Homework 3

Jiaxin Li[1] and Sunjun Gu[2]

[1]jli2274@wisc.edu

[2]sgu59@wisc.edu

1 Question 1

Algorithm 1 CanSeg

Input: An intefer i where $1 \leq i \leq n+1$
Output: True if $S[i...n]$ can be segmented into words at all; Else false
0: **procedure** CanSeg(i)
1: **if** $CanSegArr[i] \neq Null$ **then**
2: **return** CanSegArr[i]
3: **end if**
4: **if** $i = n+1$ **then**
5: $CanSegArr[i] \leftarrow True$
6: **return** $True$
7: **end if**
8: $SegPass \leftarrow False$
9: **for** $k = i, ..., n$ **do**
10: **if** $isWord(i, k)$ and $CanSeg[k+1]$ **then**
11: $SegPass \leftarrow True$
12: **end if**
13: **end for**
14: $CanSegArr[i] \leftarrow SegPass$
15: **return** $SegPass$
15: **end procedure**=0

Algorithm 2 StringSeg

Input: An String S[1,...,n]
Output: True if $S[1...n]$ can be segmented into words at all; Else false
0: **procedure** StringSeg($S[1, ..., n]$)
1: $CanSegArr[1, ..., n+1] \leftarrow$ a new 1-d Boolean value array with Null initialization
2: **return** $CanSeg(1)$
2: **end procedure**=0

∗IsWord

Assume that we have access to this subroutine IsWord(i, j) algorithm.

where **IsWord**(**i**, **j**)

Input: Indices i, j with $i \leq j$.

Output: True if $S[i, ..., j]$ is a "word" in the foreign language.Else, False

Runtime: $O(1)$

Proof of Correctness :

We want to prove that the given the string $S[1, ..., n]$, the StringSeg algorithm could correctly determine whether it can be segmented into word at all. In this algorithm, after creat a new Boolean value array to store the segment-ability of different length of string S, then it calls on the CanSeg algorithm and return CanSeg(1) as the result. Therefore, StringSeg has the same correctness as CanSeg. We first prove the correctness of CanSeg.

The proposition here CanSeg(i) is that this algorithm can determine whether the given string $S[i, ..., n]$ be segmented into words at all(True) or not(False) and store the boolean value on the array.

Base case : When $i = n + 1$, we define the empty String $S[n + 1, n]$ as segmentable,so it returns True and thus CanSeg($n + 1$) is correct.

Induction Step : By strong induction, suppose $CanSeg(i)$ is correct for all $k \leq i \leq n + 1$.($k > 1$) Such that given a String $S[i, ..., n]$, the algorithm can determine whether the given string can be segmented into words at all.

To prove the correctness of $CanSeg(k - 1)$ where k-1 represents i:

The given string $S[k - 1, ..., n]$ has the start index k-1 and end index n. $k \leq i \leq n + 1$ ($k > 1$) $\rightarrow 1 \leq k - 1 \leq n$

Because initialized all the value to Null when create CanSegArr. And $k - 1 \leq n \rightarrow i \neq n + 1$. So skip Line 1 to 7, $line8 \sim line15$ will be executed.

Line 8: creat a temporary boolean value to record whether the String is segmentable and initialize it to False.

Line 9 $\sim$ 13: The $S[k - 1, ..., n]$ is segmentable if and only if there is some index j($k - 1 \leq j \leq n$) such that $S[k - 1, ..., j]$ is a word and $S[j + 1, ..., n]$ is segmentable(or empty, which we defined as segmentable).Therefore, use for loop to parse the string into the first word with different length and the remaining string and check whether this string is also segmentable. If these two conditions are met at the same time($isWord(i, j)$ checks whether $S[k-1, ..., j]$ is a word and $CanSeg(j+1)$ checks whether $S[j + 1, ..., n]$ is segmentable), indicating that the string $S[k - 1, ..., n]$ is segmentable.
By the given correctness of $isWord(k - 1, j)$ and the induction hypothesis that $CanSeg(j + 1)$ correctly works, it could prove that the $CanSeg(k - 1)$ correctly works as well.

