CS 6150: Dynamic Programming

Pre-submission deadline: Nov 1, 2014

This assignment has 8 questions, for a total of 100 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Cutting Cloth	$12\frac{1}{2}$	
Dance, dance !	$12\frac{1}{2}$	
Palindromes	$12\frac{1}{2}$	
Time for geometry	$12\frac{1}{2}$	
Bitmaps	$12\frac{1}{2}$	
Thirsty Students	$12\frac{1}{2}$	
Nadira	$12\frac{1}{2}$	
Solitaire	$12\frac{1}{2}$	
Total:	100	

Question 1: Cutting Cloth ..... **[12½]**

This is from Dasgupta/Papadimitriou/Vazirani, Exercise 6.14

You're given a rectangle piece of cloth with dimensions $X \times Y$ (X, Y are positive integers), and a list of n products that can be made using the cloth. For each product $i \in [1 \ldots n]$ you need a piece of cloth of dimension $a_i \times b_i$ and the final value of this product is c_i. Assume that all a_i, b_i, c_i are positive integers.

You have a machine that can take a rectangular piece of cloth and cut it into *two* pieces either horizontally or vertically. Design a strategy to maximize the return for cutting the cloth: i.e a strategy to cut the cloth into pieces so that the sum of values of the pieces is maximized. It's fine to duplicate a product (and sell many copies of it).

The next seven questions are from Erickson (`http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf`)

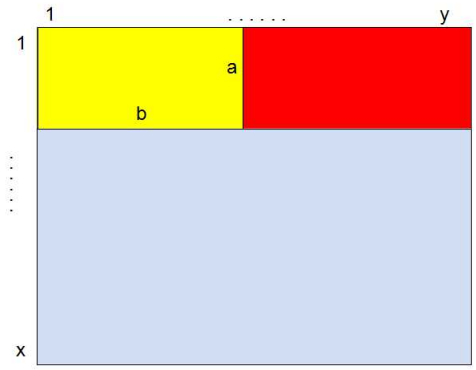

Figure 1

Figure 1

Solution:

Algorithm 1: max value of cloth

```
1:  MAX_VALUEOFCLOTH(int x,int y, int product_a[1...n], product_b[1...n], product_c[1...n]):
2:  dp[1..x][1..y] ← 0;
3:  cut[1..x][1..y] ← 0;
4:  for i ← 1 to x do
5:    for j ← 1 to y do
6:      for k ← 1 to n do
7:        max ← 0, which;
8:        a ← product_a[k], b ← product_b[k], c ← product_c[k];
9:        if a ≤ i and b ≤ j then
10:         max ← max{dp[i − a][j] + dp[a][j − b], max} + c;
11:         if max is updated then
12:           which ← k;
13:         end if
14:       end if
```

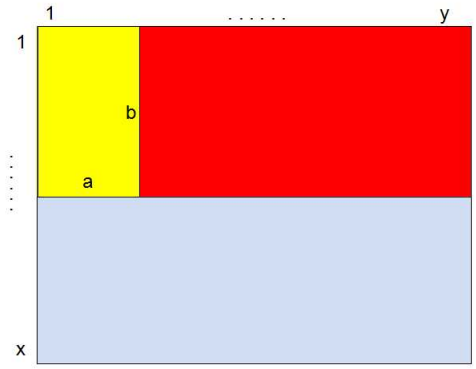

Figure 2

Figure 2

```
15:         if b ≤ i and a ≤ j then
16:           max ← max{dp[i − b][j] + dp[b][j − a], max} + c;
17:           if max is updated then
18:             which ← k + n;
19:           end if
20:         end if
21:       end for
22:       cut[i][j] ← which;
23:     end for
24: end for
25: OUTPUT_STRATEGY(cut[1..x][1..y],1,1,product_a[1..n],product_b[1..n]);
26: return dp[x][y];
 1: OUTPUT_STRATEGY(cut[1..x][1..y],int i,int j,int product_a[1..n], int product_b[1..n]):
 2: whcih ← cut[x][y];
 3: if which! = 0 then
 4:   a ← product_a[which], b ← product_b[which];
 5:   if which > n then
 6:     output "cut the cloth starting from the position (i, j) and get a cloth b × a to produce
        product which";
 7:     OUTPUT_STRATEGY(cut[1..b][a + 1..y],i,j + a,product_a[1..n], product_b[1..n]);
 8:     OUTPUT_STRATEGY(cut[b + 1..x][1..y],i + b,j,product_a[1..n], product_b[1..n]);
 9:   end if
10:   if which ≤ n then
11:     output "cut the cloth starting from the position (i, j) and get a cloth a × b to produce
        product which";
12:     OUTPUT_STRATEGY(cut[1..a][b + 1..y],i,j + b,product_a[1..n], product_b[1..n]);
13:     OUTPUT_STRATEGY(cut[a + 1..x][1..y],i + a,j,product_a[1..n], product_b[1..n]);
14:   end if
15: end if
```

Claim: x and y represent the dimension of the cloth. $product\_a[1..n]$ and $product\_b[1..n]$ store the dimensions and b the product needed and $product\_c[1..n]$ stores the n products' values. $dp[1..x][1..y]$ will store the results of sub-problems, e.g., $dp[i][j]$ stores the largest value a cloth with dimensions $i \times j$ can get. And $cut[1..x][1..y]$ stores the strategy information, $cut[i][j]$ is the first product the machine should cut from the left top corner of cloth $i \times j$. Its value is from 1 to $2n$. If the product is k and its dimension is $a \times b$, $cut[i][j] = k$ means the machine should cut a piece $a \times b$ from the cloth as Figure 1 shows. And $cut[i][j] = k + n$ means the machine should cut a piece $b \times a$ from the cloth as Figure 2 shows. Moreover, in function OUTPUT_STRATEGY, parameters i and j is the position of the sub-cloth in the original cloth. Take Figure 1 as an example, the red piece's starting position is $(1, b + 1)$ in the original cloth, so $i = 1$ and $j = b + 1$.
EXPLANATION: The algorithm solves the problem using dynamic programming. The largest value cloth $x \times y$ can be achieved with the results of the sub-problems. It always cut the cloth from the left top corner. Take Figure 1 shows, the machine cuts a piece $a \times b$ from the left top corner and then the red piece and the blue piece are the sub-problems. The algorithm uses the known results of these sub-problems to try producing different products in order to choose the optimal one. For each sub-problem, it will repeat the process. Line 9 to Line 20 means the cloth can be cut in two approaches - vertical or horizontal as Figure 1 and Figure 2 shows and the algorithm will choose the best one. Then the algorithm will call function OUTPUT_STRAGEGY to output the strategy of cutting the cloth. The function output the results recursively. As Figure 1 shows, it will output the product produced by the yellow piece and then recursively call itself to output the products produced by the red and the blue piece.

Proof. Firstly, the approach of cutting cloth of this algorithm is optimal can be proved. If we want to make a product k, we had better cut the needed piece from any corner of the cloth. It's obviously no matter how we cut the piece, the remaining area is constant. Cutting from the corners can maintain the integrity of the remaining cloth material. More integrated the cloth is, more values the cloth can achieve obviously. Moreover, without generality, the algorithm can achieve same largest value by cutting from any corners. Hence cutting from the left top corner is optimal approach. Secondly, the Dynamic Process in the algorithm can be proved by Induction Method.
Base Case:$dp[0][0] = 0$ is obviously true.
Hypothesis: Make a hypothesis that $\forall k_1 \leq i\ ,\ k_2 \leq j\ and\ !(k_1 = i \&\& k_2 = j)$, $dp[k_1][k_2]$ stores the largest value of cloth with dimensions $k_1 \times k_2$ correctly, i.e., if $product\_a[k] \leq i\ and\ product\_b[k] \leq j$, $dp[i - product\_a[k]][j]$ is the largest value the cloth with dimensions $(i - product\_a[k]) \times j$ can achieve and $dp[product\_a[k]][j - product\_b[k]]$ is the largest value cloth with dimensions $product\_a[k] \times (j - product\_b[k])$ can achieve.
Induction: For a cloth with dimension $i \times j$. For any product k, as long as its size is smaller then the cloth, it should be tried obviously. And if product k's two dimensions are different, there are two cutting ways as Figure 1 and Figure 2 shows. So both the two approaches should be tried. If the machine cuts the piece with dimensions $a \times b$ from the cloth the way as Figure 1 shows, then the remaining part will be divided to two pieces - one with dimensions $product\_a[k] \times (j - product\_b[k])$ just like the red piece in Figure 1 and another with dimensions $(i - product\_a[k]) \times j$ like the blue piece in Figure 1.Since we have known the largest values of clothes with size smaller than $i \times j$ as a hypothesis, then the largest value the cloth $i \times j$ can achieve if we chooses to produce product k firstly is $dp[i - product\_a[k]][j] + dp[product\_a[k]][j - product\_b[k]] + product\_c[k]$. Each product should be tried to be cut firstly to get different values in order to judge which one is the largest. In addition, we could produce the same products for many times. So all kinds of products can be tried in the iteration process as long as its size satisfies the requirements. So $dp[i][j] = max\{dp[i - product\_a[k]][j] + dp[product\_a[k]][j - product\_b[k]] + product\_c[k] | k \in 1, ..., n\}$. Hence the algorithm can figure out $dp[i][j]$ correctly, i.e., $dp[x][y]$ is the largest value the cloth $x \times y$ can achieve by induction.
Moreover, for all kinds of clothes with dimensions $i \times j$, $cut[i][j]$ stores the product that should be

