Recitation 7 (HW6)

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

Let $T(n)$ denote the running time of Approach 1 to find the nth Fibonacci number discussed in the lecture. Use strong induction to show that $T(n) = \Omega(2^{n/2})$.
Hint: Use the inequality $T(n) \geq T(n-2)$ which holds for all $n \geq 3$.

Problem 1

Problem 1

Let $T(n)$ denote the running time of Approach 1 to find the nth Fibonacci number discussed in the lecture. Use strong induction to show that $T(n) = \Omega(2^{n/2})$.
Hint: Use the inequality $T(n) \geq T(n-2)$ which holds for all $n \geq 3$.

$$T(n) = T(n-1) + T(n-2) + O(1) \qquad n \geq 3$$

Problem 1

$$T(n) = T(n-1) + T(n-2) + O(1) \qquad n \geq 3$$

In order to prove T(n) = Omega(2^(n/2)), we need to prove that

T(n) >= c1 * 2^(n/2) for some positive c1

Problem 1

$$T(n) = T(n-1) + T(n-2) + O(1) \qquad n \geq 3$$

In order to prove T(n) = Omega(2^(n/2)), we need to prove that

T(n) >= c1 * 2^(n/2) for some positive c1

Base Case:

n=1 -> T(1) = 1 >= c1 * 2^(1/2)

n=2 -> T(2) = 1 >= c1 * 2^1

Thus, when **c1<=(1/2)** , it holds for base case.

Problem 1

$$T(n) = T(n-1) + T(n-2) + O(1) \qquad n \geq 3$$

In order to prove T(n) = Omega(2^(n/2)), we need to prove that

T(n) >= c1 * 2^(n/2) for some positive c1

Induction Step:

Assume it holds for **n=k-1, k-2**   ->   **T(k-1) >= c1 * 2^((k-1)/2), T(k-2) >= c1 * 2^((k-2)/2)**

Problem 1

$$T(n) = T(n-1) + T(n-2) + O(1) \qquad n \geq 3$$

In order to prove T(n) = Omega(2^(n/2)), we need to prove that

T(n) >= c1 * 2^(n/2) for some positive c1

Induction Step:

Assume it holds for **n=k-1, k-2** -> **T(k-1) >= c1 * 2^((k-1)/2), T(k-2) >= c1 * 2^((k-2)/2)**

So, **T(k) = T(k-1) + T(k-2) + O(1) >= c1 * 2^((k-1)/2) + c1 * 2^((k-2)/2) + c2**

>= c1 * 2^k * (1/sqrt(2) + 1/2) + c2. Larger than 1

Thus, it always holds that **T(k) >= c1 * 2^k.** Thus, it also holds for **n = k.**

Problem 2

Problem 2

Given two strings $S[1 \dots n]$ and $T[1 \dots m]$, let $\mathrm{LCS}(n, m)$ denote the length of the longest common substring of $S[1 \dots n]$ and $T[1 \dots m]$. Note that unlike a subsequence, a substring is required to occupy consecutive positions within the original strings.

(a) Find the recursion that $\mathrm{LCS}(n, m)$ satisfies. Fully justify your answer.

(b) Identify the base cases for your recursion in part (a) and find their corresponding values. Justify your answer.

(c) Write the pseudo-code for the bottom-up DP algorithm to compute $\mathrm{LCS}(n, m)$.

(d) Find and justify the time complexity of your algorithm in the form of $\Theta(.)$.

Problem 2

(a) $LCS(n, m) = \max\limits_{\substack{i \in [1,n] \\ j \in [1,m]}} (lcs(i, j))$

$lcs(i, j)$: the length of longest common substring of $S[1 \ldots i]$ and $T[1 \ldots j]$, ending with $S[i]$ and $T[j]$.

$$lcs(i,j) = \begin{cases} 0 & \boxed{i=0 \text{ or } j=0} \in \text{Base case} \\ & \text{or} \\ & i,j>0,\ S[i] \neq T[j] \in \text{case (i)} \\ lcs(i-1, j-1) + 1 & i,j>0,\ S[i] == T[j] \in \text{case (ii)} \end{cases}$$

Problem 2

(b)

