edu>
To: Colby Saxton <cas264@case.edu>,Brendan Dowling <bld43@case.edu>,Seohyun Jung <sxj393@case.edu>,Jian Liu <jxl2184@case.edu>,Aimee Chau <axc693@case.edu>,Sheng Guan <sxg967@case.edu>,Huixian Yang <hxy298@case.edu>
Date Sent: Sat, 9 Nov 2019 21:02:27 -0500
Date Received: Sat, 9 Nov 2019 18:02:46 -0800 (PST)
Attachments: breakout9.pdf

Attached are the notes for Monday, and here is the quiz solution:

Create a single source and connect to all the supply nodes and a single sink and connect the distribution nodes to it. Give every edge infinite capacity and every node (except the source and sink) capacity 1. I showed them the conversion of node capacities to normal flow on Wednesday, so they can do this conversion so now we have a graph where the "node" edges are the only edges with capacity 1 and the rest of the edges have capacity infinite. Run FF and find the minimum cut by doing a DFS/BFS from the source on the residual graph. Since the maximum flow is n (assume n vertices and m edges), the run time is O(nm).

Assume we have a flow of size K. By the max flow/min cut theorem, there are K edges we can cut, and they must be "vertex" edges. So if we remove those vertices from the original graph, we separate the source nodes from the distribution nodes. (Technically we separate the source from the sink, but the cut can't happen at the source or sink, and so it is either an internal node or a supply/distribution node, but that will still separate the remaining supply nodes from the distribution nodes).

Assume we can separate the graph by removing K nodes. Then in the modified graph, we can also separate the graph by removing those K "vertex" edges. Since each "vertex" edge has capacity 1, there must be a cut of size K.

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, November 11, 2019

1. As usual, begin by handing out the quiz and collect it quiz at 3:40.

2. Break the students into groups of 2 or 3 and assign them this problem: We want to identify if a graph has any cycles of odd length. That is, a path that starts at a vertex v and returns to that vertex visiting no vertex more than once, and the number of vertices on the cycle is odd. (You can motivate this by telling them that many NP-hard problems like Vertex Color, Independent Set, and Coloring, can be solved in polynomial time if the graph has no odd length cycles.)

3. Let them think for a minute, and then give them the hint that we can reduce this problem to Breadth First Search. Here is the BFS algorithm we did in lecture:

```
BFS(graph G)
1  mark all vertices as unvisited
2  set P[v] = null for all vertices v
3  for each v in G
4      if v is unvisited
5          BFS-Tree(G, v)

BFS-Tree(graph G, vertex s)
6  Q: an empty queue
7  mark s as visited
8  add s to Q
9  while Q is not empty
10     v → remove from Q
11     for each vertex w that is a neighbor of v
12         if w is unvisited then
13             mark w as visited
14             set P[w] = v
15             add w to the Q
```

4. The first idea they may come up with is to do a BFS from each vertex until it returns to the vertex. However, this will only give the smallest cycle. There could be an odd one that is longer.

5. A key point they should realize quickly (maybe with your help) is that any edge that is not part of the BFS tree must form a cycle. Where in the algorithm do we find a non-BFS tree edge? At line 12 if w is already visited. (They might thing that they can just then see how big the cycle with that edge is, but that still takes too long: up to m non-tree edges, and tracing the cycle is a minimum of $O(n)$. We want a $O(m)$ algorithm total.) You can try giving a simple graph of say 7 vertices, and trace a BFS. They may notice a pattern on how the non-tree edges connect in the tree.

6. Here is the proof you should help them to: Run a BFS, and keep track of the "level" of each vertex in the BFS tree. (Let L[v] = the level of the vertex, start L[s] = 0, and each time a vertex w is marked as visited, set L[w] = L[v]+1.) There are three possibilities

   - The non-tree edge connects vertices on the same tier
   - The non-tree edge connects vertices on adjacent tiers
   - The non-tree edge connects vertices on tiers further than 1 away

   The third is impossible. If vertex u connects to v more than 1 tier below, v would have been added when u was visited.

If a non-tree edge connects two vertices of the same tier we have an odd length cycle. Let the edge connect a and b, and let x be the common parent of a and b in the BFS-tree. Let k be the length of the path from x to a, and likewise from x to b, and we have a cycle of size $2k + 1$.

If a non-tree edge connects a and b on adjacent tiers, let x be the common parent, and let k be the length from x to a, and $k + 1$ the length from x to b, and we have a cycle of $2k + 2$.

Subject: Breakout notes and quiz solution
From: Harold Connamacher <hsc21@case.edu>
To: Colby Saxton <cas264@case.edu>,Aimee Chau <axc693@case.edu>,Brendan Dowling <bld43@case.edu>,Huixian Yang <hxy298@case.edu>,Seohyun Jung <sxj393@case.edu>,Jian Liu <jxl2184@case.edu>,Sheng Guan <sxg967@case.edu>
Date Sent: Sun, 3 Nov 2019 18:00:53 -0500
Date Received: Sun, 3 Nov 2019 15:01:13 -0800 (PST)
Attachments: breakout8.pdf

Dear TAs,

Attached are the breakout notes. Here are some solutions:

1) A simple reduction to Dijkstra.
For each edge, if the edge is compromised, give it weight 1 and if not compromised give it weight 0. Use Dijkstra to find the shortest path from s to t. The run time of Dijkstra is O(m log n) and the reduction is O(m) to give O(m log n).

The path uses K compromised edges if and only if the cost of the shortest s - t path in the new graph is K. Assume the path has K compromised edges, then it used K edges of weight 1 and the rest of weight 0. Assume the shortest path is length K, then it used K edges of weight 1, and these are the K compromised edges.

(Technically this solution also requires them to show that Dijkstra works with 0 cost edges. In class we claimed the edge costs had to be positive, but you can ignore that issue.)

2) The same idea but more rigorous.
Let W = m.
Any edge that is compromised gets weight W while an uncompromised edge gets weight 1. Run Dijkstra. Since summing the weights is O(log m) which is the same as the cost of maintaining the sorted costs, so the run time is still O(m log n). The path from s-t uses K compromised edges if and only if the length of the s-t path is greater than KW and less than (K+1)W. The rest of the proof is identical to above.

3) Or they do a greedy algorithm/proof based on Dijkstra.
Let L[v] = infinity where L[v] will store the number of compromised edges on the best path found so far. Let P[v] = null where P[v] stores the previous vertex to v on the path from s. Set L[s] = 0 and T = empty.
Until t is added to T, get the vertex v not in T with smallest L value. That is the least compromised path to v. For each vertex a adjacent to v, if edge (a,v) is not compromised and L[a] > L[v] then set L[a] = L[v] and P[a] = v. If edge (a,v) is compromised and L[a] > L[v] + 1 then set L[a] = L[v] + 1 and P[a] = v.

Now, the proof is almost identical to the one from last week. Here is how the swap version should start:

Assume we have an optimal solution T* that gives the least compromised path from s to all vertices. Suppose our algorithm matches T* on the path to the first
k vertices added to T. Now we find a different path that T* to vertex k+1. (And the rest follows the proof from last week.)

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, November 4, 2019

1. As usual, please give the quiz at the start of the breakout session, and end it at 3:40.
2. Place the students into groups and give them this problem:

   We are given a graph with costs on the edge, and now we want a *maximum* spanning tree. That is a spanning tree (a tree that includes every vertex of the graph) where the sum of the costs of the edges of the tree is as large as possible.
3. The already saw Kruskal's algorithm for minimum spanning tree last week. The question is how does changing from minimizing to maximizing change the problem? Can we still find a polynomial time algorithm or is it now NP-complete?
4. Here are two solutions to help guide them to. The first is a direct greedy algorithm and the second is a reduction.
5. We can adopt Kruskal's algorithm to this problem: Start with a empty set T. Sort the edges by cost so that $c(e_1) \geq c(e_2) \geq \ldots \geq c(e_m)$. Consider the edges in order and add each edge to T as long as adding the edge does not produce a cycle in T. Tbe run time will be the same as Kruskal's: $O(m \log n)$.
6. The similarity of this algorithm and Kruskal's suggests that we should be able to reduce the problem to Kruskal's. See if they can find the reduction.
7. Here it is: We need to convert largest cost edges to smallest cost edges. Find a value W that is greater than or equal to the largest cost of any edge of G. Then create a "new" graph G' that is the same as G but for each edge e, if $c(e)$ is the cost of edge e in G, $W - c(e)$ is the cost of edge e in G'. Now run Kruskal and return the tree found. The running time of the conversion is $\Theta(m)$ and so the total running time is $O(m \log m)$.

   Proof: Prove that G has a spanning tree with cost greater than or equal to K if and only if G' has a spanning tree with cost less than or equal to $(n-1)W - K$. Assume G' has a spanning tree with cost less than or equal to $(n-1)W - K$.

   $$\begin{aligned} cost_G(T) &= \sum_{e \in T} c(e) = (n-1)W + \sum_{e \in T}(c(e) - W) = (n-1)W - \sum_{e \in T}(W - c(e)) \\ &= (n-1)W - cost_{G'}(T) <= (n-1)W - ((n-1)W - K) = K \end{aligned}$$

   Assume G has a spanning tree with cost greater than or equal to K.

   $$cost_{G'}(T) = \sum_{e \in T}(W - c(e)) = (n-1)W - \sum_{e \in T} c(e) = (n-1)W - cost_G(T) >= (n-1)W - K$$
8. Here is a proof of the direct algorithm if you have time at the end: Assume there is an optimal tree T^*. Assume our algorithm matches T^* on the first k edges considered by our algorithm. Assume our algorithm makes a different choice on edge $k+1$. The only possibility is that we add edge $k+1$ while it is not in T^*. Consider $T^* + e_{k+1}$. It must have a cycle. There must exist some edge on that cycle that is not in the first $k+1$ edges we considered. Let that edge be e_x. By our greedy choice, $c(e_{k+1}) \geq c(e_x)$. So consider $T^{**} = T^* - e_{k+1} + e_x$. The cost of T^{**} is greater than or equal to the cost of T^*. We must now prove that T^{**} is still a tree. Consider arbitrary vertices a and b. If the path from a to b in T^* did not use edge e_x, that path still exists in T^{**}. If the path does use edge e_x, then we can still find a path from a to b in T^{**} by first going from a to the first endpoint of e_x, then around the cycle created by adding e_{k+1} to the other end point of e_x, and from there to b.

Subject: Quiz solution
From: Harold Connamacher <hsc21@case.edu>
To: Sheng Guan <sxg967@case.edu>,Jian Liu <jxl2184@case.edu>,Brendan Dowling <bld43@case.edu>,Huixian Yang <hxy298@case.edu>,Colby Saxton <cas264@case.edu>,Seohyun Jung <sxj393@case.edu>,Aimee Chau <axc693@case.edu>
Date Sent: Mon, 28 Oct 2019 15:47:38 -0400
Date Received: Mon, 28 Oct 2019 12:47:55 -0700 (PDT)

Let V = emptyset, and for each vertex v of G, let L[v] = infinity for v != h and L[h] = t_0. Let P[v] = null for all v.

```
Repeat :
   Let v be the vertex of V(G) - V that has the smallest L value.
   Add v to V.
   For each edge (v,a) of G, set L[a] = min{L[a], L[v] + f_{v,a}(L[v])}, and if this changed L[a], we set P[a] = v.
Until (v = w) or until all vertices chosen.
```

We can get the path by following P[w], P[P[w]], P[P[P[w]]], ..., h.

The running time of the algorithm is identical to Dijkstra's. Every edge is considered at most twice, and we have to maintain a sorted list of L. We can use a heap for that and so O(m log n).

Proof 1: (Greedy stays ahead)
We prove that each time we add a vertex v to V, then L[v] is the earliest/least cost we can reach v from h when starting at time t_0.

Let O[v] be the true earliest we can reach v and order the vertices so that O[v_1] <= O[v_2] <= ....

Consider the first vertex in this ordering where O[i] < T[i]. The optimal path to i reaches i by travelling an edge (j,i). Since O[j] < O[i], we know by our assumption that T[j] = O[j]. However, that means that j was added to V before i, and when we added j, we set L[i] <= L[j] + f_{j,i}(L[j]) = O[i]. This contradicts the assumption that O[i] < L[i].

Proof 2: (Swap with optimal)
Suppose that the tree of least cost paths from h to all other vertices matches an optimal tree for the first k vertices, in the order our algorithm added them, but then the optimal algorithm uses a different path to vertex k+1. Let x be the vertex immediately preceding vertex (k+1) on the optimal path and y be the vertex immediately preceding (k+1) on the greedy path. Either x < k+1 or x > k+1 (ordered by how they were added to V by greedy). If x < k+1, then by the induction hypothesis, L[x] = the optimal time we can reach O, and then greedy considered L[x] + f(x,k+1) and L[y] + f(y, k+1) and found that L[y] + f(y, k+1) <= L[x] + f(x, k+1). Thus we can change O to have y precede k+1 instead of x, and the time to k+1 can only decrease, and thus the time to any vertex after k+1 also can only decrease. Suppose x > k+1. Then there are two vertices x' and x'' with x' < k+1 and x'' > k+1 where the edge (x',x'') is on the optimal path to k+1. By the same argument as above, we know L[y]+f(y,k+1) <= L[x'] + f(x',x''). By the induction hypothesis, L[x'] is the optimal time to x', and so changing O so the optimal path to k+1 goes from y can only decrease the time to k+1, and thus also only decrease the time to any vertex reached from k+1.

