Recitation 6 (HW5)

Online: Xinyi Zhao xz2833@nyu.edu

GCASL 475: Yifan Jin yj2063@nyu.edu

New York University

Basic Algorithms (CSCI-UA.0310-005)

Problem 1

Problem 1 (19 points)

Recall that the bottom-up dynamic programming algorithm for finding the nth Fibonacci number required $\Theta(n)$ extra space. Modify the algorithm to develop a linear-time bottom-up DP approach with $O(1)$ extra space. You have to write the pseudo-code of your algorithm.

Hint: Note that in order to compute $F(i)$, we only need to know $F(i-1)$ and $F(i-2)$, and do not require the previous Fibonacci numbers. Use this remark to replace the auxiliary array of size n used in the algorithm covered in the lecture by two auxiliary variables.

Problem 1

② Fib(n)

```
memo[1...n] = {1}          memo[1...n] = [1, 1, ..., 1]
for i=3 to n
    memo[i] = memo[i-1] + memo[i-2]
return memo[n]
```

When computing memo[i], we need **memo[i-2]** and **memo[i-1].**

Problem 1

② Fib(n)

```
memo [1...n] = {1}          memo [1..n] ← [1,1,...,1]

  for i=3 to n

      memo[i] = memo[i-1] + memo[i-2]

  return memo[n]
```

When computing memo[i], we need **memo[i-2]** and **memo[i-1].**

After computing **memo[i]**, in order to compute **memo[i+1]**, what values do we need?

Problem 1

② Fib(n)

```
memo [1...n] = {1}          memo [1...n] ← [1,1,...,1]

 for i=3 to n

     memo [i] = memo[i-1] + memo [i-2]

 return memo [n]
```

When computing memo[i], we need **memo[i-2]** and **memo[i-1].**

After computing **memo[i]**, in order to compute **memo[i+1]**, what values do we need?

Memo[i-1] and **memo[i]. (memo[i-2] no longer needed)**

Problem 1

Define X as current **memo[i-2]**, Y as current **memo[i-1]**, Z as current **memo[i]**

```
Fib(n):
    if(n<=2)
        return 1
    X=1, Y=1, Z
    for i = 3 to n:
        Z = X+Y
        X = Y
        Y = Z
    return Z
```

Compute memo[i]

Move **memo[i-1]** to **memo[(i+1)-2]** for next i+1

Move **memo[i]** to **memo[(i+1)-1]** for next i+1

Problem 2

Problem 2 (9+3+15 points)

Recall *the rod cutting problem* discussed in the lecture: Consider a rod of length n inches and an array of prices $P[1 \dots n]$, where $P[i]$ denotes the price of any i inches long piece of rod. Now suppose we have to pay a cost of \$1 per cut.

Thus, for example, if we cut the rod into k pieces of lengths $n_1, n_2, \dots, n_k$, this means that we made $k-1$ cuts, which gives their final selling price as $P[n_1] + \cdots + P[n_k] - (k-1)$.

Let $\textsc{MaxPrice}(n)$ denote the maximum selling price we can get this way among all possible options we have to make the cuts (note that one possible option is to make no cut and sell the whole rod of length n).

(a) Find the recursion that $\textsc{MaxPrice}(n)$ satisfies. In other words, you should write $\textsc{MaxPrice}(n)$ in terms of $\textsc{MaxPrice}(n-1)$, $\textsc{MaxPrice}(n-2)$, $\dots$, $\textsc{MaxPrice}(0)$. Fully justify your answer.

(b) Identify the base case for your recursion in part (a) and find its corresponding value. Justify your answer.

(c) Write the pseudo-code for the bottom-up DP algorithm to compute $\textsc{MaxPrice}(n)$. Find and justify the time complexity of your algorithm in the form of $\Theta(.)$.

Problem 2

(a) Find the recursion that MaxPrice(n) satisfies.

What is the subproblem?

Problem 2

(a) Find the recursion that MaxPrice(n) satisfies.