Explain for the correctness of $CanSeg(j + 1)$:
By induction hypothesis, $CanSeg(i)$ is correct for all $k \leq i \leq n + 1$ and $k - 1 \leq j \leq n$. Therefore, $\rightarrow k \leq j + 1 \leq n + 1$,which meets the requirement of induction hypothesis. That is, $CanSeg(j + 1)$

returns True if $S[j+1, ..., n]$ is segmentable which works as empty string or recursively calls on both isWork and CanSeg.
As long as there is one method of string division whose $isWord(i, j)$ and $CanSeg(j+1)$ are both True, $S[k-1, ..., n]$ is segmentable and then set its SegPass boolean value to True.

Line 14: Store the SegPass value which means $S[k-1, ..., n]$ is segementable(True) or not(False) on the (k-1)-th index of CanSegArr. Therefore, if we want to check different part of string S($S[i, ..., n]$) is segmentable or not, we could just check the value on the i-th index of CanSegArr.

Line 15: Return the SegPass value which means $S[k-1, ..., n]$ is segementable(True) or not(False). Therefore, the segment-ability of $S[k-1, ..., n]$ would be returned correctly by the algorithm.

Overall,$CanSeg(j+1)$ and $isWord(i, j) \rightarrow CanSeg(k-1)$.
By mathematical inductive principle, $CanSeg(i)$ is true for all $1 \leq i \leq n$.
From the above explanation, StringSeg algorithm could call on the $CanSeg(1)$ to check whether the String $S[1, ..., n]$ is segmentable or not. Due to the correctness of $CanSeg(1)$, StringSeg is correct as well.

Proof of Time Complexity :

We first analysis the runtime of CanSeg algorithm. For local work, from Line1 to Line7, if $CanSegArr[i] \neq Null$ or $i = n+1$, the algorithm checks the condtion then assign or return the value.These lines only run in constant time. From Line 9, the algorithm needs to examine $n - i + 1 = O(n)$ times splits of the S to word and remaining strings. The examination and assignment and return work on other lines only takes constant time. And the constant time could be ignored. Therefore, the total local work is O(n). In addition, on Line10, there are recursively calls on isWord and CanSeg algorithm. From the given information, the isWord algorithm only takes constant time. And toally O(n) unique calls to CanSeg to compute. Therefore the time complexity is the call times $\times$ local work = O(n)·O(n) = O(n^2).

The StringSeg algorithm fisrt creat and initialize a 1-d array(Line1) with O(n) time and then call and return the CanSeg(1) algorithm(Line2) which runs in O(n^2) time. The total time complexity = O(n) + O(n^2) = O(n^2).

Combine the above analysis, the algorithm we designed solves 'string parses to words' problem in O(n^2) time.

2 Question 3

The algorithm is on the next page

Proof of Correctness:
This algorithm uses **Bottom-up** dynamic programming. Let us firstly explain the notations used

Algorithm 3 MaxDistance

Input: An integer e and an array $d[1, ..., n]$ of $n \geq 1$ positive integers
Output: The maximum distance you can run subject to the constraints

```
0: procedure MaxDistance(e, d)
1: T[0,1,...e],R[0,1,...e] ← new arrays of integers with length e+1
2: T'[0,1,...e], R'[0,1,...e] ← new arrays of integers with length e+1
3: for i = 1 → e do
4:    T[i]←0
5:    R[i]←0
6:    T'[i]←0
7:    R'[i]←0
8: end for
9: n ← length of d
10: for i = 1 → n do
11:    for j = 0 → e do
12:       if j > i then
13:          break
14:       end if
15:       if j > (n − i) then
16:          break
17:       end if
18:       if j == 0 then
19:          T'[j] ← 0
20:          R'[j] ← Max(T[j + 1], R[j + 1], R[0])
21:       else if j == e then
22:          T'[j] ← T[j − 1] + d[i]
23:          R'[j] ← 0
24:       else
25:          if j != 1 T'[j] ← T[j − 1] + d[i] else: T'[j] ← R[0] + d[i]
26:          R'[j] ← Max(T[j + 1], R[j + 1])
27:       end if
28:    end for
29:    T ← T'
30:    R ← R'
31: end for
32: return R'[0]
32: end procedure=0
```