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

Subject: Re: Breakout notes for tomorrow
From: Sheng Guan <sxg967@case.edu>
To: Harold Connamacher <hsc21@case.edu>
Cc: Jian Liu <jxl2184@case.edu>, Aimee Chau <axc693@case.edu>, Huixian Yang <hxy298@case.edu>, Colby Saxton <cas264@case.edu>, Brendan Dowling <bld43@case.edu>, Seohyun Jung <sxj393@case.edu>
Date Sent: Mon, 28 Oct 2019 14:12:38 -0400
Date Received: Mon, 28 Oct 2019 11:12:23 -0700 (PDT)

Hi Professor Harold:

Can you send the quiz solution to us? I believe I will be asked several questions after I read the problem.

Best,
Sheng

On Sun, Oct 27, 2019 at 11:59 AM Harold Connamacher <hsc21@case.edu> wrote:

> Hello TAs,
>
> Here are the breakout notes for tomorrow. You can pick the quizzes up from me between 12-2 tomorrow, or after my 132 class ends (3:05). I will send the quiz solution and rubric later. Also, you will have a slightly larger group this week because Colby is out of town.
>
> Regards,
>
> Harold Connamacher
> Associate Professor
> Computer and Data Sciences
> Case Western Reserve University

Subject: Breakout notes for tomorrow
From: Harold Connamacher <hsc21@case.edu>
To: Jian Liu <jxl2184@case.edu>,Aimee Chau <axc693@case.edu>,Huixian Yang <hxy298@case.edu>,Sheng Guan <sxg967@case.edu>,Colby Saxton <cas264@case.edu>,Brendan Dowling <bld43@case.edu>,Seohyun Jung <sxj393@case.edu>
Date Sent: Sun, 27 Oct 2019 11:59:01 -0400
Date Received: Sun, 27 Oct 2019 08:59:19 -0700 (PDT)
Attachments: breakout7.pdf

Hello TAs,

Here are the breakout notes for tomorrow. You can pick the quizzes up from me between 12-2 tomorrow, or after my 132 class ends (3:05). I will send the quiz solution and rubric later. Also, you will have a slightly larger group this week because Colby is out of town.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, October 28, 2019

1. As usual, please give the quiz at the start of the breakout session, and end it at 3:40.

2. Place the students into groups and give them this problem:

   Vertex Cover is a graph G and a number K. We want to find K vertices such that every edge is connected to one of the vertices. (Ex: Can we place K 360-degree cameras at different vertices and guarantee that each edge is being watched?)

   Dominating Set is a graph G and a number K. We want to find K vertices such that every vertex is either one of the K or connected to one of the K. (Ex: Can we place K cell towers at different vertices and guarantee that every vertex is within one edge of a tower?)

   Reduce the Vertex Cover problem to the Dominating Set problem.

3. You will definitely need to give some hints as they work to help them. Hint one after they have a few minutes to understand the problem: they are given a G and K for the Vertex Cover problem and have to convert it to a new G' and K' for the Dominating Set problem.

4. The next hint: they need to be creating a new graph for the Dominating Set problem. They cannot use the same graph of the Vertex Cover problem. They might be able to use the same K, but they should not assume that. Thee is a good chance that they need to change the K as well.

5. The next hint: what should the new graph look like? We need to convert a "covering" of vertices into a "covering" of edges. So there probably needs to be a vertex of the new graph for each edge of the original graph as well as for each vertex of the original graph.

6. The next hint is a *key point*: They do not get to decide how the *Dominating Set* algorithm finds the solution. For example, they can't force it to pick only vertices of the new graph that correspond to vertices of the old graph. Instead, they must either create the graph so a hitting set of size K' can never use an "edge" vertex or to allow it to pick "edge" vertices but still be able to convert that choice back to the vertex cover of the original graph.

7. With about 10-15 minutes left, depending on how they are doing, give them a conversion. If you see a good conversion created by the students, use that one. Otherwise, give them this one:

   We are given G and K for Vertex Cover. Let G have n vertices and m edges. We create a G' with $n + m$ vertices and $3m$ edges. For each vertex a of G, we create a vertex a of G'. For each edge ab of G we create a vertex v_{ab} of G'. For each edge ab of G, we create three edges for G': ab, av_{ab}, and bv_{ab}. We set $K' = K$ and ask if G' has a dominating set of size K'.

   Ask them to try to do the "if and only if" proof to show that there exists a dominating set of size K on graph G' if and only if there exists a vertex cover of size K on the original graph G.

8. Here is the solution:

   Assume G' has a dominating set of size K. If any vertex v_{ab} is part of the dominating set, remove v_{ab} from the dominating set and add either a or b. Since v_{ab} only connects to a and b, and a connects to both b and v_{ab} (as well as other vertices), we can make this change and we still have a dominating set. Every edge vertex v_{ab} is connected to one of the K vertices, and thus the same K vertices are incident to every edge of G.

   Assume G has a vertex cover of size K. First note that every vertex not in the K must be connected to one of the K vertices. Since every edge is covered, those same vertices in G' will be connected to all of the edge vertices v_{ab}. Thus every vertex of G' is dominated by those K vertices.

Subject: Homework rubric and solutions
From: Harold Connamacher <hsc21@case.edu>
To: Seohyun Jung <sxj393@case.edu>,Brendan Dowling <bld43@case.edu>,Aimee Chau <axc693@case.edu>,Colby Saxton <cas264@case.edu>,Huixian Yang <hxy298@case.edu>,Jian Liu <jxl2184@case.edu>,Sheng Guan <sxg967@case.edu>
Date Sent: Fri, 25 Oct 2019 17:52:13 -0400
Date Received: Fri, 25 Oct 2019 14:52:31 -0700 (PDT)

For each question, the student's answers should have the following:
1. A conversion of one problem to another instead of directly trying to solve the original problem.
2. The reduction needs to go in the correct direction.
3. A correct reduction.
4. An "if-and-only-if" proof structure with reasonable rigor. For example, the proof must have clear assumptions.
5. The "if" direction is correct.
6. The "only-if" direction is correct.
7. A correct running time for their reduction (or reduction + algorithm) is given.
8. A good justification of their running time.

Here is the grade to assign:

8: Perfect. All 8 above achieved.
7: Very good. All 8 points attempted, and at least 6 of the 8 points achieved.
6: Good. Achieved at least 1, 2, 4 and either 5 or 6.
5: Acceptable. At least two of 1, 2, 4, 5 and 6.
4: Poor. Attempted points 1 and 4.
3: Poor-. There is some attempt at doing a reduction.
2: There some attempt at an algorithm but not a reduction.
0: There is nothing useful to grade.

Problem 1:
We reduce the airplane problem to the Star Wars problem. Each airplane will be come a transmission interval. Plane i with arrival time t_i and flight limit d_i will become interval [t_i, t_i + d_i], or more correctly: (t_i + d_i)/2 +/- d_i / 2. Each time at which you can land an airplane will become a reception time: x_i = 3i. The reduction is linear time.

Now we prove that there is a transmission match if and only if we can land every airplane.

(->) If there is a transmission match, then we can place each x_i into some interval t_j +/- epsilon_j. That means landing time x_i falls into the interval [t_j, t_j + d_j] in which plane j will be at our airport and able to land. So every plane gets a landing time.

(<-) For the other direction, assume there is a way to land every plane. That means for each plane j, we landed it at some time X, and X must be between when the plane arrived at the airport t_j and when it ran out of fuel t_j + d_j. Therefore, we matched transmission X/3 uniquely with interval [t_j, t_j + d_j]

Problem 2:
They have to convert Hamiltoninan Cycle to Longest Path.

Given G, create G' by taking G and adding three new vertices. New vertex a is connected to some arbitrary vertex v of G. New vertex b is connected to every neighbor of a in G, and new vertex z is connected to b. We set K to n+2 and ask if there is a path of n+2 edges in G'. The reduction is clearly linear time since we add three vertices and at most n edges.

(->) Assume G has a Hamiltonian cycle, then G' has a path from a to v, the cycle will return to v from some vertex u, but u is connected to b and then b to z. That gives a path from a to z that is 1 edge (a,v), n-1 edges (v to u), and 2 edges (u to b to z).

(<-) Assume G' has a path of n+2 vertices. Since there are only n+3 vertices in G', the path must hit every vertex, and since a and z are degree 1, the path must start at a and end at z. Therefore, there is a path from a to v to some vertex u to b to c, and we know v to u hits every vertex and there is an edge from u to v. Adding that edge gives us a cycle of n vertices in the original G.

Problem 3:
Given n numbers and a bound T, create n objects. For number x, create an object with weight x and value x. Make the knapsack capacity C=T and the bound B=T. The running time of this reduction is linear in the size of the input (we make one object for each item, and we are using the same numeric values).

(->) Suppose we have a solution to knapsack (a subset of objects whose total value is >= B and total weight is <= C. Because the sum of the values of the objects <= C = T$and the sum of the weights of the objects >= B = T, we know that the sum of the values/weights of the objects = T. Thus using those same values for the numbers of the subset gives a subset that sums to T.

(<-) Suppose we have a subset of numbers that sums to T. Choose those same objects for the knapsack, and their value is T >= B and the weight is T <= C.

Problem 4:
Create a network with m+n+2 nodes. There will be nodes s and t. For each of the n regions, create a node. For region i, let node n_i have an edge from s with capacity N_i. Create m nodes for each of the safe places. For safe place j, let node m_j have an edge to t with capacity M_j. For each region node n_i and safe space node m_j, create an edge from n_i to m_j if and only if we can get from region i to safe space j in under 24 hours. Give that edge infinite capacity. Run flow, and the flow from edge i to edge j is the number of people to send from region i to safe space j. The run time is O(nm) to create the network (n+m+2 vertices and at most n+m+nm edges), and by Edmunds-Karp, the flow will take O((n+m)(nm)^2) for a total running time of O(n^2m + nm^2).

(->) Suppose we can evacuate K people safely. Then we can create a flow of size K. Send a flow from s to n_i equal to the number of people leaving that region. Since that number is at most N_i, the flow is under the edge capacity. Send the flow from i to j equal to the number of people who go from region i to safe space j. The capacity of that edge is infinite. The total flow out of node i will equal to flow in. The total flow into node m_j will equal the number of people arriving, and send that flow on to the sink. Since that number is under M_j, it is less than that edge capacity.

(<-) Suppose we have a flow of size K. Then for each edge from n_i to m_j that has flow on it, say that flow is x, send x people from region i to safe space j. Since the total flow into the sink (and thus into the safe space nodes) is equal to K, we successfully moved K people into the safe spaces.

Problem 5:
Using the straight line route and distance d_1, and for each cell tower location t_i, calculate a_i and b_i where a_i is the first point at which cell 1 enters the range of tower i and b_i is the point on the route at which cell 1 leaves the range of tower 1. This can be done in time O(n), Now, create a network. For each cell and each tower create a vertex. Have an edge from cell vertex i

to tower vertex j if cell i is in the range of tower j, and give that edge weight 1. Have a vertex start and an edge of capacity 1 from start to each cell vertex, and have a vertex sink and an edge of capacity 1 from each tower vertex to sink. Now run flow. Since the maximum flow is n and since there are at most n^2 edges, by Ford Fulkerson, this flow takes O(n^3) time. Now, move the cell 1. Suppose cell 1 is connected to tower x, it can stay connected to tower x until it reaches point b_x. Now, we remove that 1 flow from source to 1 to tower x to sink, and we change the edges from cell 1 to the towers based on the towers it can now reach. We keep the rest of the graph and flow the same (so we have a flow of n-1). Now we see if we can adjust to reconnect cell 1. This requires a single iteration of the augmenting path to go from a flow of size (n-1) to a flow of size n, and so is done in O(n^2) time. We repeat this process until cell 1 reaches its destination. If there ever is a situation where we can't find an augmenting path to get a flow of n, it is impossible to connect all the cells to the towers. We have to readjust the flow at most n times for a total run time of O(n) + O(n^3) + O(n^3) = O(n^3).

(->) Suppose at each step it is possible to connect all the cells to the towers, then we can create a flow of size n at each point of the path by sending 1 unit of flow to each cell, and that cell sending the unit to the tower it is connected to and from there to the sink.

(<-) Suppose we were able to find a flow of size n at each of the b_x points on the path. Then there is 1 unit of flow going to each cell node and 1 unit from that node to a tower node. Each tower node receives only 1 unit of flow, and so we connect that cell to that tower. Each cell is now connected, and each tower gets 1 cell. We can keep this connection until cell 1 leaves the range of the tower it is connected to, and that only happens at the b_k points.

Problem 6:
Here is a reduction from Vertex Cover. Given G with n vertices and m edges and K as an instance of Vertex Cover, we create a rumor instance as follows. Create n+m actors, one for each vertex and edge of the graph. An actor j follows i if and only if edge j is incident to vertex i. We set k=K and b=m+k. Set p(a) = 1/2 for every actor a. The construction takes O(n+m) time.