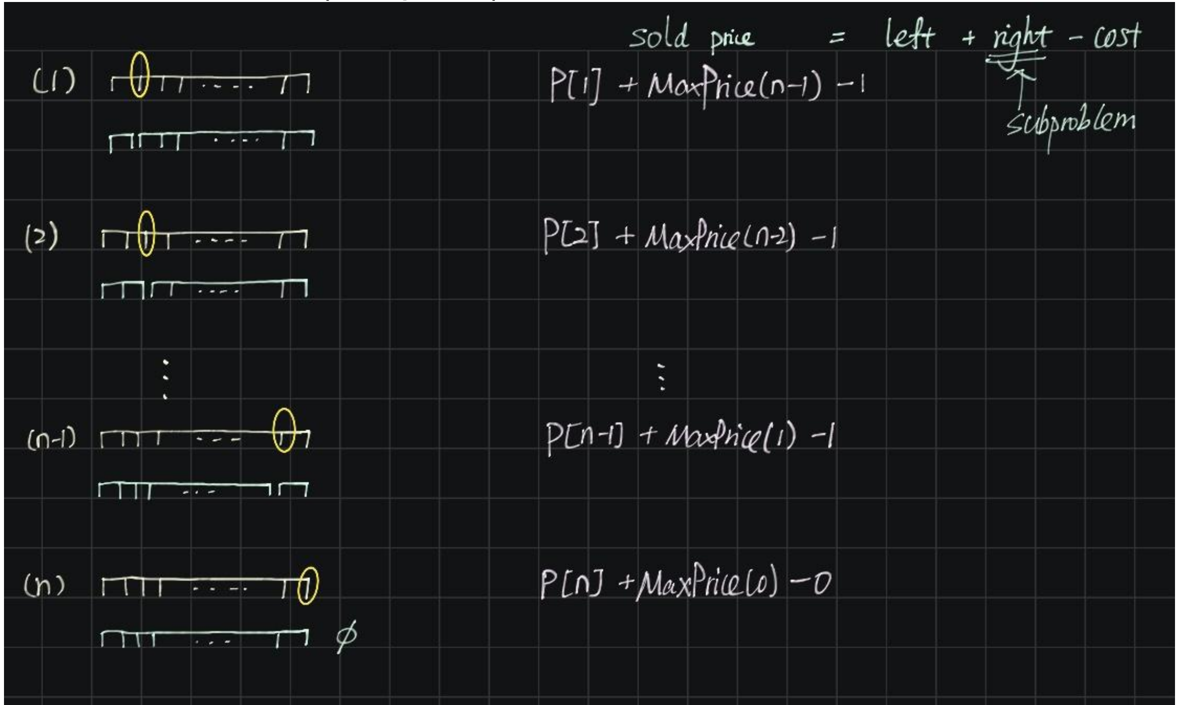
What is the subproblem?

=> MaxPrice(i) := max price we can get by cutting a rod of length i, i=0,1,...,n

Problem 2

(a) Find the recursion that MaxPrice(n) satisfies.

What are the options for the first cut (first piece)?



Problem 2

(a) Find the recursion that MaxPrice(n) satisfies.

Take a maximum of those options to get MaxPrice(n).

⇒ Recursion :

$$\text{MaxPrice}(n) = \max\begin{pmatrix} P[1] + \text{MaxPrice}(n-1) - 1, \\ P[2] + \text{MaxPrice}(n-2) - 1, \\ \vdots \\ P[n] + \text{MaxPrice}(0) \end{pmatrix}, \quad n \geq 1$$

Problem 2

(b) Identify the base case for your recursion in part (a) and find its corresponding value.

What is the smallest subproblem? => when n = 0, MaxPrice(0) = 0;

Then we can calculate:

MaxPrice(1) = P[1]+MaxPrice(0);
MaxPrice(2) = max(P[1]+MaxPrice(1)-1, P[1]+MaxPrice(0));
…

Problem 2

(b) Identify the base case for your recursion in part (a) and find its corresponding value.

Base Case:

MaxPrice(0) = 0

Problem 2

(c) Write the pseudo-code for the bottom-up DP algorithm to compute MaxPrice(n). Find and justify the time complexity of your algorithm in the form of Θ(.).

(Bottom-up DP)

```
MaxPrice(n):                                           // memo[i] = MaxPrice(i)
    memo[0 1 ... n] = [0, ..., 0]      // size n+1    , base case memo[0] = 0

    for i = 1 to n:
        memo[i] = p[i]        // no cut
        for j = 1 to i-1:      // one cut in this step
            memo[i] = max(memo[i], p[j] + memo[i-j]-1)

    return memo[n]
```

Problem 2

(c) Time complexity.

(Bottom-up DP)

MaxPrice(n) : // memo[i] = MaxPrice(i)

memo[0 1 ⋯ n] = [0, ⋯, 0] // size n+1, base case memo[0] = 0

TC = $\sum_{i=1}^{n} \theta(i)$

for i = 1 to n :

memo[i] = p[i] // no cut

TC = $\theta(i)$

for j = 1 to i-1 : // one cut in this step

memo[i] = max (memo[i], p[j] + memo[i-j]-1)

return memo[n]

TC: $T(n) = \sum_{i=1}^{n} \theta(i) = \theta(n^2)$ $(0 + 1 + 2 + \cdots + n-1)$

SC: $SC = \theta(n)$ (space of memo[0⋯n])

Problem 3

Problem 3 (9+3+15 points)

Consider the array $A[1 \dots n]$ consisting of n non-negative integers. There is a frog on the last index of the array, i.e. the nth index of the array. In each step, if the frog is positioned on the i^{th} index, then it can make a jump of size at most $A[i]$ towards the beginning of the array. In other words, it can hop to any of the indices $i, \dots, i - A[i]$.
The goal is to develop a DP-based algorithm to determine whether the frog can reach the 1st index of the array.
For $i = 1, 2, \dots, n$, define the subproblem $\textsc{CanReach}(i)$ to denote true or false depending on whether the frog can reach the 1st index of the array when it is currently standing on the ith index (so $\textsc{CanReach}(i) =$ true, if it can reach the 1st index, and $\textsc{CanReach}(i) =$ false, if it cannot reach the 1st index).