cut firstly correctly since the most suitable one is chose in the iterations process. Hence, the function OUTPUT_STRATEGY is correct as well. □

*Analysis.*There are three layers of iterations of this algorithm. For every pair i and j, we should implement the comparing process of these products and its time complexity is $O(xyn)$. The function OUTPUT_STRATEGY outputs all pieces of clothes in the optimal strategy and its cost can't exceed $O(xyn)$. So the total time cost is $O(xyn)$ And the space complexity is $O(xy)$.

Question 2: Dance, dance ! ..... **[12½]**
Problem #4

Solution:

Algorithm 2: Max_Score of the dancer

```
1: MAX_SCORE OF DANCING(Score[1,...,n],Wait[1,...,n])
2: int DP[1,...,n,...,n+wait[n]+1];
3: memset(DP,0,sizeof(DP));
4: for i = n to 1 do
5:    DP[i] = max(DP[i+wait[i]+1]+Score[i],DP[i+1]);
6: end for
7: return DP[1];
```

Claim:$DP[1,...,n]$is an array storing the maximum total scores, $DP[i]$ represents the maximum total score the dancer could achieve between song i and song n.
EXPLANATION: The algorithm solves the problem using dynamic programming. For each song, there are two chooses - dance or not dance. If she chooses to dance, it corresponds to the subproblem MAX_SCORE OF DANCING($Score[i+wait[i]+1,...,n]$), $wait[i+wait[i]+1,...,n]$ and if she chooses not to dance, it corresponds to the subproblem MAX_SCORE OF DANCING($Score[i+1,...,n], Wait[i+1,...,n]$). The algorithm makes the decision by comparing the maxi scores of the two chooses and choose the larger one. The recursive equation is as Line 5 shows.

Proof. We could prove the DP process by Induction Method,
Base case: For $k=n$, since $k+wait[k]+1>n$, $DP[k]=max\{DP[k+wait[k]+1]+Score[k], DP[k+1]\}=maxScore[k],0=Score[k]$;
Hypothesis: Let's assume that for $n \geq k > i$, $DP[k]$ stores the maximum total score the dancer could achieve between song k and song n;
Induction: Now for song i, the dancer two chooses, either to dance it or not dance it. If she dances it, we have to wait between song $i+1$ and song $i+wait[i]$. Since we have known the maximum total score the dancer could achieve between song $i+wait[i]+1$ and song n. So the dancer could achieve $DP[i+wait[i]+1]+Score[i]$ if choosing to dance song i. If she doesn't dance this song, she couldn't achieve the score of this song and the maximum total score between song i and song n is the same to the that between song $i+1$ and song n($DP[i+1]$). So the dancer just compares these two situations and get the larger one, then $DP[i]=max(DP[i+wait[i]+1]+Score[i], DP[i+1])$. In conclusion, this DP process is true and we could get the maximum total value by returning $DP[1]$. □

*Analysis.*For this algorithm, we just implement $O(n)$ comparisons and the time complexity is $O(n)$ and the space complexity is $O(n)$.

Question 3: Palindromes ..... **[12½]**
Problem #5 (a)

Solution:

Algorithm 3: longest palindrome subsequence

```
1: longest_parl_subs(str[1, ..., n]):
2: DP[n][n];
3: for i = 1 to n do
4:    DP[i][i] = 1;
5: end for
6: for i = n to 1 do
7:    DP[i][i + 1] = (str[i] == str[i + 1])?2 : 1;
8:    for j = i + 2 to n do
9:       if str[i] == str[j] then
10:         DP[i][j] = DP[i + 1][j − 1] + 2;
11:      else
12:         DP[i][j] = max{DP[i + 1][j], DP[i][j − 1]};
13:      end if
14:   end for
15: end for
16: return DP[1][n];
```

Claim:$DP[i][j]$ stores the largest palindrome sequence length of string $str[i, ..., j]$.
EXPLANATION: For problem $longest\_parl\_subsstr[i..j]$, if $str[i] = str[j]$, there is only one subproblem that is $longest\_parl\_subsstr[i + 1..j - 1]$. If $str[i]! = str[j]$, there are two chooses and the corresponding two subproblems are $longest\_parl\_subsstr[i+1..j]$ and $longest\_parl\_subsstr[i..j-1]$. The recursive equations are as Line 10 and Line 12 show.

Proof. Let's use Induction Method to prove it,
Base Case: For $i = j$, it's obviously true that $DP[i][j] = 1$. And for $j = i + 1$, if $str[i] == str[j]$, it's true that $DP[i][j] = 2$ and if $str[i]! = str[j]$, it's true that $DP[i][j] = 1$;
Hypothesis:$\forall str[k_1, ..., k_2] \subset str[i, ..., j]$, $DP[k_1][k_2]$ stores the largest palindrome sequence length of $str[k_1, ..., k_2]$. We could imagine that all elements in $str[k_1, ..., k_2]$ that couldn't find its symmetric element have disappeared and we don't need to care the remaining ones since they have found their symmetric elements.
Induction: For $str[i, ..., j]$, if $str[i] == str[j]$, Since $DP[i + 1][j - 1]$ stores the largest palindrome sequence length of $str[i + 1, ..., j - 1]$, and we don't need to care the situation in $str[i + 1, ..., j - 1]$, so $DP[i][j] = DP[i + 1][j - 1] + 2$ is the largest palindrome sequence length of $str[i, ..., j]$. Then if $str[i]! = str[j]$, we could consider $str[i, ..., j]$ as $str[i] + str[i + 1, ..., j]$ or $str[i, ..., j - 1] + str[j]$. For the first case, $str[i]$ could be ignored since it's single and it couldn't find its symmetric one in $str[1 + 1, ..., j]$ and it's similar to the second case. So $DP = max\{DP[i + 1][j], DP[i][j - 1]\}$.
In conclusion, this algorithm is true. $DP[1][n]$ is the largest palindrome sequence length of $str[1, ..., n]$. □

*Analysis.*The space complexity is $O(n^2)$. However, we only use half of the total space. Given the time complexity, we need $[n-(n-1)]+[n-(n-2)]+\cdots+[n-(n-i)]+\cdots+[n-(n-(n-1))]+[n-(n-n)] = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ iterations. So the time complexity is $O(n^2 * c) = O(n^2)$.

(b)

Solution:

Algorithm 4: shortest palindrome supersequence

```
1: SHORTEST_PLD_SPS(str[1, ..., n]):
2: DP[n][n];
3: for i = 1 to n do
4:    DP[i][i] = 1;
5: end for
6: for i = n to 1 do
7:    DP[i][i + 1] = (str[i] == str[i + 1])?2 : 3;
8:    for j = i + 2 to n do
9:       if str[i] == str[j] then
10:         DP[i][j] = DP[i + 1][j − 1] + 2;
11:      else
12:         DP[i][j] = min{DP[i + 1][j] + 2, DP[i][j − 1] + 2};
13:      end if
14:   end for
15: end for
16: return DP[1][n];
```

Claim:$DP[i][j]$ stores the shortest palindrome supersequence length of string $str[i, ..., j]$.
EXPLANATION: Line 6 to Line 15 is a dynamic programming process. For the problem dealing with $str[i...j]$, if $str[i] = str[j]$, there recursive equation is as Line 10 shows and the subproblem is $str[i + 1...j - 1]$. And if $str[i]! = str[j]$, the recursive equation is as Line 12 shows and the subproblems are $str[i + 1...j]$ and $str[i...j - 1]$.

