CSORW4231_001_2022_1 - ANALYSIS OF ALGORITHMS I

hw3

Benjamin Jenney

TOTAL POINTS

138 / 170

QUESTION 1

Problem 1 55 pts

1.1 Part (a) **30 / 30**

1.2 Part (b) **25 / 25**

QUESTION 2

2 Problem 2 **20 / 20**

QUESTION 3

Problem 3 30 pts

3.1 Part (a) **15 / 14**

3.2 Part (b) **13 / 16**

QUESTION 4

4 Problem 4 **8 / 20**

1 Should be 1-(1-p)^n

QUESTION 5

Problem 5 45 pts

5.1 Part (a) **5 / 5**

5.2 Part (b) **5 / 5**

5.3 Part (c) **10 / 10**

5.4 Part (d) **2 / 10**

5.5 Part (e) **0 / 15**

QUESTION 6

6 Extra Credits **5 / 0**

Analysis of Algorithms
HW 3

bej2117

March 2022

Problem 1

(a) Clearly define the subproblems in English.
(b) Explain the recurrence in English. (This counts as a proof of correctness; feel free to give an inductive proof of correctness too for practice but points will not be deducted if you don't.) Then give the recurrence in symbols.
(c) State boundary conditions.
(d) Analyze time.
(e) Analyze space.
(f) If you're filling in a matrix, explain the order to fill in subproblems in English.
(g) Give pseudocode.

Part A

Can we make change with the given denominations such that the sum over the set of chosen coins equals v with repetitions?

Subproblems
Much like the single source shortest paths problem if we know a shortest path from s to t, we know we can remove an incoming edge to t, say $(x, t) \in E$, that is on that path and we can rest assured that the path s to x is a shortest path and that $weight(s \text{ to } x) + weight(x, t)$ is the weight of the shortest path s to t should we reconstruct the shortest path s to t. Similarly if we had a set of denominations who's sum was V, and we removed one of those denominations, call the removed denomination c_i, from the set, we know the sum of the remaining set would be the answer to the subproblem $V - c_i$.

(1) Claim $V - c_i$ is an optimal solution. if $sum(c_1, c_2, ..., c_{i-1}, c_i) = V$, then $sum(c_1) = V - sum(c_2, ..., c_{i-1}, c_i)$ and $sum(c_1, c_2, ..., c_{i-1}) = V - sum(c_i)$. and so on. Thus $V - c_i$, is an optimal solution. Notice that we only care that the given denominations sum up to V so any one optimal solution is just as good as any other, as long as the set of chosen coins adds up to V. Furthermore

since addition is commutative order does not matter.

Boundary Condition Since a boundary condition for $V = 0$ is so natural we will use a bottom-up approach in which $OPT(V) = True$ if $\exists sum(\{c_1..c_i)\} = V$ for $i = 1, 2, ..., V$. Clearly, if $V = 0$, then trivially $OPT(V) = True$ since we can always take a set with no coins.

Recurrence The recursion for this is straightforward and is very similar to the knapsack problem with repetitions. If we choose a coin then that coin $M[i - C[j]] = 1$ in the first iteration say $V_i = 1$, and $c_j = 1$ then $M[1] = M[0]$ which is our boundary condition which means $M[1] = 1$, which means c_1 was a solution to a sub problem. Else we move on: $M[i = V_i - C[j]] = 0$. By the end $M[V] = 1$ if there is a way to make V with denoms $c_1...c_n$.
(2)

$$M[j] = \begin{cases} M[i - C[j]], \text{ if } i \geq C[j] \\ False, \text{ otherwise} \end{cases}$$

Where i enumerates $1, 2, ..., V$ and j is a coin in $c_1, c_2, ..., c_n$. WE have to check that $i \geq C[j]$ since obviously if $C[j] \geq V - c_j$ then $C[j]$ can not be part of a sum that equals V.

From (1) and (2) we can build our pseudocode:

Algorithm 1: *makeChange*$(C = [c_1, ..., c_n], V)$

```
1  Let M, prev be arrays of size V;
2  Let M[0..V] = False; // O(V)
3   Let prev[0..V] = 0; // O(V)
4   M[0] = True; // boundary condition
5   ind = 0 for i = 1, 2, ..., V do
6   |  for j = 1, 2, ..., n do
7   |  |  if i ≥ C[j] then
8   |  |  |  M[j] = M[i − C[j]]
9   |  |  end
10  |  |  ind = j
11  |  end
12  |  prev[i] = ind
13 end
14 if M[n] == True then
15 |  Reconstruct(C, prev, V)
16 end
```

We can then reconstruct the values recursively using the values stored in M using *prev*, and working our way backwards through the solution path.