(a) Find the recursion that $\textsc{CanReach}(n)$ satisfies. In other words, you should write $\textsc{CanReach}(n)$ in terms of some of $\{\textsc{CanReach}(n-1),\ \textsc{CanReach}(n-2),\ \dots,\ \textsc{CanReach}(1)\}$. Fully Justify your answer.

Hint: What options does the frog have for the first jump? It can jump to any of the following indices: $n, \dots, n - A[n]$.

(b) Identify the base case for your recursion in part (a) and find its corresponding value. Justify your answer.

(c) Write the pseudo-code for the bottom-up DP algorithm to compute $\textsc{CanReach}(n)$. Justify that the run-time of your algorithm is $O(n^2)$ in the worst-case.

Problem 3

(a) Find the recursion that $\textsc{CanReach}(n)$ satisfies. In other words, you should write $\textsc{CanReach}(n)$ in terms of some of $\{\textsc{CanReach}(n-1), \textsc{CanReach}(n-2), \ldots, \textsc{CanReach}(1)\}$. Fully Justify your answer.

Hint: What options does the frog have for the first jump? It can jump to any of the following indices: $n, \ldots, n - A[n]$.

Problem 3

(a) Find the recursion that $\textsc{CanReach}(n)$ satisfies. In other words, you should write $\textsc{CanReach}(n)$ in terms of some of $\{\textsc{CanReach}(n-1), \textsc{CanReach}(n-2), \ldots, \textsc{CanReach}(1)\}$. Fully Justify your answer.

Hint: What options does the frog have for the first jump? It can jump to any of the following indices: $n, \ldots, n - A[n]$.

Assume we already know the value of **CANREACH(n-1, n-2, … , n-A[n])**, indicating whether we can reach the 1st index from n-1, n-2, … n-A[n].

Problem 3

(a) Find the recursion that CANREACH(n) satisfies. In other words, you should write CANREACH(n) in terms of some of {CANREACH($n-1$), CANREACH($n-2$), ..., CANREACH(1)}. Fully Justify your answer.

Hint: What options does the frog have for the first jump? It can jump to any of the following indices: $n, \dots, n - A[n]$.

Assume we already know the value of **CANREACH(n-1, n-2, ... , n-A[n])**, indicating whether we can reach the 1st index from n-1, n-2, … n-A[n].

Thus, if any of **CANREACH(n-1, n-2, ... , n-A[n])** is true, we can jump from **n** to that corresponding index, then reach the 1st index, which means **CANREACH(n) = true.**

Problem 3

(a) Find the recursion that CanReach(n) satisfies. In other words, you should write CanReach(n) in terms of some of {CanReach($n-1$), CanReach($n-2$), ..., CanReach(1)}. Fully Justify your answer.

Hint: What options does the frog have for the first jump? It can jump to any of the following indices: $n, \ldots, n - A[n]$.

Assume we already know the value of **CANREACH(n-1, n-2, … , n-A[n])**, indicating whether we can reach the 1st index from n-1, n-2, … n-A[n].

Thus, if any of **CANREACH(n-1, n-2, … , n-A[n])** is true, we can jump from **n** to that corresponding index, then reach the 1st index, which means **CANREACH(n) = true.**

Otherwise, we can never reach the 1st index.

Problem 3

(a) Find the recursion that CANREACH(n) satisfies. In other words, you should write CANREACH(n) in terms of some of {CANREACH($n-1$), CANREACH($n-2$), ..., CANREACH(1)}. Fully Justify your answer.

Hint: What options does the frog have for the first jump? It can jump to any of the following indices: $n, \ldots, n - A[n]$.

Assume we already know the value of **CANREACH(n-1, n-2, ... , n-A[n])**, indicating whether we can reach the 1st index from n-1, n-2, … n-A[n].

Thus, if any of **CANREACH(n-1, n-2, ... , n-A[n])** is true, we can jump from **n** to that corresponding index, then reach the 1st index, which means **CANREACH(n) = true.**

Otherwise, we can never reach the 1st index.

Thus, **CANREACH(n) = CANREACH(n-1) or ... or CANREACH(n-A[n])**

Problem 3

(b) Identify the base case for your recursion in part (a) and find its corresponding value. Justify your answer.

Problem 3

(b) Identify the base case for your recursion in part (a) and find its corresponding value. Justify your answer.