Proof. Let's use Induction Method to prove it:
Base Case:$\forall str[k] \subseteq str[i, .., j]$, it's obviously true that $DP[k][k] = 1$. And $\forall str[k, k + 1] \subseteq str[i, ..., j]$, it's true that $DP[k][k+1] = 3$ if $str[k]! = str[k+1]$ because we have to add one element to keep palindrome and $DP[k][k + 1] = 2$ if $str[k] == str[k + 1]$ since we don't need to add any elements;
Hypothesis: For $\forall str[k_1, ..., k_2] \subset str[i, ..., j]$, $DP[k_1, ..., k_2]$ has been calculated. We could imagine this situation as that all the elements in $str[k_1, ..., k_2]$ have found their symmetric elements and for the single one except the middle element, we have added its symmetric element;
Induction: If $str[i] == str[j]$, $DP[i][j] = 2+DP[i+1][j-1]$ is true since $str[i]andstr[j]$ have found their symmetric element and could just image they have both disappeared. For $str[i + 1, j - 1]$, we have finished their pairing. Then the shortest supersequence of string $str[i, ..., j]$ is equal to that the shortest supersequence of string $str[i + 1, ..., j - 1]$ adds this pair of elements. If $str[i]! = str[j]$, we could decompose $str[i, ..., j]$ as $str[i] + str[i + 1, ..., j]$ or $str[i, ..., j - 1] + str[j]$. For the first case, all the elements in $str[i + 1, ..., j]$ have finished their pairing, so $str[i]$ couldn't find its and we should add its symmetric one. Hence $DP[i][j] = DP[i + 1, ..., j] + 2$. Similarly, we could get $DP[i][j] = DP[i][j - 1] + 2$ for the second case. So, $DP[i][j] = max\{DP[i + 1, ..., j] + 2, DP[i, ..., j - 1] + 2\}$.
In conclusion, $DP[i][j]$ could always store the shortest supersequence length of string $str[i, ..., j]$. Hence $DP[1][n]$ is the final answer. □

*Analysis.*The space complexity is $O(n^2)$. However, we only use half of the total space. Given the time complexity, we need $[n-(n-1)]+[n-(n-2)]+\cdots+[n-(n-i)]+\cdots+[n-(n-(n-1))]+[n-(n-n)] = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ iterations. So the time complexity is $O(n^2 * c) = O(n^2)$.

(c)

Solution:

Algorithm 5: smallest number of palindromes

```
1: SMALLEST_NUMOFPLR(str[1, ..., n]):
2: DP[n][n];
3: for i = 1 to n do
4:    DP[i][i] = 1;
5: end for
6: for i = n to 1 do
7:    for j = i to n do
8:       min = n + 1;
9:       for k = 1 to j − i do
10:         if ISSYMMETRIC(str[i, ..., i + k]) then
11:            min =_min {1 + DP[i + k + 1][j], min};
12:         end if
13:      end for
14:   end for
15: end for
```

Claim: $DP[i][j]$ stores the smallest number of palindromes of $str[i, ..., j]$. And ISSYMMETRIC($str[i, ..., j]$) is a function that could help judge whether $str[i, ..., j]$ is a palindrome.
EXPLANATION: Line 6 to Line 15 is the dynamic programming process. For the problem dealing with $str[i...j]$, the recursive equation is as Line 9 to Line 13 shows and the subproblems are $str[i + k + 1...j]$, $k \in 1, .., j - i$.

Proof. We could use Induction Method to prove it.
Base Case: For $k_1 = k_2$, it's obviously true that $DP[k_1][k_2] = 1$;
Hypothesis: $\forall str[k_1, ..., k_2] \subset str[i, ..., j]$, $DP[k_1][k_2]$ stores the smallest number of palindromes of $str[k_1, ..., k_2]$. And we could imagine that we have decomposed the strings;
Induction: For $str[i, ..., j]$, we have many decomposing ways. $\forall str[i, ..., i + k] \subseteq str[i, ..., j]$, if it's a palindrome and we decide to cut it from the whole string, since $DP[i+k+1]$ has stored the smallest number of palindromes and the string $str[i + k + 1][j]$ has been decomposed. So $DP[i + k + 1] + 1$ is the smallest number of palindromes for this way and this 1 represents $str[i, ..., i + k]$. However, it may not be the optimal solution. $str[i, ..., i + k] + str\_tmp \subseteq str[i + k + 1, ..., j]$ may be another palindrome and $DP[i+k_1+1]+1$ may be smaller than $DP[i+k+1]+1$. So we try all the symmetric strings $str[i, ..., i + k]$ to find the smallest number of palindromes.
In conclusion, $DP[i][j]$ could tell us the smallest number of palindromes. We could get the answer by returning $DP[1][n]$. □

Analysis. The space complexity is $O(n^2)$. For the time complexity, We should implement $O(n^3)$ times comparing and judging operations. The time complexity of ISSYMMETRIC($str[i, ..., j]$) is $O(n)$ because we should compare the left element and right element one by one. So the total time complexity is $O(n^4)$.

Question 4: Time for geometry ..... **[12½]**
Problem #7
(a)

Solution:

Algorithm 6: largest subset of L between lines

Require: A set L of n line segments, where each segment has one endpoint on line $y = 0$ and one endpoint on line $y = 1$;

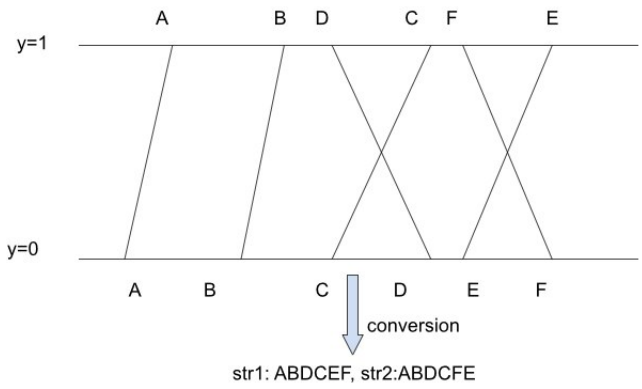

Figure 3

Ensure: The largest subset of L in which no pair of segments intersects.

1: convert the endpoints of the set L on line $y = 0$ into uppercase letters according to their orders, say, the first one becomes to A, the second one becomes to B and so on;
2: for each line in set L, we convert its endpoint on line $y = 1$ to the same uppercase letter as its endpoint on line $y = 0$, say if the endpoint of a line l on $y = 0$ is A then its endpoint on $y = 1$ is A as well, then we could consider the uppercase letters on line $y = 0$ as string s1 and the uppercase letters on line $y = 1$ as string s2;
3: create $DP[1...n][1...n]$; % n is the length of these two strings.
4: create $LCS[1..n][1..n]$;
5: **for** $i = 0$ to n **do**
6: $DP[0][i] = 0$;
7: $LCS[0][i] =$ "";
8: $DP[i][0] = 0$;
9: $LCS[i][0] =$ "";
10: **end for**
11: **for** $i = 1$ to n **do**
12: **for** $j = 1$ to n **do**
13: **if** $s2[j] == s1[i]$ **then**
14: $DP[i][j] = DP[i-1][j-1] + 1$;
15: $LCS[i][j] = LCS[i-1][j-1] + s1[i]$;
16: **else**
17: $DP[i][j] = max\{DP[i-1][j], DP[i][j-1]\}$;
18: **if** $DP[i-1][j] > DP[i][j-1]$ **then**
19: $LCS[i][j] = LCS[i-1][j] + s1[i]$;
20: **else**
21: $LCS[i][j] = LCS[i][j-1] + s1[i]$;
22: **end if**
23: **end if**
24: **end for**

25: **end for**
26: return $DP[n][n]$ and $LCS[n][n]$

Claim:$LCS[i][j]$ stores the largest common substring of $s1[1..i]$ and $s2[1..j]$. $DP[i][j]$ stores the longest common substring length of strings $s1[1..i]$ and $s2[1..j]$.