(->) If there is a solution to the rumor problem, then we create a set S of the vertices G whose actors are in k planted rumors. This set has at most K elements. Since only "edge" actors follow anyone, the only way to get k+m actors posting the rumor is to have all edges receive the rumor. Thus the vertices of S must cover every edge of G.

Likewise, if we have a solution of Vertex Cover, then we plant the rumor at the actors that correspond to the vertices of the cover. Since every edge is covered, each of the m "edge" actors' will receive the rumor from one of the two actors it is following and will post the rumor. Thus we end up with m+k actors posting the rumor.

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

Subject: Breakout notes and quiz solution for Oct 14
From: Harold Connamacher <hsc21@case.edu>
To: Jian Liu <jxl2184@case.edu>,Huixian Yang <hxy298@case.edu>,Aimee Chau <axc693@case.edu>,Brendan Dowling <bld43@case.edu>,Colby Saxton <cas264@case.edu>,Seohyun Jung <sxj393@case.edu>,Sheng Guan <sxg967@case.edu>
Date Sent: Fri, 11 Oct 2019 17:42:42 -0400
Date Received: Fri, 11 Oct 2019 14:43:02 -0700 (PDT)
Attachments: breakout6.pdf

Dear TAs,

For the quiz solution, they have to convert Hamiltoninan Cycle (Given a graph, is there a cycle of that hits every vertex exactly once) to Longest Path (given a graph, and a number K, is there a path of K edges).

Given G, create G' by taking G and adding three new vertices. New vertex a is connected to some arbitrary vertex v of G. New vertex b is connected to every neighbor of a in G, and new vertex z is connected to b. We set K to n+2 and ask if there is a path of n+2 edges in G'. The reduction is clearly linear time since we add three vertices and at most n edges.

(->) Assume G has a Hamiltonian cycle, then G' has a path from a to v, the cycle will return to v from some vertex u, but u is connected to b and then b to z. That gives a path from a to z that is 1 edge (a,v), n-1 edges (v to u), and 2 edges (u to b to z).

(<-) Assume G' has a path of n+2 vertices. Since there are only n+3 vertices in G', the path must hit every vertex, and since a and z are degree 1, the path must start at a and end at z. Therefore, there is a path from a to v to some vertex u to b to c, and we know v to u hits every vertex and there is an edge from u to v. Adding that edge gives us a cycle of n vertices in the original G.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, October 14, 2019

1. As usual, start with the quiz, and end the quiz at 3:40.
2. Place the students into groups and give them this problem:

   You have p puppies who need to be adopted and q people who want to adopt puppies. Each person indicates which puppies they want (in an unordered list). Give an efficient algorithm that assigns puppies to people so that the maximum number of possible puppies are adopted and every person who gets a puppy, gets one they want.

   The students need to come up with an algorithm to solve the problem, give the running time, and prove the algorithm is correct.
3. You want to complete the solution by 3:40. So for the next 15 minutes, give them hints and show parts of the solution as you go.
   - They should be reducing this problem to network flow to solve the problem.
   - You may need to remind them of the basic network flow problem (a graph with capacities and we are pushing as much fow from s to t as we can). They have only seen it once.
   - What should the vertices represent? What should the edges represent? What should the flow represent?
   - Let there be vertices for the puppies and people, have the edges be whether a person wants a puppy.
   - Have a unit of flow be a "puppy being adopted".
   - Here is the algorithm:

     Create a vertex for each puppy, a vertex for each person, a source vertex and a sink vertex. Create a directed edge from the source to each puppy vertex, create a directed edge from each person to the sink vertex, and create a directed edges from each puppy vertex to every person who wants that puppy. Give all edges capacity 1. We want directed edges to make sure the flow goes in one direction from puppies to people. It is not essential for this problem but it will it easier to extract the solution. If there is flow from puppy i to person j then puppy i is adopted by person j.
   - The proof should be of the form: "There exists a flow of f units in this network if and only if it is possible for f different puppies to be adopted by f different people".
   - The proof is as follows:

     ($\rightarrow$) Assume we have a flow of size f. Since the residual paths will all have capacity 1, there will be no fractional flows in this network. Each edge will have a flow of 1 or 0. Each puppy will get at most 1 flow because there is a single edge in of capacity 1. Each person receives at most 1 flow because there is a single out edge from each person node with capacity 1. Every puppy that gets a flow will give it to a single person, and that is the person we give the puppy to. Because the edges are directed, the flow is only moving from the puppies to the people. Therefore, if there is a flow of f, we have exactly f single flows moving from f puppies to f people.

     ($\leftarrow$) Assume we can match f puppies to f people. For each puppy that is adopted, the person wanted the puppy so there is an edge from that puppy to that person. We push 1 unit of flow along that edge. Each person received either 1 or 0 unit of flow, and we push that unit to the sink. Each puppy that pushes a unit of flow, receives it from the source and no other puppy receives a flow. *The key point:* at each vertex (except source and sink), the flow in equals the flow out, and the total flow on the network is equal to f
   - What is the running time? We can use either Ford Fulkerson ($O(|f^*|m)$) or Edmunds-Karp ($O(nm^2)$). Number of edges in the graph is $O(pq)$, the number of vertices is $O(p+q)$, and the maximum flow is $O(p+q)$. As a result, Ford-Fulkerson gives $O((p+q)pq)$ and Edmunds-Karp gives $O((pq)^2(p+q))$. So, the best bound is Ford-Fulkerson: $O(p^2q+pq^2)$.

Subject: Re: Breakout notes and quiz solution for Monday
From: Roxanne Yang <hxy298@case.edu>
To: Harold Connamacher <hsc21@case.edu>
Date Sent: Sun, 6 Oct 2019 17:03:14 -0400
Date Received: Sun Oct 06 21:03:14 UTC 2019

Hi Professor Connamacher,

Thanks! It was an amazing experience :)

12:30 to 2 should work best for me! I will see you at Nord 326 tomorrow.

Best,
Roxanne (Huixian) Yang

Case Western Reserve University, Class of 2020
Computer Science B.S. Candidate

On Sun, Oct 6, 2019 at 2:21 PM Harold Connamacher <hsc21@case.edu> wrote:

> Dear Roxanne,
>
> I hope you had a great time at Grace Hopper. My schedule tomorrow is pretty tight, but you can catch me at my office between 12 and 12:30, I will be at a new student meet-and-greet from 12:30 to 2 in Nord 326, and then either before or after my 2:15-3:05 class in Schmitt. If you can't catch me any of those times, I will bring the quizzes to your breakout room.
>
> Regards,
>
> Harold Connamacher
> Associate Professor
> Computer and Data Sciences
> Case Western Reserve University
>
> On Sat, Oct 5, 2019 at 3:50 PM Roxanne Yang <hxy298@case.edu> wrote:
>
>> Thank you for the information! When can I pick up my quizzes?
>>
>> Best,
>> Roxanne Yang
>>
>> On Sat, Oct 5, 2019 at 2:29 PM Harold Connamacher <hsc21@case.edu> wrote:
>>
>>> Dear TAs,
>>>
>>> Attached are the notes for the breakout session. Here is the quiz solution:
>>>
>>> We reduce the airplane problem to the Star Wars problem. Each airplane will be come a transmission interval. Plane i with arrival time t_i and flight limit d_i will become interval [t_i, t_i + d_i], or more correctly: (t_i + d_i)/2 +/- d_i / 2. Each time at which you can land an airplane will become a reception time: x_i = 3i. The reduction is linear time.
>>>
>>> Now we prove that there is a transmission match if and only if we can land every airplane. If there is a transmission match, then we can place each x_i into some interval t_j +/- epsilon_j. That means landing time x_i falls into the interval [t_j, t_j + d_j] in which plane j will be at our airport and able to land. So every plane gets a landing time. For the other direction, assume there is a way to land every plane. That means for each plane j, we

> > landed it at some time X, and X must be between when the plane arrived at the airport t_j and when it ran out of fuel t_j + d_j. Therefore, we matched transmission X/3 uniquely with interval [t_j, t_j + d_j]
> >
> > For grading, the key thing to watch for (besides that they get the reduction pieces correct), is that they are doing a correct if and only if proof. This reduction is simple enough that a clever proof writer can do both if and only-if directions at the same time. That is okay, but it needs to be explicit.
> >
> > Regards,
> >
> > Harold Connamacher
> > Associate Professor
> > Computer and Data Sciences
> > Case Western Reserve University
>
> --
> Roxanne (Huixian) Yang
>
> Case Western Reserve University, Class of 2020
> Computer Science B.S. Candidate

Subject: Re: Breakout notes and quiz solution for Monday
From: Harold Connamacher <hsc21@case.edu>
To: Roxanne Yang <hxy298@case.edu>
Date Sent: Sun, 6 Oct 2019 14:20:54 -0400
Date Received: Sun, 6 Oct 2019 11:21:12 -0700 (PDT)

Dear Roxanne,

I hope you had a great time at Grace Hopper. My schedule tomorrow is pretty tight, but you can catch me at my office between 12 and 12:30, I will be at a new student meet-and-greet from 12:30 to 2 in Nord 326, and then either before or after my 2:15-3:05 class in Schmitt. If you can't catch me any of those times, I will bring the quizzes to your breakout room.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

On Sat, Oct 5, 2019 at 3:50 PM Roxanne Yang <hxy298@case.edu> wrote:

> Thank you for the information! When can I pick up my quizzes?
>
> Best,
> Roxanne Yang
>
> On Sat, Oct 5, 2019 at 2:29 PM Harold Connamacher <hsc21@case.edu> wrote:
>
>> Dear TAs,
>>
>> Attached are the notes for the breakout session. Here is the quiz solution:
>>
>> We reduce the airplane problem to the Star Wars problem. Each airplane will be come a transmission interval. Plane i with arrival time t_i and flight limit d_i will become interval [t_i, t_i + d_i], or more correctly: (t_i + d_i)/2 +/- d_i / 2.   Each time at which you can land an airplane will become a reception time: x_i = 3i.   The reduction is linear time.
>>
>> Now we prove that there is a transmission match if and only if we can land every airplane. If there is a transmission match, then we can place each x_i into some interval t_j +/- epsilon_j. That means landing time x_i falls into the interval [t_j, t_j + d_j] in which plane j will be at our airport and able to land. So every plane gets a landing time. For the other direction, assume there is a way to land every plane. That means for each plane j, we landed it at some time X, and X must be between when the plane arrived at the airport t_j and when it ran out of fuel t_j + d_j. Therefore, we matched transmission X/3 uniquely with interval [t_j, t_j + d_j]
>>
>> For grading, the key thing to watch for (besides that they get the reduction pieces correct), is that they are doing a correct if and only if proof. This reduction is simple enough that a clever proof writer can do both if and only-if directions at the same time. That is okay, but it needs to be explicit.
>>
>> Regards,
>>
>> Harold Connamacher
>> Associate Professor

> > Computer and Data Sciences
> > Case Western Reserve University
>
> --
> Roxanne (Huixian) Yang
>
> Case Western Reserve University, Class of 2020
> Computer Science B.S. Candidate

Subject: Re: Breakout notes and quiz solution for Monday
From: Roxanne Yang <hxy298@case.edu>
To: Harold Connamacher <hsc21@case.edu>
Date Sent: Sat, 5 Oct 2019 15:50:02 -0400
Date Received: Sat Oct 05 19:50:02 UTC 2019

Thank you for the information! When can I pick up my quizzes?

Best,
Roxanne Yang

On Sat, Oct 5, 2019 at 2:29 PM Harold Connamacher <hsc21@case.edu> wrote:

> Dear TAs,
>
> Attached are the notes for the breakout session. Here is the quiz solution:
>
> We reduce the airplane problem to the Star Wars problem. Each airplane will be come a transmission interval. Plane i with arrival time t_i and flight limit d_i will become interval [t_i, t_i + d_i], or more correctly: (t_i + d_i)/2 +/- d_i / 2.   Each time at which you can land an airplane will become a reception time: x_i = 3i.   The reduction is linear time.
>
> Now we prove that there is a transmission match if and only if we can land every airplane. If there is a transmission match, then we can place each x_i into some interval t_j +/- epsilon_j. That means landing time x_i falls into the interval [t_j, t_j + d_j] in which plane j will be at our airport and able to land. So every plane gets a landing time. For the other direction, assume there is a way to land every plane. That means for each plane j, we landed it at some time X, and X must be between when the plane arrived at the airport t_j and when it ran out of fuel t_j + d_j. Therefore, we matched transmission X/3 uniquely with interval [t_j, t_j + d_j]
>
> For grading, the key thing to watch for (besides that they get the reduction pieces correct), is that they are doing a correct if and only if proof. This reduction is simple enough that a clever proof writer can do both if and only-if directions at the same time. That is okay, but it needs to be explicit.
>
> Regards,
>
> Harold Connamacher
> Associate Professor
> Computer and Data Sciences
> Case Western Reserve University

--
Roxanne (Huixian) Yang

Case Western Reserve University, Class of 2020
Computer Science B.S. Candidate

Subject: Breakout notes and quiz solution for Monday
From: Harold Connamacher <hsc21@case.edu>
To: Aimee Chau <axc693@case.edu>,Colby Saxton <cas264@case.edu>,Sheng Guan <sxg967@case.edu>,Brendan Dowling <bld43@case.edu>,Huixian Yang <hxy298@case.edu>,Jian Liu <jxl2184@case.edu>,Seohyun Jung <sxj393@case.edu>
Date Sent: Sat, 5 Oct 2019 14:29:14 -0400
Date Received: Sat, 5 Oct 2019 11:29:34 -0700 (PDT)
Attachments: breakout5.pdf

Dear TAs,

Attached are the notes for the breakout session. Here is the quiz solution:

We reduce the airplane problem to the Star Wars problem. Each airplane will be come a transmission interval. Plane i with arrival time t_i and flight limit d_i will become interval [t_i, t_i + d_i], or more correctly: (t_i + d_i)/2 +/- d_i / 2. Each time at which you can land an airplane will become a reception time: x_i = 3i. The reduction is linear time.