According to the definition of **CANREACH,** which case can be determined for sure?

Problem 3

(b) Identify the base case for your recursion in part (a) and find its corresponding value. Justify your answer.

According to the definition of **CANREACH,** which case can be determined for sure?

When n=1, the 1st index (itself) can necessarily be reached, **CANREACH(1) = true**

Problem 3

(c) Write the pseudo-code for the bottom-up DP algorithm to compute CanReach(n). Justify that the run-time of your algorithm is $O(n^2)$ in the worst-case.

Problem 3

(c) Write the pseudo-code for the bottom-up DP algorithm to compute CANREACH(n). Justify that the run-time of your algorithm is $O(n^2)$ in the worst-case.

CANREACH(n) = CANREACH(n-1) or … or CANREACH(n-A[n])

CANREACH(1) = true

Problem 3

(c) Write the pseudo-code for the bottom-up DP algorithm to compute CANREACH(n). Justify that the run-time of your algorithm is $O(n^2)$ in the worst-case.

CANREACH(n) = CANREACH(n-1) or ... or CANREACH(n-A[n])

CANREACH(1) = true

```
define CANREACH[1...n] = [FALSE, FALSE.... FALSE]

CAMREACH[1] = TRUE

for i = 2 to n:                    i - A[i]

    for j = max(1, n-A[n])  to  i-1

        if CANREACH[j] == TRUE

            CANREACH[i] = TRUE

return CANREACH[n]
```

Problem 3

(c) Write the pseudo-code for the bottom-up DP algorithm to compute CANREACH(n). Justify that the run-time of your algorithm is $O(n^2)$ in the worst-case.

CANREACH(n) = CANREACH(n-1) or ... or CANREACH(n-A[n])

CANREACH(1) = true

O(i) in the worst case
For each j loop

```
define CANREACH[1...n] = [FALSE, FALSE.... FALSE]

CAMREACH[1] = TRUE

for i = 2 to n:                  i - A[i]

    for j = max(1, n-A[n])  to  i-1

        if CANREACH[j] == TRUE

            CANREACH[i] = TRUE

return CANREACH[n]
```

Problem 3

(c) Write the pseudo-code for the bottom-up DP algorithm to compute CANREACH(n). Justify that the run-time of your algorithm is $O(n^2)$ in the worst-case.

CANREACH(n) = CANREACH(n-1) or … or CANREACH(n-A[n])

CANREACH(1) = true

```
define CANREACH[1...n] = [FALSE, FALSE.... FALSE]

CAMREACH[1] = TRUE

for i = 2 to n:                    i - A[i]

    for j = max(1, n-A[n])  to  i-1

        if CANREACH[j] == TRUE

            CANREACH[i] = TRUE

return CANREACH[n]
```

O(i) in the worst case
For each j loop

In the worst case, the total

Running time is O(n^2)

Problem 4

Problem 4 (3+9+15 points)

Suppose we want to find the number of ways to make change for n cents with the use of dimes (10 cents), nickels (5 cents), and pennies (1 cent). Note that the ordering of coins in the change matters! (see the examples below)

For $i = 1, \ldots, n$, define the subproblem $\textsc{MakeChange}(i)$ to denote the number of ways to make change for i cents with the use of dimes (10 cents), nickels (5 cents), and pennies (1 cent).

Below, you can find the number of ways to make change for i cents for $i = 1, \ldots, 7$:

- $i = 1, \ldots, 4$: There is only one way to do that. We have to change only using pennies.
- $i = 5$: There are two ways: $1+1+1+1+1$ (use 5 pennies), or 5 (use one nickel).
- $i = 6$: There are three ways: $1+1+1+1+1+1$ (use 6 pennies), or $1+5$ (one penny and one nickel), or $5+1$ (one nickel and one penny).
- $i = 7$: There are four ways: $1+1+1+1+1+1+1$ (use 7 pennies), or $1+1+5$ (two pennies and one nickel), or $1+5+1$ (one penny, one nickel, and one penny), or $5+1+1$ (one nickel and two pennies).

(a) Find the values of $\textsc{MakeChange}(8)$ and $\textsc{MakeChange}(9)$.

