Steven Proctor
A01514516

CS 5050
11/10/17

#1 NUMBER OF FEASIBLE KNAPSACK SUBSETS

Create a dynamic programming algorithm to compute the number of feasible subsets for a given knapsack problem in $O(n * M)$ time.

PSEUDO CODE:

```
int feasibleKnapsackSubsets(int[] A, int M) {
    int[M] B = {1, 0, 0,  ...,  0}          // for M elements

    for (int i = 0; i < A.length; i++) {
        for (int k = M; k >= A[i]; k−−) {
                B[k] = B[k] + B[k − A[i]];
        }
    }

    return B[M];
}
```

DESCRIPTION:

This algorithm works nearly identically to the knapsack problem discussed in class. The only difference is that instead of saving a 1 or 0 to the array when the element fits in a knapsack of size k, it saves the sum of $B[k]$ and $B[k - A[i]]$. This way it tracks the number of ways a knapsack of size k can be filled. If only $B[k] > 0$ or $B[k - A[i]] > 0$ there are only $B[k]$ or $B[k - A[i]]$ subsets that can fit in a knapsack of size k. If both $B[k]$ and $B[k - A[i]]$ are greater than 0 then there are $B[k] + B[k - A[i]]$ number of subsets that will fit in a knapsack of size k.

ANALYSIS:

Because the algorithm is the same algorithm developed in class, it will take $O(n * M)$ time. There are M sizes of knapsacks, and each one is visited at most n times for each element of $A[n]$. The space in this case is $O(M)$ because all data is stored in the array $B[M]$

$$Time : O(n * M)$$

$$Space : O(M)$$

#2 MINIMIZED DIFFERENCE SUBSET SUM

Given an array $A[n]$ find a subset whose sum minimizes the difference between the subset's sum and a given value M in $O(n * K)$ time. Where K is the sum of all the elements in $A[n]$.

PSEUDO CODE:

```
List<int> minimizedDifferenceSubset(int[] A, int M) {
    int[sum(A)] B = {1, 0, 0,  ...,  0};            // for K elements
    for (int i = 0; i < A.length; i++)
        for (int k = sum(A); k >= A[i]; k−−)
                B[k] = Math.max(B[k], B[k − A[i]]);

    sum = B[M] == 1 ? M : findClosestSum(B, M);
    int i = A.length;
    List<int> C;
    while (sum > 0) {
        if (B[i] == 1 && B[k − A[i]) == 1) {
            C.add(A[i]);
            sum −= A[i];
        }
        i−−;
    }
    return C;
}

int findClosestSum(int[] B, int M) {
    int left , right;
    for (int i = M + 1; i < B.length; i++) {
        if (B[i] == 1) { right = i; break; }
    }
    for (int i = M − 1; i >= 0; i−−) {
        if (B[i] == 1) { left = i; break; }
    }
    return Math.min(Math.abs(M−left), Math.abs(M−right));
}
```

DESCRIPTION:

This algorithm uses the same method as the knapsack algorithm to determine all possible sums of the values contained in $A[n]$. $findClosestSum()$ locates the sum which is closest to the given value M. The values that make up the subset with the minimized difference are saved to a List $C[1 \leq j \leq n]$ and returned to the user.

ANALYSIS:

Finding the sum of all the elements in $A[n]$ takes $O(n)$ time. The for-loops iterate $O(n * K)$ times. $findClosestSum()$ takes at most $O(K)$ time. Saving the subset to $C[j]$ takes at most $O(n)$ time.

$$Time : O(n * K + 2 * n + K) = O(n * K)$$

$$Space : O(K + n)$$

#3 MAXIMED VALUE KNAPSACK SUBSET

Repeat the knapsack problem, looking for a subset whose total size is at most M. This time each item has a value associated with it. Find a subset that maximizes the value of the subset in $O(n*M)$ time without exceed a total size of M.

PSUEDO CODE:

```
int maximumValuedSubset(int[] A, int M) {
    int[M] B = {1, 0, 0,  ...,  0}          // for M elements

    for (int i = 0; i < A.length; i++) {
        for (int k = M; k >= A[i]; k−−) {
                B[k] = Math.max(B[k], (B[k − A[i]] + A[i].value));
        }
    }

    return maxNumInArray(B) − 1;
}
```

DESCRIPTION:

This algorithm is extremely similar to the first algorithm. It uses the same method to solve for viable subsets which don't exceed a size of M. However, the decision process for deciding which value to store has changed. In this algorithm the total value of the subset that has already been determined to fit a knapsack of size k ($B[k]$) is compared against the total value of a subset of size k which includes the value of the new number $A[i]$ ($B[k - A[i]] + A[i].value$). The value of the subset with the greatest value is stored at $B[k]$. The final array $B[k]$ contains the total value for every subset that does not exceed size M. The largest value in the array is the total value of the maximized value subset.

ANALYSIS:

Because the algorithm is nearly identical the algorithm in problem one, the time and space complexity is the same. This algorithm does have to scan through $B[k]$ one additional time to retrieve the maximum value, but this does not change the total running time of the algorithm.

$$Time : O((n + 1) * M) = O(n * M)$$

$$Space : O(M)$$

#4 LONGEST INCREASING SUBSEQUENCE

Given an Array $A[n]$ of random positive integers, design an algorithm that returns the length and elements of the longest monotonically increasing subsequence in $O(n^2)$ time.

PSUEDO CODE:

```
int[] longestIncreasingSubset(int[] A) {
    int[A.length] B = {1, 0, 0, ..., 0}          // for A.length elements
    for (int i = 0; i < A.length; i++) {
        for (int j = 0; j < i; j++) {
            if (A[i] > A[j] && B[i] < (B[j] + 1)) {
                B[i] = B[j] + 1;
            }
        }
    }

    int max = maxValue(B);
    int[max] C;
    for (int i = B.length − 1; i >= 0; i−−) {
        if (max <= 0) { break; }
        if (B[i] == max) {
            if (C[C.length] == 0) {
                C[max − 1] = A[i];
            } else if (A[i] < C[max]) {
                C[max − 1] = A[i]
            }
            max−−;
        }
    }
    return C;
}
```

DESCRIPTION:

This algorithm tracks the longest subsequence by checking to see if the current element is greater than all the elements before it. If the element is greater, the length of the previous element's longest subsequence plus one is stored at the current elements location in $B[n]$. This value gets overwritten if the current element can be added to a subsequence with a total length greater than that currently stored at that element's location in $B[n]$. Finally the longest subsequence is stored in $C[1 \leq k \leq n]$ by counting down from the maximum value (k) stored in $B[n]$ until the element containing 1 has been added to $C[k]$. The length of $C[k]$ is the length of the longest subsequence.

ANALYSIS:

In order the find the length of the longest subarray, a nested for-loop with a time of $O(n^2)$ is used. The max value of $B[n]$ must be found, which takes $O(n)$ time. Lastly, the longest subsequence must be saved to $C[k]$ with a worst case time of $O(n)$.

$$Time : O(n^2 + 2 * n) = O(n^2)$$

$$Space : O(n + k)$$