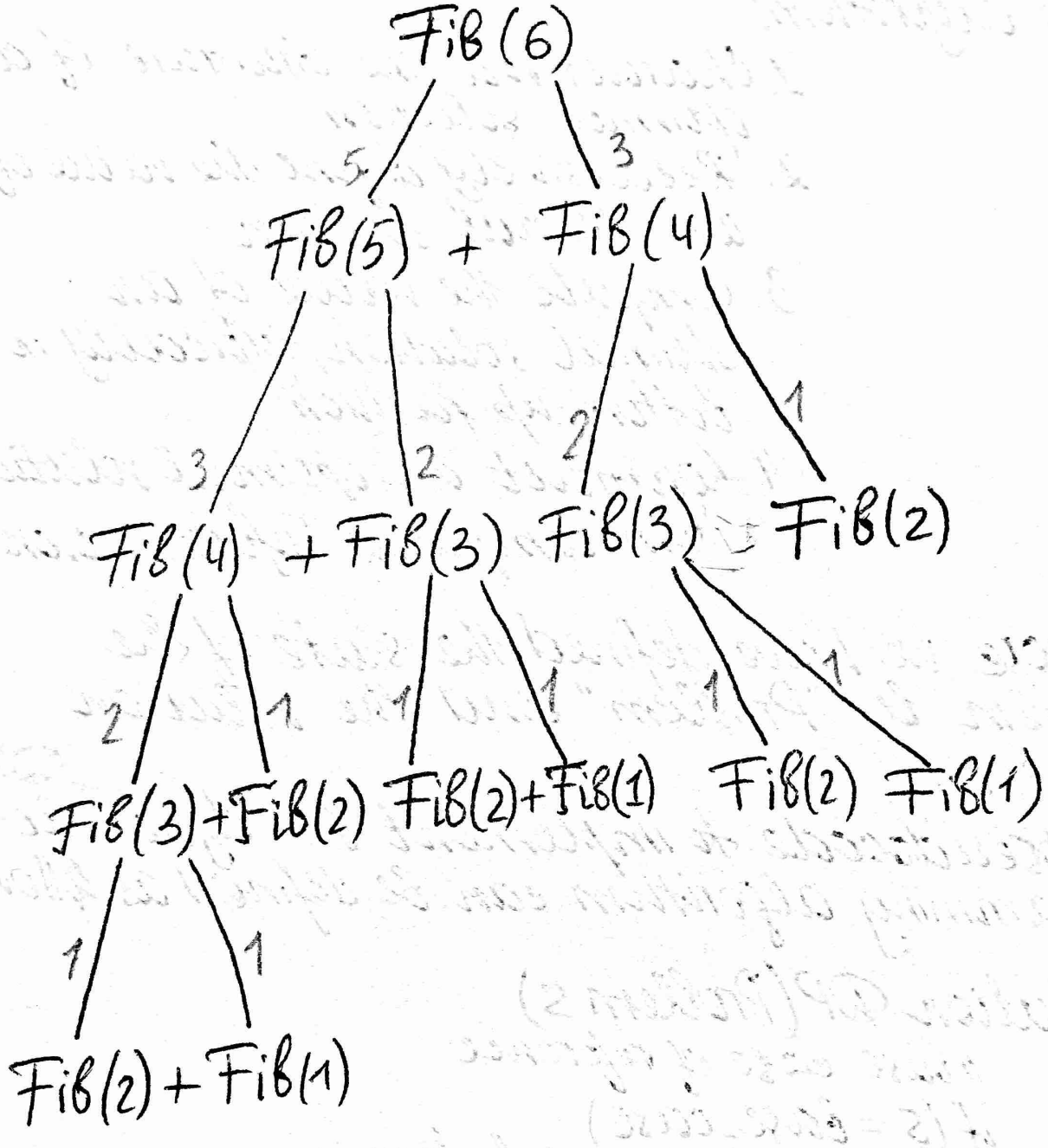
Problem 1



Recursive relation: $T(n) = T(n-1) + T(n-2) + 1$

$T(n) = 0$ for $n = 1, 2$

$$\Rightarrow a^n = a^{(n-1)} + a^{(n-2)} \quad / \div a^{(n-2)}$$

$$a^2 = a + 1 \Rightarrow a \approx 1.61$$

$\Rightarrow$ Runtime $O(n^{1.61})$

Problem 2)

Suppose, we defined the state of the problem by the class Problem and the solution by the class Solution.

