Problem 1

We know a path from the origin s to i is not unique if their exists two paths from s to i with the same weight . We can modify the utility *Update* for Dijkstra's algorithm to test if the path s to i is unique by checking if $dist[v] == dist[u] + w_{uv}$ and updating *unique* accordingly.

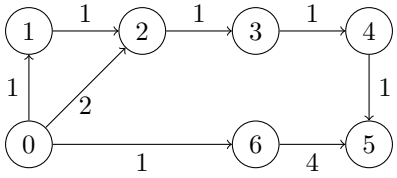

Figure 1: example of a directed graph with non-unique paths 0 to 2, and 0 to 5. Therefore we expect the solution to this graph to be $unique = [1, 1, 0, 1, 1, 0, 1]$ where $unique[i]$ corresponds to $i \in V$ and $s = 0$.

Algorithm 1: $Dijkstra_{v3}(G = (V, E, w), s \in V)$

```
1  Let unique, dist, prev be arrays of size |V|;
2  for each v ∈ V do
3  |  dist[v] = ∞;
4  |  prev[v] = NIL;
5  |  unique[v] = 1; // Assume all paths are unique
6  end
7  Q = BuildQueue(V; dist);
8  S = ∅;
9  while Q is not empty do
10 |  u = ExtractMin(Q);
11 |  S = S ∪ {u};
12 |  for (u, v) ∈ E do
13 |  |  Update(u, v);
14 |  end
15 end
```

Algorithm 2: $Update(u, v)$

```
1 if dist[v] == dist[u] + w_uv then
2 |   unique[v] = 0;
3 end
4 if dist[v] > dist[u] + w_uv then
5 |   dist[v] = dist[u] + w_uv;
6 |   prev[v] = u;
7 |   DecreaseKey(Q, v, dist[v]);
8 end
```

correctness

the general correctness of Dijkstra has been given in class and in slides. The correctness of the min-heap implementation seen above is given in CLRS page 659. From these we will assume that $dist[v] = \delta(s, v) \forall v \in V$.
As the graph from figure 1 shows it is possible to have a path $s-u-v$ and a path $s-x-v$ where $\delta(s, v) = w(s-u-v) = w(s-x-v)$ and this path will be a shortest path from $s-v$ (take for instance the paths $0-6-5$ and $0-3-5$ from figure 1). Due to the greedy nature of Dijkstra we will travel either $s-x-v$ or $s-u-v$ first and given the correctness of Dijkstra, which ever path we choose, $dist[v]$ will equal $\delta(s, v)$ after every edge in either the path $s-x-v$ or $s-u-v$ has been relaxed.
Say Dijkstra chose $s-u-v$ first. Then Upon a later iteration, since Dijkstra checks all shortest paths, we will inevitably travel the path $s-x-v$. Since $dist[v] = w(s-u-v)$ already we can simply check whether $dist[v] == w(s-x-v)$. If so we know that the path $s-u-v$ is not unique, nor is the path $s-x-v$ for that matter and this test will happen for all paths in G. Using the triangle inequality we can derive we can show $dist[v] \leq \delta(s, v)$ which by induction $\delta(s-u) \leq \delta(s-x) + \delta(x-v) = \delta(s-u) \leq dist[u] + w_{uv}$

Analysis

In the case that the graph is sparse $DijkstraUnique$ is $O(|E|log|V|)$, if the graph is dense say $|E| > |V|$ then it is $O(|V|log|V|)$, and if it is a complete graph then we have $|E| = |V|(|V|-1) = O(V^2 logV)$ which is clearly worse than $O(V^2)$ we would get from $Dijkstra-v2$. However, from what I can tell from lecture and the book, such dense graphs would be rare, and it's generally considered that $Dijkstra-v3$ is the best to use in practice, often times even better than fibanocci heaps which are mostly kept to the realms of theory. Since we can't know apriori what the graph will be, it is good to go with the most likely gain which is $O(mlogn)$ which is smaller than $O(n^2)$. As for our modifications: the lines 1,2 on only add a constant term, and we initialize *unique* in the same loop that we initialize *prev* and *dist*. So in the worst case $DijkstraUnique = \Theta(mlogn)$ in the worst case, when the graph is not dense

(which is the more likely case) and $Dijkstra Unique = \Theta(n^2 logn)$ in the worst case of a complete graph.