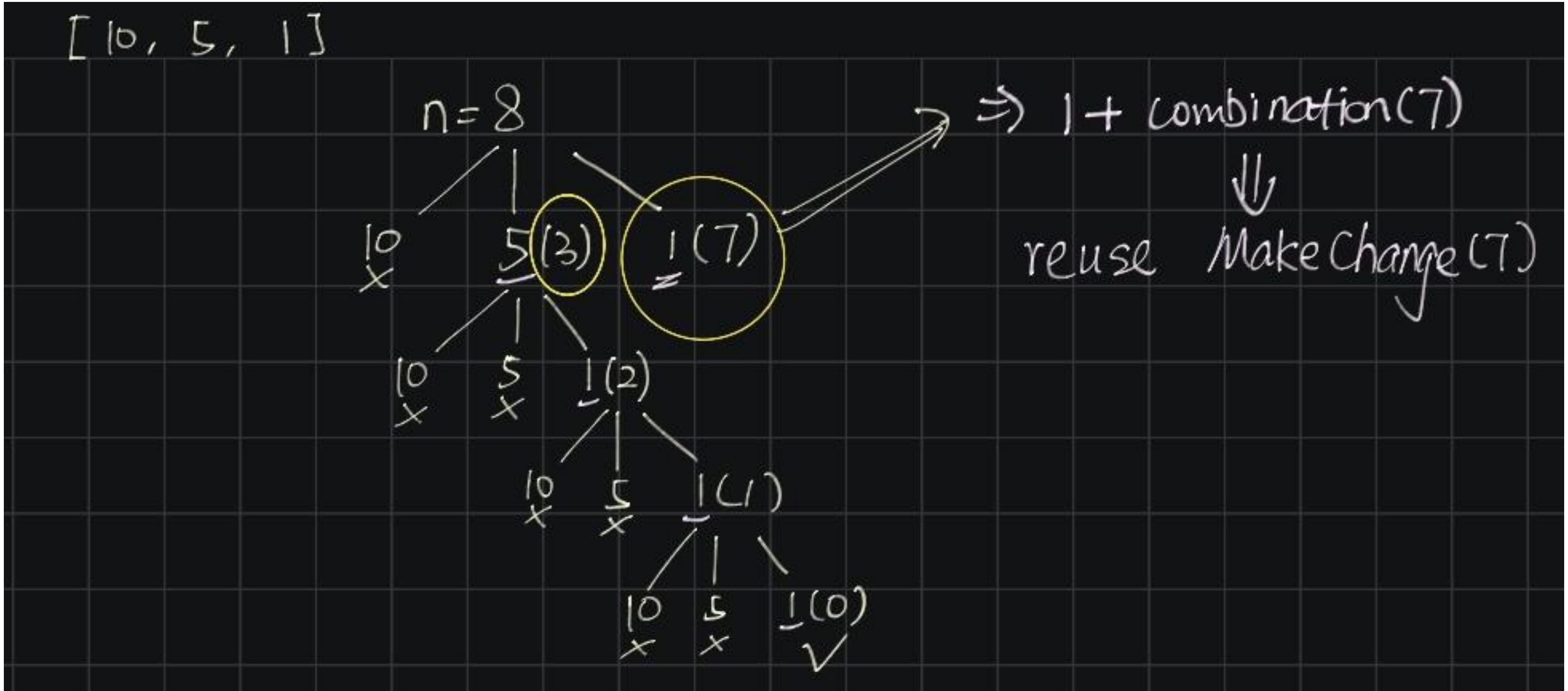
(b) Find the recursion that $\textsc{MakeChange}(n)$ satisfies for $n \geq 10$. In other words, you should write $\textsc{MakeChange}(n)$ in terms of some of $\{\textsc{MakeChange}(n-1), \textsc{MakeChange}(n-2), \ldots, \textsc{MakeChange}(1)\}$. Fully justify your answer.

Hint: What options do we have for the first coin of the change?

(c) Write the pseudo-code for the bottom-up DP algorithm to compute $\textsc{MakeChange}(n)$. Find and justify the time complexity of your algorithm in the form of $\Theta(.)$.