Now we prove that there is a transmission match if and only if we can land every airplane. If there is a transmission match, then we can place each x_i into some interval t_j +/- epsilon_j. That means landing time x_i falls into the interval [t_j, t_j + d_j] in which plane j will be at our airport and able to land. So every plane gets a landing time. For the other direction, assume there is a way to land every plane. That means for each plane j, we landed it at some time X, and X must be between when the plane arrived at the airport t_j and when it ran out of fuel t_j + d_j. Therefore, we matched transmission X/3 uniquely with interval [t_j, t_j + d_j]

For grading, the key thing to watch for (besides that they get the reduction pieces correct), is that they are doing a correct if and only if proof. This reduction is simple enough that a clever proof writer can do both if and only-if directions at the same time. That is okay, but it needs to be explicit.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, October 7, 2019

1. As usual, we start with the quiz. Please end the quiz at 3:40.

2. Place the students into groups and give them this problem:

   You have to divide up n items between you and your sibling. Each item i has two values. There is the value a_i that you give the item and the value b_i that your sibling gives the item. You sibling will be unhappy if he/she does not get items that total to at least B total value (as summed by their b_i values).

   You want an algorithm that will choose the items for yourself to maximize the sum of the a_i values of the items you are getting such that the sum of the b_i values for the items you leave for your sibling is at least B.

   We want to solve this problem by *reducing* it to another similar problem.

3. Let the groups take just a little time figuring out what problem is close to this one. You will probably have to give hints because we just started reductions and they will not remember all the problems from class.

4. The problem that is closest to this problem is Knapsack. The hint is that we choosing values to take and we have a limit imposed on how many items we can choose. You should give them the formal definition of Knapsack since most students will have forgotten the details:
   We have n items. Each item has a weight w_i and a value v_i, and you have a bag with capacity C. You want to maximize the value of the items you take such that the sum of their weight is at most C.

5. Now, you should help them create the reduction. The key points are that we have to take the n items, values a_i and b_i and turn it into n items with value v_i weight w_i and a bag with capacity C.

6. The easier part is to see that the n items for the Knapsack problem will be the same n items for the sibling problem. The next easiest part is to see that for each item, we can set the value $v_i = a_i$ because that is what we want to maximize, and we can set the weight $w_i = b_i$ since that is what we want to limit. The trickiest part is to get the capacity correct. If we want to leave total b value (or w weight) B behind, then we want to take at most $\sum_{i=1}^{n} b_i - B$ value. So we set $C = \sum_{i=1}^{n} b_i - B$.

7. This reduction is clearly worst case linear time since we just run through the objects (actually we don't have to change the objects at all) and do a single summation of n values. Even if we allow the values to be really big, the summation is still linear in terms of the number of bits in the values.

8. Now we have to do an "if-and-only-if" proof that the reduction is correct. Let the groups try to form the proof statement before you do.

9. We need to prove:
   The sibling problem has a solution that gives me total value at least X if and only if there is a solution to the knapsack problem with total value at least X.
   You can remind them that this implies that a maximizing the sibling problem is the same as maximizing the knapsack, but we write it this way because it is easier to prove for a specific value than to prove the value is the maximum.

10. Proof: Assume the knapsack problem returns total value X. Then it selected items that sum up to X, and we choose the same items from the sibling problem to get value X. The items weigh at most $C = \sum_{i=1}^{n} b_i - B$ so we leave our sibling at least $\sum_{i=1}^{n} b_i - C = B$ value.

    Assume there is a way to choose total value X items for you while leaving your sibling at least B value. Fill a knapsack with those same items and you get value X in your knapsack. The sibling value of the items you take is at most $\sum_{i=1}^{n} b_i - B$. Since this value is also C, the values you take fit in the knapsack.

Subject: 340 Homework 2 rubric and solutions
From: Harold Connamacher <hsc21@case.edu>
To: Jian Liu <jxl2184@case.edu>,Sheng Guan <sxg967@case.edu>,Aimee Chau <axc693@case.edu>,Brendan Dowling <bld43@case.edu>,Colby Saxton <cas264@case.edu>,Huixian Yang <hxy298@case.edu>,Seohyun Jung <sxj393@case.edu>
Date Sent: Fri, 4 Oct 2019 11:30:52 -0400
Date Received: Fri, 4 Oct 2019 08:31:10 -0700 (PDT)

Dear TAs,

The grading rubric for this homework is the same as the prevous one.

For each question, check the following points:
1. The algorithm is precise and clear.
2. The algorithm used the correct technique (greedy/DP/divide and conquer) for the problem. It is not enough to say they are using a certain technique. It is very common for a student to say they are using DP, but the algorithm recurrence is actually a greedy algorithm.
3. The algorithm is correct.
4. They provide a proof that has reasonable rigor. For example, there are clear assumptions made. The biggest mistake is for students to just describe what their algorithms do and call that a "proof".
5. They use a proof appropriate for the algorithm type they are trying to write (not necessarily for the correct algorithm).
6. The proof is correct.
7. The correct runtime for their algorithm (not for the correct algorithm) is given.
8. A reasonable justification of the running time is given.

Here is the grade to assign:

8: Perfect. All 8 above achieved.
7: Very good. All 8 points attempted, and at least 6 of the 8 points achieved.
6: Good. Achieved at least 1, 2, 4 and 5.
5: Acceptable. At least two of 1, 2, 4, and 5.
4: Poor. Attempted points 1 and 4.
3: Poor-. Attempted either point 1 or 4.
2: There some attempt at an algorithm.
0: There is nothing useful to grade.

Problem 1:
Create a set {1,....n} corresponding to all the indeces of the array. Let x be the bits of the missing number. x starts out as empty. Let k=log n. Request the kth bit of each entry at an index in the set. If there are more 1 bits returned than 0's, then the missing number starts with a 0. Append 0 to x and remove from the set all those indeces that returned a 1. Otherwise is starts with a 1 and append 1 to x and remove from the set all those indeces that returned a 0. Decrement k. If k=0 x stores the missing number.

The total number of steps is O(n) in the first round, O(n/2) in the second round and so on. This is O(n + n/2 + n/4 + ...) = O(n).

Proof: The base case is when n=1. There will be a single bit, and if we query it we see what is missing. Assume we can find the missing number from a set of k digits numbers. Consider a set of k+1 digit numbers. If the set is even is even there should be the same number of entries that start with a 0 as start with a 1. If the set is odd, there should be one more entry that starts with a 0 as starts with a 1. If there are more 1's than 0's, then we know that the missing number starts with a 0. Otherwise it must start with a 1. We remove all those indeces from the set that corresponds to the majority answer (what the missing number cannot be). Then we have effectively looking at a smaller set of (k-1) digit numbers. By the induction hypothesis, we find the missing number.

Problem 2:
Sort the pizzas by value so that v_1 >= v_2 >= .... >= v_n. Then run through the list in order. For pizza i, we schedule it as late as possible and still meet the deadline. So we go to the deadline d_i and work from there to time 0 and schedule the pizza in the first available slot. If there is none available, we schedule the pizza as late as possible in the schedule. So we go to time n and work backwards scheduling it in the first available slot.

The runtime is O(n log n) to sort, O(n) to run through the list, but at worst case I may have to run through O(n) slots before finding a free slot. For a total run time of O(n^2).

Proof: Assume we have an optimal schedule O, and if we consider the pizzas in the order scheduled by greedy, O and G schedule the first k pizzas the same but differ on the time for pizza k+1. Let pizza x be scheduled in O at the time when G schedules k+1. Create schedule O* where O and O* are the same except that O* swaps the scheduling of pizzas x and k+1. O* now matches G on k+1 pizzas, but we have to prove that O* is just as good a schedule as O. Here are the cases:

Case 1: O schedules k+1 before G/O* does.

Case 1a: O schedules k+1 before its deadline but O* does not. This is impossible since G will schedule k+1 before the deadline if possible.

Case 1b: O and G/O* both schedule k+1 after the deadline. O and O* still pay the penalty for pizza k+1, but pizza x is moved earlier on O* than on O. So at worst O* pays the same penalty as O.

Case 1c: O and G/O* both schedule k+1 before the deadline. Neither O nor O* pays a penalty for pizza k+1, and pizza x is moved earlier in the schedule in O* than in O. So at worst O* pays the same penalty as O.

Case 2: O schedules k+1 after G/O* does.

Case 2a: O schedules k+1 before its deadline. This is impossible. If O schedules k+1 before its deadline, there is an available slot before its deadline, and G put k+1 in the last available slot before its deadline.

Case 2b: O schedules k+1 after its deadline. Since G puts k+1 in the last available slot, G (and therefore O*) must put k+1 before its deadline. Now in O*, pizza x is scheduled later than in O. So O* may have to pay the penalty on pizza x if it now misses its deadline. However, O pays the penalty of pizza k+1. So penalty(O*) <= penalty(O) - v_{k+1} + v_x. Since v_{k+1} >= v_x, we have penalty(O*) <= penalty(O).

Problem 3:
Let T[i,j] be the maximum profit we can earn if we do j buys and sells and the last sell occurred at time i or earlier. We let T[i,0] = 0 for all i,
T[i,j] = max{T[i-1,j], max{T[x, j-1] + 1000(p(x+1)-p(i))} for all 2(j-1)<=x< i}

Prove that T[i,j] is the maximum profit from making j buys and sells that complete by time i. Assume T[i-1,j] stores the maximum profit from making j buys by time i-1 and T[x,j-1] stores the maximum profit from making j-1 buys by time x for all times less than i. We have two choices. If we do not complete a sale at time i, then the best we can do is the maximum profit at time i-1, and by the induction hypothesis, T[i-1,j] stores this. If we complete a sale at time i, then we purchased those shares at time some time x+1, and we must have completed the previous j-1 sale at some time at or before x. Since we are buying at time x+1, the want the maximum profit possible at time x, and by the induction hypothesis T[x,j-1] stores this value. We take the maximum over all these choices.

Here is how to get the solution from the table. This will be needed in the homework, but don't deduct points if the students forget this in the quiz. The solution is the largest value T[n,m] for all m from 0 to k. We then can work backwards from here. Start with s=n and t=k. From T[s,t], if T[s,t] = T[s-1,t] then we did not complete a sale at time t. Otherwise, we go search for all x from 2(t-1) to s for the T[x,t-1] that stores T[s,t] - 1000(p(x+1,t-1), and that means we bought at time x+1 and sold at time t. We then set s=x and t=t-1 and repeat. (Another solution is to have a second table that stores the choices made at each step. If T[i,j] = T[i-1,j], the set S[i,j] = 0 to indicate no sale. Otherwise, set S[i,j] = x+1 to record that we bought at time x+1 and sold at time j.

The size of the table is n x k, and it takes up to time n to go back to find each value for a running time of O(n^2 k).

Problem 4:
Consider the x_i values from smallest to largest, and match each x_i to the interval t_j that overlaps with x_i and, of those that overlap, has the smallest t_j + \epsilon_j.

This algorithm is O(n^2) because for each of the n times, we may have to run through n intervals to test for overlap and to store the minimum end time.

Proof: Let G be the matching our greedy algorithm produces and let O be the matching that some optimal algorithm produces. Consider the values from smallest to largest and suppose that G and O match the first k-1 values to the same intervals, but now assume that G matches the x_k to interval t_a while O matches x_k to interval t_b and O matches some other value x_p to t_a.
Now we have the following:

t_b - \epsilon_b <= x_k	because O matches x_k to t_b
<= x_p	because G considers the values in order
<= t_a + \epsilon_a	because O matches x_p to t_a
<= t_b + \epsilon_b	because G matched x_k to t_a instead of t_b

Therefore, we can create O' by matching x_k to t_a instead of t_b and matching x_p to t_b instead of t_a. Since all other matchings stayed the same, O' has the same number of matchings as O and it agrees with G on the first k values.

Problem 5:
Let T[i, j] = true if the first i+j characters of s is an interleaving of i characters of s1^p and j characters of s2^q for any non-negative p and q.

We have two choices, either match the next character of s with s_1 or match the next character of s with s_2. The recurrence is:
T[0,0] = true
T[i, j] = (T[i-1, j] || s[i+j] == s1[i % s1.length]) &&
    (T[i, j-1] || s[i+j] == s2[j % s2.length])

Let n be the length of s. To get a solution, we look at all T[x,y] where x+y = n and see if any are true. The running time is O(n^2) since we have an n x n table and it takes O(1) to fill in each entry.

