93/100

Ruben Tegnide
HW3
Q.1

(1) **(Finding elements near the median)** (10 points) Given an unsorted array A of n distinct numbers and an integer k, where $1 \leq k \leq n$, design an algorithm that finds the k numbers in A that are *closest in value* to the median of A in $\Theta(n)$ time.

(Note: Finding the numbers that are closest in *value* to the median has no relationship in general with how these numbers are ordered in the unsorted array A.)

10/10

First we want to find the median. In order to do so we use the kth_smallest algorithm where we pass in $\left\lfloor \frac{n+1}{2} \right\rfloor$ (where n is the length of the array) as k which will give us the median of the array. ✓

Now that we know the median we can find the absolute difference between the median and every other element in the array. We will create an additional array that will hold arrays storing the absolute difference ✓ between the median and the elements value and the element value itself. Then we will iterate over the array and skip over the first element that matches our median. At every position we append the absolute difference between the median and the elements value as well as the elements value (for recovery purposes) within the newly created array

why? it could contain median don't skip it.

$A' = \overset{1}{|}\, A_1 \,|\, A_2 \,|\, A_3 \,|\, A_4 \,|\, A_5 \,|\, \underbrace{A_6}_{*} \,|\, A_7 \,\overset{n}{|}$ *Let's say A_6 came out to be the median

new array of arrays $= [[|A_1 - A_6|, A_1], [|A_2 - A_6| A_2] \cdots [|A_5 - A_6|, A_5], [|A_7 - A_6|, A_7]]$

Once we have this new array we can find the k^{th} smallest difference by using the kth_smallest algorithm on the first element of every sub array within our new array and we pass in our k from the original problem. This will give us the element with the kth smallest difference. With this element we partition the new array around this element and have elements with differences smaller than the pivot to go on its right and the others to go on its left.

$[\, A'[i][1] < k \;\vdots\; k \;\vdots\; A'[i][1] > k \,]$

↳ kth smallest difference

Our newly partitioned array will have all differences less than the kth smallest difference on the left side of the array so now we can return all elements in A' at positions $A'[1][2] - A'[k][2]$ and we will have all k values closest to the median.

We know this works because we first find the median and then get all the differences between each element and the median. Then we find the k^{th} smallest element of the remaining elements and partition around that then return the first k elements. The first k elements must contain the elements that are closest to the median because their differences are minimized and any elements to the right of the first k elements will have larger differences. We also know there won't be any elements to the left of the first k elements because we partitioned around k so adding any more elements would surpass

Ruben Tequida

Q. 1

our desired k.

Time Analysis

```
closest (A, k)
    n = A.length
    median := kth_smallest ($\lfloor \frac{n+1}{2} \rfloor$)  - finds median                          } $\Theta(n)$
    med_index := A.indexof (median)  - index of median in original array    } $\Theta(n)$
    A' := new array of arrays
    for i := 1 to n+1 do
        if i != med_index do
            A'.append ([|A[i] - median|, A[i]])  } $\Theta(1)$
        end
    end
    k_val = kth_smallest (k)  - find $k^{th}$ smallest in A'                 } $\Theta(n)$
    partition (A', k_val)  - partition A' around k_val                       } $\Theta(n)$
    closest = new array
    for i := 1 to k+1 do
        closest.append (A'[i][2])                   } $\Theta(1)$
    end
    return closest
```

for loop over i from 1 to $n+1$: $\sum_{1 \le i \le n} \Theta(1) = \Theta(n-1+1) = \Theta(n)$

for loop over i from 1 to $k+1$: $\sum_{1 \le i \le k} \Theta(1) = \Theta(k)$

✓

$$
\begin{aligned}
T(n) &= \Theta(n) + \Theta(n) + \Theta(n) + \Theta(n) + \Theta(n) + \Theta(k) \\
&= 5\Theta(n) \underbrace{\cancel{+\Theta(k)}}_{\to \text{ less than } n} \\
&= 5\Theta(n) = \Theta(n)
\end{aligned}
$$

Ruben Tequida
Q. 2

(2) **(Finding quantiles)** (20 points) For a set S of n numbers and an integer k, where $1 \le k \le n$, the *kth-quantiles* of S are $k-1$ elements from S whose ranks in S divide the sorted set into k groups that are of equal size (to within one unit).