Algorithm 2: *Reconstruct*$(M, prev, V)$

```
1 if V <= 1 then
2 |  return;
3 end
4 print(M[prev[V]], end =',' );
5 Reconstruct(M, prev, V − M[prev[V]])
```

Correctness *Reconstruct* walks backwards through the dp dag. Prev remembers the indices for each subproblem If $M[prev[V]] = 1$ then $C[j]$ was a solution to a subproblem, and $V - M[prev[V]]$ is the value of $C[j]$. Notice we do not run Reconstruct if $M[n] \neq 1$, as no solution was found.
time complexity
Clearly initializing the arrays takes $O(n) + O(n)$ and the Two for loops take $O(Vn)$, Reconstruct takes $T(V) = T(V-1) + O(1) = \emptyset(V)$ in the worst case. $\emptyset(Vn)$ clearly dominates, therefore $makeChange = \Theta(Vn)$ in the worst case.
space
Clearly, M is $O(V)$ and prev is $O(v)$, giving a total of $O(V)$

part B

(a) Clearly define the subproblems in English.
Basically the same argument from part A. If $Sum(c_1, c_2, ..., c_{i-1}, c_i) = V$, then clearly $Sum(c_1, c_2, ..., c_{i-1}) = V - c_i$. **(b) Explain the recurrence in English. Then give the recurrence in symbols.**
For each coin you can move back in the table to see if an already usable coin $< i$ has already been a solution or if the current coin is already usable, if neither then M[i][j] will be false. Clearly if i ¡ C[j] then we can only be true if the current coin was already a solution to a subproblem else it will be false.

$M[n][v] = 1$ if coins chosen sum to V.
$M[i][j] = M[i][j] = M[i][j-1]$ if $i >= c_j$, else $M[i][j] = M[i - C[j-1]][j - 1] or M[i][j-1]$

(c) State boundary conditions.
M[0][0:n] = 1
M[0:V][0] = 0
(d) Analyze time.
Subproblems = nV, time per subproblem = O(1) therefore $O(nV)$.
(e) Analyze space.
clearly we have an V by n table so $O(nV)$.
(f) If you're filling in a matrix, explain the order to fill in subproblems in English.
the outer loop goes through the coins by row, the inner goes from 1 to V filling up column by column.
(g) Give pseudocode.

1.1 Part (a) **30 / 30**

Algorithm 2: *Reconstruct*$(M, prev, V)$

```
1 if V <= 1 then
2 |  return;
3 end
4 print(M[prev[V]], end =',' );
5 Reconstruct(M, prev, V − M[prev[V]])
```

Correctness *Reconstruct* walks backwards through the dp dag. Prev remembers the indices for each subproblem If $M[prev[V]] = 1$ then $C[j]$ was a solution to a subproblem, and $V - M[prev[V]]$ is the value of $C[j]$. Notice we do not run Reconstruct if $M[n] \neq 1$, as no solution was found.
time complexity
Clearly initializing the arrays takes $O(n) + O(n)$ and the Two for loops take $O(Vn)$, Reconstruct takes $T(V) = T(V-1) + O(1) = \emptyset(V)$ in the worst case. $\emptyset(Vn)$ clearly dominates, therefore $makeChange = \Theta(Vn)$ in the worst case.
space
Clearly, M is $O(V)$ and prev is $O(v)$, giving a total of $O(V)$

part B

(a) Clearly define the subproblems in English.
Basically the same argument from part A. If $Sum(c_1, c_2, ..., c_{i-1}, c_i) = V$, then clearly $Sum(c_1, c_2, ..., c_{i-1}) = V - c_i$. **(b) Explain the recurrence in English. Then give the recurrence in symbols.**
For each coin you can move back in the table to see if an already usable coin $< i$ has already been a solution or if the current coin is already usable, if neither then M[i][j] will be false. Clearly if i ¡ C[j] then we can only be true if the current coin was already a solution to a subproblem else it will be false.

$M[n][v] = 1$ if coins chosen sum to V.
$M[i][j] = M[i][j] = M[i][j-1]$ if $i >= c_j$, else $M[i][j] = M[i - C[j-1]][j - 1] or M[i][j-1]$

(c) State boundary conditions.
M[0][0:n] = 1
M[0:V][0] = 0
(d) Analyze time.
Subproblems = nV, time per subproblem = O(1) therefore $O(nV)$.
(e) Analyze space.
clearly we have an V by n table so $O(nV)$.
(f) If you're filling in a matrix, explain the order to fill in subproblems in English.
the outer loop goes through the coins by row, the inner goes from 1 to V filling up column by column.
(g) Give pseudocode.