Proof. Assume T[i-1, j] indicates whether the first i+j-1 characters of s is an interleaving of i-1 characters from s1^p and j characters from s2^q, and assume T[i,j-1] indicates whether the first i+j-1 characters of s is an interleaving of i characters from s1^p and j-1 characters from s2^q. Are the first i+j characters of s an interleaving of i characters from s1 and j characters from s2? If so, then the (i+j)th character is either from s1 or from s_2. If it is from s1, then it must match the (i % s1.length) character of s1 and the remaining i+j-1 characters are an interleaving of i-1 chars from s1^p for some p and j chars from s2^q for some q. By the induction hypothesis, T[i-1,j] stores this value. The same reasoning applies if the (i+j)th character is from s2.

Problem 6:
If the set S is only one interval {[a,b]} with height h, then return the skyline [(0,0),(a,h),(b,0),(1,0)].

Otherwise, split the set S in half arbitrarily and recurse on each half. The result of the two recursive calls are the skylines [(0, a_0), (y_1, a_1), ..., (y_p, a_p), (y_{p+1},0)] and [(0, b_0), (z_1,b_1), ..., (z_q, b_q), (z_{q+1}, 0)]. We now merge the two skylines together. The first point of the output is (0, max {a_0,b_0}). We run through both skylines from left to right. Suppose we are considering points (y_i, a_i) and (z_j, b_j).

Case 1: If y_i < z_j, add point (y_i, max {a_i, b_j}) and go to the next point in sequence 1.
Case 2: If y_i > z_j, add point (z_j, max {a_i, b_j}) and go to the next point in sequence 2.
Otherwise: add point (y_i, max {a_i, b_j}) and go to the next point in both sequence 1 and 2.

The result does not have maximal intervals, so we run through the output a second time, and if we have a point (x_i, c_i), (x_{i+1}, c_{i+1}) with c_i = c_{i+1}, we remove the second point. (Technically, we must check to see that this lies inside an interval, so we would need to record for each point whether we added it from skyline 1 or 2, but you can ignore that technicality in the grading).

The proof is induction on the size of S. The base case of S=1 is trivial.

For S > 1, we split the list in two and assume that each half's skyline contains the maximal subintervals of a maximal height. Consider interval [x_i,x_{i+1}] in the output skyline. When the algorithm added the point (x_i,c_i), it chose the maximum of skyline y and z at that point. Under the I.H. that each skyline is maximal, then c_i is the maximal height at this point. Because we removed duplicates, the point (x_{i+1},c_{i+1}) is only included where one of the skyline changes and only if c_{i+1} != c_i. Thus, the interval is a maximal interval with height c_i.

The running time is T(n) = 2T(n/2) + cn and so T(n) is Theta(n log n).

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

Subject: Re: Breakout notes and quiz solution
From: Roxanne Yang <hxy298@case.edu>
To: Harold Connamacher <hsc21@case.edu>
Date Sent: Mon, 30 Sep 2019 13:23:46 -0400
Date Received: Mon Sep 30 17:23:46 UTC 2019

Thank you so much!

Best,
Roxanne (Huixian) Yang

Case Western Reserve University, Class of 2020
Computer Science B.S. Candidate

On Mon, Sep 30, 2019 at 12:57 PM Harold Connamacher <hsc21@case.edu> wrote:

> Here you are.
>
> Regards,
>
> Harold Connamacher
> Associate Professor
> Computer and Data Sciences
> Case Western Reserve University
>
> On Mon, Sep 30, 2019 at 12:39 PM Roxanne Yang <hxy298@case.edu> wrote:
>
>> Hi Professor Connamacher,
>>
>> I just realized I forgot to bring my quizzes to school :(Could you send me the file and I can print it myself? Thank you so much!
>>
>> Best,
>> Roxanne (Huixian) Yang
>>
>> Case Western Reserve University, Class of 2020
>> Computer Science B.S. Candidate
>>
>> On Mon, Sep 30, 2019 at 12:22 PM Harold Connamacher <hsc21@case.edu> wrote:
>>
>>> Hi Brendan,
>>>
>>> You are correct that there are two typos. The t should be t-1, and the max should be a min.
>>>
>>>   T[t,l,d] = T[t-1, l, d-1] + f_1 - min{f_l, d_l * (d - 1)}
>>>
>>> which is equivalent to what you have.
>>>
>>> Thanks,
>>>
>>> Harold Connamacher
>>> Associate Professor
>>> Computer and Data Sciences
>>> Case Western Reserve University

On Mon, Sep 30, 2019 at 12:02 PM Brendan Dowling <bld43@case.edu> wrote:

> Professor Connamacher,
>
> Just to make sure I understand correctly, why is the recurrence for T[t,l,d]
> T[t,l,d] = T[t,l,d-1] + f_l - max{f_l, d_l * (d - 1)}
> rather than
> T[t,l,d] = T[t-1,l,d-1] + max{f_l -f_l, f_l - d_l * (d - 1)}
> The first part (T[t-1,l,d-1])seems necessitated because you wouldn't be referencing the same time as a max and would need to reference the prior time,
> and the second part (changing the max) seems to be needed as you force the fish gained to be 0 at any time no matter what with the writing of the first recurrance (the max of the two options is always going to be greater than or equal to f_l).
>
> I get the feeling that this might be an error but i wanted to make sure,
>
> Brendan
>
> On Fri, Sep 27, 2019 at 5:14 PM Harold Connamacher <hsc21@case.edu> wrote:
>
>> Hello TAs,
>>
>> For the breakout, you are going to do the fishing problem. I like it because you can show an "incorrect" recurrence set up and explain what simply optimizing the subproblems does not give you an optimal solution. I did that in Friday's lecture, and they need to see it again because a lot of people did not give a good DP proof on the test.
>>
>> Quiz Answer:
>>
>> Consider the x_i values from smallest to largest, and match each x_i to the interval t_j that overlaps with x_i and, of those that overlap, has the smallest t_j + \epsilon_j.
>>
>> This algorithm is O(n^2) because for each of the n times, we may have to run through n intervals to test for overlap and to store the minimum end time.
>>
>> Proof: Let G be the matching our greedy algorithm produces and let O be the matching that some optimal algorithm produces. Consider the values from smallest to largest and suppose that G and O match the first k-1 values to the same intervals, but now assume that G matches the x_k to interval t_a while O matches x_k to interval t_b and O matches some other value x_p to t_a. Now we have the following:
>>
>> ```
>> t_b - \epsilon_b <= x_k                 because O matches x_k to t_b
>>                  <=  x_p                because G considers the values in order
>>                  <= t_a + \epsilon_a    because O matches x_p to t_a
>>                  <= t_b + \epsilon_b    because G matched x_k to t_a instead of t_b
>> ```
>>
>> Therefore, we can create O' by matching x_k to t_a instead of t_b and matching x_p to t_b instead of t_a. Since all other matchings stayed the same, O' has the same number of matchings as O and it agrees with G on the first k values.
>>
>> Regards,
>>
>> Harold Connamacher
>> Associate Professor
>> Computer and Data Sciences
>> Case Western Reserve University

Subject: Re: Breakout notes and quiz solution
From: Harold Connamacher <hsc21@case.edu>
To: Roxanne Yang <hxy298@case.edu>
Date Sent: Mon, 30 Sep 2019 12:57:00 -0400
Date Received: Mon, 30 Sep 2019 09:57:20 -0700 (PDT)
Attachments: quiz4.pdf

Here you are.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

On Mon, Sep 30, 2019 at 12:39 PM Roxanne Yang <hxy298@case.edu> wrote:

> Hi Professor Connamacher,
>
> I just realized I forgot to bring my quizzes to school :(Could you send me the file and I can print it myself? Thank you so much!
>
> Best,
> Roxanne (Huixian) Yang
>
> Case Western Reserve University, Class of 2020
> Computer Science B.S. Candidate
>
> On Mon, Sep 30, 2019 at 12:22 PM Harold Connamacher <hsc21@case.edu> wrote:
>
>> Hi Brendan,
>>
>> You are correct that there are two typos. The t should be t-1, and the max should be a min.
>>
>> T[t,l,d] = T[t-1, l, d-1] + f_1 - min{f_l, d_l * (d - 1)}
>>
>> which is equivalent to what you have.
>>
>> Thanks,
>>
>> Harold Connamacher
>> Associate Professor
>> Computer and Data Sciences
>> Case Western Reserve University
>>
>> On Mon, Sep 30, 2019 at 12:02 PM Brendan Dowling <bld43@case.edu> wrote:
>>
>>> Professor Connamacher,
>>>
>>> Just to make sure I understand correctly, why is the recurrence for T[t,l,d]
>>> T[t,l,d] = T[t,l,d-1] + f_l - max{f_l, d_l * (d - 1)}
>>> rather than
>>> T[t,l,d] = T[t-1,l,d-1] + max{f_l -f_l, f_l - d_l * (d - 1)}
>>> The first part (T[t-1,l,d-1])seems necessitated because you wouldn't be referencing the

same time as a max and would need to reference the prior time,
and the second part (changing the max) seems to be needed as you force the fish gained to be 0 at any time no matter what with the writing of the first recurrance (the max of the two options is always going to be greater than or equal to f_l).

I get the feeling that this might be an error but i wanted to make sure,

Brendan

On Fri, Sep 27, 2019 at 5:14 PM Harold Connamacher <hsc21@case.edu> wrote:

Hello TAs,

For the breakout, you are going to do the fishing problem. I like it because you can show an "incorrect" recurrence set up and explain what simply optimizing the subproblems does not give you an optimal solution. I did that in Friday's lecture, and they need to see it again because a lot of people did not give a good DP proof on the test.

Quiz Answer:

Consider the x_i values from smallest to largest, and match each x_i to the interval t_j that overlaps with x_i and, of those that overlap, has the smallest t_j + \epsilon_j.

This algorithm is O(n^2) because for each of the n times, we may have to run through n intervals to test for overlap and to store the minimum end time.

Proof: Let G be the matching our greedy algorithm produces and let O be the matching that some optimal algorithm produces. Consider the values from smallest to largest and suppose that G and O match the first k-1 values to the same intervals, but now assume that G matches the x_k to interval t_a while O matches x_k to interval t_b and O matches some other value x_p to t_a. Now we have the following:

```
t_b - \epsilon_b <= x_k                because O matches x_k to t_b
                 <=  x_p               because G considers the values in order
                 <= t_a + \epsilon_a   because O matches x_p to t_a
                 <= t_b + \epsilon_b   because G matched x_k to t_a instead of t_b
```

Therefore, we can create O' by matching x_k to t_a instead of t_b and matching x_p to t_b instead of t_a. Since all other matchings stayed the same, O' has the same number of matchings as O and it agrees with G on the first k values.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EEACS 340: Algorithms Quiz 4

Name & Case ID (8 points): ______________________________

Question 1 (8 points): You have n values (message receive times) $x_1, \ldots, x_n$, and you have n intervals (possible message transmission times) $t_1 \pm \epsilon_1, \ldots, t_n \pm \epsilon_n$.

Design an algorithm that matches each x_i value to one interval $t_j \pm \epsilon_j$ such that every value and interval is matched and for each matching of value x_i and interval $t_j \pm \epsilon_j$, we have $t_j - \epsilon_j \leq x_i \leq t_j + \epsilon_j$. Or indicate that no such matching of values and intervals is possible.

Prove your algorithm correct and state and justify its running time.

Subject: Re: Breakout notes and quiz solution
From: Roxanne Yang <hxy298@case.edu>
To: Harold Connamacher <hsc21@case.edu>
Date Sent: Mon, 30 Sep 2019 12:39:01 -0400
Date Received: Mon Sep 30 16:39:01 UTC 2019

Hi Professor Connamacher,

I just realized I forgot to bring my quizzes to school :(Could you send me the file and I can print it myself? Thank you so much!

Best,
Roxanne (Huixian) Yang

Case Western Reserve University, Class of 2020
Computer Science B.S. Candidate

On Mon, Sep 30, 2019 at 12:22 PM Harold Connamacher <hsc21@case.edu> wrote:

> Hi Brendan,
>
> You are correct that there are two typos. The t should be t-1, and the max should be a min.
>
> T[t,l,d] = T[t-1, l, d-1] + f_1 - min{f_l, d_l * (d - 1)}
>
> which is equivalent to what you have.
>
> Thanks,
>
> Harold Connamacher
> Associate Professor
> Computer and Data Sciences
> Case Western Reserve University
>
> On Mon, Sep 30, 2019 at 12:02 PM Brendan Dowling <bld43@case.edu> wrote:
>
>> Professor Connamacher,
>>
>> Just to make sure I understand correctly, why is the recurrence for T[t,l,d]
>> T[t,l,d] = T[t,l,d-1] + f_l - max{f_l, d_l * (d - 1)}
>> rather than
>> T[t,l,d] = T[t-1,l,d-1] + max{f_l -f_l, f_l - d_l * (d - 1)}
>> The first part (T[t-1,l,d-1])seems necessitated because you wouldn't be referencing the same time as a max and would need to reference the prior time,
>> and the second part (changing the max) seems to be needed as you force the fish gained to be 0 at any time no matter what with the writing of the first recurrance (the max of the two options is always going to be greater than or equal to f_l).
>>
>> I get the feeling that this might be an error but i wanted to make sure,
>>
>> Brendan
>>
>> On Fri, Sep 27, 2019 at 5:14 PM Harold Connamacher <hsc21@case.edu> wrote:
>>
>>> Hello TAs,

For the breakout, you are going to do the fishing problem. I like it because you can show an "incorrect" recurrence set up and explain what simply optimizing the subproblems does not give you an optimal solution. I did that in Friday's lecture, and they need to see it again because a lot of people did not give a good DP proof on the test.