Given an unsorted array A of n distinct numbers, design an algorithm that finds the kth-quantiles of A in $O(n \log k)$ time.

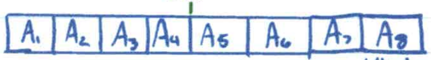
(Note: As an illustration, the 4-quantiles of a set of scores are the values that define the 25-, 50-, and 75-percentile cutoffs. Similarly, the 10-quantiles of a set are the values that define the 10-, 20-, 30-, ..., and 90-percentile cutoffs.)

20/20

We want to use a divide and conquer approach that will divide the problem into smaller quantiles we need to find until we are at the point where $k==1$. At this point we return nothing becaus a k-quantile will be the entire array as 1 group.

If k is greater then 1 we want to find the quantile separator that will be closest to the middle of the array. We will do this by finding which quantile will be closest to the middle without going over it $mid\_q = \lfloor k/2 \rfloor$

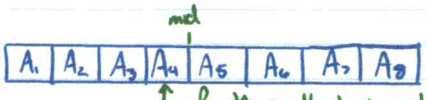
array of 8 elements with $k = 4$



the quantile separator closest to mid will be $\lfloor 4/2 \rfloor = 2$

Next we want to find element that would be at the position mid-q if the array were sorted. So we use the kth smallest algorithm and pass in $\lfloor mid\_q \cdot n/k \rfloor$. This will find one of our quantile separators so we store it.

ok.



fourth smallest in sorted order (2nd quantile separator)

Using $\lfloor mid\_q \cdot n/k \rfloor = \lfloor 2 \cdot 8/4 \rfloor = 4$ we find we want to find the 4th smallest element

Next we partition the array around the element we just found so all values smaller than it are on its left and larger values on its right.

Then, we want to recursively call the quantile finding algorithm but on the left half of the array up to the quantile we just found and on the right half going from the quantile +1 to the end. In both the left and right case we pass the list of found quantiles as well as a new k equal to $\lceil k/2 \rceil$ for the right portion and $\lfloor k/2 \rfloor$ for the left portion.

Within these recursive calls we will find all the quantiles that are to the right of the middle of the array if $k > 1$ or do nothing if $k = 1$ effectively finding all quantiles.

Ruben Tequida
Q. 2

We know this works because find one quentile everytime we all the function and then divide the problem into subproblems. We work on the left side of the array with a k value equel to $\lfloor k/2 \rfloor$ (which removes the quantile we just found to the left of the middle) end $\lceil k/2 \rceil$ for the right side to get all the quntiles on the right side of the middle. Dividing k in half as well as dividing the array in half easures we find all k-quantiles.

Time Analysis

```
Q ( A, k, found)
        if k != 1 do
        n = A.length
        mid-q := $\lfloor k/2 \rfloor$
        val-q := ( kth_smallest ( A, $\lfloor$mid-q · n/k$\rfloor$ ) )   } $\Theta(n)$
         found.append (val-q)
         partition (A, val-q)                                           } $\Theta(n)$
          Q(A[1 : $\lfloor$mid-q · n/k$\rfloor$], $\lfloor k/2 \rfloor$, found)   } $T(k/2)$
         Q(A[mid-q · n/k + 1 : n], $\lceil k/2 \rceil$, found)           } $T(k/2)$
```

✓

$T(n) = 2T(k/2) + 2\Theta(n)$

$\log_2 2 = 1 \qquad c=1 \qquad d=0$

$\rightarrow \theta(n^c \log^{d+1} n) = n \log k$

Ruben Tequida

Q. 3

(3) **(Answering dynamic kth-smallest queries) (bonus)** (10 points) Given a set of n numbers, the kth-smallest element can be found in $\Theta(n)$ time using the algorithm we learned in class. Suppose we have a dynamic set S of numbers that changes over time, and we want to be able to efficiently support the operations of

(a) *inserting* a new number into set S, and
(b) *finding* the kth-smallest element in the current set S, for any k.