Algorithm 3: $CanMakeChange2(C = [c_1, c_2, ..., c_n], v)$

```
let n = len(C);
Let M be a matrix of size M[n + 1, V + 1];
for j in range(n+1) do
    M[0][j] = 1;
end
for i in range(V+1) do
    M[i][0] = 0;
end
for j in range(1,n+1) do
    for i in range(1,V+1) do
        if i < C[j − 1] then
            M[i][j] = M[i][j-1];
        end
        else
            M[i][j] = M[i][j-1] or M[i - C[j-1]][j-1];
        end
    end
end
return M[n][v]
```

Problem 2

We shall show that the arbitrage problem is polynomial reducible to detecting negative edge weights.
Let X = the Arbitrage Problem and Y = negative weight cycle detection.
Claim 1: The table R is a graph
R is an adjacency matrix. This is easy to see if you consider that R[i][j] corresponds to a weight for a currency c_i, c_j, then i, j correspond to nodes c_i, c_j and (c_i, c_j) form an edge. Therefore $G = R$ s.t. each (i,j) pair form an edge with weight R[i][j] for i,j from 1 to n. Notice we are only changing the input by name R itself is a graph. **Claim 2:** an input for the arbitrage problem x for X is equivalent to an input y for Y for the negative edge detection problem. Per the problem statement an arbitrage opportunity exists $R[i_1, i_2] * R[i_2, i_3] * ... * R[k_1, i_k]R[i_k, i_1] > 1$.

$$R[i_1, i_2] * R[i_2, i_3] * ... * R[i_{k-1}, i_k]R[i_k, i_1] > 1$$
$$\leftrightarrow ln(R[i_1, i_2]) + ln(R[i_1, i_2]) + ... + ln(R[i_{k-1}, i_k]) + ln(R[i_k, i_1]) > 1$$
$$\leftrightarrow -ln(R[i_1, i_2]) - ln(R[i_1, i_2]) - ... - ln(R[i_{k-1}, i_k]) - ln(R[i_k, i_1]) < 0$$

Clearly, this shows that for an input x an arbitrage opportunity exists if and only if a negative weight graph exists.

1.2 Part (b) **25 / 25**

Algorithm 3: $CanMakeChange2(C = [c_1, c_2, ..., c_n], v)$

```
let n = len(C);
Let M be a matrix of size M[n + 1, V + 1];
for j in range(n+1) do
    M[0][j] = 1;
end
for i in range(V+1) do
    M[i][0] = 0;
end
for j in range(1,n+1) do
    for i in range(1,V+1) do
        if i < C[j − 1] then
            M[i][j] = M[i][j-1];
        end
        else
            M[i][j] = M[i][j-1] or M[i - C[j-1]][j-1];
        end
    end
end
return M[n][v]
```

Problem 2

We shall show that the arbitrage problem is polynomial reducible to detecting negative edge weights.
Let X = the Arbitrage Problem and Y = negative weight cycle detection.
Claim 1: The table R is a graph
R is an adjacency matrix. This is easy to see if you consider that R[i][j] corresponds to a weight for a currency c_i, c_j, then i, j correspond to nodes c_i, c_j and (c_i, c_j) form an edge. Therefore $G = R$ s.t. each (i,j) pair form an edge with weight R[i][j] for i,j from 1 to n. Notice we are only changing the input by name R itself is a graph. **Claim 2:** an input for the arbitrage problem x for X is equivalent to an input y for Y for the negative edge detection problem. Per the problem statement an arbitrage opportunity exists $R[i_1, i_2] * R[i_2, i_3] * ... * R[k_1, i_k]R[i_k, i_1] > 1$.

$$R[i_1, i_2] * R[i_2, i_3] * ... * R[i_{k-1}, i_k]R[i_k, i_1] > 1$$
$$\leftrightarrow ln(R[i_1, i_2]) + ln(R[i_1, i_2]) + ... + ln(R[i_{k-1}, i_k]) + ln(R[i_k, i_1]) > 1$$
$$\leftrightarrow -ln(R[i_1, i_2]) - ln(R[i_1, i_2]) - ... - ln(R[i_{k-1}, i_k]) - ln(R[i_k, i_1]) < 0$$

Clearly, this shows that for an input x an arbitrage opportunity exists if and only if a negative weight graph exists.

Claim 3: The transformation from x to y can be done in polynomial time.