Problem 2

In order to keep the running count of edges from a path $s - i$ we simply have to keep a running count of $(u, v) \in E$ in the path $s - i$. Since shortest paths are defined by their weights and we are only interested in shortest paths with the minimum number of edges we are essentially using number of edges as a tie breaker. So we can test the case in which $dist[v] == dist[u] + w_{uv}$ and $minedges[u] + 1 < minedges[v]$ thus optimizing for ties with the fewest edges from s to i.

Algorithm 3: $DijkstraMinEdges(G = (V, E, w), s \in V)$

```
1  Let dist, prev, minedges be arrays of length |V|;
2  for each v ∈ V do
3  |  dist[v] = ∞;
4  |  prev[v] = NIL;
5  |  minedges[v] = 0;
6  end
7  dist[s] = 0;
8  Q = BuildQueue(V; dist);
9  S = ∅;
10 while Q is not empty do
11 |  u = ExtractMin(Q);
12 |  S = S ∪ {u};
13 |  for (u, v) ∈ E do
14 |  |  minedges[v] = minedges[u] + 1;
15 |  |  if dist[v] == dist[u] + w_uv and minedge[u] + 1 < minedge[v]
   |  |   then
16 |  |  |  minedges[v] = minedges[u] + 1;
17 |  |  |  prev[v] = u
18 |  |  end
19 |  |  if dist[v] > dist[u] + w_uv then
20 |  |  |  dist[v] = dist[u] + w_uv;
21 |  |  |  minedges[v] = minedges[u] + 1;
22 |  |  |  prev[v] = u;
23 |  |  |  Q.DecreaseKey(v, dist[v])
24 |  end
25 |  return minedges
26 end
```

Correctness

claim: $minedges[v] = minedges[u] + 1$ for all $(u, v) \in E$. Dijkstra finds all shortest paths $\delta(s, v) \forall v \in V$. As such Dijkstra travels all $(u, v) \in E$. We set all values in $minedges$ to 0 to begin because no edges have been traversed yet. If the graph consists only of a source node s then we correctly return $minedges[s] = 0$. If there is an edge $(s, x) \in E$, then $minedges[s] = minedges[x] + 1 = 1$, again this is correct since there is only a single edge from s to x. For the inductive step we see for any edge $(x, u) \in E$ where x is some vertex in the path from $s - v$ (between s and v) there is at most one edge from x *to* u so we increase number of edges on the path $s - u$ by one. By way of inductive hypothesis we add one more vertex v to the path $s - u$ such that $(u, v) \in E$ then we will increase the edge count $minedges[v] = minedges[u] + 1$, and this pattern will adhere to all paths in G since our modification does not disturb the functionality of Dijkstra's Algorithm, Notice we have not changed any of the logic of $Update$ except to update $prev$ with the shortest path that also has the minimum numbers of elements, as Dijkstra does not depend on $prev$ to find shortest paths, this has no effect on the functionality of Dijkstra's shortest path algorithm, and addition of the array $minedge$ only adds a linear term to the space and time complexity.

Analysis

Building the min heap requires $O(n)$ time. Initialising the arrays, $dist[v], prev[v], minedges[v]$ takes $\Theta(n)$ time. We traverse every edge in G calling $Q.DecreaseKey, Q.ExtractMin = O(logn)$ giving us $O((n + m)logn)$ as discussed in lecture and the slides. Check explanation in Analysis from problem 1 for the advantages and disadvantages of using a binary min heap priority queue implementation.

The addition of $minedges$ adds $\Theta(n)$ space and maintains the linear space complexity of Dijkstra's Shortest paths Algorithm. In update on lines 15 to 17 we have only added a trivial constant term, and on 21 we have only added one constant time term. Therefore $DijkstraMinEdge = \Theta((n + m)logn)$ in the worst case, for sparse graphs, and $\Theta(n^2 logn)$ in the very worst case that the graph is complete.