We could support both operations by representing S as an unsorted array A. Inserting a new element would take just $\Theta(1)$ time. Finding the kth-smallest element, however, would take $\Theta(n)$ time. If there are many kth-smallest operations executed on set S, performing all of them will take a large amount of total time. We might like to speed up the time to find the kth-smallest, at a slight increase in the time to insert an element.

Design an algorithm that (a) supports inserting an element into S in $O(\log n)$ time, and (b) supports finding the kth-smallest element of S in $O(\log n)$ time as well.
(Hint: Consider modifying a balanced binary search tree data structure.)

18/20

Ruben Tequida
Q.4

(4) **(Longest increasing subsequence)** (20 points) Given a string S of numbers, an *increasing subsequence* of S is any subsequence T of S such that the numbers in T, read left-to-right across T, are strictly increasing. For example, if $S = (3, 1, 6, 2, 5, 4)$, an increasing subsequence of S is $T = (1, 2, 4)$.

Design a dynamic programming algorithm that finds the *longest* increasing subsequence of a string S of length n in $\Theta(n^2)$ time.

(Note: Do *not* solve this problem by reducing it to the longest common subsequence problem. Instead design a dynamic programming algorithm from first principles.)

6/6 1. There are two ways an optimal solution can begin
Select the last element or not select it

Case 1 A[0:i-1] < A[0:i] where i is the length of the array we want to solve up to. If A[i] is larger than the largest value in the subproblem A[0:i-1] then we include A[i]

$$A = [\underbrace{\underbrace{\overset{0}{*}, *}_{\text{solution to subroblem } A[0:i-1]}, \overset{i}{*}}_{\text{A[i] is included in the solution if it is greater in value than the greatest selected value for } A[0:i-1]} \ldots *, \overset{n}{*}]$$

Case 2 A[0:i-1] > A[0:i] so A[i] isn't part of the solution so we don't count it in this subarray

8/10 2. Next we want to develop a recurrence equation for when A[i] is part of the solution and when it is not. To do this we would need to calculate the solutions to a subarray ending in i-1

We would create a new array to hold the max runs for each subproblem with array length n where n = A.length

max_runs = new Array[n] , max_val = 0,

max_runs[1, n] := 0 (Initialize all elements to 0)

Your structure should not depends on evaluation order.

$$\underset{0 \le i \le n}{\text{max\_runs}[i]} := \begin{cases} \underset{0 \le j \le i}{\text{max\_val}} := \begin{cases} \max(\text{max\_runs}[j], \text{max\_runs}[i]) \text{ if } A[i] > A[j] & , j < i \\ \text{max\_val} + 1 & , j == i \end{cases} & , i \ge 1 \\ 1 & , i == 0 \end{cases}$$

2/2 3. We want to compute the max_runs for each subproblem as i goes from 0 to n so we can see which pairs of numbers will increase the max-val total (indicating an increase in the longest increasing subsequence) or keep it the same (indicating no change to the longest increasing subsequence)

Ruben Tequida

Q.4

```
fillMaxRuns(A, max-runs) begin
    if A.length != 0 do
        max-run[0] = 1
        for i := 1 to n-1 do
            max-val = 0
            for j := 0 to i-1 do
                if A[j] < A[i] do
                    max-val = max(max-val, A[j])
                end
            end
            max-run[i] = max-val + 1
        end
        return max-runs
    end
end
```

2/2 4. In order to recover the sequence of numbers that formed the longest subsequence of increasing numbers we first find the largest element in max-runs and record its index. Then, we will go from that index to the left in the array to find the very next element that has a value of exactly 1 less than the max. Continue to do so until you hit the element with a value of 1.

```
find-seq(A, max-runs) begin
    cur-max := 0
    n = max-runs.length
    for i := n-1 down to 0 do
        if max-runs[i] > cur-max do
            cur-max = max-runs[i]
            max-index = i
        end
    end
    seq = new Array
    seq.append(A[max-index])
    for x := cur-max-1 down to 1 do
        max-index -= 1
        for y := max-index down to 0 do
            if max-runs[y] == x do
                seq.append(A[y])
                break
            end
        end
    end
    return seq.reverse
```

Done in order to return list in increasing order rather than decreasing order.