Algorithm 4: Transform(G)

```
1 for i = 1..n do
2 |   for j = 1..n do
3 |   |   G[i][j] = -ln(G[i][j]);
4 |   end
5 end
```

Correctness
Transform goes row by row transforming each weight in G, updating all values in G in accordance with claim 2. It's correctness is apparent.
Time
Clearly, Transform(G) takes $\Theta(n^2)$ time.

Claim 4: Assuming Claim 1,2,3 are correct than we can write an algorithm that gives a yes/no answer to the arbitrage problem by detecting a negative weight.

Algorithm 5: Forditrage(G)

```
1  Let dist be a n array set to infinity; // O(n)
2  for k in range(n) do
3  |   for i in range(n) do
4  |   |   for j in range(n) do
5  |   |   |   update(i,j,G[i][j]) // (
6  |   |   |   O(1))
7  |   |   end
8  |   end
9  end
10 for i in range(n) do
11 |   for j in range(n) do
12 |   |   if dist[j] > dist[i] + G[i][j] then
13 |   |   |   return True
14 |   |   end
15 |   end
16 end
17 return False
```

Correctness
Forditrage is essentially just Bellman Ford for an adjacency matrix, it's correctness follows from proofs of Bellman-Ford's correctness from class and in CLRS.

time
The three nested for loops dominate. Bellmanford is (nm) in the worst case and we have n nodes and m = nxn edges in G which is R. Therefore Forditrage is $\Theta(n^3)$ in the worst case.

By way of correctness we have shown claims 1,2,3,4 to be correct. And have shown that X is polynomial reducible to Y. ■

2 Problem 2 **20 / 20**

0.1 Problem 3

0.2 Part A

With help from T.A. Yushan An
For Bob's claim to be true two things must be true:

(1) d[v] must equal the true wait of from Alices graph: $\delta(v) = d[u] + weight(\forall(v,u) \in E)$ in the proposed $P = v - t$ Path i.e for every outgoing edge from v in P.

(2) As such the path P must also exist.

In order to check if Bob's values can be in P for all v, we can iterate through all the edges once for all edges in G checking if $d_v = \delta(v)$. We can then Form a new graph, *bobsGraph*, and run a BFS *from* t to see if the graph is connected. If Bob is right then BFS will visit each node in *bobsGraph* and $d_v = \delta(v)$ for all v in P. This BFS will be exactly the same as the one covered in class except at the end of execution it will return the *explored* array.

Algorithm 6: *CheckBob*$(G = (V, E), dist, t)$

```
1  Let bobsGraph be an empty adjacency list representation of a graph;
   // clearly the below forloop is O(m)
2  for each (v,u) ∈ G.E do
3  |  if dist[v] > dist[u] + weight((v,u)) then
4  |  |  return "Bob is wrong"
5  |  end
6  |  if dist[v] == dist[u] + weights((v,u)) then
7  |  |  bobsGraph[v].append(u) // include edge in bobsGraph
8  |  end
9  end
   // transpose the graph.  O(n+m)
10 for v in range(len(bobsGraph)) do
11 |  for u in range(len(bobsGraph[v])) do
12 |  |  bobsGraph[u].append(v)
13 |  end
14 end
15 explored = BFS(bobsGraph = (V,E), t); // O(n+m)
16 for i in range(explored) do
17 |  if explored[i] == ∞ then
18 |  |  return "Bob is wrong"
19 |  end
20 end
21 return "Bob is right"
```

Correctness
The correctness of *CheckBob* follows from the Correctness of the Optimality of shortest paths. By exploring each edge in G we ensure to check all edges

in for all edges in all $v - t$ paths in G, if for any of those edges $dist[v] > dist[u] + weight((v,u))$ then $\delta(v) = dist[u] + weight((v,u)$, which is to say $dist[v] \neq \delta(v)$. Thus Bob must be incorrect and reject. If Bob is correct then $dist[v] = \delta(v)$ and if the new graph *bobsGraph* is connected, then the $v - t$ of those edges such that $dist[v] = \delta(v - t)$.

Part B

Having read the portion the chapter on all pairs shortest paths in CLRS, I was already aware of the solution to this question. Essentially we could just use Lemma 25.1 from pg. 701. to show that we can re-weight the edges while maintaining integrity of shortest paths. I will reproduce the proof here, as it is elegant and I understand it. I also give a brief intuitive understanding of the equation $w((v,u))' = w((v,u)) + d(v) - d(u)$.

Claim: for all pairs of vertices $(v,u) \in V$, a path p is shortest from u to v using weight function w iff p is also a shortest path using w'.