in following proof:
Let $T(i, j)$ be the maximum possible distance that can be reached **after** i-th minute and at the end of this minute, the exhaustion level is exactly j, given the person **runs** in the i-th minute.
Let $R(i, j)$ be the maximum possible distance that can be reached **after** i-th minute and at the end of this minute, the exhaustion level is exactly j, given the person **rests** in the i-th minute.
To be noticed, the arrays used in this algorithm T, R, T' and R' are not 2 dimensional but 1 dimensional to meet the space complexity requirement. If we fill T(i,j) into a i by j table, then array T used in the pseudo code should be one column of this table, and T' should be the column next to T. So in the pseudo codes, the first index of T(i,j) and R(i,j) is omitted.
Because the problem requires the exhaustion level at the end of the run must be 0, so the last minute (n-th minute) must be resting. Therefore, the aim of this algorithm is finding the correct value of **R(n,0)**
We use the mathematical induction to prove this:
Proposition The proposition here P(n) is this algorithm could return correct value of T(i,j) and R(i,j) when i = n.
Base Case:
when $n = 1$.
Firstly, the arrays T[0,1,...e], R[0,1,...e], T'[0,1,...e] and R'[0,1,...e] are all initialized with zeros (Line 1$\sim$ 8), then the for loop in line 10 $\sim$ 31 are executed exactly once.

When the exhaustion level j = 0 (line 18$\sim$ 20), it means the person must rest in the first minute. The maximum distance that can be reached is 0. So R(1,0) should be 0, which means R'[0] should be 0. Line 20 assign the max value among T[1], R[1] and R[0] to R'[0]. T[1],R[1] and R[0] are all 0. So the algorithm can give correct value of R(1,0). Because running in the first minute will elevate j to one, which means T(1,0) is an impossible state under such restriction. So T(1,0) should be 0 as well. In line 11, T'[0] is assigned with 0, which is correct.

When the exhaustion level j = 1 (line 24 $\sim$ 27), it means the person must run in the first minute. The maximum distance that can be reached is d[1]. So T(1,1) i should be d[1], and R(1,1) is impossible state, R(1,1) should be 0. In line 25, T'[1] is assigned with T[0]+d[1], which is exactly d[1]. In line 18, R'[1] is assigned with Max(T[2], R[2]) = 0. So R(1,1) and T(1,1) are both correct.

When the exhaustion level j = 2$\rightarrow$ e,T(1,j) and R(1,j) are all impossible states, which means T(1,j) = R(1,j) = 0. Line 12 $\sim$ 13 are executed, the for loop break immediately when j increases to 2, so T(1,j) and R(1,j) ($j \geq 2$) are all remain 0, which are correct.

Overall, the T(1,j) and R(1,j) are all correct for $0 \leq j \leq e$, P(1) is true.

Inductive Step:

Suppose P(k) ($k \geq 1$) is true, which means T(k,j) and R(k,j) are all correct for $0 \leq j \leq e$. Accordingly, because the previous column of values are stored in the "memory" array T and R at the end of each cycle (line 29 and 30). So assuming P(k) is true equals to assuming the values in T and R are all correct.

To prove P(k+1) is true:
The column T(i = k, j) corresponds to array T in the algorithm.
The column R(i = k, j) corresponds to array R in the algorithm.
The column T(i = k+1, j) corresponds to array T' in the algorithm.
The column R(i = k+1, j) corresponds to array R' in the algorithm.

First special situation: $j > i$, such states are invalid. Because after i-th minute, even the person didn't rest at all, the maximum exhaustion level that can be reached is i, which is less than current j. So when j increases to larger than i, the for loop that iterates j breaks immediately, then all T'[j] and R'[j] remain the initial value 0, which are correct.

Second special situation: $j > (n - i)$, such states are also invalid. Because after i-th minute, even the person takes all of the remaining minutes (n-i minutes) to rest, he or she can not get 0 exhaustion level at the end of the run. so once j reaches to larger than n-i, the for loop breaks immediately, all T'[j] and R'[j] remain the initial value 0, which are correct

In other cases:
When $j = 0$, line 18 $\sim$ 20 are executed, the person must be resting in the (k+1)th minute, so T(k+1, 0) must be invalid, which means T'[0] should be 0. In line 19, T'[0] is assigned with 0, which is correct.
For R(k+1,0), there are 3 possible previous state:
(1 and 2) The person rested in the k-th minute, and his exhaustion level after k-th minute could be 1 or 0 to ensure the exhaustion level after the rest in (k+1)th minute can reach 0. then R(k+1,0) = R(k,0) or R(k,1).
(3) The person run in the k-th minute, and his exhaustion level after k-th minute must be 1 ensure the exhaustion level after the rest in (k+1)th minute can reach 0. Therefore, R(k+1,0) = T(k,1)
To make the distance largest at current state, we take the max value among these three previous state.
This is done by line 20.