Quiz Answer:

Consider the x_i values from smallest to largest, and match each x_i to the interval t_j that overlaps with x_i and, of those that overlap, has the smallest t_j + \epsilon_j.

This algorithm is O(n^2) because for each of the n times, we may have to run through n intervals to test for overlap and to store the minimum end time.

Proof: Let G be the matching our greedy algorithm produces and let O be the matching that some optimal algorithm produces. Consider the values from smallest to largest and suppose that G and O match the first k-1 values to the same intervals, but now assume that G matches the x_k to interval t_a while O matches x_k to interval t_b and O matches some other value x_p to t_a. Now we have the following:

```
t_b - \epsilon_b <= x_k                because O matches x_k to t_b
                 <=  x_p               because G considers the values in order
                 <= t_a + \epsilon_a   because O matches x_p to t_a
                 <= t_b + \epsilon_b   because G matched x_k to t_a instead of t_b
```

Therefore, we can create O' by matching x_k to t_a instead of t_b and matching x_p to t_b instead of t_a. Since all other matchings stayed the same, O' has the same number of matchings as O and it agrees with G on the first k values.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

Subject: Re: Breakout notes and quiz solution
From: Harold Connamacher <hsc21@case.edu>
To: Brendan Dowling <bld43@case.edu>,Seohyun Jung <sxj393@case.edu>,Aimee Chau <axc693@case.edu>,Colby Saxton <cas264@case.edu>,Huixian Yang <hxy298@case.edu>,Jian Liu <jxl2184@case.edu>,Sheng Guan <sxg967@case.edu>
Date Sent: Mon, 30 Sep 2019 12:22:14 -0400
Date Received: Mon, 30 Sep 2019 09:22:33 -0700 (PDT)

Hi Brendan,

You are correct that there are two typos. The t should be t-1, and the max should be a min.

T[t,l,d] = T[t-1, l, d-1] + f_1 - min{f_l, d_l * (d - 1)}

which is equivalent to what you have.

Thanks,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

On Mon, Sep 30, 2019 at 12:02 PM Brendan Dowling <bld43@case.edu> wrote:

> Professor Connamacher,
>
> Just to make sure I understand correctly, why is the recurrence for T[t,l,d]
> T[t,l,d] = T[t,l,d-1] + f_l - max{f_l, d_l * (d - 1)}
> rather than
> T[t,l,d] = T[t-1,l,d-1] + max{f_l -f_l, f_l - d_l * (d - 1)}
> The first part (T[t-1,l,d-1])seems necessitated because you wouldn't be referencing the same time as a max and would need to reference the prior time,
> and the second part (changing the max) seems to be needed as you force the fish gained to be 0 at any time no matter what with the writing of the first recurrance (the max of the two options is always going to be greater than or equal to f_l).
>
> I get the feeling that this might be an error but i wanted to make sure,
>
> Brendan
>
> On Fri, Sep 27, 2019 at 5:14 PM Harold Connamacher <hsc21@case.edu> wrote:
>
>> Hello TAs,
>>
>> For the breakout, you are going to do the fishing problem. I like it because you can show an "incorrect" recurrence set up and explain what simply optimizing the subproblems does not give you an optimal solution. I did that in Friday's lecture, and they need to see it again because a lot of people did not give a good DP proof on the test.
>>
>> Quiz Answer:
>>
>> Consider the x_i values from smallest to largest, and match each x_i to the interval t_j that overlaps with x_i and, of those that overlap, has the smallest t_j + \epsilon_j.

> > This algorithm is O(n^2) because for each of the n times, we may have to run through n intervals to test for overlap and to store the minimum end time.
> >
> > Proof: Let G be the matching our greedy algorithm produces and let O be the matching that some optimal algorithm produces. Consider the values from smallest to largest and suppose that G and O match the first k-1 values to the same intervals, but now assume that G matches the x_k to interval t_a while O matches x_k to interval t_b and O matches some other value x_p to t_a. Now we have the following:
> >
> > ```
> > t_b - \epsilon_b <= x_k                 because O matches x_k to t_b
> >                  <=  x_p                because G considers the values in order
> >                  <= t_a + \epsilon_a    because O matches x_p to t_a
> >                  <= t_b + \epsilon_b    because G matched x_k to t_a instead of t_b
> > ```
> >
> > Therefore, we can create O' by matching x_k to t_a instead of t_b and matching x_p to t_b instead of t_a. Since all other matchings stayed the same, O' has the same number of matchings as O and it agrees with G on the first k values.
> >
> > Regards,
> >
> > Harold Connamacher
> > Associate Professor
> > Computer and Data Sciences
> > Case Western Reserve University

Subject: Breakout notes and quiz solution
From: Harold Connamacher <hsc21@case.edu>
To: Seohyun Jung <sxj393@case.edu>,Jian Liu <jxl2184@case.edu>,Aimee Chau <axc693@case.edu>,Huixian Yang <hxy298@case.edu>,Colby Saxton <cas264@case.edu>,Sheng Guan <sxg967@case.edu>,Brendan Dowling <bld43@case.edu>
Date Sent: Fri, 27 Sep 2019 17:13:55 -0400
Date Received: Fri, 27 Sep 2019 14:14:13 -0700 (PDT)
Attachments: breakout4.pdf

Hello TAs,

For the breakout, you are going to do the fishing problem. I like it because you can show an "incorrect" recurrence set up and explain what simply optimizing the subproblems does not give you an optimal solution. I did that in Friday's lecture, and they need to see it again because a lot of people did not give a good DP proof on the test.

Quiz Answer:

Consider the x_i values from smallest to largest, and match each x_i to the interval t_j that overlaps with x_i and, of those that overlap, has the smallest t_j + \epsilon_j.

This algorithm is O(n^2) because for each of the n times, we may have to run through n intervals to test for overlap and to store the minimum end time.

Proof: Let G be the matching our greedy algorithm produces and let O be the matching that some optimal algorithm produces. Consider the values from smallest to largest and suppose that G and O match the first k-1 values to the same intervals, but now assume that G matches the x_k to interval t_a while O matches x_k to interval t_b and O matches some other value x_p to t_a.
Now we have the following:

t_b - \epsilon_b	<= x_k	because O matches x_k to t_b
	<= x_p	because G considers the values in order
	<= t_a + \epsilon_a	because O matches x_p to t_a
	<= t_b + \epsilon_b	because G matched x_k to t_a instead of t_b

Therefore, we can create O' by matching x_k to t_a instead of t_b and matching x_p to t_b instead of t_a. Since all other matchings stayed the same, O' has the same number of matchings as O and it agrees with G on the first k values.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, October 12, 2018

1. Begin by handing out the quiz. The students are to work alone with no notes, phones, etc. Collect the quiz at 3:40 so that you have 30 minutes to do the exercise below.

2. Place the students into groups of 3 and give them this problem (taken from the International Programming Competition): John is going fishing. He has h hours available, and there are n lakes available all reachable on a single, one-way road. John starts at lake 1 and can finish at any lake he wants. He can only travel from one lake to the next one, but he does not have to stop at a lake unless he wishes to. For each $i = 1, \ldots, n-1$, t_i is the number of minutes it takes to travel from lake i to lake $i+1$. All times are in 5-minute intervals. For each lake i, f_i is the number of fish he expects to catch in his first 5 minutes of fishing at the lake, ($f_i \geq 0$). Each 5 minutes of fishing decreases the number of fish expected to be caught in the next 5 minute interval by a constant amount d_i, $d_i \geq 0$. If the number of fish expected to be caught in a 5-minute interval is less than d_i then no fish will be caught at this lake after this interval ends.

   Design an algorithm to plan John's trip in order to maximize the expected number of fish that he will catch.

3. Let them know that this is a dynamic programming question. They need to think about what they need to store in the table, what choices they have to make, and what information they need to make that choice.

   What are the choices that John must make? At each 5 minute interval, John must decide if he will stay at this lake or move on to the next lake. These are the only two choices.

   To get an dynamic programming algorithm they must do the following:

   - Show that for each choice they want to do the optimal on the resulting subproblem.
   - Give a recurrence relation.
   - Prove the recurrence relation is correct.
   - Show how to extract the solution from the table.
   - Give the running time of the algorithm.

   They should now try to complete these tasks.

4. Here is a likely recurrence. If they do not come up with it, or something like it, show it to them so they can see that dynamic programming requires more than "optimize each subproblem and take the max". We actually have to think about how to combine the optimal subproblems.

   Let $T[l,t]$ be the maximum number of fish caught if we are at lake l and time t. So

   $$T[l,t] = max\{T[l-1,t-1], T[l,t-1] + \text{ fish caught at this lake}\}$$

   If we maximize each subproblem, do we get the maximum number of fish? In order words, if we are at lake l at time t, do we want to maximize the number of fish caught before this time?

   No! The amount of fish we catch at this lake depends on how long we have been here. If we just arrived, we will catch more fish, and so it may make sense to not maximize the number of fish if we can get to this lake later. We need to adjust how we are thinking about the problem.

   However, we can change the argument: If we are at lake l at time t, and we have been at this lake for d intervals already, then we do want to maximize the amount of fish caught before. That is because the amount of fish we can catch at this moment is fixed, and if we catch less than the maximum up to this point, we can replace that with a plan that catches the maximum and we catch more fish. That tells us we need a 3-dimensional table!

5. Here is a correct recurrence

   Let $T[t, l, d]$ be the number of fish caught if John is at lake l at time interval t, and has been at this lake for d intervals. For $T[t, l, 0]$ we want to come from lake $l-1$ at t_l intervals ago, but we don't know how we should have stayed at that lake, so we look at all possibilities. For $T[t, l, d]$ with $d > 0$ we know we stayed at the lake.

$$T[t, l, 0] = \max_{d \in \{0,\ldots,12h\}} \{T[t - t_{l-1}, l-1, d]\}$$
$$T[t, l, d] = T[t, l, d-1] + f_l - \max\{f_l, d_l * (d-1)\}$$

   We need some initial values in our table. We both need to set that we are starting at lake 1 at time interval 0. We also need to prevent John going to any lake faster than he can travel there. We can either have if statements in the loops to prevent illegal values, or we can put error values into the table.

$$T[0, 1, 0] = 0$$
$$T[t, l, d] = -\infty \quad \text{if } t < \sum_{i=1}^{l-1} t_i$$

6. The proof of correctness is as follows:

   Assume $T[x, l-1, d]$ stores the maximum number of fish caught at time x, lake $l-1$ and y intervals spent at lake $l-1$ for all $x < t$ and all d. The maximum number of fish at time t, lake l, and interval 0 is the maximum we could have achieved from the previous lake. This is $\max\{T[t - t_{l-1}, l-1, d]\}$, and by the I.H. the table stores the maximum value so $T[t, l, 0]$ stores the maximum value.

   Likewise, assume $T[t-1, l, d-1]$ stores the maximum number of fish we can catch by time $t-1$ if we are at lake t and we have been nere for $d-1$ intervals. If we are at time t, lake l and inteval d for $d > 0$, we will catch $f_l - (d-1)d_l$ fish this interval, and to maximize the total number of fish, we need to maximize the fish caught at time $t-1$, lake l, and duration $d-1$. By the induction hypothesis, this value is stored in $T[t-1, l, d-1]$.

7. How do we get the solution out of the table?

   The maximum number of fish John can expect to catch is found by searching the entire table for the largest value. Then, we can work backwards from that value. Suppose $T[t, l, d]$ is the largest value. Then we know John ended his day spending d intervals at lake l. That means John arrived at lake l at time $t-d$, and we record that John must leave lake $l-1$ at time $t-d-t_l$. At what time did he arrive at lake $l-1$? Let $t' = t-d$, the time that John arrived at lake l. We look at $T[t' - t_{l-1}, l-1, d^*]$ for all possible d^* and find the one that has the same value as $T[t', l, 0]$. That tells us how long we were at lake $l-1$. Repeating this process will give us the schedule for John.

8. What is the running time?

   The table is size $n \times 12h \times 12h$ (because there are 12 5-minute intervals in an hour) and it takes up to $12h$ steps to fill in each table value. Thus the running time is $\Theta(nh^3)$.

Subject: Looking for student interested in mobile app development
From: Harold Connamacher <hsc21@case.edu>
To: CDS-UG <cds-ug@case.edu>
Date Sent: Wed, 25 Sep 2019 10:31:45 -0400
Date Received: Wed, 25 Sep 2019 07:32:10 -0700 (PDT)

A local community college is developing a mobile app to help with suicide prevention and mental health promotion. They are looking for a student who does iOS and Android development and is available to do a few freelance hours of work per week. The job will involve customizing the app for different campuses. If you are interested, please send me an email.

