

Solutions for Homework Assignment #3

**Answer to Question 1.**

For all  $i \in [1..n]$ , define

$M[i]$  = maximum number that is the sum of a contiguous subsequence of  $S$  that ends in position  $i$

We claim that the following is a recursive formula to compute  $M[i]$  for all  $i \in [1..n]$ .

$$M[i] = \begin{cases} S[1], & \text{if } i = 1 \\ \max(S[i], S[i] + M[i - 1]), & \text{if } i > 1 \end{cases}$$

This is obvious for  $i = 1$ . For  $i > 1$ , let  $x_i$  be the contiguous subsequence of  $S$  that ends in position  $i$  and has the maximum possible sum among all such subsequences. Thus,  $x_i = yS[i]$ , where  $y$  is the (possibly empty) prefix of  $x_i$  up to and excluding the last element,  $S[i]$ . There are two possibilities.

CASE 1.  $y$  is empty. Then  $x_i = S[i]$  and obviously  $M[i] = S[i]$ .

CASE 2.  $y$  is nonempty. Therefore,  $y$  is a contiguous subsequence of  $S$  that ends in position  $i-1$  (otherwise  $x_i$  would not be a contiguous subsequence of  $S$  that ends in position  $i$ ). Furthermore,  $y$  has the maximum sum among all such subsequences of  $S$ ; for if there was a contiguous subsequence  $y'$  of  $S$  that ends in position  $i-1$  and has sum strictly greater than  $y$ , then  $y'S[i]$  would be a contiguous subsequence of  $S$  that ends in position  $i$  and has sum strictly greater than  $x_i$ , contrary to the definition of  $x_i$ . So,  $y = x_{i-1}$  and therefore  $x_i = x_{i-1}S[i]$ . Thus, in this case  $M[i] = M[i-1] + S[i]$ .

Note that, by definition, the quantity we are interested in (the maximum sum of all elements of a contiguous subsequence of  $S$ ) is simply  $\max\{M[i] : i \in [1..n]\}$ . Therefore, our dynamic programming algorithm computes  $M[1], M[2], \dots, M[n]$ , in this order, using the above recursive formula, and returns the maximum of these values.

Expressed in pseudocode, the algorithm is as follows ( $m$  keeps track of the maximum sum of contiguous subsequences of  $S$  ending in positions examined so far).

```
M[1] := S[1]
m := M[1]
for i := 2 to n do
    M[i] := max(S[i], M[i - 1] + S[i])
    m := max(m, M[i])
return m
```

The correctness of the algorithm follows by the previous argument regarding the recursive formula for  $M[i]$  given above. The running time of the algorithm is obviously  $\Theta(n)$ .

## Answer to Question 2.

I noticed, albeit too late to do anything about it, that in “rewording” the question as stated in DPV, I inadvertently changed the question itself. I intended to ask for a longest palindrome that is a **subsequence** of  $S$ , not a **substring** of  $S$ . (The difference is that a substring is a **contiguous** subsequence.) The hint was misleading for the question I actually asked, which may have led some of you to read the question as (correctly) stated in DPV and hence solve the correct problem. However, I can only expect you to answer the questions I ask, and not the questions I meant to ask! The only reasonable way that I can think of to deal with this situation is to ignore this question and mark the assignment out of 40 (instead of 50) points. I apologize for this error, and the confusion I have caused.

What follows is a solution to the question I should have asked: Finding in quadratic time a maximum length subsequence that is a palidrome.

a. For any  $i, j \in [1..n]$  such that  $i \leq j$ , define

$$P[i, j] = \text{maximum length of a palidrome that is a subsequence of } S[i..j]$$

The quantity we wish to compute, i.e., the maximum length of a palidrome that is a subsequence of  $S$ , is simply  $P[1, n]$ .