Problem 3

The pool is fixed and the bike and run portions are like free variables. None of the other contestants are dependent on how long the bike + run portions take, and all contestants can do all portions in parallel with whoever finished swimming before them. Furthermore, It doesn't matter how long it takes a contestant to finish the pool, as I will prove below. What is important is how long it takes for the biking + running portions, since the contestant who takes the longest to finish the biking and running portions would benefit the rest by going first, so every one else, in the mean time, can finish the largest portion of their own race while she takes her time with the biking and running.

From this informal understanding we can try making a greedy choice, finding $max\{b_i + r_i\}$ for all $i \in contestants$ as the solution to our optimal sub-problem.

Although we are looking for maximum elements we can still use a min-heap based priority queue by the equation $min\{-1(b_i + r_i)\}$. This is a constant time transformation, and since Max-heaps are symmetrical to Min-heaps, maintains an equivalent global order as a max heap by way of symmetry (if for contestants k, l $b_k + r_k > l_k + r_k$ then $-(b_k + r_k) < -(l_k + r_k)$ as a consequence contestant k will appear before contestant l in the min heap and will pop off first maintaining the correct decreasing order.)

As for some implementation details, we are inserting pairs into a priority queue, where the first element in the pair is the key for which we are optimizing, and i is the label we give each contestant. For the first contestant in A we assign $i = 1$ for the second $i = 2$ and so on to the last contestant $i = A.length$.

Algorithm 4: $OrderContestants(A)$

```
1 Q = PriorityQueue() of size A.length;
2 for i = 1 to A.length do
3 |  Q.insert((−(b_i + r_i), i));
4 end
5 Let optimalOrder be an array of size A.length;
6 for i = 1 to Q.size() do
7 |  optimalOrder[i] = Q.extractMin()[2];
8 end
9 return optimalOrder
```

Correctness

1. Consider the case in a race with two contestants C_i, C_j where $i < j$.
C_i finishes the race in $p_i + b_i + r_i$.
C_j finishes the race in $p_i + p_j + b_j + r_j$
Therefore the total finish time $= p_i + p_j + b_j + r_j$
2. Consider the case that $j < i$
Then C_j finishes in $p_j + b_j + r_j$

and C_i finishes in $p_j + p_i + b_i + r_i$
Therefore the total finish time $= p_i + p_j + b_i + r_i$
3. And in the case that $i == j$ we would still be taking the $max\{b_i + r_i, b_j + r_j\}$.

In all three cases the optimum finish time is not dependent on the pool portion of the race but the biking and running portions. Therefore the total finish time is minimized by starting C_i if $b_i + r_i \geq b_j + r_j$, or C_j if $b_j + r_j \geq b_i + r_i$. Thus showing that $max\{(b_i + r_i), (b_j + r_j)\}$ is an optimal greedy solution to a sub-problem.
Let S^* be an arbitrary feasible optimal solution, and let S^G be the optimal solution suggested here. Since we are creating an optimal schedule, based on optimal finish time, if $S^* \neq S^G$, it must be because it is optimizing finish time by some different ordering of the contestants. Let C_k, C_l be contestants that appear in decreasing order in S^G, assuming that they appear in some other order in S^* than clearly they can be swapped to approach an ordering closer to S^G and this process can be applied iteratively for all contestants given that S^* is simply a different ordering from S^G. If either C_k, C_l were not in S^* it would be because they were not yet added to S^* and they could be placed in S^* without affecting optimal substructure by way of swaps. Furthermore if S^* optimized by incorporating pool times, or biking and running times in increasing order, then, by the equations given in case 1,2 above, S^* would be made strictly better for having made these swaps closer to the ordering in S^G.