EXPNATION: Line 1 - 2 converts the largest subset of L between lines problem into a largest common substring problem. The algorithm map each line to a distinct uppercase letter and then maps the endpoints in $y = 0$ and $y = 1$ to two strings. Figure 3 shows an example, there are six lines between $y = 1$ and $y = 0$, the algorithm maps A, B, C, D, E, F to them respectively and then converts the endpoints in two lines to two strings "$ABCDEF$" and "$ABDCFE$". Line 3 - 25 is the Dynamic Programming for solving the largest common substring problem.

Proof. This problem is equal to the largest common substring problem. According to the algorithm, we now have two strings, e.g.s1:ABCDEF,s2:ABDCFE as Figure 3 shows. s1 corresponds to the endpoints on line $y = 0$ and s2 corresponds to the endpoints on line $y = 1$. The letters in each string are distinct because each letter represents a line. Since the uppercase letters in each string are distinct, if we connect the same letters of the two lines, then line A:"A-A" represents line 1, line B:"B-B" represents line 2 and so on, hence we could get the same lines as the input and we may consider each letter as a line. Since the letters' order of largest common substring in the two strings are same, e.g. ABCE, then these lines "A-A", "B-B", "C-C" and "E-E" will not intersect. However, we may not sure whether line A, line B,line C,line D is the largest subset of L. If there is another line X that will not intersect with line A, line B, line C and Line D, then it implies the order of A, B, C, D, X in the two strings are same, hence we could get a longer common substring which is contradict to the hypothesis. So the largest common substring could represent the largest subset of L, in which each line won't intersect with each other.
Now we just need to prove the correctness of process handling largest common substring problem. we could use Induction Method to prove it:
Now let's consider $s1[1..i]$ and $s2[1..j]$.
Base Case:$DP[0..n][0] = DP[0][0..n] = 0$ is obviously true;
Hypothesis:$DP[i-1][j-1]$, $DP[i][j-1]$ and $DP[i-1][j]$ are true and $LCS[i-1][j-2]$, $LCS[i-1][j]$ and $LCS[i][j-1]$ store the largest common substring correctly;
Induction: If $s1[i] = s2[j]$ and we have handled $s1[1..i-1]$ and $s2[1..j-1]$ which means the largest common substring of $s1[1..i-1]$ and $s2[1..j-1](LCS[i-1][j-1])$ in either string is in the same order, then it implies that letters of $LCS[i-1][j-1] + s1[i]$ in either string are in the same order, i.e.$LCS[i-1][j-1] + s1[i]$ is common substring. Since the letters are distinct, it's impossible that $s1[i]$ find its same letter before $s2[j]$, hence $s1[i]$ doesn't appear in $LCS[i-1][j-1]$ or $LCS[i-1][j]$ or$LCS[i][j-1]$. Hence $LCS[i][j-1] = LCS[i-1][j] = LCS[i-1][j-1]$ and $DP[i][j-1] = DP[i-1][j] = DP[i-1][j-1]$. So $DP[i][j] = max\{DP[i-1][j-1]+1, DP[i-1][j], DP[i][j-1]\} = DP[i-1][j-1]+1$ and $LCS[i][j] = LCS[i-1][j-1] + s1[i]$.
If $s1[i] != s2[j]$, then $LCS[i][j] = max\{LCS[i-1][j-1], LCS[i-1][j], LCS[i][j-1]\}$ and $DP[i][j] = max\{DP[i-1][j-1], DP[i-1][j], DP[i][j-1]\}$. It's obvious that $DP[i-1][j-1] \leq DP[i][j-1]$ and $DP[i-1][j-1] \leq DP[i-1][j]$, so $DP[i][j] = max\{DP[i-1][j], DP[i][j-1]\}$ and $LCS[i][j] = max\{LCS[i-1][j], LCS[i][j-1]\}$.
In conclusion, this algorithm runs correctly. At last, if each letter represents a line, then we could get this largest subset of L from $LCS[n][n]$. □

*Analysis.*The time complexity is $O(n^2)$. We create two two-dimensional arrays. For LCS, the string length in $LCS[i][j]$ - $P[i][j] \leq min\{i, j\} = O(n)$, so the space needed for $LCS[n][n]$ is no more than $2*\Sigma_{i=1}^{n}\Sigma_{j=1}^{i} j = 2\Sigma_{i=1}^{n}\frac{(1+i)i}{2} \leq n^3$. So the time space is $O(n^3)$.

(b)
credit to the idea of Kaiqiang Wang and do by myself.

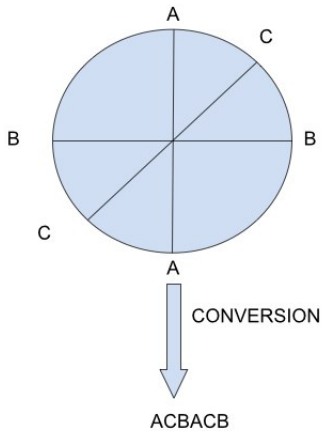

Figure 4

Solution:

Algorithm 7: largest subset of L in a circle

Require: n line segments, where the endpoints of each segment lie on a unit circle and the 2n endpoints are distinct;

Ensure: The largest subset of L in which no pair of segments intersect.

1: First, map each segment to a distinct uppercase letter, which means convert the endpoints of a line segment to a uppercase letter and different line segments is mapped to different uppercase letter;%then there will be a series of uppercase letters on the unit circle and each uppercase letter will appear twice to represent the two endpoints of a line segment.
2: Secondly, convert the unit circle into a line to get a string $str[1..2n]$, e.g. "ABCDABCD";%n is the number of the line segments.
3: Create $DP[1..2n][1..2n]$;
4: $memset(DP, 0, sizeof(DP))$;
5: Create $LLS[1..2n][1..2n]$;
6: $memset(LLS, 0, sizeof(LLS))$;
7: **for** $L = 1$ to $2n - 1$ **do**
8: **for** $i = 1$ to $2n - L$ **do**
9: $k =$FIND($str[i + 1..i + L]$,$str[i]$);
10: **if** $k! = -1$ **then**
11: $DP[i][i + L] = max\{DP[i + 1][i + L], DP[i][k - 1] + DP[k + 1][i + L] + 1\}$;
12: **if** $DP[i + 1][i + L] > DP[i][k - 1] + DP[k + 1][i + L] + 1$ **then**
13: $LLS[i][i + L] = LLS[i + 1][i + L]$;
14: **else**
15: $LLS[i][i + L] = LLS[i][k - 1] + str[i] + LLS[k + 1][j]$;
16: **end if**
17: **else**
18: $DP[i][i + L] = DP[i + 1][i + L]$;
19: $LLS[i][i + L] = LLS[i + 1][i + L]$;
20: **end if**

21: **end for**
22: **end for**
23: return $LLS[1][2n]$ and $DP[1][2n]$;

Claim: In the algorithm, $DP[1..2n][1..2n]$ stores the largest number of pairs of same characters that don't intersect with each other in the strings. $LLS[1..2n][1..2n]$ stores the largest set of such pairs. The function $k = \text{FIND}(str[i+1..j], str[i])$ is a function that will return the index of char $str[i]$ in the substring $str[i+1..j]$. If the return value is -1, it means char $str[i]$ can't be found in this substring. This function can be easily implemented, hence it's not stated.

EXPLANATION: This algorithm converts the problem into a DP problem of string. The conversion is as figure 3 shows. **Here, "intersect" means the same letters appear alternately, e.g., "ABAB"**. Line 7 to Line 22 is the dynamic programming part. The first-level loop is the stage of DP, L is the substring's length. In the second-level loop, i is the starting index of the substring. For the problem dealing with $str[i, .., i+L]$, there are two cases. If the same letter with $str[i]$ can be found in $str[i+1, .., i+L]$, the recursive equation of DP is as Line 11 shows and the recursive equation of LLS is as Line 12 to Line 16 shows. If the same letter with $str[i]$ can't be found in $str[i+1, ..., i+L]$, the recursive equation of DP is as Line 18 shows and the recursive equation of LLS is as Line 19 shows.