Ruben Tequilda

Q.4

We know this works because we are starting with an optimal solution where the next element in the sequence will be included in the solution or not. We then find the optimal solution for the minimum subproblem of an array with two elements and keeping track of how many numbers are smaller than A[i] for every element of A. Then, we find the total number of increasing number sequences found for the next element in A by using the solutions found for the previous subproblem. Once our new array holds all the total lengths of increasing numbers we can recover the original numbers that composed the longest subsequence by finding the max element in max-runs and then going to the left and finding the next consecutive number less than the current max.

~~Time analysis.~~

fillMaxRuns

```
for i := 1 to n-1 do
    max_val = 0
    for j := 0 to i-1 do
        if A[j] < A[i] do
            max_val = max(max_val, A[j])
        end
    end
    max_run[i] = max_val + 1
end
```

$\Theta(1)$

$\sum_{0 \le j \le i-1} \Theta(1) = \Theta(i-1-0+1)$

$= \Theta(i)$

$$\sum_{1 \le i \le n-1} \Theta(i) = \Theta \sum_{1 \le i \le n-1} i = \Theta\left(\frac{n^{1+1}}{1+1} + \Theta(n^1)\right) = \Theta\left(\frac{n^2}{2} + \cancel{\Theta}(n)\right) = \Theta(n^2)$$

Ruben Tequida
Q.4

FindSeq

```
for i := n-1 downto 0 do
    if max_runs[i] > cur_max do
        cur_max = max_runs[i]
        max_index = i
    end
end

for x := cur_max-1 to 1 do
    max_index -= 1
    for y := max_index downto 0 do
        if max_runs[y] == x do
            seq.append(A[y])
            break
        end
    end
end
```

$$\Theta(1) \qquad \sum_{0\le i\le n-1}\Theta(1) = \Theta(n-1+0+1) = \Theta(n)$$

$$\Theta(1) \qquad \sum_{0\le y\le max\_index}\Theta(1) = \Theta(max\_index - 0 + 1) = \Theta(max\_index)$$

$$\sum_{1\le x\le \underbrace{cur\_max}} \Theta(max\_index) \rightarrow \sum_{1\le x\le n}\Theta(n) = \Theta\left(n\sum_{1\le x\le n}1\right) = \Theta(n(n-1+1)) = \Theta(n^2)$$

if the array were in sorted order cur_max would equal n so we can upper bound cur_max to n. In this case max_index would be at n-1 so we can upper bound max_index to n as well

$$T(n) = \underbrace{\Theta(n^2)}_{\text{from FillMaxRuns}} + \underbrace{\Theta(n) + \Theta(n^2)}_{\text{from FindSeq}} = \Theta(n^2)$$

Ruben Tequida

Q.5

(5) **(Editing strings)** (30 points) Given two strings $A[1:m]$ and $B[1:n]$, the *edit distance* between A and B is the minimum cost of a script that edits A into B. A script is a series of edit operations, each edit operation has a non-negative cost, and the cost of a script is the sum of the costs of its operations.

27/30

The allowed edit operations in a script are:

- *copy*, which leaves a character unchanged, and has cost 0,
- *substitute*, which replaces a character a with another character b, and has cost c_{sub},
- *insert*, which adds a character a into a string, and has cost c_{ins},
- *delete*, which removes a character a from a string, and has cost c_{del}, and
- *transpose*, which replaces two adjacent characters ab in a string by the characters ba, and has cost c_{tra}.

Design a dynamic programming algorithm to compute the edit distance between A and B and recover the corresponding edit script in $\Theta(mn)$ time. You may assume that an optimal script never edits a given character more than once. The costs of operations are part of the input to your algorithm.