Analysis

Q is the same min-heap implementation discussed in class. Therefore in the worst case $Q.insert = \Theta(logn)$, $Q.ExtractMin = \Theta(logn)$. In the For loop on line 2 we call $Q.insert$ n times and in the For loop on line 6 we call $Q.ExtractMin$ n times, giving in the worst case $\Theta(nlogn) + \Theta(logn) = \Theta(nlogn)$. We could have used other sorting methods such as $MergeSort$ but we would have not achieved a runtime on a better order and would have created overhead on the call stack. And the Priority Queue is convenient as it sorts easily by keys.

Problem 4

Method and Recursion

We can use a DP algorithm that optimizes the max word size of words found in the dictionary, so that 'i' 'it' becomes 'it'. We know that the first segment starts at the beginning (or end if you work backwards), but we do not know where the optimal segments end (or start if you work backwards). Here I use a bottom up approach. As such we have to exhaustively, but cleverly, guess all possible segments, remembering when we get a good guess. Here I use a Boolean array, L, that represents each character, and is equal to one if the newly scanned character is part of a valid word, and an array *opt* which remembers the position of the last scanned character that forms the end of a valid word, we update *opt* and L based on whether the previous character in L $L[j]$ was also part of a word segment and whether the current segment itself forms a word this gives us the recursion:

$$L[i] = 1 \text{ and } opt[j] = i \text{ if } (L[j] == 1 \text{ and } Dict(s[j:i])) == True$$

where $j < i \leq n$ and s is the document.$L[j]$ This gives us the global solution

$$L[n] = 1 \text{ if } s \text{ is reconstructable, else } L[n] = 0$$

. Since if any segment guessed does not turn into a word, then all proceeding segments are invalidated. For example: if s = 'itzwas' becomes 'z', then 'zw', 'zwa', then 'zwas', none of which are valid words. We also update $opt[j] = i$ so we remember where the optimal word segment ends, for when we reproduce a valid construction from the corrupted document. We know this gives an optimum substructure since if on any iteration we no longer have a valid word, we will not update $L[i]$ or $opt[j]$ which means we have reached the max length of the previous valid word, and i will continue to grow giving us the boundary of the next longest valid word.
Our boundary condition is $L[0] = 1$, Since we know we start from the beginning of the document.

Input: The document broken document s.
Output: If document can be reconstructed we return a pair consisting of a boolean $L[n]$ that it can be restructured, and the valid reconstruction. $reconstruction$

Algorithm 5: $IsValidString(s)$

```
1  n = s.length;
   // adding 1 to make up for zero indexing
2  Let L and opt be arrays of size n initialized to zeros;
3  L[0] = 1;
   // Boundary Condition
4  for i = 1ton do
5  |  for j = 0toi) do
6  |  |  if L[j]andDict(s[j : i]) then
7  |  |  |  L[i] = 1;
8  |  |  |  opt[j] = i;
9  |  |  end
10 |  end
11 end
12 j = 0;
13 Let reconstruction be a linked list;
   // Linked list for O(1) appending
14 . if L[n] == 1 then
15 |  Print YES;
16 |  for each i ∈ opt do
17 |  |  if j < i then
18 |  |  |  reconstruction.append(s[j : i]);
19 |  |  |  j = i;
20 |  |  end
21 |  end
22 |  return Pair(L[n], reconstruction)
23 end
24 else
25 |  Print NO;
26 |  return L[n] // L[n] == False if document is invalid as
   |     expressed in the recurrence
27 end
```

Analysis

We have $j < i \leq n$ sub-problems giving us $\Theta(n^2)$ in the worst case. and it takes $O(1)$ to compute each subproblems. on lines 1-3 we initialize 2 arrays of size n with zeros taking $O(n)$ time and $O(2n)$ space. We also have a For loop for giving the valid reconstruction, that only works given the document is reconstructable. we create a linked list and a set $j = 0$ on lines 13 and 14 which take $O(1)$. In the for loop the test runs n times, and the operations in the body are clearly $O(1)$ giving us $O(n)$ for the reconstruction. The Print statements on lines 15 and 25 are clearly constant time. Taking it all together we get $\Theta(n^2) + O(n) + c$ where c is a small constant. Therefore $IsValidString = \Theta(n^2)$ in the worst case.