Proof. (1)Firstly, the problem of largest set of lines that don't intersect with each other in a circle can be converted to finding a largest subset of pairs of same uppercase letters that these pairs don't intersect with each other.

Prove: Every pair of same letters corresponds to a line in the circle as figure 3 shows.Two pairs of same letters intersect if and only if the two corresponding lines intersect. Since the orders of letters are same with the endpoints in the circle, if line AA intersects with line BB, then one and only one endpoint of line BB will appear between two A. Hence, the order is A,B,A,B. Hence, the two pairs of same letters will appear alternately in the string. Conversely, if two pairs intersect(e.g., the order is A,B,A,B) it implies the endpoints of line AA and line BB appear in the circle alternately, so the lines intersect with each other. Hence, the problem can be converted to finding the largest set of letters that don't appear alternately for any two letters.

(2)Then, the DP process in the algorithm can find the largest set correctly.

Prove(Induction Method):

Base Case: For each substring $str[k, k+1]$ with length 2 in string $str[i..j]$, if $str[k] = str[k+1]$, then the largest set is just $(str[k])$. This algorithm correctly handles the problem, it updates $DP[k][k+1]$ as 1 and updates $LLS[k][k+1]$ as $(str[k])$. If $str[k] = str[k+1]$, then the set is empty. The algorithm correctly updates $DP[k][k+1]$ and $LLS[k][k+1]$.

Hypothesis: For any substring with length smaller than $m-1$, e.g., $str[t..t+h]$in string $str[i..j]$, the algorithm can correctly find the largest set and store it in $LLS[t][t+h]$ and store the correct largest set size in $DP[t][t+h]$.

Induction: For any substring $str[t..t+m-1]$ with length m in string $str[i..j]$, if the first letter $tmp = str[t]$ appears again the sth position in the substring, then any pair of same letters appear only once in the interval bounded by the two tmp will be separated. Hence, only the pairs in the interval or outside the interval will be remained. Hence, if choose to add tmp into the largest set, the largest set consists of $LLS[t][s-1]$, $LLS[s+1][t+m-1]$ and the letter $str[k]$. However, there is an other choose that tmp is not added to the largest set. Then, any pairs in $LLS[t+1][t+m-1]$ will not be separated, since $str[t]$ is not selected and $LLS[t+1][t+m-1]$ correctly stores the largest set of substring $str[t+1][t+m-1]$ as a hypothesis, hence the largest set is same with $LLS[t+1][t+m-1]$. There are only two chooses for substring $str[t..t+m-1]$ and the two chooses produce two different sets, so the largest set of this substring is just the largest one. The algorithm compares them and chooses the larger one, hence it's right and $DP[t][t+m-1]$ and $LLS[t][t+m-1]$ can correctly store the true largest set and largest set size. In addition, if the first letter $tmp = str[t]$ doesn't

appear again in the substring, then, it can be ignored and the largest set of $str[t..t+m-1]$ is just the largest set of $str[t+1..t+m-1]$. The algorithm correctly deals with this situation.
Conclusion: The algorithm can find the largest set of letters that don't appear alternately, and the each pair of same letters corresponds to one line, so the largest set of lines that don't intersect with each other in the circle can be found. □

*Analysis.*The conversion of problem just needs to traversal the endpoints and the time cost is $O(n)$. There are two levels of loops in the algorithm and in each loop, the function FIND($str[i+1..i+L], str[i]$) will be called. The function's cost is $O(L)$. Then the major part is just some comparisons and the cost is $O(1)$. Hence the total time cost is $O((2n-1)*1+(2n-2)*2+(2n-3)*3+...+2*(2n-2)+1*(2n-1)) = O(n^3)$. $O(n^3)$ is just the worst case, if the function FIND uses *hashmap* or some other high efficient data structures, the time cost for finding can be reduced to $O(1)$ and the total time cost is $O(n^2)$. For the space cost, $LLS[1..2n][1..2n]$ uses largest extra space. Each element in the array is a set, hence the cost is $O(n^3)$.

Question 5: Bitmaps ..... **[12½]**
Problem #8 (only parts (a) and (b))

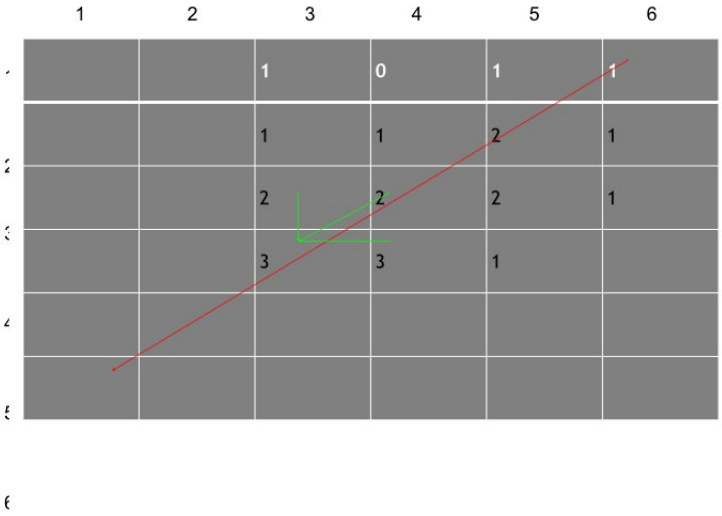

Figure 5

(a)

Solution:

Algorithm 8: The algorithm of finding maximum area of a solid square

```
1: MAX_SQUARE(M[1,...,m][1,...,n]):
2: DP[1,...,m][1,...,n];
3: memset(DP, 0, sizeof(DP));%initialize array DP.
4: DP[1][n] = (M[1][n] == 1)?1 : 0;
5: max = DP[1][n];
6: for i = 1 to m do
```

```
7:    for j = n to 1 do
8:      if M[i][j] == 1 then
9:        if i − 1 ≥ 0 and j + 1 ≤ n then
10:         DP[i][j] = min{DP[i − 1][j], DP[i][j + 1], DP[i − 1][j + 1]} + 1;
11:         max = max{DP[i][j], max};
12:       end if
13:       if (i == 1 and j + 1 ≤ n)or (j == n and i − 1 ≥ n)  then
14:         DP[i][j] = 1;
15:         max = max{DP[i][j], max};
16:       end if
17:     end if
18:   end for
19: end for
20: return max;
```

EXPLANATION:$DP[i][j]$ represents the largest edge length of a solid square that $M[i][j]$ is the left bottom bit. The algorithm solves the problem using DP. The direction of DP is from the right top corner to the left bottom corner. For each problem, the algorithm chooses the minimum of three subproblems for the result of the problem. Figure 5 shows an example. The grid can be considered as the array $DP[1..n][1..n]$. The red lines illustrates the DP direction For cell$(4, 3)$, it has three chooses as shown in green line in the figure and since the minimum of three chooses is 2, the algorithm will modify the value of the cell as 3. The algorithm deals with corner cases in line 13 to line 16.