We claim that the following is a recursive formula to compute  $P[i, j]$  for all  $i, j \in [1..n]$  such that  $i \leq j$ .

$$P[i, j] = \begin{cases} 1, & \text{if } i = j \\ \max(P[i, j-1], P[i+1, j]), & \text{if } i < j \text{ and } S[i] \neq S[j] \\ 2, & \text{if } i = j-1 \text{ and } S[i] = S[j] \\ 2 + P[i+1, j-2], & \text{if } i < j-1 \text{ and } S[i] = S[j] \end{cases}$$

For  $i = j$  this is obvious since, in that case,  $S[i..j]$  consists of a single character, so the entire  $S[i..j]$  is a palidrome. So, suppose that  $i < j$ , and let  $x$  be a subsequence of  $S[i..j]$  that is a palidrome and has maximum length among all such subsequences. There are three possibilities.

CASE 1.  $S[i] \neq S[j]$ . Therefore  $x$  is either a subsequence of  $S[i..j-1]$  or of  $S[i+1..j]$  (otherwise  $x$  would start with  $S[i]$  and end with  $S[j]$  and so it would not be a palidrome). Furthermore,  $x$  is the longest such subsequence (otherwise there would be an even longer subsequence of  $S[i..j]$  than  $x$  that is a palidrome, contrary to the definition of  $x$ ). So, in this case,  $P[i, j] = |x| = \max(P[i, j-1], P[i+1, j])$ .

CASE 2.  $S[i] = S[j]$  and  $i = j-1$ . In this case,  $S[i..j]$  consists of two identical characters, and so the entire  $S[i..j]$  is a palidrome of length 2. Therefore, in this case,  $P[i, j] = |x| = |S[i..j]| = 2$ .

CASE 3.  $S[i] = S[j]$  and  $i < j-1$ . Note that, in this case,  $i+1 \leq j-1$  and so  $S[i+1..j-1]$  is a nonempty sequence. So,  $x = S[i]yS[j]$ , where  $y$  is a subsequence of  $S[i+1..j-1]$  that is a palidrome and has maximum length among all such subsequences (otherwise, we could find an even longer subsequence of  $S[i..j]$  than  $x$  that is a palidrome, contrary to the definition of  $x$ ). Therefore  $P[i, j] = |x| = 2 + |y| = 2 + P[i+1, j-2]$ .

As noted above, the quantity we are interested in (the maximum length of a subsequence of  $S$  that is a palidrome) is simply  $P[1, n]$ . Therefore, our dynamic programming algorithm computes  $P[i, j]$  for all pairs  $i, j \in [1..n]$  such that  $i \leq j$ , in order of increasing  $j-i$ , using the above recursive formula, and then returns  $P[1, n]$ .

Expressed in pseudocode, the algorithm is as follows:

```

for  $i := 1$  to  $n$  do  $P[i, i] := 1$ 
for  $d := 1$  to  $n - 1$  do
  for  $i := 1$  to  $n - d$  do
     $j := i + d$ 
    if  $S[i] \neq S[j]$  then
      if  $P[i, j - 1] \geq P[i + 1, j]$  then  $P[i, j] := P[i, j - 1]$ 
      else  $P[i, j] := P[i + 1, j]$ 
    else
      if  $i = j - 1$  then  $P[i, j] := 2$ 
      else  $P[i, j] := 2 + P[i + 1, j - 1]$ 
return  $P[1, n]$ 

```

The correctness of the algorithm follows by the previous argument regarding the recursive formula for  $P[i, j]$  given above. The running time of the algorithm is obviously  $\Theta(n^2)$ .