Let $w(p)' = w(p) + d(v) - d(u)$. then

$$\begin{aligned} w(p)' &= \Sigma_{i=1}^{k} w'(v_{i-1}, i) \\ &= \Sigma_{i=1}^{k} (w(v_{i-1}, i) + d(v) - d(u)) \\ &= \Sigma_{i=1}^{k} ((w(v_{i-1}, i)) + d(v) - d(u) (\text{prop. A.9}) \\ &= w(p) + d(v_0) - d(v_k). \end{aligned}$$

Therefore any path p from v_0 to v_k has $w'(p) = w(p) + d(v) - d(u)$. Furthermore, since "$d(v_0)$ and $d(v_k)$ do not depend on the path, if one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using w'."(pg. 702). Therefore, $w(p) = \delta(v_0, v_k)$ iff $w'(p) = \delta(v_0, v_k)$. ■

Intuitively the equation $w(p)' = w(p) + d(u) - d(v)$ suggests that if you have an edge from from from v to u and an outgoing edge from u you can take the difference of their weights they balance out so to speak. The proof above shows that this method will produce the same shortest graphs in G and the reweighted graph G'.

Psuedocode

With our formula we can make a new graph G' with t' and then add all edges and vertices from G s.t $E' = \{G.E \cup (t', v) \forall v \in V\}$ with weights = 0. Since t' is a source node with no incoming edges and outgoing edges equal to 0, shortest paths in G' will be the same as in G. finally since $d(v)$ is the weight of a shortest path up to v, then $d(v) \leq d(u) + w(v,u)$ which gives us $0 \leq d(u) + w(u,v) - d(v) = w'(v,u)$.

we can then use the weight function w' to update all the weights for G' We can then run $Dijkstra(G', w', t')$

3.1 Part (a) **15 / 14**

in for all edges in all $v - t$ paths in G, if for any of those edges $dist[v] > dist[u] + weight((v,u))$ then $\delta(v) = dist[u] + weight((v,u)$, which is to say $dist[v] \neq \delta(v)$. Thus Bob must be incorrect and reject. If Bob is correct then $dist[v] = \delta(v)$ and if the new graph *bobsGraph* is connected, then the $v - t$ of those edges such that $dist[v] = \delta(v - t)$.

Part B

Having read the portion the chapter on all pairs shortest paths in CLRS, I was already aware of the solution to this question. Essentially we could just use Lemma 25.1 from pg. 701. to show that we can re-weight the edges while maintaining integrity of shortest paths. I will reproduce the proof here, as it is elegant and I understand it. I also give a brief intuitive understanding of the equation $w((v,u))' = w((v,u)) + d(v) - d(u)$.

Claim: for all pairs of vertices $(v,u) \in V$, a path p is shortest from u to v using weight function w iff p is also a shortest path using w'.

Let $w(p)' = w(p) + d(v) - d(u)$. then

$$\begin{aligned} w(p)' &= \Sigma_{i=1}^{k} w'(v_{i-1}, i) \\ &= \Sigma_{i=1}^{k} (w(v_{i-1}, i) + d(v) - d(u)) \\ &= \Sigma_{i=1}^{k} ((w(v_{i-1}, i)) + d(v) - d(u) (\text{prop. A.9}) \\ &= w(p) + d(v_0) - d(v_k). \end{aligned}$$

Therefore any path p from v_0 to v_k has $w'(p) = w(p) + d(v) - d(u)$. Furthermore, since "$d(v_0)$ and $d(v_k)$ do not depend on the path, if one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using w'."(pg. 702). Therefore, $w(p) = \delta(v_0, v_k)$ iff $w'(p) = \delta(v_0, v_k)$. ■

Intuitively the equation $w(p)' = w(p) + d(u) - d(v)$ suggests that if you have an edge from from from v to u and an outgoing edge from u you can take the difference of their weights they balance out so to speak. The proof above shows that this method will produce the same shortest graphs in G and the reweighted graph G'.