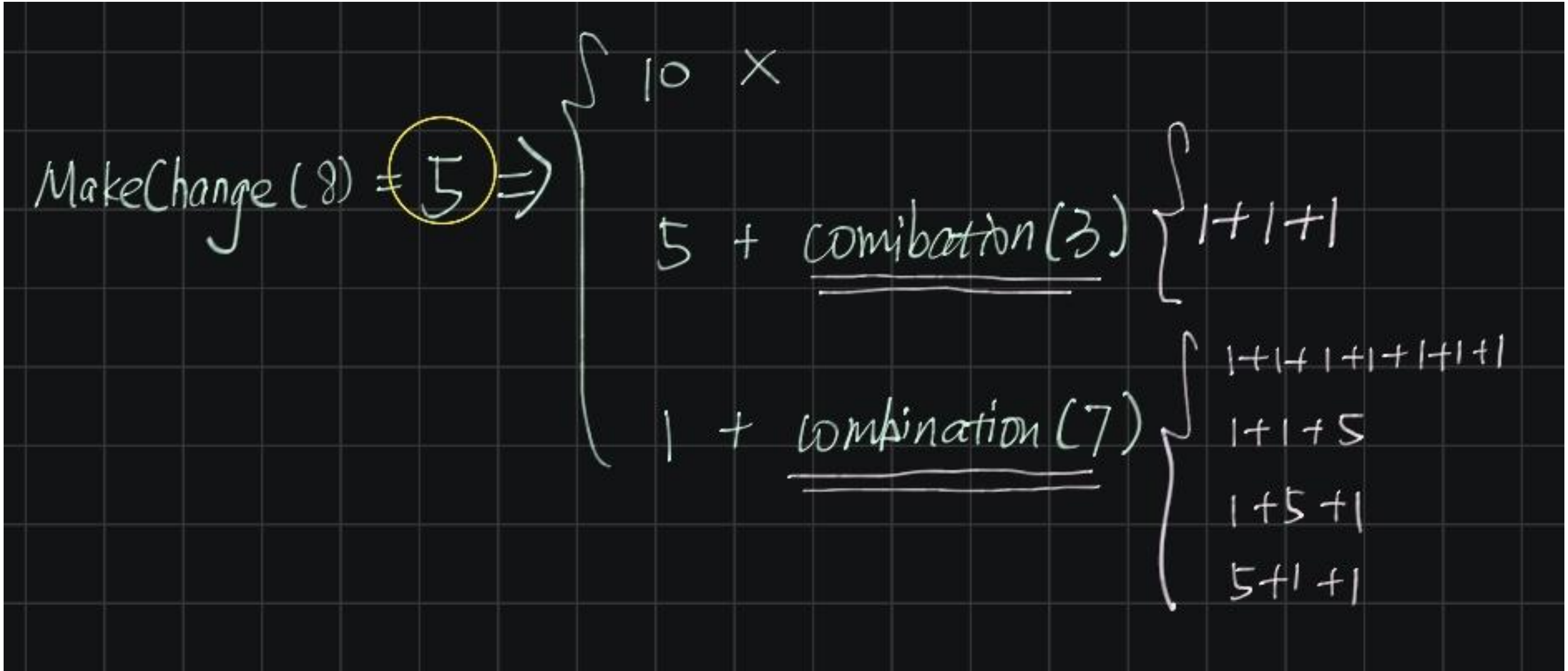
Problem 4

(a) Find the values of MakeChange(8) and MakeChange(9).