Proof. We could prove it using Induction Method:
This algorithm updates the values of $DP[m][n]$ from the right top to the left bottom by row and row and from the right to the left for each row.
Base Case: For the first row and the last column, given this algorithm, $DP[i][j] = 1$ if $M[i][j] = 1$. It's obviously true since in this algorithm, $DP[i][j]$ represents the largest edge length of a solid square that $M[i][j]$ is the left bottom bit. This implies the bits could only expand to the up or right direction to form a larger square. So the first row and last column couldn't expand to a square whose area is larger than 1.
Let's consider the bits$(i, j) \in M[2, ..., m][1, ..., n - 1]$.
Hypothesis: Assume $DP[i - 1][j]$, $DP[i][j + 1]$ and $DP[i - 1][j + 1]$ has the largest areas of solid squares where each of them is the left bottom bit;
Induction: For $M[i][j]$, Since we know the correctness of $DP[i-1][j]$, $DP[i][j+1]$ and $DP[i-1][j+1]$, we could imagine that there is a solid square with edge length $DP[i][j+1]$ on its right, a solid square with edge length $DP[i - 1][j]$ above it and a solid square with edge length $DP[i - 1][j + 1]$ on its upper right corner. And these squares are the largest near $M[i][j]$. And if $M[i][j] = 0$, then thus solid square doesn't exist, hence $DP[i][j] = 0$. If $M[i][j] = 1$, then $DP[i][j] = min\{DP[i - 1][j], DP[i][j+1], DP[i-1][j+1]\}+1$ is obviously true. If we could find a solid square with a larger edge length than $DP[i - 1][j + 1] + 1$, let's assume its edge length $DP[i - 1][j + 1] + 1 + k$, since the solid square $M[i - DP[i - 1][j + 1], ..., i - 1][j + 1, ..., j + DP[i - 1][j + 1]]$ whose left bottom bit is $M[i - 1][j + 1]$ is in $M[i - DP[i - 1][j + 1] - k, ..., i][j, ..., j + DP[i - 1][j + 1] + k]$, then $M[i - DP[i - 1][j + 1], ..., i - 1][j + 1, ..., j + DP[i - 1][j + 1]]$ could expand to a larger solid square $M[i-1+DP[i-1][j+1]-k, ..., i-1][j+1, ..., j+1+DP[i-1][j+1]+k]$ which is contradict to the Hypothesis, so $DP[i][j] \leq DP[i-1][j+1]+1$. Similarly, we could get $DP[i][j] \leq DP[i-1][j]+1$ and $DP[i][j] \leq DP[i][j + 1] + 1$. Hence $DP[i][j] = min\{DP[i - 1][j], DP[i][j + 1], DP[i - 1][j + 1]\} + 1$ is true.
In conclusion, for every bit(i, j), $DP[i][j]$ could store the largest edge length of a solid square that $M[i][j]$ is the left bottom bit successfully. For each solid square in $M[1, ..., m][1, ...n]$, its left bottom is bound to be in $M[1, ..., m][1, ...n]$. By this algorithm, we have figured out all the solid squares'

edge length and stored them in $DP[1,...,m][1,...,n]$. And we just compare them and then find out the maximum. □

*Analysis.*We update every $DP[i][j]$, so the time complexity is $O(mn)$. And the space complexity is obviously $O(mn)$.

(b)

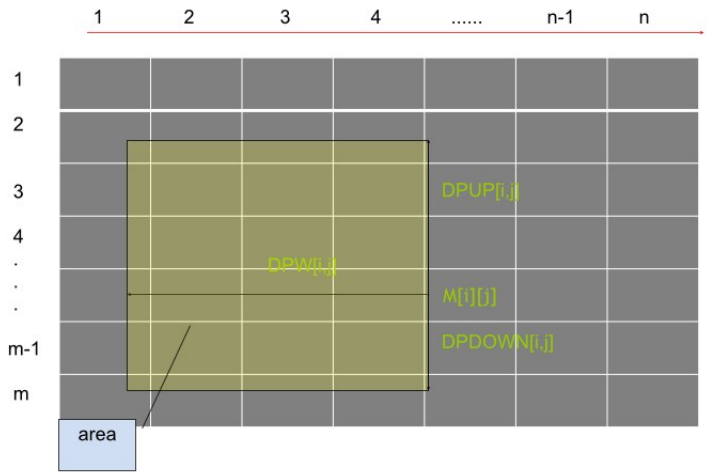

Figure 6

Solution:

Algorithm 9: Algorithm to find maximum area of a solid block

Require: The $m \times n$ bitmap - $M[1,...,m][1,...,n]$ of 0s and 1s;

Ensure: The maximum area max of a solid block in M which is a subarray of the form $M[i,...i'][j,...,j']$ containing only 1s;

1: create three arrays $DPUP[1,...,n]$, $DPDOWN[1,...,n]$, $DPW[1,...,m][1,...,n]$;

2: For each specific bit $M[i][j]$, we try to expand it in the left direction continuously until we meet a 0. And then store the biggest width in $DPW[i][j]$. We do this by column and column, so we have $DP[i][j] = DP[i][j-1] + 1$;

3: % Scan the bitmap from left column to right column and for a specific column we scan from top to bottom.

4: **for** $j = 1$ to n **do**

5: for the width - $DPW[i][j]$ of a specific bit, we move it in the up and down directions to find the largest height ($h\_up - h\_down$) with the constrains that it will not meet 0. And we store the biggest distance the width of bit(i,j) could move up in $DPUP[i]$ and the biggest distance to move down in $DPDOWN[i][j]$.

6: **for** $i = 1$ to m **do**

7: $DPUP[i] = i$;

8: **while** $(DPUP[i] \geq 1)$ and $(DPW[i][j] \leq DPW[DPUP[i]-1][j]$ **do**
9: $DPUP[i] = DPUP[DPUP[i]-1]$;
10: **end while**
11: **end for**
12: **for** $i = m$ to 1 **do**
13: $DPDOWN[i] = i$
14: **while** $(DPDOWN[i] \leq m)$ and $(DPW[i][j] \leq DPW[DPDOWN[i]+1][j]$ **do**
15: $DPDOWN[i] = DPDOWN[DPDOWN[i]+1]$;
16: **end while**
17: calculate the largest solid block $M[i][j]$ could form by $DPW[i][j] \times (DPDOWN[i] - DPUP[i]+1)$. And update max if we get a larger solid block;
18: **end for**
19: **end for**
20: return max;

EXPLANATION: $DPW[i][j]$ stores the most left side the cell $M[i][j]$ can expand to(if value of the left side cell in the grid is 1,$DPW[i][j]$ can add one). $DPUP[i][j]$ is the most up side the cell can expand to and $DPDOWN[i][j]$ is the most bottom side the cell can expand to. Figure 6 shows the three values. The algorithm figures out $DPW[i][j]$ by traversing the grids and figures out $DPUP[i][j]$ and $DPDOWN[i][j]$ using DP. Then it calculates the largest area of each cell can expand to by these three values and updates the largest area of solid block in the grid if necessary.

Proof. For each bit(i, j) in $M[1, ..., m][1, ..., n]$, we will have its largest width filled with 1s expanding to its left in this algorithm and **I will use the width of bit(i, j) to represent this largest width next**. So $DPW[1, ..., m][1, .., n]$ represent all widths of possible largest solid blocks in this bitmap. For each possible largest solid block, we just need to find its largest height and then multiply its width.
Line 6 to line 18 describes the Dynamic Programming process of finding these largest heights. For the up direction, $DPW[DPUP[i]-1][j] \geq DPW[i][j]$ tells us that the width above current top of this height is larger $DPW[i][j]$ and $DPW[i][j]$ won't meet 0. And since we have got the top the width of bit$(DPUP[i]-1, j)$ could arrive - $DPUP[DPUP[i]-1]$, we could move current top to $DPUP[DPUP[i]-1]$ because the width of bit(i, j) won't meet 0 in the up direction. If we meet any 0s, it means there is 0s in the way that the width of bit$(DPUP[i]-1, j)$ move to its top and it's a contradiction. However, if $DPW[DPUP[i]-1][j] < DPW[i][j]$, this block couldn't expand any more obviously because there is some 0s on the left of the width of bit$(DPUP[i]-1, j)$ and it couldn't expand any more as well if it has arrived the bitmap's edge. It's similar to the situation of $DPDOWN[i][j]$.
Since we could calculate each block's area correctly, we just need to compare them and find the largest one. □

*Analysis.*For every bit, we need to find its width, so we should implement the process for $O(mn)$ times. For each process, we just need constant operations. So the time complexity of this process is $O(mn)$. Then for each width of a specific bit, we will find the largest range it could move up and down and it may be the all the rows. So this process's time complexity is $O(m)$. There are $O(mn)$ bits, so the time complexity is $O(m^2n)$. For the whole algorithm, the time complexity is $O(m^2n + mn) = O(m^2n)$. And the space complexity is $O(m + m + mn) = O(mn)$.

Question 6: Thirsty Students ..... **[12½]**
Problem #9 **Credit the idea to Liang Zhang and do by myself**.

Solution:

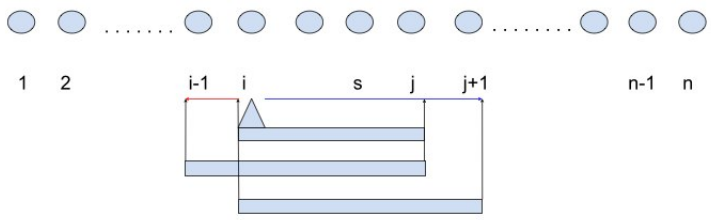

Figure 7

Algorithm 10: Least soda of thirty students

Require: The number of exits n, the numbers of students dropped off at each exit $drop[1..n]$, the current location s, and the travel time between successive exits $tarvel\_time[1..n-1]$.