Psuedocode

With our formula we can make a new graph G' with t' and then add all edges and vertices from G s.t $E' = \{G.E \cup (t', v) \forall v \in V\}$ with weights = 0. Since t' is a source node with no incoming edges and outgoing edges equal to 0, shortest paths in G' will be the same as in G. finally since $d(v)$ is the weight of a shortest path up to v, then $d(v) \leq d(u) + w(v,u)$ which gives us $0 \leq d(u) + w(u,v) - d(v) = w'(v,u)$.

we can then use the weight function w' to update all the weights for G' We can then run $Dijkstra(G', w', t')$

3.2 Part (b) **13 / 16**

Problem 4

Part A

$Pr[E_1] = p = 1/n =$ the probability that we choose a specific element from U
$Pr[E_2] = 1 - p = 1 - 1/n =$ The probability that we choose any other element from U
$Pr[E_3] = (1-p)^n = (1-1/n)^n =$ The probability we choose nothing in S
$Pr[E_4] = 1 - p(1-p) = 1 - (1-1/n)^n(1-1/n)^n =$ The probability we choose at least one element in T.

Let $P(n) = Pr[E_3]$

plugging in $x = -1$ to remark 1

$$e^{-1}\left(1 - \frac{(-1)^2}{n}\right)^n \leq P(n) = \left(1 - \frac{1}{n}\right)^n \leq e^{-1} \tag{1}$$

Plugging in values for $n \geq 2$ we see that $P(n)$ we see that $P(n)$ increases monotonically, and has a lower bound of $\frac{1}{4}$

$$\begin{aligned} P(n) &= \left(1 - \frac{1}{2}\right)\ ()^2 = \frac{1}{4} \\ &= \left(1 - \frac{1}{3}\right)^3 = \frac{8}{27} = 0.2962 \\ &= \left(1 - \frac{1}{4}\right)^4 = \frac{81}{256} = 0.3164 \\ &\vdots \end{aligned} \tag{2}$$

taking the y = limit of P(n) we get:

$$
\begin{aligned}
y &= \lim_{n\to\infty}\left(\left(1\ -\ \frac{1}{n}\right)^{n}\right) \\
lny &= \lim_{n\to\infty}\left(e^{n\ln\left(1-\frac{1}{n}\right)}\right) \\
lny &= \lim_{n\to\infty}\left(n\ln\left(1-\frac{1}{n}\right)\right) \\
lny &= \lim_{n\to\infty}\left(\frac{\ln\left(1-\frac{1}{n}\right)}{\frac{1}{n}}\right) \\
lny &= \lim_{n\to\infty}\left(\frac{\frac{1}{n(n-1)}}{-\frac{1}{n^2}}\right) \\
lny &= \lim_{n\to\infty}\left(-\frac{1}{1-\frac{1}{n}}\right) \\
lny &= -1 \\
y &= e^{-1} \\
\lim_{n\to\infty} P\left(n\right) &= e^{-1}
\end{aligned}
\tag{3}
$$

technically $1/4 \leq Pr[E_4]$ which is an apparent solution to the problem, given the problem statement. More interestingly however we have shown that since $P(n)$ converges on $1/e$ we can deduce that $Pr[E_4]$ converges on $1 - \frac{1}{e^2}$ for large values of n

4 Problem 4 **8 / 20**

1. Should be 1-(1-p)^n

Problem 5

Part A

Let event $E_{i,t}$ = person i attempts to access the computer at step t. Let the complement $\overline{E_{i,t}}$ be the event that i does not attempt to access the computer at step t. $Pr[E_{i,t}] = p$ as per the problem statement. As such $Pr[\overline{Ei,t}] = 1-p$. Success for person i in accessing the computer at time step t is equivalent to person i attempting to access the computer and every other $n-1$ people not attempting to access the computer at time step t. Since people attempting to access, or not access, the computer are independent events we can take the product of the probabilities of both happening for a specific step in order to get the probability that person i succeeds:

$$Pr[\text{person } i \text{ succeeds at step t}] = Pr[E_{i,t}] * Pr[\overline{E^{i,t}}]^{n-1} = p * (1-p)^{n-1}$$

Part B

Intuitively $p = 1/n$ would maximize our chances of success since we want to maximize the chance any one, and only one, person can use the computer. To prove this we can take the derivative of the equation for $Pr[\text{person } i \text{ succeeds at step t}]$ found in part A and plug in $p = 1/n$ and set it to zero.