Base case: $i=0$ or $j=0$

$\Rightarrow lcs(i,j) = 0$

Problem 2

(C) pseudo-code (Bottom-up DP)

```
LCS(S[1...n], T[1...m]):
    memo[0...n][0...m] = [0]
    longest = 0

    for i = 1 to n:
        for j=1 to m:
            if S[i] == T[j]:
                memo[i][j] = 1+ memo[i-1][j-1]

            longest = max(longest, memo[i][j])

    return longest
```

notes

Problem 2

(d)

$TC = \Theta(nm)$

```
LCS(S[1...n], T[1...m]):
    memo[0...n][0...m] = [0]     ← includes bace cases
    longest = 0                  ← result

    for i = 1 to n:
        for j=1 to m:
            if S[i] == T[j]:                      ← case(i)
                memo[i][j] = 1+ memo[i-1][j-1]
                                                  ← else 0 , case(ii)
            longest = max(longest, memo[i][j])

    return longest
```

nm iterations

m iterations

O(1) time

notes

Problem 3

Problem 3

Alice and Bob want to play the following game by alternating turns: They have access to a row of n coins of values $v_1, \ldots, v_n$, where n is even. In each turn, a player selects either the first or the last coin from the row, removes it from the row, and receives the value of the coin. Alice starts the game.
Devise a dynamic programming algorithm to determine the maximum possible amount of money Alice can definitely win (assume that Bob will play in such a way to maximize the amount he gets). State a $\Theta(.)$ expression for the running time of your algorithm.

Problem 3

Problem 3

Alice and Bob want to play the following game by alternating turns: They have access to a row of n coins of values $v_1, \ldots, v_n$, where n is even. In each turn, a player selects either the first or the last coin from the row, removes it from the row, and receives the value of the coin. Alice starts the game.
Devise a dynamic programming algorithm to determine the maximum possible amount of money Alice can definitely win (assume that Bob will play in such a way to maximize the amount he gets). State a $\Theta(.)$ expression for the running time of your algorithm.

Game problem:

- Some players take turns to earn profits
- All players take the global optimal strategy to maximize their profits

Problem 3

Problem 3

Alice and Bob want to play the following game by alternating turns: They have access to a row of n coins of values $v_1, \ldots, v_n$, where n is even. In each turn, a player selects either the first or the last coin from the row, removes it from the row, and receives the value of the coin. Alice starts the game.
Devise a dynamic programming algorithm to determine the maximum possible amount of money Alice can definitely win (assume that Bob will play in such a way to maximize the amount he gets). State a $\Theta(.)$ expression for the running time of your algorithm.

Game problem:

Usually the total sum of profits is certain,

so each player tries to minimize others' profits in order to maximize theirs.

Problem 3

Problem 3

Alice and Bob want to play the following game by alternating turns: They have access to a row of n coins of values $v_1, \ldots, v_n$, where n is even. In each turn, a player selects either the first or the last coin from the row, removes it from the row, and receives the value of the coin. Alice starts the game.
Devise a dynamic programming algorithm to determine the maximum possible amount of money Alice can definitely win (assume that Bob will play in such a way to maximize the amount he gets). State a $\Theta(.)$ expression for the running time of your algorithm.

Simple greedy strategy fails. E.g 4 7 5 3

The optimal strategy takes 3 first and then take 7 (total = 10), while the greedy strategy takes 4 first and then take 5 (total = 9).

Problem 3

Problem 3

Alice and Bob want to play the following game by alternating turns: They have access to a row of n coins of values $v_1, \ldots, v_n$, where n is even. In each turn, a player selects either the first or the last coin from the row, removes it from the row, and receives the value of the coin. Alice starts the game.
Devise a dynamic programming algorithm to determine the maximum possible amount of money Alice can definitely win (assume that Bob will play in such a way to maximize the amount he gets). State a $\Theta(.)$ expression for the running time of your algorithm.

The most difficult and important question:

What is the DP problem? (decides the recursive formula).

Problem 3

Problem 3

Alice and Bob want to play the following game by alternating turns: They have access to a row of n coins of values $v_1, \ldots, v_n$, where n is even. In each turn, a player selects either the first or the last coin from the row, removes it from the row, and receives the value of the coin. Alice starts the game.
Devise a dynamic programming algorithm to determine the maximum possible amount of money Alice can definitely win (assume that Bob will play in such a way to maximize the amount he gets). State a $\Theta(.)$ expression for the running time of your algorithm.