(Hint: Order the operations in an edit script so they occur left-to-right across string A, and then examine the possible ways in which an optimal script could end.)

1) The optimal solution will consist of selecting between 5 options for each position in A.

8/9.

Case 1 $A_n == B_n$ which will result in a copy ✓

$$A = A_1 A_2 A_3 \cdots A_{n-1} \boxed{A_n}$$

$$B = \underbrace{\underbrace{B_1 B_2 B_3 \cdots B_{n-1}}_{\text{find optimal solution to subproblem}} \boxed{B_n}}_{\text{current subproblem optimized by a copy since } A_n = B_n} \quad > =$$

Case 2 $A_{n-1} == B_n$ will result in a transpose

$$A = A_1 A_2 A_3 \cdots \overset{\longleftrightarrow}{\boxed{A_{n-1}} A_n}$$

$$B = \underbrace{B_1 B_2 B_3 \cdots B_{n-1}}_{\text{optimal solution to subproblem where } A_n \text{ and } A_{n-1} \text{ are transposed}} \boxed{B_n} \quad > =$$

Case 3 $A_n \,!= B_n$ and $A_{n-1} \,!= B_n$

subcase 1 $|A| > |B|$ then delete

$$A = A_1 A_2 A_3 \cdots \boxed{A_{n-1}} \boxed{A_n} \qquad m < n$$

$$B = B_1 B_2 B_3 \cdots B_{m-1} \boxed{\boxed{B_m}} \qquad > \,!= \quad > \,!=$$

delete A_n

Ruben Tequida

Q.5

subcase 2 $|A| < |B|$ then insert

$A = A_1 A_2 A_3 \cdots \boxed{A_{n-1}} \boxed{A_n}$ $\quad m > n$

$B = B_1 B_2 B_3 \cdots B_{m-1} \boxed{\boxed{B_m}}$

$>! =$

$>! =$

insert letter after A_n

subcase 3 $|A| == |B|$ then substitute

$A = A_1 A_2 A_3 \cdots \boxed{A_{n-1}} \boxed{A_n}$ $\quad m == n$

$B = B_1 B_2 B_3 \cdots B_{m-1} \boxed{\boxed{B_m}}$

$>! =$

$>! =$

change A_n to B_m

2) Next we will develop a recurrence equation for filling out our table costs[|B|, |A|]. Our table will hold every letter of A in the columns and every letter of B in the rows. Then, we can compare every sub string of A with every substring of B to see what the minimum cost of transforming A[1, n-1] to B[1, m-1] so we can use that to develop an optimal solution for the entire problem.

13/15

$$\text{dist}[i,j] := \min \left\{ \begin{array}{lll} \text{dist}(i, j-1) + C_{DEL} & & , i \geq 0, j \geq 1 \\ \text{dist}(i-1, j) + C_{INS} & & , i \geq 1, j \geq 0 \\ \text{dist}(i-1, j-1) + \left\{ \begin{array}{lll} 0 & \text{if} & A[j] = B[i] \\ C_{TRA} & \text{if} & A[j-1] = B[i] \\ C_{SUB} & \text{else} & \end{array} \right\} & & , i \geq 1, j \geq 1 \\ 0 \quad d[i-2, i-2] & & \checkmark \; , i == 0 \text{ and } j == 0 \end{array} \right]$$

3) In order to evaluate which combination of operations would result in the minimum cost we construct a 2D array that will hold the cost of previous subproblems in order to solve the current problem.