For the space complexity we have 2 arrays of size n and a linked list of size equal to the number of segments which is in the very unlikely but not impossible case size n. Therefore in the worst case total space is $O(3n + c) = O(n)$, where c is a small constant.

Problem 5

In collaboration with T.A. Shyam Padya

Method and Boundary Conditions

Here we use a DP solution with a table that stores the probabilities That Alice wins for all $i, j < n$. Building the table takes $\Theta(n^2)$ time. We start by filling the table of probabilities Ps final column with ones corresponding with the cases that Alice's number of wins $= n$, i.e. $P[0..n][n] = 1$. We then fill the bottom row of the table with zeros representing all the cases that Bob's wins $= n$ i.e. $P[n][0..n] = 0$ since if Bob's wins equal n then Alice has no chance of winning, and if Alice's wins equal n she has no chance of losing. These are our boundary conditions.

$$\begin{bmatrix} p_{0,0} & p_{1,0} & p_{2,0} & \dots & 1 \\ p_{0,1} & p_{1,1} & \dots & \dots & 1 \\ p_{0,2} & \vdots & \ddots & \dots & \vdots \\ \vdots & \vdots & \dots & \ddots & 1 \\ 0 & 0 & \dots & 0 & p_{n,n} \end{bmatrix}$$

Figure 2: where i is the number of games won by Alice, and j is the number of games won by Bob in $p_{i,j}$. Notice $p_{n,n}$ is an impossible value given the rules of the game and is only included in the figure to make the pattern clear.

Recurrence

If Alice wins $i = n$ games then Alice has won all games and $P(i, 0...j) = 1$ for all j games Bob might have won, and if Bob wins n games $P(0..i, j) = 0$. If Alice loses a match then we have the probability that she will win the next two matches is the probability $P(i-1, j)$ and if she wins then the probability that she will win the next match is $P(i, j-1)$. Since we don't know which will happen we have to take the average of both. Giving us the recurrence:

$$P(n, m) = (1/2) * P(n-1, m) + (1/2) * P(n, m-1)$$

and the optimal sub problem:

$$P[i][j] = (1/2) * P[i-1][j] + (1/2) * P[i][j-1]$$

where, $i, j < n$ (strictly less because boundary conditions are filled first), Where $P[i][j]$ represents the probability of Alice winning for any $i, j < n$ and the term $(1/2) * P[i-1][j]$ is the case in which Alice loses, from her winning record of $i = n$, and the term $(1/2) * P[i][j-1]$ is the case that Bob loses. We are finished building the table when $i = 0$ and $j = 0$. We will use a utility function *BuildTable* for generating the probability table corresponding to the

probabilities that Alice will win n games first.

Input: i, j, n where i equals Alice's current win count and j equals Bob's current win count and n equals the target winning record.
Output: $P[i][j]$ which is the chance Alice will win given $n - i, n - j$;

Algorithm 6: $ChanceAliceWins(i, j, n)$

```
1 if i >= n and j >= n then
2 |  return NIL
3 end
4 Let P be an nxn array of arrays;
5 BuildTable(P, i, j, n);
6 return P[i][j]
```

Algorithm 7: $BuildArray(P, i, j, n)$

```
  // Update P with Boundary conditions.
1 for i = n down to 0 do
2 |  P[n − 1][i] = 0;
3 |  P[i][n − 1] = 1;
4 end
  // Apply recurrence.
5 for i = n − 1 down to 1 do
6 |  for j = n − 1 down to 1 do
7 |  |  P[i][j] = (1/2) ∗ P[i − 1][j] + (1/2) ∗ P[i][j − 1];
8 |  end
9 end
```

Analysis

We have $\Theta(n^2)$ sub-problems since we can not know what i, j will be passed apriori, and each solution can be looked up from the table in $O(1)$ time giving an over all time of $\Theta(n^2)$. The table P has dimension nXn and any other space required for $ChanceAliceWins$ is some small constant c giving us $ChanceAliceWins = \Theta(n^2) + c = \Theta(n^2)$ space complexity.