Ensure: A strategy that drops the students off so that they drink as little soda as possible.

1: $Left[1..s][1..n-s+1] \leftarrow 0$;
2: $Right[1..s][1..n-s+1] \leftarrow 0$;
3: $chooseL[1..s][1..n-s+1] \leftarrow 0$;
4: $chooseR[1..s][1..n-s+1] \leftarrow 0$;
5: **for** $L \leftarrow n$ to 1 **do**
6: **for** $m \leftarrow 0$ to $L-1$ **do**
7: $i \leftarrow k-m$;
8: $j \leftarrow k+(L-m-1)$;
9: $unknown \leftarrow 0$;
10: $travelTime\_toRight \leftarrow 0$;
11: $travelTime\_toLeft \leftarrow 0$;
12: **for** $stop \leftarrow 1$ to $i-1$ **do**
13: $unknown \leftarrow unknown + drop[stop]$;
14: **end for**
15: **for** $stop \leftarrow j+1$ to n **do**
16: $unknown \leftarrow unknown + drop[stop]$;
17: **end for**
18: **for** $to\_right \leftarrow i$ to j **do**
19: $travelTime\_toRight \leftarrow travelTime\_toRight + travel\_time[to\_right]$;
20: **end for**
21: **for** $to\_left \leftarrow j-1$ to $i-1$ **do**
22: $travelTime\_toLeft \leftarrow travelTime\_toLeft + travel\_time[to\_left]$;
23: **end for**
24: **if** $Left[i-1][j] + unknown * travel\_time[i-1] < Right[i][j+1] + unknown * travelTime\_toRight$ **then**

```
25:       Left[i][j] ← Left[i − 1][j] + unknown ∗ travel_time[i − 1];
26:       chooseL[i][j] ← i − 1;
27:     end if
28:     if Left[i − 1][j] + unknown ∗ travel_time[i − 1] ≥ Right[i][j + 1] + unknown ∗
        travelTime_toRight then
29:       Left[i][j] ← Right[i][j + 1] + unknown ∗ travelTime_toRight;
30:       chooseL[i][j] ← j + 1;
31:     end if
32:     if Right[i][j+1]+unknown∗travle_time[j] < Left[i−1][j]+unknown∗travelTime_toLeft
        then
33:       Right[i][j] ← Right[i][j + 1] + unknown ∗ travle_time[j];
34:       chooseR[i][j] ← j + 1;
35:     end if
36:     if Right[i][j+1]+unknown∗travle_time[j] ≥ Left[i−1][j]+unknown∗travelTime_toLeft
        then
37:       Right[i][j] ← Left[i − 1][j] + unknown ∗ travelTime_toLeft;
38:       chooseR[i][j] ← i − 1;
39:     end if
40:     if i = j then
41:       call SEARCH_PATH(chooseL[1..s][1..n − s + 1], chooseR[1..s][1..n − s + 1]);
42:     end if
43:   end for
44: end for
 1: SEARCH_PATH(int chooseL[1..s][1..n − s + 1], int chooseR[1..s][1..n − s + 1])
 2: i ← s, j ← s;
 3: judge ← "Left";
 4: while i! = 1 and j! = n do
 5:   if judge = "Left" then
 6:     k ← Left[i][j];
 7:   end if
 8:   if judge = "Right" then
 9:     k ← Right[i][j];
10:   end if
11:   output "Next step, the bus should turn" + judge + "and stop at" + station[k];
12:   if k = j + 1 then
13:     j ← j + 1;
14:     judge ← "Right";
15:   end if
16:   if k = i − 1 then
17:     i ← i − 1;
18:     judge ← "Left";
19:   end if
20: end while
```

EXPLANATION: The algorithm solves the problem using dynamic programming. $Left[i][j]$ stores the least soda that students will continue to drink after the bus has stopped at all stations in the interval(i, j)(including i and j), dropped corresponding students and stops at station i at last. $Right[i][j]$ stores the least soda value if the bus stops at station j at last. $1 \leq i \leq s$ and $s \leq j \leq n$. If the bus stops at the left side of the interval (i, j), then it has two chooses - turn left to station $i-1$ and turn right to station $j+1$ as Figure 7 shows and the two chooses are based on two subproblems. Hence the value of $Left[i][j]$ can be achieved by comparing the two chooses. $Left[i-1][j]$ and $Right[i][j+1]$ have the results of the two subproblems, hence $Left[i][j]$ is achieved by comparing

these two results in this algorithm.
If the bus turns left to station $i-1$, since $Left[i-1][j]$ has been solved firstly, then the least soda value is the soda amount students still on the bus when leaving station i will drink between station $i-1$ and station i pluses the amount students still on the bus when leaving station $i-1$ will drink $Left[i-1][j]$. And it's similar to another choose. The choose is between Line 24 to Line 31. *unknown* is the number of students still on the bus after stopped all stations between i and j. *to_right* is the total travel time the bus travels from station i to station $j+1$. The chooses for $Right[i][j+1]$ is between Line 32 to Line 39. *to_left* is the total travel time the bus travel from station j to station $i-1$.
$chooseL[i][j]$ stores the choose of the bus in the subproblem $L[i][j]$. And $chooseR[i][j]$ stores the choose in the subproblem $R[i][j]$.
When $i=j=s$, the algorithm calls function SEARCH_PATH to search the travel path of the bus.

Proof. Firstly, for each subproblem, the bus will stop either the most left side or the most right side. It's obvious that when a bus passes through a station, it should drop the students off, otherwise, the students will continue to drink soda. Make an assumption the bus stops in an internal station k finally with least soda amount, the bus has stopped at the most left station i and the most right station j already for sure. Hence, the bus has passed through all stations including k before having reached both station i and station j otherwise it won't reach both sides. Since the bus stops at k finally, it should be the first time the bus passes through k, otherwise there is no need for it to visit station k again. Hence, the bus haven't stopped at both i and j. That's a contradiction and the bus can't stop at an internal station with least soda amount. Hence, for each subproblem, the bus stops in either side at last.
Conclusion1: If station i is the most left station the bus has visited and station j is the most right station the bus has visited, it must have visited all the stations between them and finally stops at the two stations in order to minimize the soda amount.
Then, the major dynamic programming part of the algorithm can be proved by Induction Method.
Base Case: If all stations $1,..,n$ have been visited, then no student is on the bus. Hence the least soda amount in future is 0. Hence $Left[1][n]=Right[1][n]=0$.
Hypothesis: If station $i-1$ is the most left station bus has visited and j is the most right station bus has visited, then all stations between them have been visited, corresponding students have been dropped and the bus stops at either $i-1$ or j given **Conclusion1**. Make a hypothesis that $Left[i-1][j]$, $Right[i][j+1]$ are correct, i.e., $Left[i-1][j]$ is the least soda amount students will drink after the bus has visited all stations $i-1,..j$ and stops at $i-1$ eventually, $Right[i-1][j]$ is the least soda amount students will drink if stops on j eventually and so on. Similarly, make a hypothesis that $Left[i][j+1]$ and $Right[i-1][j]$ are correct as well.
Induction: If station i and station j is separately the most left and the most right stations the bus has stopped, given **Conclusion1**, it has visited all stations between i and j. Suppose there are N students left on the bus now. If it stops at i, it has two chooses - turn left or turn right.
If it turns left, since $1,...,i-1$ haven't been visited, station $i-1$ will become the first station the bus will stop. When the bus leaves station i, the N students will drink $N \times travel\_time[i-1]$ soda before stopping at station $i-1$. Since $Left[i-1][j]$ is the least soda amount the students will drink after the bus has visited $i-1,i,...,j$ and stops at $i-1$ as a hypothesis, hence students will drink more $Left[i-1][j]+N \times travel\_time[i-1]$ if turns left.
If it turns right, since $i+1,...,j$ all have been visited and $j+1,..,n$ haven't been visited, it will stop at station $j+1$ firstly. After the bus leaves station i and turns right, the students will drink $N \times (travel\_time[i]+travel\_time[i+1]+...+travel\_time[j])$ soda. Since $Right[i][j+1]$ is the least soda amount the students will drink after the bus visiting $i,...,j,j+1$ and stops at $j+1$, students will drink more $Right[i][j+1]+N \times (travel\_time[i]+travel\_time[i+1]+...+travel\_time[j])$ soda if the bus turns right. Hence least soda amount after the bus visits $i,...,j$ and stops at i is the minimum of the two chooses and $Left[i][j]$ is correct.
Similarly, $Right[i][j]$ is correct.