**b.** We also compute  $Q[i, j]$ , a longest subsequence of  $S[i..j]$  that is a palindrome, at the same time that we compute  $P[i, j]$ , the length of this subsequence. This is done as follows, where **CONCAT** is a function that returns the concatenation of two sequences.

```

for  $i := 1$  to  $n$  do  $P[i, i] := 1; Q[i, i] = S[i]$ 
for  $d := 1$  to  $n - 1$  do
  for  $i := 1$  to  $n - d$  do
     $j := i + d$ 
    if  $S[i] \neq S[j]$  then
      if  $P[i, j - 1] \geq P[i + 1, j]$  then  $P[i, j] := P[i, j - 1]; Q[i, j] := Q[i, j - 1]$ 
      else  $P[i, j] := P[i + 1, j]; Q[i, j] := Q[i + 1, j]$ 
    else
      if  $i = j - 1$  then  $P[i, j] := 2; Q[i, j] := S[i, j]$ 
      else  $P[i, j] := 2 + P[i + 1, j - 1]; Q[i, j] := \text{CONCAT}(S[i], \text{CONCAT}(Q[i + 1, j - 1], S[j]))$ 
return  $Q[1, n]$ 

```

### Answer to Question 3.

The inputs are values of the bit array  $A[1..n, 1..n]$ .

For a clot  $(i_1, i_2, j_1, j_2)$ , define the lower-right corner (LRC) to be  $(i_2, j_2)$  and define the size of the clot to be  $i_2 - i_1 + 1$  (which equals  $j_2 - j_1 + 1$ ).

For  $1 \leq i, j \leq n$  define  $C(i, j)$  to be the maximum possible size of a clot amongst all clots whose LRC is  $(i, j)$ .  $\max\{C(i, j) : 1 \leq i, j \leq n\}$  is the size of the largest clot.

We now give (and justify) a recursive formula to compute  $C(i, j)$  Let  $1 \leq i, j \leq n$ .

CASE 1.  $i = 1$  or  $j = 1$ . Then  $C(i, j) = A(i, j)$ .

CASE 2.  $i > 1, j > 1$ , and  $A(i, j) = 0$ . Then clearly  $C(i, j) = 0$ .

CASE 3.  $i > 1, j > 1$ , and  $A(i, j) = 1$ . Let  $m = \min(C(i - 1, j), C(i, j - 1))$ . (The figure on the next page may help clarify the discussion in this case.)

First, note that  $C(i, j) \leq m + 1$ . For, if  $C(i, j) \geq m + 2$ , then the entire  $(m + 2) \times (m + 2)$  square whose LRC is  $(i, j)$  would consist entirely of 1s; therefore, the two  $(m + 1) \times (m + 1)$  squares whose LRCs are  $(i - 1, j)$  and  $(i, j - 1)$  would both consist entirely of 1s. Thus,  $\min(C(i - 1, j), C(i, j - 1)) \geq m + 1$ , contradicting the definition of  $m$ .

Next, we show that  $C(i, j) \geq m$ . The  $(m + 1) \times (m + 1)$  square with LRC  $(i, j)$  consists entirely of 1s, with the possible exception of  $A(i - m, j - m)$ . This is because, by definition of  $m$ , the two  $m \times m$  squares with LRCs  $(i - 1, j)$  and  $(i, j - 1)$  consist entirely of 1s. Together with the bit  $A(i, j)$  which is 1 (by the hypothesis of the case), this covers the entire  $(m + 1) \times (m + 1)$  square with LRC  $(i, j)$ , with the exception of  $A(i - m, j - m)$ . So, in this case, if  $A(i - m, j - m) = 1$  then  $C(i, j) = m + 1$ ; otherwise,  $C(i, j) = m$ .

Based on the preceding discussion, the algorithm below computes the values of  $C$ .

