**Chapter 6 – Practice Exercises**

**Exercise 6.1**

**Input:** a1,…,an

**Goal:** Substring w/ max sum

1.) **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let T(i) = maximum sum of a contiguous subsequence ending at ai

2.) **State recurrence for entries of your table in terms of smaller subproblems.**

**Base Case:** T(0)

**General Recurrence:** T(i) = max{0, T(i-1)} + a[i], where 1 ≤ i ≤ n

3.) **Write pseudocode for your algorithm to solve this problem.**

T(0) = 0

for i = 1 to *n* do

T(i) = *max*{0, T(i-1)} + a[i]

return *max*{T(.)}

4.) **State and analyze the running time of your algorithm.**

We have one loop through *n* inputs to set the table values, and a second loop to find the max. Overall run time is linear O(n).

**Exercise 6.2**

**Input:** a[1…n] where *n* is hotels

**Goal:** You want to plan your trip to minimize the total penalty - that is, the sum, over all travel days, of the daily penalties.

1.) **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

T(j) = the minimum penalty incurred to get to a*j*.

2.) **State recurrence for entries of your table in terms of smaller subproblems**.

**Base Case:** T(0) = 0

**General Recurrence:** T(j) = *min*{T(i) + (200 – (aj – ai))2, where 1 ≤ i < j, 1 ≤ j ≤ n

**3.) Write pseudocode for your algorithm to solve this problem.**

T(0) = 0

for j = 1 to n do

T(j) = (200 – aj)2

for i = 1 to j - 1 do

T(j) = *min*{T(j), T(i) + (200 – (aj – ai))2}

return T(n)

**4.) State and analyze the running time of your algorithm.**

We have two loops through n and j - 1 inputs so overall run time is O(n2).

**Reference:** <https://edstem.org/us/courses/42665/discussion/3335441>

**Exercise 6.3**

**Input:** m1, m2,…,mn where *n* is possible locations

**Goal:** Compute the maximum expected total profit subject to the given

constraints.

1.) **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let T(i) = the maximum profit expected at location pi

2.) **State recurrence for entries of your table in terms of smaller subproblems.**

**Base Case:** T(0) = 0

**General Recurrence:** T(i) = *max*{T(j) + pi, pi}, where 1 ≤ i ≤ n, 1 ≤ j < i, and mi – mj >= k

3.) **Write pseudocode for your algorithm to solve this problem.**

T(0) = 0

for i = 1 to n do

T[i] = p[i]

for j = 1 to i – 1 do

if m[i] – m[j] > k then

T(i) = *max*{T[i], T[j] + p[i]}

return *max*{T}

4.) **State and analyze the running time of your algorithm.**

We have two loops through n and i - 1 inputs so overall run time is O(n2).

**References:** <http://users.eecs.northwestern.edu/~dda902/336/hw6-sol.pdf>; https://edstem.org/us/courses/42665/discussion/3354188

**Exercise 6.4**

**Input:** s[1…n] **where *n* is characters**

**Goal:** Determine whether the string s[.] can be reconstituted as a sequence of valid words

1.) **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let T(i) = the TRUE/FALSE answer to the following problem: Can the string s1, s2,…,si be broken into a sequence of valid words?

2.) **State recurrence for entries of your table in terms of smaller subproblems.**

**Base Case:** T(0) = TRUE

**General Recurrence:** False ∨{dict(s[j…i]) ∧ E(j – 1)}, where 1 ≤ i ≤ n, 1 ≤ j ≤ i

3.) **Write pseudocode for your algorithm to solve this problem.**

a.) Base pseudocode:

T(0) = TRUE

for i = 1 to *n* do

T(i) = FALSE

for j = 1 to *i* do

if T(i) ∨ {dict(s[j…i]) ∧ E(j – 1)} then

T(i) = TRUE

*prev*(i)= j // This is an addition used for part *b*

return T(n)

b.) In the event that the string is valid, make your algorithm output the corresponding sequence of words:

*words* = “”

i = n

while i > 0: do

*words* = s[*prev*(i)…i] + “” + *words*

i = *prev*(i) – 1

return *words*

4.) **State and analyze the running time of your algorithm.**

We have one loop through *n* inputs to set the table values, and a second loop to find the max. Overall run time is linear O(n).

**Exercise 6.8**

**Input:** *Substring #1* – x1…xi, *Substring #2* – y1…..yi

**Goal:** Find the length of their longest common substring

1.) **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let T(i, j) = length of x1….xi with y1….yj where we only consider substrings with xi = yj as the last letter.

2.) **State recurrence for entries of your table in terms of smaller subproblems.**

**Base Case:** T(i, 0) = 0 for any 0 ≤ i ≤ n, T(0, j) = 0 for any 0 ≤ j ≤ n

**General Recurrence:** T(i, j) = {1 + T(i – 1, j – 1) if xi = yj, 0 if xi ≠ yj}, where 1 ≤ i ≤ n, 1 ≤ j ≤ m

3.) **Write pseudocode for your algorithm to solve this problem.**

for i = 0 to n do

T(i, 0) = 0

for j = 0 to m do

T(0, j) = 0

for i = 1 to n do

for j = 1 to m do

if xi = yj

T(i, j) = {1 + T(i – 1, j – 1)}

else

T(i, j) = 0

return *max*{T(. , .)}

4.) **State and analyze the running time of your algorithm.**

The two base case loops are linear O(n) and O(m). The nested loops establish the

values for T and find the maximum dominate the run time, which is O(nm).