Conclusion2: The dynamic programming part is correct as shown. Hence when the bus is at station s and hasn't visited any other stations, the minimum of the chooses - turns left or turns right is just the least soda amount students will drink. □

*Analysis.*As the algorithm shows, the dynamic programming part costs $O(n^2)$ time and the searching part costs $O(n)$. Hence the total time cost is $O(n^2)$. And the space cost is $O(n^2)$.

Question 7: Nadira ..... **[12½]**
Problem #10

Solution:
(a)
For example, we now should give change \$455 to the customer. Using the greedy algorithm, we will at first choose \$365, and then \$52, \$28,\$7 and three \$1, so 7 bills in total. However, we only need 5 \$91 bills.

(b)

Algorithm 11: Recursive algorithm of minimum bills

```
1: int b_sort[8] = {1, 4, 7, 13, 28, 52, 91, 365};
2: MIN_BILLSBYREV(k):
3: if k == 0 then
4:    return 0;
5: end if
6: min = MAX_INT;
7: for i ← 8 to 1 do
8:    if k − b_sort[i] ≥ 0 then
9:       tmp ←MIN_BILLSBYREV(k − b_sort[i]);
10:      if 1 + tmp < min then
11:         min ← tmp + 1;
12:      end if
13:   end if
14:   i ← i − 1;
15: end for
16: return min;
```

EXPLANATION:MAX_INT is the maximum positive integer. This algorithm pcalls itself recursively. The array $b\_sort[8]$ will store the eight kinds of denominations.

Proof. We could prove it by Induction Method:
Base Case: For $k_1 = 1||4||7||13||28||52||91||365$, this algorithm will return 1, which is obviously correct;
Hypothesis: $\forall k_1 < k$, let's assume that MIN_BILLSBYREV(k_1) will return the minimum number of bills needed for the k_1 Dream Dollars;
Induction: Now for $k_1 = k$, for $i \leq 8$, if $k \geq b\_sort[i]$ and we choose to use this kind of denomination, then the customer needs another $k - b\_sort[i]$ Dream Dollars. However, now we have known the fewest bill needed of $k - b\_sort[i]$ Dream Dollars. So for this choose, we just need to add another one bill. However, we have other 7 chooses. So we should compare them and choose the minimum one. Hence, this process is correct. □

*Analysis.*For this recursive algorithm, we just need constant space complexity. Let's

assume the operation needed is $T(k)$. So we could get a recursion that $T(k) \leq T(k-365)+T(k-91)+T(k-52)+T(k-28)+T(k-13)+T(k-7)+T(k-4)+T(k-1)$ and we assume k is large. Hence the time complexity is $O(c^n)$ and it works very slowly especially when k is large.
(c)

Algorithm 12: Dynamic Programming Al of minimum number of bills needed

```
1: b_sort[8] = {1, 4, 7, 13, 28, 52, 91, 365};
2: MIN_BILLSBYDP(k):
3: DP[1..k] ← 0;
4: for i ← 1 to k do
5:    min = i;
6:    for j ← 1 to 8 do
7:       if i > b_sort[j] then
8:          if DP[i − b_sort[j]] + 1 < min then
9:             min ← DP[i − b_sort[j]] + 1;
10:         end if
11:      end if
12:   end for
13:   DP[i] ← min;
14: end for
```

Claim:$DP[k]$ stores the minimum number of bills needed. The array $b\_sort[8]$ will store the eight kinds of denominations.
EXPLANATION: For problem dealing with money i in total, the recursive equation is as Line 6 to Line 10. There are eight chooses and corresponding eight subproblems, the optimal solution is the best one of them.

Proof. We could prove it using Induction Method.
Base Case: For this algorithm, $DP[0] = 0$, it's obviously true;
Hypothesis: $\forall k_1 < k$, let's assume that $DP[k_1]$ is the minimum number of bills needed for k_1 Dream Dollars;
Induction: Now for $k_1 = k$, let's assume it's larger than 365. So it has eight chooses at first, we could give change $1, 4, 7, 13, 28, 52, 91, 365$ to the customer firstly. Then we have known the minimum number of bills needed for the remaining Dream Dollars. Hence, we just need to add another bill for each situation and compare them. So $DP[k]$ will store the minimum number of bills needed for k Dream Dollars. □

*Analysis.*The space complexity is $O(n)$. And the time complexity is $O(n * 8) = O(n)$.

Question 8: Solitaire ..... **[12½]**
Problem #12

Solution:

Algorithm 13: maximum score of Vankin's Mile

```
1: MAX_SCOREOFVMILE(score[1, ..., n][1, ..., n]):
2: DP[n][n];%initialize the array for storing DP information
3: DP[n][n] = score[n][n];
4: max = DP[n][n];
5: for i = n to 1 do
```

```
6:    for j = n to 1 do
7:      if j + 1 ≤ n and i + 1 ≤ n then
8:        DP[i][j] = max{DP[i + 1][j], DP[i][j + 1]} + score[i][j];
9:        max = max{DP[i][j], max};
10:     end if
11:     if j == n and i + 1 ≤ n then
12:       DP[i][j] = max{DP[i + 1][j] + score[i][j], score[i][j]};
13:       max = max{DP[i][j], max};
14:     end if
15:     if i == n and j + 1 ≤ n then
16:       DP[i][j] = max{score[i][j], score[i][j] + DP[i][j + 1]};
17:       max = max{DP[i][j], max};
18:     end if
19:   end for
20: end for
21: return max;
```

Claim: $DP[i][j]$ represents that maximum score we could achieve starting at the position (i, j) in the grid.

EXPLANATON: There are three cases. If the starting position is on the right edge of the grid, the recursive equation is as Line 12 shows. If the starting position is on the bottom edge of the grid, the recursive equation is as Line 16 shows. For the two cases, there are two chooses - continue to move or stop since the edge has been reached. If the starting position is not on the bottom or right edges, the recursive equation is as Line 8 shows, and there are two chooses: turn right or turn down. If it chooses to turn right, the subproblem is what's the max score if starting at the right position and if it chooses to turn down, the subproblem is what's the max score if starting at the down position. For each case, the solution is the optimal one among all the chooses.

Proof. We could use Induction Method to prove it and let's pick any point(i, j) in the grid:

Base Case: If we start at point (n, n), then we have no choose because there is no square on right or left, so $DP[n][n] = score[n][n]$;

Hypothesis: $\forall$ points$(k_1, k_2) \in score[i, .., n][j, ..., n]$ except $score[i][j]$, $DP[k_1][k_2]$ has the maximum score if we start at point(k_1, k_2);

Induction: Now we start at point(i, j). If $i < n$ and $j < n$, then we have two chooses - go a square right and go a square down. And we have to pick a choose since the token has to be off the edge of the board. $DP[i + 1][j]$ and $DP[i][j + 1]$ could tell us the maximum score we could achieve starting on point$(i + 1, j)$ and point$(i, j + 1)$. And we just choose the larger one and then add $score[i][j]$ to get the maximum score starting on point(i, j). If $i = n$ and $j < n$, then we have arrived the edge of the board, hence we could choose to move a square right or choose to do nothing. Since the value of grid square could be negative, zero or positive and $DP[i][j + 1]$ could tell us the maximum score we could achieve continuing to move right, so we just compare the chooses. So is similar to the situation that $i < n$ and $j = n$. Hence the algorithm is still correct for point(i, j).

In conclusion, we could know the maximum score starting on any point(i, j) in the grid and hence we could get the maximum score of a special grid just by comparing all the values $DP[i][j]$. □

*Analysis.*We implement the algorithm from the right bottom of the grid to the left top of the grid row by row and from the most right one to the most left one for each row. So the time complexity is $O(n^2)$ and the space complexity is obviously $O(n^2)$.