The most difficult and important question:

Interval/Range DP

What is the DP problem? (decides the recursive formula).

Since players only pick the first or last, at any time the remaining coins are always **consecutive from v[i] to v[j]** where 1<=i<=j<=n.

Define DP[i,j] as the maximum money the current player can win **from v[i] to v[j]**.

Problem 3

DP[i,j] as the maximum money the current player can win **from v[i] to v[j]**.

What options/choice do we currently have?(under the condition **from v[i] to v[j]**)

What subproblem does each choice correspond to?

How to make a decision among all choices using the result of subproblems?

Problem 3

DP[i,j] as the maximum money the current player can win **from v[i] to v[j]**.

What options/choice do we currently have?(under the condition **from v[i] to v[j]**)

Since we can only select first or last, the choice is either v[i] or v[j].

What subproblem does each choice correspond to?

DP[i+1, j] for choosing v[i] , **DP[i,j-1]** for choosing v[j]

How to make a decision among all choices using the result of subproblems?

Assume we already know **DP[i+1, j]** and **DP[i,j-1]** (See next slide)

Problem 3

How to make a decision among all choices using the result of subproblems?

Both **DP[i+1,j]** and **DP[i,j-1]** represent the maximum money the **opponent** can make.

Recall that **each player tries to minimize others' profits in order to maximize theirs.**

Problem 3

How to make a decision among all choices using the result of subproblems?

Both **DP[i+1,j]** and **DP[i,j-1]** represent the maximum money the **opponent** can make.

Recall that **each player tries to minimize others' profits in order to maximize theirs.**

Strategy: **If DP[i+1,j] < DP[i,j-1] , select v[i]**

else , select v[j]

Problem 3

How to compute DP[i,j]?

To simplify, assume we know how to compute the total sum of **v[i], v[i+1],...,v[j-1],v[j]**.

Define as **SUM[i,j]**.

Both **DP[i+1,j]** and **DP[i,j-1]** represent the maximum money the **opponent** can make.

Problem 3

How to compute DP[i,j]?

To simplify, assume we know how to compute the total sum of **v[i], v[i+1],...,v[j-1],v[j]**.

Define as **SUM[i,j]**.

Both **DP[i+1,j]** and **DP[i,j-1]** represent the maximum money the **opponent** can make.

Thus, if we **select v[i]**, the total money we can get is **SUM[i,j] - DP[i+1,j],**

Similarly, if we **select v[j]**, the total money we can get is **SUM[i,j] - DP[i,j-1],**

Problem 3

What is the base case?

Remember usually in base case, we don't (and don't need to) make choice.

Problem 3

What is the base case?

Remember usually in base case, we don't (and don't need to) make choice.

DP[i,i] = v[i] for **1<=i<=n**

Problem 3

Another point: How to write codes?

```
Initialization / Base Case

for i from 1 to n:
    for j from i to n:
        .....
```

Problem 3

Another point: How to write codes?

```
Initialization / Base Case

for i from 1 to n:
    for j from i to n:
        .....
```

WRONG!!!

In order to compute DP[i,j], we need to value **DP[i+1,j]** and **DP[i,j-1].**

But in the above iteration order, **DP[i+1,j]** hasn't been computed.

Problem 3

Another point: How to write codes?

Recall that the length of **range(i,j)** is 1 larger than that of **range(i+1,j)** and **range(i,j-1)**.

Thus, compute the result of range in increasing order.

```
Initialization / Base Case

for len from 2 to n:
    for i from 1 to n:
        define j = i + len - 1
        if j>n:
            break
        .....
```

Problem 3

```
define DP[n,n]

for i from 1 to n:
    DP[i,i] = v[i]

for len from 2 to n:
    for i from 1 to n:
        define j = i + len - 1
        if j>n:
            break
        DP[i,j] = SUM[i,j] -
                    min(DP[i+1,j], DP[i,j-1])

return DP[1,n]
```

Running Time: theta(n^2)

Problem 4

Problem 4

A palindrome is a non-empty string that spells the same forward and backward. As an example, "civic" is a palindrome. Given the string $S[1 \dots n]$, we want to find the length of the longest palindromic subsequence of S. For example, for the string "character", the answer is 5 since the longest palindromic subsequence is "carac".