Regards,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

Subject: Breakout session notes and rubric for tomorrow
From: Harold Connamacher <hsc21@case.edu>
To: Colby Saxton <cas264@case.edu>,Aimee Chau <axc693@case.edu>,Sheng Guan <sxg967@case.edu>,Brendan Dowling <bld43@case.edu>,Huixian Yang <hxy298@case.edu>,Jian Liu <jxl2184@case.edu>,Seohyun Jung <sxj393@case.edu>
Date Sent: Sun, 22 Sep 2019 18:43:16 -0400
Date Received: Sun, 22 Sep 2019 15:43:34 -0700 (PDT)
Attachments: breakout3.pdf

Dear 340 TAs,

Please stop by and pick up the quiz tomorrow. My day is a little crazy, but you can find me at these times:
  between 10-11 in my office
  between 12-12:15 in my office
  right at 2pm in my office before I go to class
  as soon as my class ends at 3:05 in Schmitt Lecture Hall

The breakout notes are attached, and here is the solution to the quiz for tomorrow.

Technically, to be completely correct, the answer should also show how to find the k-shot strategy for the table. However, if the students give a correct recurrence, proof, and runtime, give them full marks, but write on their quiz that they should not forget to include creating the stategy from the table.

Let T[i,j] be the maximum profit we can earn if we do j buys and sells and the last sell occurred at time i or earlier. We let T[i,0] = 0 for all i,
T[i,j] = max{T[i-1,j], max{T[x, j-1] + 1000(p(x+1)-p(i))} for all 2(j-1)<=x< i}

Prove that T[i,j] is the maximum profit from making j buys and sells that complete by time i. Assume T[i-1,j] stores the maximum profit from making j buys by time i-1 and T[x,j-1] stores the maximum profit from making j-1 buys by time x for all times less than i. We have two choices. If we do not complete a sale at time i, then the best we can do is the maximum profit at time i-1, and by the induction hypothesis, T[i-1,j] stores this. If we complete a sale at time i, then we purchased those shares at time some time x+1, and we must have completed the previous j-1 sale at some time at or before x. Since we are buying at time x+1, the want the maximum profit possible at time x, and by the induction hypothesis T[x,j-1] stores this value. We take the maximum over all these choices.

Here is how to get the solution from the table. This will be needed in the homework, but don't deduct points if the students forget this in the quiz. The solution is the largest value T[n,m] for all m from 0 to k. We then can work backwards from here. Start with s=n and t=k. From T[s,t], if T[s,t] = T[s-1,t] then we did not complete a sale at time t. Otherwise, we go search for all x from 2(t-1) to s for the T[x,t-1] that stores T[s,t] - 1000(p(x+1,t-1), and that means we bought at time x+1 and sold at time t. We then set s=x and t=t-1 and repeat. (Another solution is to have a second table that stores the choices made at each step. If T[i,j] = T[i-1,j], the set S[i,j] = 0 to indicate no sale. Otherwise, set S[i,j] = x+1 to record that we bought at time x+1 and sold at time j.

The size of the table is n x k, and it takes up to time n to go back to find each value for a running time of O(n^2 k).

Regards,

Harold Connamacher

Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, September 23, 2019

1. Begin by handing out the quiz. The students are to work alone with no notes, phones, etc. Collect the quiz at 3:40 so that you have 30 minutes to do the exercise below.

2. Ask the students to get in groups of 2 or 3 so they can work together to solve this problem: You have a set of n DNA samples. You want to see if there is a variant of a particular gene that exists in a majority of the n samples. (We simplify this to say a gene is a particular location on a chromosome.) Because of how DNA has lots of variants, all you are able to do is run a test on two samples, and the test returns *true* if the two samples have the same gene variant on the chromosome in question and *false* if they have a different gene variant. Design an algorithm that determines if there is a gene variant that exists in a majority (50% + 1) of the n samples.

3. As last week, have them work in groups, and then periodically have the groups share what they think the next step should be. The goal is for the groups working as separate groups and as the class as a whole figure out most of the task without your telling them. Here are the key points to try to lead them to.

4. First, brute force is a $O(n^2)$ algorithm, and so the goal is to beat brute force. The next question is what should be returned? Simply returning *true* or *false* is not very descriptive. The students should realize that they should return either a DNA sample that represents the majority gene or an indication that no such majority exists.

5. The algorithm is to split into halves, find a majority element of the first sample and one of the second (if they exist). Then you need to check the up to 2 candidate majority genes against all n samples to see if any is $50\% + 1$. Then return that element or indicate that there is no majority element.

6. The run time is $T(n) = 2T(n/2) + cn$ for some constant c, and by the Master Theorem (or by tracing the recursion tree or by noticing this is the same recursion as mergesort) the runtime is $\Theta(n \log n)$.

7. The proof is by induction. The key part is to show that a majority element of the whole set must be a majority in one of the two halves.

   Assume the algorithm correctly returns a sample containing the majority gene or indicates that there is no majority gene on a set of up to n genes. Consider a set of $n+1$ genes. Split it in half, and by induction the algoritm correcty returns a sample containing the majority gene of each half. Since we are comparing the sample returned against all the n samples, clearly if any of the two have a majority gene, we are done. Suppose there is a majority gene that is not in one of the (up to) two samples returned. Since that sample is not majority in the first half, it occurs in at most $\frac{1}{2}\left\lfloor\frac{n+1}{2}\right\rfloor$ of the first half samples. Since that sample is not majority in the second half, it occurs in at most $\frac{1}{2}\left\lceil\frac{n+1}{2}\right\rceil$ of the second half samples. So, that gene will occur in at most

$$\frac{1}{2}\left(\left\lfloor\frac{n+1}{2}\right\rfloor + \left\lceil\frac{n+1}{2}\right\rceil\right) = \frac{n+2}{2}$$

   of the $n+1$ samples and is not a majority gene.

Subject: Re: Quiz solution and breakout session notes
From: Harold Connamacher <hsc21@case.edu>
To: Seohyun Jung <sxj393@case.edu>
Cc: Aimee Chau <axc693@case.edu>, Sheng Guan <sxg967@case.edu>, Colby Saxton <cas264@case.edu>, Brendan Dowling <bld43@case.edu>, Huixian Yang <hxy298@case.edu>, Jian Liu <jxl2184@case.edu>
Date Sent: Tue, 17 Sep 2019 02:10:17 -0400
Date Received: Mon, 16 Sep 2019 23:10:36 -0700 (PDT)

Dear Seohyun,

The brute force algorithm for this problem is O(n log n). Their algorithm needs to beat that in order to be considered "correct". Please be sure to write on their quiz if their algorithm fails to beat brute force so they know to look for something better.

Thanks,

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

On Mon, Sep 16, 2019 at 5:45 PM Seohyun Jung <sxj393@case.edu> wrote:

> Dear Prof. Connamacher,
>
> For the quiz the students took today, we noticed that there are a number of people who wrote down less efficient algorithms than the solution that you gave us.
>
> Should we consider these algorithms incorrect?
>
> Best Regards,
> **Seohyun Jung,** *Class of 2020*
> Computer Science B.S./M.S. & Mathematics B.A. Candidate, Case Western Reserve University
> Undergraduate Teaching Assistant, Algorithms
>
> On Fri, Sep 13, 2019 at 9:23 AM Harold Connamacher <hsc21@case.edu> wrote:
>
>> Attached is the breakout notes, and here is the solution to the quiz. I will need a volunteer (or more) to collect the graded quizzes and return them to the class on Wednesday. Students usually show up 10-15 minutes early so please go to the room between 3:05 and 3:10 so you can give the quizzes back as the students come in, and you do not take up much of the lecture time.
>>
>> Answer:
>>
>> Create a set {1,....n} corresponding to all the indeces of the array. Let x be the bits of the missing number. x starts out as empty. Let k=log n. Request the kth bit of each entry at an index in the set. If there are more 1 bits returned than 0's, then the missing number starts with a 0. Append 0 to x and remove from the set all those indeces that returned a 1. Otherwise is starts with a 1 and append 1 to x and remove from the set all those indeces that returned a 0. Decrement k. If k=0 x stores the missing number.
>>
>> The total number of steps is O(n) in the first round, O(n/2) in the second round and so on. This is O(n + n/2 + n/4 + ...) = O(n).

Proof: The base case is when n=1. There will be a single bit, and if we query it we see what is missing. Assume we can find the missing number from a set of k digits numbers. Consider a set of k+1 digit numbers. If the set is even is even there should be the same number of entries that start with a 0 as start with a 1. If the set is odd, there should be one more entry that starts with a 0 as starts with a 1. If there are more 1's than 0's, then we know that the missing number starts with a 0. Otherwise it must start with a 1. We remove all those indeces from the set that corresponds to the majority answer (what the missing number cannot be). Then we have effectively looking at a smaller set of (k-1) digit numbers. By the induction hypothesis, we find the missing number.

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

Subject: Re: Quiz solution and breakout session notes
From: Seohyun Jung <sxj393@case.edu>
To: Harold Connamacher <hsc21@case.edu>
Cc: Aimee Chau <axc693@case.edu>, Sheng Guan <sxg967@case.edu>, Colby Saxton <cas264@case.edu>, Brendan Dowling <bld43@case.edu>, Huixian Yang <hxy298@case.edu>, Jian Liu <jxl2184@case.edu>
Date Sent: Mon, 16 Sep 2019 17:44:49 -0400
Date Received: Mon, 16 Sep 2019 14:45:03 -0700 (PDT)

Dear Prof. Connamacher,

For the quiz the students took today, we noticed that there are a number of people who wrote down less efficient algorithms than the solution that you gave us.

Should we consider these algorithms incorrect?

Best Regards,
Seohyun Jung, *Class of 2020*
Computer Science B.S./M.S. & Mathematics B.A. Candidate, Case Western Reserve University
Undergraduate Teaching Assistant, Algorithms

On Fri, Sep 13, 2019 at 9:23 AM Harold Connamacher <hsc21@case.edu> wrote:

> Attached is the breakout notes, and here is the solution to the quiz. I will need a volunteer (or more) to collect the graded quizzes and return them to the class on Wednesday. Students usually show up 10-15 minutes early so please go to the room between 3:05 and 3:10 so you can give the quizzes back as the students come in, and you do not take up much of the lecture time.
>
> Answer:
>
> Create a set {1,....n} corresponding to all the indeces of the array. Let x be the bits of the missing number. x starts out as empty. Let k=log n. Request the kth bit of each entry at an index in the set. If there are more 1 bits returned than 0's, then the missing number starts with a 0. Append 0 to x and remove from the set all those indeces that returned a 1. Otherwise is starts with a 1 and append 1 to x and remove from the set all those indeces that returned a 0. Decrement k. If k=0 x stores the missing number.
>
> The total number of steps is O(n) in the first round, O(n/2) in the second round and so on. This is O(n + n/2 + n/4 + ...) = O(n).
>
> Proof: The base case is when n=1. There will be a single bit, and if we query it we see what is missing. Assume we can find the missing number from a set of k digits numbers. Consider a set of k+1 digit numbers. If the set is even is even there should be the same number of entries that start with a 0 as start with a 1. If the set is odd, there should be one more entry that starts with a 0 as starts with a 1. If there are more 1's than 0's, then we know that the missing number starts with a 0. Otherwise it must start with a 1. We remove all those indeces from the set that corresponds to the majority answer (what the missing number cannot be). Then we have effectively looking at a smaller set of (k-1) digit numbers. By the induction hypothesis, we find the missing number.
>
> Harold Connamacher
> Associate Professor
> Computer and Data Sciences

Subject: Re: Quiz solution and breakout session notes
From: Roxanne Yang <hxy298@case.edu>
To: Harold Connamacher <hsc21@case.edu>
Date Sent: Mon, 16 Sep 2019 16:23:44 -0400
Date Received: Mon Sep 16 20:23:44 UTC 2019

Hi Professor Connamacher,

One of my students asked me when you will be sending them the homework solutions. I can send them out for you on Canvas if you're busy!