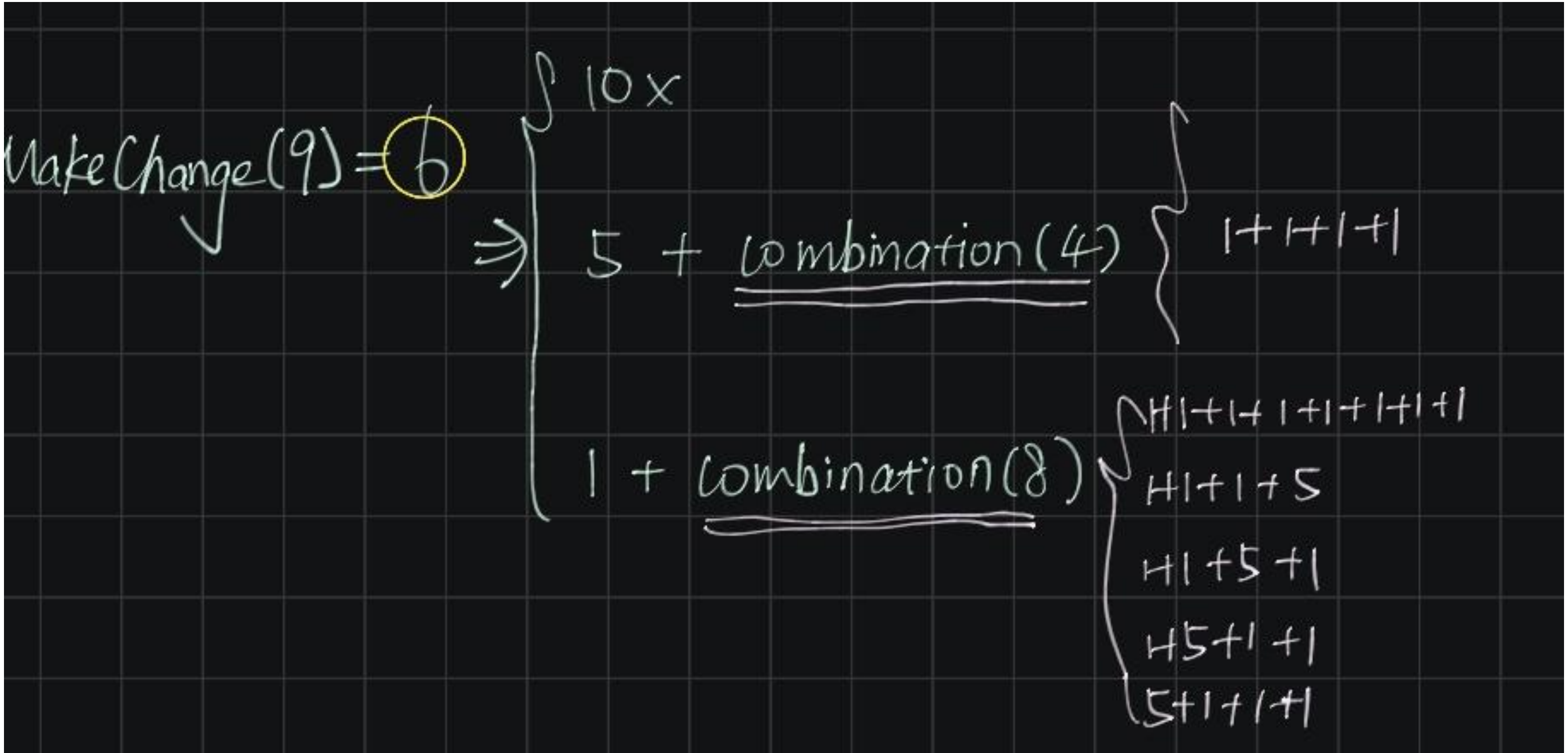
Problem 4

(a) Find the values of MakeChange(8) and MakeChange(9).



Problem 4

(a) Find the values of MakeChange(8) and MakeChange(9).



Problem 4

(b) Find the recursion that MakeChange(n) satisfies for $n \geq 10$. In other words, you should write MakeChange(n) in terms of some of {MakeChange(n−1), MakeChange(n−2), . . . , MakeChange(1)}. Fully justify your answer.

Hint: What options do we have for the first coin of the change?

Problem 4

(b) Find the recursion that MakeChange(n) satisfies for $n \geq 10$. In other words, you should write MakeChange(n) in terms of some of {MakeChange(n−1), MakeChange(n−2), . . . , MakeChange(1)}. Fully justify your answer.

Hint: What options do we have for the first coin of the change?

=> How many options for the first coin?

Problem 4

(b) Find the recursion that MakeChange(n) satisfies for $n \geq 10$. In other words, you should write MakeChange(n) in terms of some of {MakeChange(n−1), MakeChange(n−2), . . . , MakeChange(1)}. Fully justify your answer.

Hint: What options do we have for the first coin of the change?

How many options for the first coin? => 3

Try [10, 5, 1] for the first coin, then what is the subproblem??

Problem 4

(b) MakeChange(n), n ≥ 10. How many options for the first coin? => Try [10, 5, 1] for the first coin.

First coin: subproblem

$$\underset{n \geq 10}{\text{MaxChange}(n)} = \text{sum} \begin{cases} \text{choose } \underline{\underline{10}} & \& \quad \text{MaxChange}(n-10) \\ \text{choose } \underline{\underline{5}} & \& \quad \text{MaxChange}(n-5) \\ \text{choose } \underline{\underline{1}} & \& \quad \text{MaxChange}(n-1) \end{cases}$$

$$\Rightarrow \boxed{\underset{n \geq 10}{\text{MaxChange}(n)} = \text{MaxChange}(n-10) + \text{MaxChange}(n-5) + \text{MaxChange}(n-1)}$$

Base Cases: $\text{MaxChange}(0) = 1$

$\text{MaxChange}(1) = 1$

$\vdots$

$\text{MaxChange}(9) = 6$

Problem 4

(c) Write the pseudo-code for the bottom-up DP algorithm to compute MakeChange(n). Find and justify the time complexity of your algorithm in the form of Θ(.).

=> Initialize the base cases

=> Use for loop to calculate the MakeChange(n), iterating i from 10 to n.

Problem

(c)

(Bottom-up DP)

MakeChange(n):

$$\begin{array}{lccccccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \cdots \\ memo[0\ 1 \cdots n] = [& 1, & 1, & 1, & 1, & 1, & 2, & 3, & 4, & 5, & 6, & 0 & \cdots & 0] \end{array}$$

$TC = \sum_{i=10}^{n} \theta(i)$

for $i = 10$ to n:

$\theta(i) = \theta(1)$ memo[i] = memo[i-10] + memo[i-5] + memo[i-1]

return memo[n]

TC: $T(n) = \theta(n)$

SC: $SC = \theta(n)$ (space needed for memo[0 ⋯ n])

Problem 4

(c) Advanced thinking:

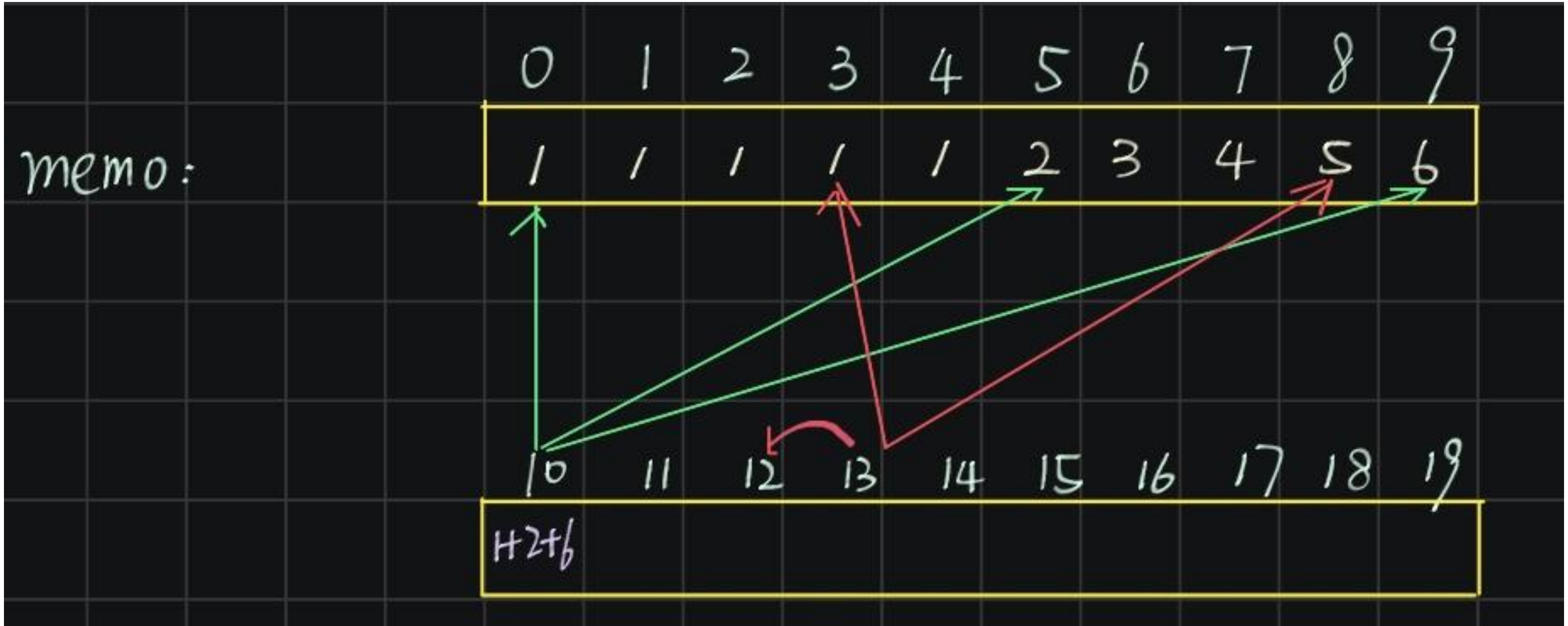
Do we really need the memo[0…n]?

Can we improve the space complexity?

Problem 4

(c) Advanced thinking:

How we improve the space complexity?



Bonus Problem 1

Bonus Problem 1

Develop an algorithm to compute the nth Fibonacci number in $O(\log n)$ time.

Bonus Problem 1

Fibonacci

$F_0 = 0 \quad , F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n-1} + F_{n-2} \\ F_{n-1} + 0 \times F_{n-2} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

$$= \quad \vdots$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Bonus Problem 1

Can we solve $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ in $O(\log n)$ time? $\quad n \geq 2$

let $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$,

$\boxed{n-1}$

even: $M^{n-1} = M^{\frac{n-1}{2}} \cdot M^{\frac{n-1}{2}}$

odd: $M^{n-1} = M \cdot M^{\lfloor \frac{n-1}{2} \rfloor} \cdot M^{\lfloor \frac{n-1}{2} \rfloor}$

$T(n) = T(\frac{n}{2}) + C$

$T(1) = C$, $\quad C > 0$

$\Rightarrow T(n) = O(\log n)$

Bonus Problem 2

Bonus Problem 2

(a) Can you develop a bottom-up DP algorithm for Problem 3 with $O(n)$ space that runs in $O(n \log n)$ time in the worst case?
If so, write the pseudo-code of your algorithm.

(b) Can you improve the run-time of your algorithm in part (a) to $O(n)$?
If so, write the pseudo-code of your algorithm.

Bonus Problem 2

(a) Can you develop a bottom-up DP algorithm for Problem 3 with $O(n)$ space that runs in $O(n \log n)$ time in the worst case?

```
CanReach(n) :
    minHeap = { }
    minHeap.add(n-A[n])

    for i in (n-1) to 2:
        if minHeap.top() > i:
            return false
        minHeap.add(i-A[i])        // O(logn)

    return minHeap.top() ≤ 1
```

Bonus Problem 2

(b) Can you improve the run-time of your algorithm in part (a) to $O(n)$?
If so, write the pseudo-code of your algorithm.

```
leftmost = n

for i = n to 1:
  If i >= leftmost:
    leftmost = min(leftmost,  i-A[i])

if (leftmost <= 1)

    return TRUE

return FALSE
```

Q & A

Thank you