(a) Let $P(i, j)$ denote the length of the longest palindromic subsequence of the string $S[i \dots j]$. Find the recursion that $P(i, j)$ satisfies. Justify your answer.

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

(c) Write the pseudo-code for the bottom-up DP algorithm to compute $P(1, n)$.

(d) Find and justify the time complexity of your algorithm in the form of $\Theta(.)$.

Problem 4

(a)

$$P(i,j) = \begin{cases} 1 & i == j \quad \text{base case} \\ P(i+1, j-1) + 2 & i < j, \quad S[i] == S[j] \quad \text{case (i)} \\ \max(P(i, j-1), P(i+1, j)) & i < j, \quad S[i] \neq S[j] \quad \text{case (ii)} \end{cases}$$

Problem 4

(b) Base case(s):

$i == j$,

$P(i, j) = 1$

Problem 4

(C) pseudo-code (Bottom-up DP)

```
LPalinS (S[1...n]):
    memo[1...n][1...n] = [0]
    for i = n to 1 :
        memo[i][i] = 1            ⇐ base case
        for j = i+1 to n:
            if S[i] == S[j] :                 ⇐ case(i)
                memo[i][j] = memo[i+1][j-1] + 2
            else:                             ⇐ case (ii)
                memo[i][j] = max(memo[i+1][j], memo[i][j-1])
    return memo[1][n]
```

Problem 4

(d)

$TC = \Theta(n^2)$

```
LPalinS (S[1...n]):

    memo[1...n][1...n] = [0]

    for i = n to 1 :

        memo[i][i] = 1                                   ⇐ base case

        for j = i+1 to n:

            if   S[i] == S[j] :                            ⇐ case(i)

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

            else:                                            ⇐ case (ii)

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

    return memo[1][n]
```

$\Theta(n^2)$

$(n-i)$ iterations

$\Theta(1)$

Problem 5

Problem 5

Recall *the longest common subsequence problem* discussed in the lecture. Directly solve Problem 4 by using the longest common subsequence problem.

Problem 5

Problem 5

Recall *the longest common subsequence problem* discussed in the lecture. Directly solve Problem 4 by using the longest common subsequence problem.

S = character carac

Recall the palindrome spells same forward and backward.

LCS always handles the forward common subsequence.

(a)

$$P(i,j) = \begin{cases} 1 & i==j \quad \text{base case} \\ P(i+1, j-1) + 2 & i<j, \quad S[i]==S[j] \quad \text{case(i)} \\ \max(P(i, j-1), P(i+1, j)) & i<j, \quad S[i] \neq S[j] \quad \text{case(ii)} \end{cases}$$

$$LCS(i,j) = \begin{cases} 0 & i=0 \text{ or } j=0 \quad \text{base cases} \\ LCS(i-1, j-1) + 1 & i,j>0 \;\&\; X[i]=Y[j] \quad \text{case (i)} \\ \max\{LCS(i-1,j), LCS(i,j-1)\} & i,j>0 \;\&\; X[i] \neq Y[j] \quad \text{case (ii)} \end{cases}$$

Problem 5

Problem 5

Recall *the longest common subsequence problem* discussed in the lecture. Directly solve Problem 4 by using the longest common subsequence problem.

Thus, define T is the reverse of S.

e.g S = character , T = retcarahc

So the LCS of S and T is indeed the longest palindrome of S.

Problem 6

Problem 6

Consider the two-dimensional array $A[1 \ldots m][1 \ldots n]$, where each entry $A[i][j]$ is filled with a positive integer-valued reward. We start from the bottom leftmost corner, i.e., $A[0][0]$, and in each step, we are allowed to move to the right adjacent cell or to the top adjacent cell, until we reach to the top rightmost corner, i.e., $A[m][n]$. We collect the reward of each cell we step on. Let $\text{MAX REWARD}(m, n)$ denote the maximum amount of reward we can collect by starting from $A[0][0]$ and reaching $A[m][n]$.

(a) Find the recursion that $\text{MAX REWARD}(m, n)$ satisfies. Fully justify your answer.

(b) Identify the base cases for your recursion in part (a) and find their corresponding values. Justify your answer.

(c) Write the pseudo-code for the bottom-up DP algorithm to compute $\text{MAX REWARD}(m, n)$.

(d) Find and justify the time complexity of your algorithm in the form of $\Theta(.)$.