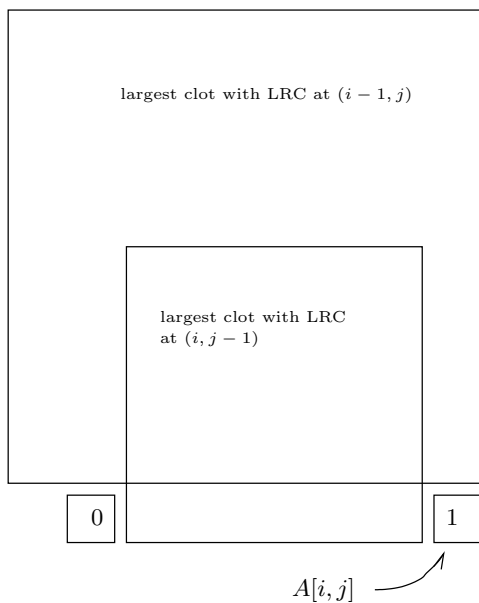
```

for  $i := 1$  to  $n$  do  $C(i, 1) := A(i, 1)$ 
for  $j := 1$  to  $n$  do  $C(1, j) := A(1, j)$ 
for  $i := 2$  to  $n$  do
  for  $j := 2$  to  $n$  do
    if  $A(i, j) = 0$  then
       $C(i, j) := 0$ 
    else
       $m := \min(C(i - 1, j), C(i, j - 1))$ 
      if  $C(i - m, j - m) = 0$  then  $C(i, j) := m$ 
      else  $C(i, j) := m + 1$ 

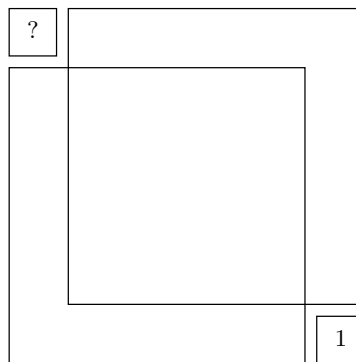
```

This algorithm takes time  $O(n^2)$ .

Now to compute an optimal clot, we go through the array  $C$  to find an  $(i, j)$  such that  $C(i, j)$  is maximum; say that  $C(i, j) = m$ . Then we return  $(i - m + 1, i, j - m + 1, j)$ . This step takes time  $O(n^2)$ , so the total time for the algorithm is  $O(n^2)$ .



Largest clots with LRC above and to the left of  $(i, j)$  have different sizes



Largest clots with LRC above and to the left of  $(i, j)$  have the same size

**Answer to Question 4.** The inputs are positive integers  $w_1, w_2, \dots, w_n, W, \Delta$ .

Let  $S \subseteq \{1, \dots, n\}$ . We define the *value* of  $S$  to be  $\sum_{i \in S} w_i$ . We say that a set  $S \subseteq \{1, \dots, n\}$  is *w-legal* if its value is at most  $w$ , and for all  $i, j \in S$  such that  $i \neq j$ ,  $|w_i - w_j| \geq \Delta$ . For  $0 \leq i \leq n$  and  $0 \leq w \leq W$ , let  $S(i, w)$  be a  $w$ -legal subset of  $\{1, \dots, i\}$  of maximum value. The set we are seeking is  $S(n, W)$ . Let  $M(i, w)$  be the value of  $S(i, w)$ .

We now give (and justify) a recursive formula to compute  $M(i, w)$ . To do so, it is convenient to first sort the first  $n$  inputs so that  $w_1 \leq w_2 \leq \dots \leq w_n$ , and to define, for each  $1 \leq i \leq n$ ,  $p(i) = \max\{j : j < i \text{ and } w_i - w_j \geq \Delta\}$ , where we take  $\max \emptyset = 0$ . (That is,  $p(i)$  is the largest index preceding  $i$  of an input that is “sufficiently far apart” from  $w_i$ .) Note that if  $i$  is in a  $w$ -legal set  $S$ ,  $p(i)$  is the largest index smaller than  $i$  that can be in  $S$ . Let  $0 \leq i \leq n$  and  $0 \leq w \leq W$ .

CASE 1.  $i = 0$ . Then clearly  $M(0, w) = 0$ .

