Amber Harding
CSCI 261

Analysis of Algorithms
Homework 5

Arthur Nunes-Harwitt

1. a

	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

(b)

Base Case:

For n = 2, A1 * A2 = A1A2

Induction:

For n = k + 1,

((A1A2….. Aj)(Aj+1 * Aj-2…. Ak+1)

((k+1) - j + 1)

(j-1) + (((k+1) - j + 1) - 1) = k - 1

Adding the outermost parentheses

(k-1) + 1 = k

N = k + 1

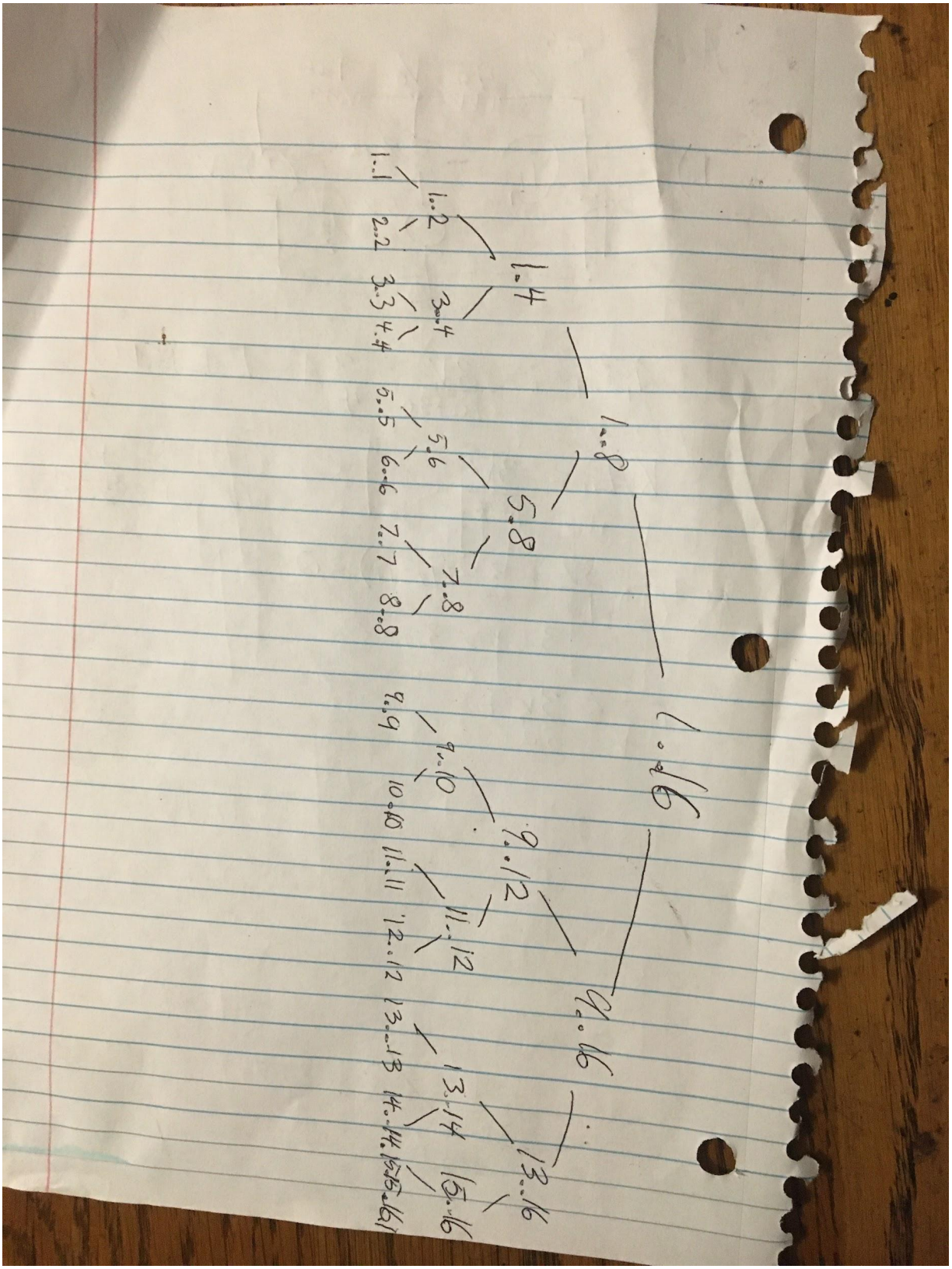
2. (a) CLRS 15.3-1

Running a recursive-matrix-chain is asymptotically more efficient than enumerating all the ways of parenthesizing the product and computing the number of multiplications for each.

The enumeration approach finds all ways to parenthesize the left half, then the right half and looks at all possible combinations of the left and right halves. The recursive-matrix-chain finds the best way to parenthesize the left half, finds the best way to parenthesize the right and then combines those results. The complexity of combining the left and right halves are O(1).

(b) **Draw the recursion tree f**or the merge-sort algorithm on an input sequence of length 16. Explain why memoization fails to speed up a good divide-and-conquer algorithm like merge-sort.

Amber Harding
CSCI 261



The merge sort algorithm performs multiplication at a single pair of matrices at a time of the array that is being sorted and then that result is returned when the same input is used again. The subproblems do not overlap and therefore memoization will not improve the runtime of the algorithm.