Best,
Roxanne (Huixian) Yang

Case Western Reserve University, Class of 2020
Computer Science B.S. Candidate

On Fri, Sep 13, 2019 at 9:23 AM Harold Connamacher <hsc21@case.edu> wrote:

> Attached is the breakout notes, and here is the solution to the quiz. I will need a volunteer (or more) to collect the graded quizzes and return them to the class on Wednesday. Students usually show up 10-15 minutes early so please go to the room between 3:05 and 3:10 so you can give the quizzes back as the students come in, and you do not take up much of the lecture time.
>
> Answer:
>
> Create a set {1,....n} corresponding to all the indeces of the array. Let x be the bits of the missing number. x starts out as empty. Let k=log n. Request the kth bit of each entry at an index in the set. If there are more 1 bits returned than 0's, then the missing number starts with a 0. Append 0 to x and remove from the set all those indeces that returned a 1. Otherwise is starts with a 1 and append 1 to x and remove from the set all those indeces that returned a 0. Decrement k. If k=0 x stores the missing number.
>
> The total number of steps is O(n) in the first round, O(n/2) in the second round and so on. This is O(n + n/2 + n/4 + ...) = O(n).
>
> Proof: The base case is when n=1. There will be a single bit, and if we query it we see what is missing. Assume we can find the missing number from a set of k digits numbers. Consider a set of k+1 digit numbers. If the set is even is even there should be the same number of entries that start with a 0 as start with a 1. If the set is odd, there should be one more entry that starts with a 0 as starts with a 1. If there are more 1's than 0's, then we know that the missing number starts with a 0. Otherwise it must start with a 1. We remove all those indeces from the set that corresponds to the majority answer (what the missing number cannot be). Then we have effectively looking at a smaller set of (k-1) digit numbers. By the induction hypothesis, we find the missing number.
>
> Harold Connamacher
> Associate Professor
> Computer and Data Sciences
> Case Western Reserve University

Subject: Quiz solution and breakout session notes
From: Harold Connamacher <hsc21@case.edu>
To: Aimee Chau <axc693@case.edu>,Sheng Guan <sxg967@case.edu>,Colby Saxton <cas264@case.edu>,Brendan Dowling <bld43@case.edu>,Huixian Yang <hxy298@case.edu>,Jian Liu <jxl2184@case.edu>,Seohyun Jung <sxj393@case.edu>
Date Sent: Fri, 13 Sep 2019 09:23:29 -0400
Date Received: Fri, 13 Sep 2019 06:23:49 -0700 (PDT)
Attachments: breakout2.pdf

Attached is the breakout notes, and here is the solution to the quiz. I will need a volunteer (or more) to collect the graded quizzes and return them to the class on Wednesday. Students usually show up 10-15 minutes early so please go to the room between 3:05 and 3:10 so you can give the quizzes back as the students come in, and you do not take up much of the lecture time.

Answer:

Create a set {1,....n} corresponding to all the indeces of the array. Let x be the bits of the missing number. x starts out as empty. Let k=log n. Request the kth bit of each entry at an index in the set. If there are more 1 bits returned than 0's, then the missing number starts with a 0. Append 0 to x and remove from the set all those indeces that returned a 1. Otherwise is starts with a 1 and append 1 to x and remove from the set all those indeces that returned a 0. Decrement k. If k=0 x stores the missing number.

The total number of steps is O(n) in the first round, O(n/2) in the second round and so on. This is O(n + n/2 + n/4 + ...) = O(n).

Proof: The base case is when n=1. There will be a single bit, and if we query it we see what is missing. Assume we can find the missing number from a set of k digits numbers. Consider a set of k+1 digit numbers. If the set is even is even there should be the same number of entries that start with a 0 as start with a 1. If the set is odd, there should be one more entry that starts with a 0 as starts with a 1. If there are more 1's than 0's, then we know that the missing number starts with a 0. Otherwise it must start with a 1. We remove all those indeces from the set that corresponds to the majority answer (what the missing number cannot be). Then we have effectively looking at a smaller set of (k-1) digit numbers. By the induction hypothesis, we find the missing number.

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University

EECS 340, Breakout Session Notes, September 9, 2019

1. Begin by handing out the quiz. The students are to work alone with no notes, phones, etc. Collect the quiz at 3:40 so that you have 30 minutes to do the exercise below.

2. Ask the students to get in groups of 2 or 3 so they can work together to solve this problem: You will be driving from Cleveland to Seattle along Interstate 90. You have an internet mapping program that lists the location of every gas station on the route, and suppose that you want to make as few stops for gas as possible. Also, to save time when stopping, you will always have the tank filled completely while you are running into the station to buy supplies. Finally, suppose this is a rental car and the company required you to pay for a full tank of gas so you would like to complete the journey with as little gas left in the tank as possible. Assuming that you know the fuel efficiency of the vehicle, give an efficient algorithm to determine at which gas stations you should stop, in order to minimize both the number of stops you take and the amount of gas remaining in the car when you return the rental.

3. As mentioned in our meeting, have them work in groups, and then periodically have the groups share what they think the next step should be. The goal is for the groups working as separate groups and as the class as a whole figure out most of the task without your telling them. Here are the key points to try to lead them to.

4. The algorithm is greedy. This is like the art guard problem, but now you have to start at the Seattle rental and work backwards. Find the most distant gas station still reachable from your last stop and mark that one to fill up.

5. The proof is to compare it to an optimal solution. Suppose that (from Seattle), G and O use the same stops up to stop k, but now G and O stop at different stations. It is impossible for O to stop at a further station since the car cannot go that far on a single tank of gas. So O stopped at a closer station to Seattle than G. Change O to O' by moving the station stopped to the one G stops at. O' still uses the same number of stations as O, and as the station is moved closer to Cleveland, there is no chance for there to be a gap opened up in the stations between this station and Cleveland where the car can't make it.

6. The running time is O(n) assuming n stations are already pre-given to you by Google/Bing maps.

Subject: Homework grading rubric
From: Harold Connamacher <hsc21@case.edu>
To: Seohyun Jung <sxj393@case.edu>,Brendan Dowling <bld43@case.edu>,Aimee Chau <axc693@case.edu>,Colby Saxton <cas264@case.edu>,Huixian Yang <hxy298@case.edu>,Jian Liu <jxl2184@case.edu>,Sheng Guan <sxg967@case.edu>
Date Sent: Fri, 13 Sep 2019 09:19:56 -0400
Date Received: Fri, 13 Sep 2019 06:20:14 -0700 (PDT)

Dear TAs,

Below is the grading rubric for the homework. What you will do is look for 8 features in in the answer. These are similar to the ones in the quiz. However, unlike the quiz it is not one point per feature. Instead, you note which features their answer has and then you give a point grade based on the rubric that follows. Please do not try to find every mistake in their answer. For example, once you see a mistake in the algorithm, then they did not achieve the "algorithm correct" feature. Now focus on looking for the other features. That prevent the grading from bogging down. If you never graded before, expect the first 10-20 you grade to take a while, but things will speed up as you get used to the student solutions and the rubric.

For each question, check the following points:
1. The algorithm is precise and clear.
2. The algorithm used the correct technique (greedy/DP/divide and conquer) for the problem. It is not enough to say they are using a certain technique. It is very common for a student to say they are using DP, but the algorithm recurrence is actually a greedy algorithm.
3. The algorithm is correct.
4. They provide a proof that has reasonable rigor. For example, there are clear assumptions made. The biggest mistake is for students to just describe what their algorithms do and call that a "proof".
5. They use a proof appropriate for the algorithm type they are trying to write (not necessarily for the correct algorithm).
6. The proof is correct.
7. The correct runtime for their algorithm (not for the correct algorithm) is given.
8. A reasonable justification of the running time is given.

Here is the grade to assign:

8: Perfect. All 8 above achieved.
7: Very good. All 8 points attempted, and at least 6 of the 8 points achieved.
6: Good. Achieved at least 1, 2, 4 and 5.
5: Acceptable. At least two of 1, 2, 4, and 5.
4: Poor. Attempted points 1 and 4.
3: Poor-. Attempted either point 1 or 4.
2: There some attempt at an algorithm.
0: There is nothing useful to grade.

Problem 1:
Place each security guard as far "right" as possible to be able to guard the left most unguarded piece of art. Guard 1 is placed at x_0 + 1. Let g_k be the last placed guard and let x_i be the smallest value such that x_i > g_k + 1. Place g_(k+1) at x_i + 1.

Proof: Let O be an optimal solution and let G be the greedy solution. Assume that O and G place the first k-1 guards at the exact same spots but place guard k at different spots. Note that G places guard k as far to the right of the next unguarded piece of art as possible so G places guard k to the right of O's placement. We can then safely move guard k in O to the right until it matches G's placement. All art that k is guarding before being moved is still be guarded by k after the move. The number of guards used by O is unchanged and every piece of art is still guarded. Running time is O(n) because we traverse the points X in order (if the student assumes the painting positions are given in order). The running time is O(n log n) if the student assumes they have to sort the painting positions.

Problem 2:
Let T[k] store the maximum profit possible after k weeks.
T[k] = max { T[k-1] + l_k, T[k-2] + h_k }.
T[0] = 0 and T[-1] = 0 or -infinity, depending on if you want to allow a high stress job on week 1. The solution max profit is at T[n], and to get the schedule, we work backwards from k=n. If T[k] = T[k-1] + l_{k} then we did the low paying job at time k and set k:=k-1. Otherwise we did the high paying job at time k and no job at time k-1, and we set k:=k-2 and repeat.

Proof: Assume T[k-1] and T[k-2] store the maximum profit possible after week k-1 and week k-2 respectively. We have two options on week k. If we take the high stress job, we can't work week k-1. Since it does not matter what jobs we work before week k-1, we should maximize the profit, and by the induction hypothesis, T[k-2] holds this value. If we are working a low stress job, then it does not matter what jobs we do before this job so we should maximize the profit possible. By the induction hypothesis, T[k-1] holds this value.

The running time is O(n) since the table has n entries and it takes O(1) to calculate each entry.

Problem 3:
Using divide and conquer, I split the n teams into two and create a round robin schedule for each. Since there is no overlap in teams, I combine each day of the first half schedule with each day of the second half schedule. Then, I "rotate" the teams of the first group with the teams of the second group, matching each up. Each step of the rotation produces n/2 games that can be played simultaneously so I schedule them all on the same day.

Proof: Assume we can correctly round robin schedule for fewer than n teams. Consider n teams split into two even-ish groups. The "rotation" correctly schedules teams in the first set against all teams in the second set in a minimal number of days. All that is left is to correctly schedule the games for teams within each set, and by the induction hypothesis, we do that.

The running time is the recurrence T(n) = 2T(n/2) + n^2 = O(n^2).

Problem 4:
Using divide-and-prune, let L1=L2=0 and H1=H2=n. Repeat, let M1=(L1 + H1)/2 and M2=(L2+H2)/2, and we request the M1-th largest element of database 1 and the M2-th largest element of database 2. Let a1 be the element returned from the first and a_2 the element returned from the second. Suppose (w.l.o.g.) a_1 =< a_2, if n is odd, we set L1=M1 and H2=M2. If n is even, we set L1=M1+1 and H2=M2. Then we repeat.

Proof: Assume the algorithm correctly returns the median element two databases with H2'-L2' = H1'-L1' < n elements each where the indeces start at L1' and L2' respectively. Given two databases with n elements each and indices starting at L1 and L2, request the (n/2)th element of each. Suppose (w.l.o.g.) that a_1 <= a_2. If n is even then there are at most 2n-2 elements less than a_1 so a_1 can't be the median. There are also at most n elements larger than a_2 so no element larger than a_2 can be the median. We set L1=M1+1 since no element of database <= a_1 can be the median and H2=M2 because no element larger than a_2 in database 2 can be the median. Now both databases have the same number of element less than n. Since we remove the same number of elements above and below the median, the median of this smaller database is the median of the whole. So we repeat and by induction hypotheses, we get the median. If n is odd, then there are at most n-1 elements smaller than a_1 and at most n-1 elements larger than a_2. Thus no element smaller than a_1 or larger than a_2 can be the median. We set

L1=M1 and H2=M2, and we remove the same number of elements from both databases. By the same induction as before, the median of the recursive call is the median of the whole, and by the induction hypothesis that is what we get.

Problem 5:
Let T[i, d] store the maximum number of events we can photograph if the telescope is at location d at time i.
Set T[0,0] = 0 and T[0,k] = -infinity for all k <> 0; T[k, 86] = -infinity for all k, and T[k, -86] = -infinity for all k
T[i, d] = max {T[i-1,d+1], T[i-1,d-1], T[i-l,d] + {1 if there is an event at location d and time i-1, 0 otherwise}}
The solution is in T[n, d*] where d* is the location of the last celestial event. To get the schedule, we work backwards from T[n.d*]. For each position T[i,d], if T[i-1,d+1] or T[i-1,d-1] equaled to T[i,d], then we must have moved the telescope to this location, otherwise we took a picture at this time.

Proof: Assume T[i-1,d-1], T[i-1,d+1], and T[i-1,d] store the maximum number of events we can shoot by minute i-1 and locations d, d-1, and d+1. We have three options. Move the telescope up, move it down, and stay and take a picture. Taking the picture gives as an event only if one is occuring where and when we are pointing. Because only the current location of the telescope matters and now what pictures we may or may not have taken before, it makes no sense to take less than the max number possible. By the induction hypothesis, T[i-1,d-1], T[i-1,d+1], and T[i-1,d] store these values.

Problem 6: The algorithm is correct. For grading this one, you can assume the algorithm and running time points are automatically achieved if they are proving the algorithm correct.

Proof: Assume there is an optimal truck loading scheme O. Assume our greedy does the same as O for the first k boxes, but on box k, we do something different. The two options are G sends the truck after loading k when O did not or O sends the truck after loading k but G does not. The first is not possible since package k+1 will not fit in the truck. So O sends the truck. Let us create O' where we do the same as O but load box k into the truck instead of sending it. That removed box k from the next truck reducing the load of that truck. Therefore O' uses the same number of trucks as O, no truck is overloaded, and G and O' now do the same on boxes up to and including k.

Harold Connamacher
Associate Professor
Computer and Data Sciences
Case Western Reserve University