CASE 2.  $i \geq 1$  and  $w_i > w$ . In this case,  $i \notin S(i, w)$  (since  $w_i$  by itself is not  $w$ -legal). Therefore,  $S(i, w) = S(i - 1, w)$ , and so  $M(i, w) = M(i - 1, w)$ .

CASE 3.  $i \geq 1$  and  $w_i \leq w$ . There are two possibilities, depending on whether  $i \in S(i, w)$ . If  $i \notin S(i, w)$  then, as in the previous case,  $S(i, w) = S(i - 1, w)$ , and so  $M(i, w) = M(i - 1, w)$ . If  $i \in S(i, w)$  then let  $R = S(i, w) - \{i\}$ . By the remark following the definition of  $p(i)$  above, we have that  $R \subseteq \{1, \dots, p(i)\}$ ; furthermore,  $R$  is  $(w - w_i)$ -legal, because  $\sum_{j \in R} w_j = \sum_{j \in S(i, w)} w_j - w_i \leq w - w_i$ . Thus,  $R = S(p(i), w - w_i)$  (otherwise, we could find a  $w$ -legal subset of  $\{1, \dots, i\}$  of value higher than that of  $S(i, w)$ ). Since  $S(i, w) = \{i\} \cup R$ , it follows that  $M(i, w) = w_i + M(p(i), w - w_i)$ .

Combining these two subcases, we have  $M(i, w) = \max(M(i - 1, w), w_i + M(p(i), w - w_i))$ .

From the above discussion, the algorithm below computes the values of array  $M$ .

```

sort  $w_1, \dots, w_n$ 
for  $i := 1$  to  $n$  do  $p(i) := \max\{j : j < i \text{ and } w_i - w_j \geq \Delta\}$ 
for  $w := 0$  to  $W$  do  $M(0, w) \leftarrow 0$ 
for  $i := 1$  to  $n$  do
  for  $w := 0$  to  $W$  do
    if  $w_i > w$  then  $M(i, w) := M(i - 1, w)$ 
    else  $M(i, w) := \max(M(i - 1, w), w_i + M(p(i), w - w_i))$ 

```

Sorting the  $n$  input numbers takes  $O(n \log n)$  time, and we can compute each  $p(i)$  in  $O(\log n)$  time using binary search. Thus, the first two steps take  $O(n \log n)$  time. The initialisation of  $M(0, w)$  takes  $O(W)$  time, and the double-nested for loop takes  $O(n \cdot W)$  time. Thus, the above algorithm runs in  $O(n(W + \log n))$  time.

For any  $i, w$  such that  $0 \leq i \leq n$  and  $0 \leq w \leq W$ , the procedure  $\text{FINDOPT}(i, w)$  below returns a set  $S(i, w)$ . Thus, to find an optimal set  $S$  as required, we call  $\text{FINDOPT}(n, W)$ .

```

FINDOPT( $i, w$ )
if  $i = 0$  then return  $\emptyset$ 
elseif  $M(i, w) = M(i - 1, w)$  then return FINDOPT( $i - 1, w$ )
else return FINDOPT( $p(i), w - w_i$ )  $\cup \{i\}$ 

```

A call to  $\text{FINDOPT}$  takes constant time plus the time for a recursive call with a smaller  $i$ . So the time for  $\text{FINDOPT}(n, W)$  is  $O(n)$ .

Therefore, the total time to find a maximum-value  $W$ -legal subset of  $\{1, \dots, n\}$  is  $O(n(W + \log n))$ .

**Answer to Question 5.** The inputs are positive integers  $w_1, v_1, \dots, w_n, v_n, W, W'$ .

Let  $S, S' \subseteq \{1, \dots, n\}$  and  $w, w'$  be non-negative integers. We say that  $(S, S')$  is a  $(w, w')$ -knapsack if  $S \cap S' = \emptyset$ ,  $\sum_{i \in S} w_i \leq w$ , and  $\sum_{i \in S'} w_i \leq w'$ . We call  $S$  and  $S'$  the *first* and *second knapsack* of  $(S, S')$ , respectively. The *value* of a  $(w, w')$ -knapsack  $S$  is defined to be  $\sum_{i \in S} v_i + \sum_{i \in S'} v_i$ .