$$\begin{aligned}
\frac{d}{dp}(p * (1-p)^{n-1}) &= (n-1)(1-p)^{n-1} * p - (1-p)^{n-2} = 0 \\
&= (1-\frac{1}{n})^{n-1} - \frac{1}{n}(n-1)(1-\frac{1}{n})^{n-2} = 0 \\
&= (1-\frac{1}{n})(1-\frac{1}{n})^{n-2} - \frac{1}{n}(n-1)(1-\frac{1}{n})^{n-2} = 0 \\
&= ((1-\frac{1}{n})^{n-2})((1-\frac{1}{n}) - \frac{1}{n}(n-1)) = 0 \\
&= ((1-\frac{1}{n})^{n-2})((1-\frac{1}{n}) - (1-\frac{1}{n})) = 0 \\
&= ((1-\frac{1}{n})^{n-2}) * 0 = 0 \\
0 &= 0
\end{aligned}$$

and to prove it is the maximum we can take the second order derivative The second order derivative of $Pr[\text{person } i \text{ succeeds at step t}]$ is:

$$-2\,(n-1)\,(1-p)^{n-2} + p\,(n-2)\,(n-1)\,(1-p)^{n-3}$$

Clearly, the highest order term is negative, so plugging in $c = 1/n$ will result in a value < 0. Therefore $p = 1/n$ is a maximum.

5.1 Part (a) **5 / 5**

Problem 5

Part A

Let event $E_{i,t}$ = person i attempts to access the computer at step t. Let the complement $\overline{E_{i,t}}$ be the event that i does not attempt to access the computer at step t. $Pr[E_{i,t}] = p$ as per the problem statement. As such $Pr[\overline{Ei,t}] = 1-p$. Success for person i in accessing the computer at time step t is equivalent to person i attempting to access the computer and every other $n-1$ people not attempting to access the computer at time step t. Since people attempting to access, or not access, the computer are independent events we can take the product of the probabilities of both happening for a specific step in order to get the probability that person i succeeds:

$$Pr[\text{person } i \text{ succeeds at step t}] = Pr[E_{i,t}] * Pr[\overline{E^{i,t}}]^{n-1} = p * (1-p)^{n-1}$$

Part B

Intuitively $p = 1/n$ would maximize our chances of success since we want to maximize the chance any one, and only one, person can use the computer. To prove this we can take the derivative of the equation for $Pr[\text{person } i \text{ succeeds at step t}]$ found in part A and plug in $p = 1/n$ and set it to zero.

$$\begin{aligned}
\frac{d}{dp}(p * (1-p)^{n-1}) &= (n-1)(1-p)^{n-1} * p - (1-p)^{n-2} = 0 \\
&= (1-\frac{1}{n})^{n-1} - \frac{1}{n}(n-1)(1-\frac{1}{n})^{n-2} = 0 \\
&= (1-\frac{1}{n})(1-\frac{1}{n})^{n-2} - \frac{1}{n}(n-1)(1-\frac{1}{n})^{n-2} = 0 \\
&= ((1-\frac{1}{n})^{n-2})((1-\frac{1}{n}) - \frac{1}{n}(n-1)) = 0 \\
&= ((1-\frac{1}{n})^{n-2})((1-\frac{1}{n}) - (1-\frac{1}{n})) = 0 \\
&= ((1-\frac{1}{n})^{n-2}) * 0 = 0 \\
0 &= 0
\end{aligned}$$

and to prove it is the maximum we can take the second order derivative The second order derivative of $Pr[\text{person } i \text{ succeeds at step t}]$ is:

$$-2\,(n-1)\,(1-p)^{n-2} + p\,(n-2)\,(n-1)\,(1-p)^{n-3}$$

Clearly, the highest order term is negative, so plugging in $c = 1/n$ will result in a value < 0. Therefore $p = 1/n$ is a maximum.

5.2 Part (b) **5 / 5**

Part C

Let $Pr[S_{i,t}] = Pr[$person i succeeds at time t$]$. We showed in Problem 4 that Plugging in the value from Part B we get $S_{i,t} = \frac{1}{n} * (1 - \frac{1}{n})^{n-1}$, as such its complement $Pr[\overline{S_{i,t}}] = 1 - PrS[i, t]$.

Plugging in $n = 2$ we get $(1 - \frac{1}{n})^{n-1} = \frac{1}{2}$.
For $n = 3$ we get $\left(1 - \frac{1}{3}\right)^{3-1} = \frac{4}{9}$.
For $n = 4$ we get $\left(1 - \frac{1}{4}\right)^{4-1}$.

as we can see for $n \geq 2$ it is monotonic. using one half of remark one and x = -1: Multiplying through by $1/n$ we get:

$$\frac{1}{en} \leq \frac{1}{n}(1 - \frac{1}{n})^{n-1} \leq \frac{1}{2n}$$

Therefore the probability that a person succeeds at time t is bounded thusly

$$\frac{1}{en} \leq Pr[S_{i,t}] \leq \frac{1}{2n}$$

this implies the probability of failure at time t is bounded as

$$1 - \frac{1}{en} \geq 1 - Pr[S_{i,t}] \geq 1 - \frac{1}{2n}$$

Since we are interested in the probability that person i did not succeed to access the computer in any of the first t = en steps, and they are independent events, we can multiply the complementary event t times giving us an upper bound of the probability of failure at all times t.