When $j = e$, line 24 $\sim$ 26 are executed, the person must be running in the (k+1)th minute, which means R(k+1, e) is invalid. Because if the person rested in the (k+1) minute, then his exhaustion level at the end of k-th minute will be e+1, which violate the restriction. Therefore, R(k+1, e) is assigned with 0 in line 23.
There are only 1 possible stage for T(k+1, e):
(1) The person run in the k-th minute, and the exhaustion level after k-th minute is e-1. Because if the person rested in the k-th minute, and the exhaustion level didn't get to 0 at the end of k-th minute, he must rest in the (k+1)th minute as well. However, we've already know the person must be running in the (k+1)th minute when $j = e$, which causes conflict.
$T(k + 1, e) = T(k, e - 1) + d[k + 1]$, which is done by line 22. Under the induction assumption T(k,e-1) is correct, so that T(k+1,e) must be correct.

When j is other valid values rather than 0 or e:

1. If the person rests in the (k+1) minute, the exhaustion level at the end of k-th minute must be j+1:

The person can either rest or run in the k-th minute:

(1) If the person run in the k-th minute, then $R(k+1, j) = T(k, j+1)$

(2) If the person rest in the k-th minute, then $R(k+1, j) = R(k, j+1)$

To make the distance largest at current state, we take the max value among these 2 possible previous state.

This is done by line 26.

2. If the person runs in the (k+1) minute, the exhaustion level at the end of k-th minute must be j-1:

The person can only run in the k-th minute if $j > 1$

So $T(k+1, j) = T(k, j-1) + d[k+1]$ This is done by line 25.

If $j = 1$, then the person must be resting in the k-th minute, and at the end of k-th minute, the exhaustion level is 0. So $T(k+1, 1) = R(k, 0) + d[k+1]$ This is also done by line 25.

Above all, given P(k) is correct, T(k,j) R(k,j) (T, R) are correct, we can get correct T(k+1, j) R(k+1, j) (T' and R') by this algorithm. So P(k)⇒ P(k+1)

By mathematical induction, P(n) is true for all n≥ 1

Proof of Time Complexity:

Line $1 \sim 8$ are executed only once in the whole procedure. Because these lines iterate i from 1 to e. So the time complexity is O(e).

Line $10 \sim 30$ is a for loop that is executed n times:

To analyze the local complexity in each cycle:

A for loop run at most e times are done by line $11 \sim 28$. In each cycle, there are only assignment operation. Max(a,b,c) or Max (a,b). Max() is a function which cost constant time to give the max value among given values. Max(a, b, c) uses 2 comparison, Max(a,b) uses 1 comparison . So in each cycle, constant time is consumed. So the time complexity of the inner for loop is O(e)

Line 29 and 30 copy 2 arrays with length e+1 to new arrays. The time complexity of these 2 line is O(e)

So the overall time complexity of the for loop is $O(n) \cdot O(e) = O(ne)$

And the overall time complexity of the algorithm is $O(ne) + O(e) = O(ne)$.

Proof of Space Complexity:

In the algorithm there are totally $n \cdot (e+1)$ different states. However, in the implementation, we don't need to store the whole table. We only need (1)two arrays with length e+1 (T and R) to store the previous minute's values. Because we only need the the T(k,$0 \to e$) and R(k,$0 \to e$) to calculate

T(k+1, $0 \rightarrow e$) and R(k+1, $0 \rightarrow e$). (2) And we also need another two arrays with length e+1 (T' and R') to store the current minute's values which we're calculating in the current cycle of for loop. So totally the data structure used in this algorithm is 4 arrays of integers with length e+1. So the overall space used is 4e+4 $O(4e + 4) = O(e)$