Problem 6

MAX_REWARD(m,n) is the maximum reward from (0,0) to (m,n) (the current destination)

What is the possible previous position before we reach (m,n)? [The choice we have]

What subproblem(s) does each choice correspond to?

How to make a decision among these choices (considering the result of subproblems)?

Problem 6

MAX_REWARD(m,n) is the maximum reward from (0,0) to (m,n) (the current destination)

What is the possible previous position before we reach (m,n)? [The choice we have]

Since each step only moves to top or right, the previous position is either (m-1,n) or (m,n-1).

What subproblem(s) does each choice correspond to?

MAX_REWARD(m-1,n) if coming from (m-1,n)

MAX_REWARD(m,n-1) if coming from (m,n-1)

Problem 6

MAX_REWARD(m,n) is the maximum reward from (0,0) to (m,n) (the current destination)

How to make a decision among these choices (considering the result of subproblems)?

Assume we have known the max reward from (0,0) to (m-1,n) **[MAX_REWARD(m-1,n)]**

as well as that from(0,0) to (m,n-1)**[MAX_REWARD(m,n-1)]**.

How to compute the MAX_REWARD(m,n) [the max reward from (0,0) to (m,n)]?

Problem 6

MAX_REWARD(m,n) is the maximum reward from (0,0) to (m,n) (the current destination)

How to make a decision among these choices (considering the result of subproblems)?

Assume we have known the max reward from (0,0) to (m-1,n) **[MAX_REWARD(m-1,n)]**

as well as that from(0,0) to (m,n-1)**[MAX_REWARD(m,n-1)]**.

How to compute the MAX_REWARD(m,n) [the max reward from (0,0) to (m,n)]?

choose the larger one, and plus A[m,n].

DP[m,n] = max(DP[m-1,n], DP[m,n-1]) + A[m,n]

Problem 6

Base Case:

A[1..i][1..j] has the reward, and the DP problem is from (0,0) to (m,n)

Problem 6

Base Case:

A[1..i][1..j] has the reward, and the DP problem is from (0,0) to (m,n)

MAX_REWARD(0,j) = MAX_REWARD(i,0) = 0 for 0<=i<=m, 0<=j<=n

Problem 6

```
define MAX_REWARD[m,n]

for i from 1 to m:
    MAX_REWARD[i,0] = 0
for j from 1 to n:
    MAX_REWARD[0,j] = 0

for i from 1 to m:
    for j from 1 to n:
        MAX_REWARD[i,j] = A[i,j] +
        max(MAX_REWARD[i-1,j], MAX_REWARD[i,j-1]

return MAX_REWARD[m,n]
```

Running Time: theta(n^2)

Problem 7

Problem 7

Given the array $A[1 \ldots n]$ consisting of n distinct integers, devise a dynamic programming algorithm to output the length of the longest increasing subsequence of A, i.e., a subsequence of A whose elements are sorted in an increasing order. Find the running time of your algorithm.

Problem 7

Example: [3, 1, 4, 2, 8, 5, 10, 6]

output = 4

lis(i): the length of the longest increasing subsequence, ending with A[i].

$$lis(i) = \begin{cases} 1 & i = 1 \quad \text{base case} \\ \max\limits_{\substack{j<i, \\ A[j]<A[i]}} (lis(j)) & i > 1 \end{cases}$$

$$LIS(n) = \max_{i \in [1,n]} (lis(i))$$

Problem 7

pseudo-code (Bottom-up DP)

```
LIS(A[1...n]):

    longest = 1                 <= result
    memo[1...n] = [0]
    memo[1] = 1                 <= base case

    for i = 2 to n:
        for j = 1 to i-1:
            if A[j] < A[i]:
                memo[i] = max(memo[i], memo[j]+1)
        longest = max(longest, memo[i])
    return longest
```

Problem 7

pseudo-code (Bottom-up DP) TC = $O(n^2)$

```
LIS(A[1...n]):

    longest = 1                ← result
    memo[1...n] = [0]
    memo[1] = 1                ← base case

O(n²) for i = 2 to n:
    (i-1) iterations  for j = 1 to i-1:
                          if A[j] < A[i]:
                              memo[i] = max(memo[i], memo[j]+1)
                      longest = max(longest, memo[i])
    return longest
```

Q & A

Thank you