(c)

Yes this problem does exhibit optimal substructure. An optimal substructure is exhibited when an optimal solution can be constructed from optimal solutions of its subproblems. This can also be applied to find the maximum amount of multiplications because optimal substructure is any optimal solution of size n.

3. (**project**) Recall F_n the recurrence that defines the Fibonacci numbers. Write a function fibDyn that computes Fibonacci numbers which implements the naive recurrence via dynamic programming.

4. Consider the 0-1 knapsack problem in CLRS chapter 16.

(a)

```
0-1-KNAPSACK(n, W)
    Initialize an (n + 1) by (W + 1) table K
    for i = 1 to n
        K[i, 0] = 0
    for j = 1 to W
        K[0, j] = 0
    for i = 1 to n
        for j = 1 to W
            if j < i.weight
                K[i, j] = K[i - 1, j]
            else
```

K[i, j] = max(K[i - 1, j], K[i - 1, j - i.weight] + i.value)

(b) (**project**) Give a dynamic programming solution to the 0-1 knapsack problem that is based on the previous problem; this algorithm should take and return the same values as the functional pseudo-code above. Implement this algorithm and call it knapsack.

(c) (**project**) Implement a function called knapsackContents that takes the same values but returns a sequence of items that maximize the knapsack's value.

5. Consider the problem of neatly printing a paragraph on the screen (or on a printer). For the project portion, put all functions in the file printPar. The input text is a sequence S of n words (represented as strings) of lengths $\ell_1, \ldots, \ell_n$ (measured in characters). The input bound M is the maximum number of characters a line can hold. The key to neatly printing a paragraph is to identify in the text sequence the lines of the paragraph so that new-lines can be placed at the end of each line. We can formalize the notion of the "badness" of a line as the number of extra space characters at the end of the line or ∞ if the bound M is exceeded. We can formalize the notion of the "badness" of a paragraph as the badness of the worst (i.e., maximum) line of the paragraph not

including the last line. Thus to identify the lines for a neat paragraph, we seek to minimize the badness of the paragraph.

(a) If a given line contains words i through j, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} \ell_k$. Write functional pseudo-code for the function $e(S, M, i, j)$ that computes the number of extra space characters at the end of a line.

S = Strings

M = Max num of characters per line

I - j = given line.

e(S, M, i, j):

finishedLine = ""

```
Line = S[i - j]

For word in line:

        If  len(word) + len(finishedLine) < M:

                finishedLine += word + " "

Return finsihedLine + len(finsihedLine)
```

(b) (**project**) Write a procedure extraSpace that implements *e*.

(c) Use the function *e* to write functional **pseudo**-code for the function *bl*(*S, M, i, j*) that computes line badness.

```
bl(S, M, i, j):

    finishedLine = ""
    LineBadness = 0

    Line = S[i - j]

    For i in range(1, M + 1):

            If  len(S[i]) + len(finishedLine) < M:

                    finishedLine += S[i] + " "

            Else:

                    LineBadness += 1

    Return LineBadness
```

(d) (**project**) Write a procedure badnessLine that implements *bl*.

(e) Write functional **pseudo**-code for the recursive function *mb*(*S, M*) that computes the minimum paragraph badness (using slicing). The base case must be that the sequence *S* is the last line (and *not* that *S* is empty).

```
mb(S, M):
    I = 0,
    J = 10
    If len(S) > 0:
        For i in range( I, M + 1):

            Badness += bl(S, M, i, j)
            Badness += bl(S,M, i + 10, j + 10)

Return badness
```

(f) Write functional pseudo-code for the recursive function $mb^0(S, M, i)$ where $mb^0(S, M, i) = mb(S[i :], M)$. The base case must be that the sequence characterized by *S* and *i* is the last line (and *not* that *S* is empty).

```
mb0(S, M, i):
    Line = ""
    For x in range (i, M)
        If len(S[x] + Line < M:
            Line += S[x]
        Else:
            mb0(S, M, i)
```

(g) (**project**) Write a recursive procedure minBad that implements the function mb^0. It

should take three parameters: *S*, *M*, and *i* a slicing index.

(h) (**project**) Write a procedure minBadDynamic that implements the function mb^0 using dynamic programming. It should take only two parameters: *S*, and *M*.

(i) (**project**) Write a procedure minBadDynamicChoice that implements the function mb^0 using dynamic programming. In addition to returning *mb*(*S, M*), it should also return the choices made. Then write a procedure printParagraph which takes two parameters: *S* and *M* that displays the words in *S* on the screen using the choices of minBadDynamicChoice. What is the asymptotic running time of your algorithm?

6. CLRS 24.1-1

Using vertex z as source:

D values:

s	t	x	y	z
∞	∞	∞	∞	0
2	∞	7	∞	0
2	5	7	9	0
2	5	6	9	0
2	4	6	9	0

π values

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
z	NIL	z	NIL	NIL
z	x	z	s	NIL
z	x	y	s	NIL

z	x	y	s	NIL

Changing the weight of edge(z, x) to 4

D values

s	t	x	y	z
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	-2

π values

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
NIL	s	NIL	s	NIL
NIL	s	y	s	t
NIL	x	y	s	t
NIL	x	y	s	t

For edge (z, x) the result is False because x.d = 4 > z.d + w(z, x) = -2 + 4