3/3

- create a 2D array $\quad$ dist = [|B|+1, |A|+1]

A = AB
B = BC

Ruben Tequida

Q.5

	'' ''	A	B
'' ''	0	C_{DEL}	C_{DEL}
B	C_{INS}	C_{SUB}	$\min\begin{pmatrix} C_{DEL} \\ C_{SUB} \end{pmatrix}$
C	C_{INS}	$\min\begin{pmatrix} C_{SUB} \\ C_{INS} \end{pmatrix}$	$\min\begin{pmatrix} C_{SUB} \\ \min\begin{pmatrix} C_{DEL} \\ C_{SUB} \end{pmatrix} \\ \min\begin{pmatrix} C_{SUB} \\ C_{INS} \end{pmatrix} \end{pmatrix}$

From the bottom-right corner we have the total cost of transforming String A into string B

4) Once we have the dist table filled out we can recover which operations produced the minimum cost by looking at which operation was chosen. Since we do not have actual values for the costs of our operations we are unable to calculate which operations would be specifically choosen. However, we can create an algorithm that tells us which path to take to recover.

3/3

If we hit an insert we go up a row dist(i-1, j)

If we hit a delete we go left a column dist(i, j-1)

If we hit a copy
If we hit a substitute } we go diagonal to the top-left dist(i-1, j-1)
If we hit a translate

Time Analysis

Evaluation Portion:

In order to fill in a table with |B| rows and |A| columns one cell at a time we need to take O(|B|·|A|) time which would be O(nm).

Recovery Portion:

While recovering we only do a number of computations equal to the number of operations done on A to get it to resemble B. The number of operations would be equal to the length of the longest string so this part would take O(max(m,n)) time

$$T(n) = O(nm) + \underbrace{O(\max(m,n))}_{\text{this will always be less than } O(mn)}$$

$$T(n) = O(mn)$$

18/20

Ruben Tejeida

Q.6

(6) **(Discrete knapsack)** (20 points) In the *discrete knapsack problem*, the input is a collection of n items with associated weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, and a capacity k. Item i has weight w_i and value v_i. All weights w_i and the capacity k are *positive integers*. The output is a subset S of the items $\{1, 2, \ldots, n\}$, called a knapsack, such that the total weight of all the items in knapsack S is at most k, and the total value of all the items in S is maximum. In other words, a solution to the discrete knapsack problem is an optimal knapsack S of items that does not exceed the weight capacity k while having the greatest possible value.

Design a dynamic programming algorithm that solves the discrete knapsack problem in $\Theta(nk)$ time.

(Hint: Examine the items in the order $1, 2, \ldots, n$, and consider knapsacks of all possible capacities.)

1. The optimal solution will consist of choosing 1 of 2 options:

5/6

first describe how problem ends. Then, remove last element in sol. There will be a sol. to a subp. Then discuss relationship between 2 sol. s.

Case 1: The last item is included in the solution
If the last item A[n] is part of the solution then we must find the optimal solution to the subproblem with items A[1 : n-1] and a capacity k' = k - A[n]. weight

$$A\ [\underbrace{*, *, *, \cdots, *, *}_{\text{Find the optimal solution with item } A[1:n-1] \text{ and a max capacity decreased by the weight of the last item}}, *]$$

↳ item will be included in optimal solution

Case 2: Item A[n] is not included in the optimal solution

$$A\ [\underbrace{*, *, *, \cdots, *, *}_{\text{Find the optimal solution with item } A[1:n-1] \text{ with the same max capacity as before}}, *]$$

↳ item will not be included in optimal solution

9 ~~10~~/10

2. Next we want to develop a recurrence equation for our table KS [n : k] which holds the max values for a given weight upto k and containing any possible items upto n.

To do this we need to calculate KS [n-1 : k] which would be the subproblem not including n and having a weight of k or k - A[n]. weight.

$$KS[i : j] := \left\{ \begin{array}{ll} \max((A[i].\text{value} + KS[i-1 : j - A[i].\text{weight}), KS[i-1 : j]) & , i \geq 1, j \geq 1 \\ \qquad\qquad \text{if } A[i].\text{weight} \leq k & \\ KS[i-1 : j] \quad \text{if } A[i].\text{weight} > k & , i \geq 1, j \geq 1 \\ 0 & , i = 0 \text{ or } j = 0 \end{array} \right\}$$