DP-Dynamic Programming

// - comment

```
DPAlgo(Problem p)
    // base case of reference
    if(p==base_case)
        return base_solution
    // if the solution has already been found
    if(contains_key(p))
        return key(p)
    // find the solutions using recurrence relationship
    Solution answer = recurrence relation(p)
    // store and return the solution
    // put() - associates the specified problem with specified
    // answer. If contains a key, old value is replaced
    put(p, answer)
        return answer
```

In general, the run-time complexity of the DP algo is O(problem space * recurrence relation)

Example on Fibonacci numbers:

```
Fibonacci(n)
    F[0] = 0
    F[1] = 1
    for j = 2 to n
    do F[j] = F[j-1] + F[j-2]
        return F[n]
```

Running time: $T(n) \in \Theta(n)$

Problem 3

1. Matrix chain multiplication problem is a problem that can be solved using dynamic programming. Its goal is to find the most efficient way to multiply matrices. It involves the question of determining the optimal sequence for performing a series of operations.

2. 2 different ways

$((A_1 A_2) A_3)$ and $(A_1 (A_2 A_3))$

3. If we multiply according to the parenthesization $((A_1 A_2) A_3)$, we get $7 \times 50 \times 10 = 3500$ ($A_1 A_2$) plus $7 \times 10 \times 100 = 7000$, resulting in total of 10500 scalar multiplications.

If we multiply according to $A_1 (A_2 A_3)$,

We get $50 \times 10 \times 100 = 50000$
$+ 7 \times 50 \times 100 = 35000$ } total 85000 scalar multiplications

(It shows that the first way of matrices multiplication is far more efficient)

4. It would take exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product.
Assuming that 2 #s of arbitrary length can be multiplied in constant time, asymptotic lower bound is $\Omega(n^3)$.

5. $O(n^3)$

6. It needs 2D array to store all the solutions of the subproblems

7. DP algo stores the solutions to the subproblem in 2D array. Thus, whenever the algo needs the already computed subproblem, it finds the solution in 2D array.

Problem 4

Let R contain max revenue for cutting rod of various size.

$R[0] = 0$

$R[1] = 2$

$R[2] = \max($cutting rod in 1 1 or keeping rod of size 2$) =$
$= \max(4, 3) = 4$

$R[3] = \max($cutting rod in 1 1 1, cutting rod in 1 2, keeping rod of size 3$) = \max(6, 4, 7) = 7$

R[4] = max(cut in 1111, 112, 13, 22, 4) =
max(8,8,9,8,8) = 9

R[5] = ... = 11
R[6] = 14
R[7] = 16
R[8] = 18

So array of optimal revenue is:
R[0] = 0
R[1] = 2
R[2] = 4
R[3] = 7
R[4] = 9
R[5] = 11
R[6] = 14
R[7] = 16
R[8] = 18

max cut will give us 18 by cutting in size 1 1 3 3

Problem 5 Beginning from 1, we will generate solution up to m, where m is the amount of money for which we want change. For a particular amount $1 <= k <= m$, we will iterate through the array A of different coins, while $A[j] <= k$, perform k - A[i] and check the value of of k-A[i] and add one for A[i] coin that is selected. This will give the min # of coins needed to

make k when one of the coins is A[i], thus, finding min for each A[i], where 1<=1<=n. So if N[k] denotes min # of coins needed to make k, the dynamic programming equation is $N[m] = 1 + \min N[m - A[i]]$.

Here is an algo for this problem:

```
ALGO(m)
1. N[0]=0; N[1]=1; B[0]={0,0....0}; B[1]={1,0,...0}
2. for i=2 to m:
3.     min = ∞
4.     for j=1 to n
5.         if i>=A[j] and min>1+N[i-A[j]]
6.             min=1+N[i-A[j]]
7.             B[i]=update(B[i-A[j]],j)*
8.     N[i]=min
9. return B[m]
```

Here B[i] corresponds to array of length n, where B[i][j] denotes the # of coins A[j] used to compute amount i.

*Also B[i]=update(B[j],k) will copy array B[j] into array B[i], while B[i][k]=B[j][k]+1. i.e. B[i] will have 1 extra A[k] coin than B[j].

The runtime is O(m·n), where n is the # of coins and m is the amount of which we want change.