**Exercise 6.18**

**Input:** Positive integers x1, x2,…, xn; another integer v

**Goal:** Make change for v, using each denomination xi at most once

1.) **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let T(i, j) = the subset of coins x1….xi form value j where xi is used at most once

2.) **State recurrence for entries of your table in terms of smaller subproblems.**

**Base Case:** T(i, 0) = 0, T(0, j) = 0 for no objects and no knapsack

**General Recurrence:** T(i, j) = { *max*{T(i – 1, j), xi + T(i – 1, j - xi)} if xi < j; T(i – 1, j) if otherwise }, where 1 < i < n, 1 < j < v

3.) **Write pseudocode for your algorithm to solve this problem.**

for i = 0 to n: T(i, 0)

for j = 0 to v: T(0, j)

for i = 1 to n do

for j = 1 to v do

if x[i] < j then

T(i, j) = *max*{T[i – 1, j], x[i] + T[i – 1, j – x[i]]}

else

T(i, j) = T[i – 1, j]

if T(n, v) = v then

return TRUE

else

return FALSE

4.) **State and analyze the running time of your algorithm.**

The two base case loops are linear O(n) and O(v). The nested loops establish the values for T and find the maximum dominate the run time, which is O(nv).

**Exercise 6.19**

**Input:** Positive integers x1, x2,…, xn; k; v

**Goal:** Make change for v using at most k coins, of denominations x1, x2,…, xn

**1.) Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let T(v) = the maximum number of coins to for the value of v

**2.) State recurrence for entries of your table in terms of smaller subproblems.**

**Base Case:** T(0) = 0

**General Recurrence:** T(v) = *max*{T(v), xi + T(v - xi)} if xi < v, where 1 < i < n

**3.) Write pseudocode for your algorithm to solve this problem.**

T(0) = 0

for v = 1 to V do

T(v) = 0

for i =1 to n do

if x[i] < v then

T(v) = *max*{T(v), x[i] + T[v – x[i]]}

if T(V) < k then

return TRUE

else

return FALSE

**4.) State and analyze the running time of your algorithm.**

The nested loops that establish the values for T(v) run in O(V) and O(n). The other step are ran in O(1), so the overall runtime is O(nV).

**Reference:** <https://iq.opengenus.org/unbounded-knapsack-problem/>

**Exercise 6.20**

**Input:** n words (in sorted order); frequencies of these words: p1, p2,…,pn

**Goal:** Find the binary search tree of lowest cost (defined above as the expected number

of comparisons in looking up a word).

**1.) Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let T(i, j) = the minimum cost binary search tree for words p1, p2,…,pn.

**2.) State recurrence for entries of your table in terms of smaller subproblems.**

**Base Case:** T(i, i) = p[i] for all 1 < i < n, T(i, j) = 0 for all 1 < i < (n + 1) and 0 < j < n and j < i

**General Recurrence:** T(i, j) = *min*{T{i, j), T(i, k – 1) + T(k + 1, j) + (p[i] +… + p[j])}, where 1 < i < j < n

**3.) Write pseudocode for your algorithm to solve this problem.**

for i = 1 to n do // This for-loop sets middle diagonal to p[i]

T(i, i) = p[i]

for i = 1 to n + 1 do // This for-loop sets bottom diagonal to 0

for j = 0 to i – 1 do

T(i, j) = 0

for w = 1 to n – 1 do

for i = 1 to n – w do

j = i + w

T(i, j) = infinity

cost = 0

for k = i to j do

cost = cost + p[k]

T(i, j) = *min*{T(i, j), T(i, k – 1) + T(k + 1, j)} + cost

return T(1,n)

**4.) State and analyze the running time of your algorithm.**

There are two base cases for loops running at O(n) and O(n2), then three nested for loops running at O(n) each. Overall runtime would be O(n3).

**Reference:** <https://edstem.org/us/courses/42665/lessons/72882/slides/390111>

**Exercise 6.26**

**Input:** x[1,…,n], y[1,…,m], and a scoring matrix z

**Goal:** Return the highest-scoring alignment

**1.) Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let T(i, j) = the maximum scoring alignment achieved of x1….xi with y1….yj

**2.) State recurrence for entries of your table in terms of smaller subproblems.**

**Base Case:** T(0,0) = 0, T(i, 0) = T(i -1, 0) + z(x[i], -) for all 1 ≤ i ≤ n, T(0, j) = T(0, j – 1) + z(-, y[j]) for all 1 ≤ j ≤ m

**General Recurrence:** T(i, j) **=** *max*{T(i -1, j -1) + z(x[i], y[j]),

T(i -1,j) + z(x[i], -),

T(i,j - 1) + z(-, y[j])}, where 1 ≤ i ≤ n, 1 ≤ j ≤ m

**3.) Write pseudocode for your algorithm to solve this problem.**

T(0, 0) = 0

for i = 1 to n do: T(i, 0) = T([i – 1], 0) + z(x[i], -)

for j = 1 to m do: T(0, j) = T(0, [j – 1] + z(-, y[j])

for i = 1 to n do

for j = 1 to m do

T(i, j) = *max*{T([i – 1], [j – 1]) + z(x[i], y[j]), T([i – 1], j) + z(x[i], -), T(i, [j – 1]) + z(-, y[j])}

return T(n , m)

**4.) State and analyze the running time of your algorithm.**

The two base case loops are linear O(n) and O(m). The nested loops establish the

values for T and find the maximum dominate the run time, which is O(nm).