derivation

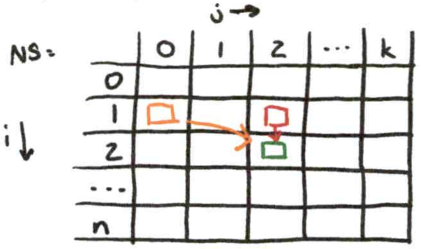
Ruben Tequida
Q. 6

3. We want to compute the maximum value we can hold in our knapsack for every possible weight capacity up to k.

2/2

```
fill NS (A, k) begin
    n = A.length
    KS = new Array(n+1, k+1)
    for i := 0 to n+1 do
        KS[i:0] := 0
    end
    for j := 1 to k+1 do
        KS[0:j] := 0
    end
    for i := 1 to n+1 do
        for j := 1 to k+1 do
            if A[i].weight ≤ k do
                KS[i:j] = max((KS[i-1:j-A[i].weight]+A[i].value), KS[i-1:j])
            end
            else do
                KS[i:j] = KS[i-1:j]
            end
        end
    end
    return KS
end
```

This will create a table that holds the max value for every combination of items up to a desired i and for a max capacity of j.



	0	1	2	...	k
0					
1	🟧		🟥		
2			🟩		
...					
n					

for i = 2 j = 2 🟩

we want to know if adding item 2 fits the max weight requirement and if it does if adding it will increase the value or be less than another combination of items that fit the weight

compare NS[i-1:j] 🟥 to A[i].value + KS[i-1:j-A[i].weight 🟧

choose the larger of the two

2/2 4. Once we have array KS completely filled out we know the max value that the knapsack can hold will be at KS[n:k]. Since this gives us the value and not the items that give us the value we can work from the bottom-right of the table to the top-left to find which items gave us that max. value.

Ruben Tequida

Q.6

```
findItems (A, NS) begin
    n := NS.length
    k := NS[0].length
    cur_val := NS[n:k]
    items = []
    while (n != 0) begin
        if NS[n-1:k] == cur_val do
            n = n-1
        end
        else do
            items.append(A[n])
            n = n-1
            k = k- A[n+1].weight
        end
    end
    return items
end
```

We know this works because we are using dynamic programming to calculate the max value we can hold in the knapsack for every possible weight from 0 – k and for all possible options of items from no items to all the items. When we are filling in the table we are computing the max value for a given weight with a number of possible items so when we add another item we can subtract that items weight from our current weight and look at the items (not including the one we just added) that give us the max-value for the new weight. So the bottom-right cell of the table will give us the max value which we can work backwards from in order to get the items that gave us that value.

Ruben Tequida
Q.6

Time Analysis

For fillNS

```
for i := 0 to n+1 do                 } Θ(n+1) = Θ(n)
    KS[i:0] := 0
end

for i := 1 to n+1 do
    for j := 1 to k+1 do
        if A[i].weight ≤ k do
            KS[i:j] = max((KS[i-1:j-A[i].weight]+A[i].value), KS[i-1:j])   } Θ(1)
        end
        else do
            KS[i:j] = KS[i-1:j]      } Θ(1)
        end
    end
end
```

$$\sum_{1\le j\le k+1} \Theta(1) = \Theta\left(\sum_{1\le j\le k+1} 1\right) = \Theta\left(k+1 \cancel{-1} \cancel{+1}\right) = \Theta(k+1) = \Theta(k)$$

$$\sum_{1\le i\le n+1} \Theta(k) = \Theta\left(\sum_{1\le i\le n+1} k\right) = \Theta\left(k \sum_{1\le i\le n+1} 1\right) = \Theta\left(k\,(n+1 \cancel{-1} \cancel{+1})\right)$$

$$= \Theta(kn+k) = \Theta(kn)$$

For findItems

```
while (n != 0) begin
    if NS[n-1:k] == cur_val do       } Θ(1)
        n = n-1
    end
    else do
        items.append(A[n])           } Θ(1)
        n = n-1
        k = k - A[n+1].weight
    end
end                                  } Θ(n+k)
```

$$T(n) = \Theta(n) + \Theta(nk) + \Theta(n+k) = \Theta(nk)$$