Problem 6

Problem Statement: Given a value N, we need a change for N cents, given that we are provided with supply of coins $\{S_1, S_2, S_3, S_4 \ldots S_n\}$

(S_1: min value coin; S_n: max value coin)

The # of coins will be minimum if the maximum sized coins are or can be accomodated in the solution. So for any max-size coin there is 2 possibilities. Either S_n will be present or not. Since we don't know whether S_n will be in the solution or not. So we consider both cases and *falce* is the one that outputs min # of coins.

Let $DP(N,S)$ be min # of coins that adds up to 'N', given that we are provided with supply of coins 'S'

$$DP(N,S) = \begin{cases} 0, & \text{if } N=0 \\ \min \begin{cases} DP(N-S_n, S-S_n)+1 \\ DP(N, S-S_n) \end{cases} & \text{if } N>0 \end{cases}$$

'$S-S_n$' - set excluding 'S_n'

"$DP(N-S_n, S-S_n)+1$" $\rightarrow$ S_n is taken, therefore 1 is added

$DP(N, S-S_n)$ - S_n is not taken.

Algorithm:

```
DP(N,s)
  if(N==0)
     return 0
  else
     if(Table(N-Sn, S-Sn)==0)
          a = DP(N-Sn, S-Sn)+1        Table(N-Sn, S-Sn)=a
     else
          a = Table(N-Sn, S-Sn)
     if(Table(N, S-Sn)==0)
```

```
        B = DP(N, S - S_n)
        Table(N, S - S_n) = B
    else
        B = Table(N, S - S_n)
    c = min(a, B)
    return c
```

Runtime: total # of different function calls =

DP(N, S)

N	S_n
N-1	$S_n - 1$
N-2	$S_n - 2$
N-3	$S_n - 3$
⋮	⋮
0	0

$(n+1) \times (m+1)$

$= (n+1)(m+1)$

Time complexity: $O(m \cdot n)$

Problem 7 First, we'll find the activities of set S_j.

If k exists in S_{ij}, then $i < k < j$ i.e. $j - i \geq 2$.

But $f_i \leq S_k$ and $f_k \leq S_j$. If we proceed with k at $j-1$ and reduce the value of k, then we stop after k reached i. We also stop the process after finding f_k. Let's start with 2 imaginary activities a with $f_0 = 0$ and a_{n+1} with $S_{n+1} = \infty$.

We are looking for mutually compatible activities in $S_{0,n+1}$ in the max size set $A_{0,n+1}$

We'll allow tables $c[0 \ldots n+1, 0 \ldots + m+1]$ as recurrence property, so $c[i,j] = |A_{ij}|$ and activity

[0...n+1, 0...n+1]. Activity k from activity [i,j] will be placed into Aij. We can put the values in tables as j-i. Sij=0, if j-i<2, then c[i,j]=0 ∀i and c[i,i+1]=0 for 0≤i≤n. The starting and ending times are arrays s and f in Recursive Activity selection and Greedy activity selection. s and f include the 2 imaginary activities and will be filtered at increased ending time.

Pseudocode:

```
Dynamic ActivitySelection (s, f, n)
step 1: Assume c[0...n+1, 0...n+1] and
        activity [0...n+1, 0...n+1] are tables for i=0 to n
step 2: c[i,i]=0
step 3: c[i,i+1]=0
step 4: c(n+1, n+1)=0
step 5: for l=2 to n+1   // l is an increasing difference between j-i
            for i=0 to n-l+1
                j=i+l
                c[i,j]=0
                k=[j-1]
            While f[i]<f[k]
                if f[i]≤s[k] and f[k]≤s[j] and
                c[i,k]+c[k,j]+1>c[i,j].
                    do c[i,j]=c[i,k]+c[k,j]+1
                        activity [i,j]=k
                        k=k-1
step 6: Print "Mutual Compatible activities of
                Max size set"
step 7: Print c[0, n+1]
step 8: Print "Activities in the set"
```

Step 9: Print_activities (c, activity, 0, n+1)
Print_activities (c, activity, i, j)

Step 10: if c[i, j] > 0
k = activity[i, j]
print k

Print_activities (c, activity, i, k)
Print_activities (c, activity, k, j)

Set of activities put in optimal solution A_{ij} are recursively printed by Print_activities function. First, activity k will be printed with the max value of c[i, j] and second, values of Activities A_{ik} and A_{kj} will be printed recursively. Recursion will be reached at minimum when

$c[i, j] = 0$, so $A_{ij} = 0$

Runtime of Greedy Activity solution is $\Theta(n)$
Runtime of Dynamic Activity solution is $O(n^3)$