$$Pr[\overline{S_{i,t}}]^t \leq 1 - (1/en)^t$$

In Problem 4 equations (1),(2),(3), we show that $(1-(1/n)^n$ increases monotonically, and converges to $\frac{1}{e}$, thus $(1 - (1/n)^n \leq \frac{1}{e}$. This implies that

$$(1 - (1/en))^{en} \leq (1/e)$$

Therefore

$$Pr[\overline{S_{i,t}}]^{en} \leq (1 - (1/en))^{en} \leq (1/e)$$

Giving us a nice upperbound on the probability that person i did not succeed to access the computer in any of the first t = en steps.

Part D

The number of steps t required so the probability that person i did not succeed in any of the the first t steps is upper bounded by an inverse polynomial in n. So we want to find t for $Pr[\overline{S_{i,t}}]^t$.

$$Pr[\overline{S_{i,t}}]^{t=en} \leq (1 - (1/en)^{t=en} \leq (1/e)$$

5.3 Part (c) **10 / 10**

Part C

Let $Pr[S_{i,t}] = Pr[$person i succeeds at time t$]$. We showed in Problem 4 that Plugging in the value from Part B we get $S_{i,t} = \frac{1}{n} * (1 - \frac{1}{n})^{n-1}$, as such its complement $Pr[\overline{S_{i,t}}] = 1 - PrS[i, t]$.

Plugging in $n = 2$ we get $(1 - \frac{1}{n})^{n-1} = \frac{1}{2}$.
For $n = 3$ we get $\left(1 - \frac{1}{3}\right)^{3-1} = \frac{4}{9}$.
For $n = 4$ we get $\left(1 - \frac{1}{4}\right)^{4-1}$.

as we can see for $n \geq 2$ it is monotonic. using one half of remark one and x = -1: Multiplying through by $1/n$ we get:

$$\frac{1}{en} \leq \frac{1}{n}(1 - \frac{1}{n})^{n-1} \leq \frac{1}{2n}$$

Therefore the probability that a person succeeds at time t is bounded thusly

$$\frac{1}{en} \leq Pr[S_{i,t}] \leq \frac{1}{2n}$$

this implies the probability of failure at time t is bounded as

$$1 - \frac{1}{en} \geq 1 - Pr[S_{i,t}] \geq 1 - \frac{1}{2n}$$

Since we are interested in the probability that person i did not succeed to access the computer in any of the first t = en steps, and they are independent events, we can multiply the complementary event t times giving us an upper bound of the probability of failure at all times t.

$$Pr[\overline{S_{i,t}}]^t \leq 1 - (1/en)^t$$

In Problem 4 equations (1),(2),(3), we show that $(1-(1/n)^n$ increases monotonically, and converges to $\frac{1}{e}$, thus $(1 - (1/n)^n \leq \frac{1}{e}$. This implies that

$$(1 - (1/en))^{en} \leq (1/e)$$

Therefore

$$Pr[\overline{S_{i,t}}]^{en} \leq (1 - (1/en))^{en} \leq (1/e)$$

Giving us a nice upperbound on the probability that person i did not succeed to access the computer in any of the first t = en steps.

Part D

The number of steps t required so the probability that person i did not succeed in any of the the first t steps is upper bounded by an inverse polynomial in n. So we want to find t for $Pr[\overline{S_{i,t}}]^t$.

$$Pr[\overline{S_{i,t}}]^{t=en} \leq (1 - (1/en)^{t=en} \leq (1/e)$$

Notice that since the three sides are positive for $n >= 2$. We can gain insight by squaring each term in the inequality.

$$(Pr[\overline{S_{i,t}}]^{en})^2 \leq ((1-(1/en)^{en})^2 \leq (1/e^2)$$

Clearly, the upperbound probability $(Pr[\overline{S_{i,t}}]^{en})$ becomes smaller for larger values of t. And since $e^{lnn} = n$. we can multiply through to get

$$(Pr[\overline{S_{i,t}}]^{en})^{ln} \leq ((1-(1/en))^{en})^{ln} \leq (e^{-lnn})$$
$$(Pr[\overline{S_{i,t}}]^{en})^{ln} \leq ((1-(1/en)^{en})^{ln} \leq n^{-1}$$

5.4 Part (d) **2 / 10**

5.5 Part (e) **0 / 15**

6 Extra Credits **5 / 0**