For any  $i, w, w'$  such that  $0 \leq i \leq n$ ,  $0 \leq w \leq W$ , and  $0 \leq w' \leq W'$ , let  $K(i, w, w')$  be a  $(w, w')$ -knapsack  $(S, S')$  of maximum value, where  $S$  and  $S'$  are subsets of  $\{1, \dots, i\}$ . We are seeking  $K(n, W, W')$ . Let  $V(i, w, w')$  be the value of  $K(i, w, w')$ .

We now give (and justify) a recursive formula to compute  $V(i, w, w')$ . Let  $0 \leq i \leq n$ ,  $0 \leq w \leq W$ , and  $0 \leq w' \leq W'$ .

CASE 1.  $i = 0$ . Then clearly  $V(0, t_1, t_2) = 0$ .

CASE 2.  $i \geq 1$ ,  $w_i > t_1$ , and  $w_i > t_2$ . Neither of the knapsacks in  $K(i, w, w')$  can contain  $i$  in this case. Thus,  $K(i, w, w') = K(i - 1, w, w')$ , and so  $V(i, w, w') = V(i - 1, w, w')$ .

CASE 3.  $i \geq 1$ ,  $w_i \leq w$ , and  $w_i > w'$ . Let  $K(i, w, w') = (S, S')$ . In this case,  $i$  can be in in the first knapsack but not in the second. There are two possibilities, depending on whether  $i \in S$ . If  $i \notin S$ , then  $K(i, w, w') = K(i - 1, w, w')$  and  $V(i, w, w') = V(i - 1, w, w')$ , as in the previous case. If  $i \in S$ , then let  $R = S - \{i\}$ . The set  $(R, S')$  is a  $(w - w_i, w')$ -knapsack of maximum value with elements from  $\{1, 2, \dots, i - 1\}$  (otherwise, we could find a  $(w, w')$ -knapsack with elements from  $\{1, 2, \dots, i\}$  of higher value than  $(S, S')$ ,

contradicting that  $(S, S') = K(i, w, w')$ . Therefore, in this subcase,  $V(i, w, w') = v_i + V(i - 1, w - w_i, w')$ . Combining the two subcases, we get  $V(i, w, w') = \max(V(i - 1, w, w'), v_i + V(i - 1, w - w_i, w'))$ .

CASE 4.  $i \geq 1$ ,  $w_i > w$  and  $w_i \leq w'$ . In this case,  $i$  can be in the second knapsack but not in the first. By an analysis similar to the preceding case, we get  $V(i, w, w') = \max(V(i - 1, w, w'), v_i + V(i - 1, w, w' - w_i))$ .

CASE 5.  $i \geq 1$ ,  $w_i \leq t_1$ , and  $w_i \leq t_2$ . In this case,  $i$  can be in either the first or the second knapsack (or in neither). By an analysis similar to the previous two cases we get  $V(i, w, w') = \max(V(i - 1, w, w'), v_i + V(i - 1, w - w_i, w'), v_i + V(i - 1, w, w' - w_i))$ .

The following algorithm computes the values of array  $V$ .

```

for  $w := 0$  to  $W$  do
  for  $w' := 0$  to  $W'$  do
     $V(0, w, w') := 0$ 
for  $i := 1$  to  $n$  do
  for  $w := 0$  to  $W$  do
    for  $w' := 0$  to  $W'$  do
      if  $w_i > w$  and  $w_i > w'$  then
         $V(i, w, w') := V(i - 1, w, w')$ 
      elseif  $w_i \leq w$  and  $w_i > w'$  then
         $V(i, w, w') := \max(V(i - 1, w, w'), v_i + V(i - 1, w - w_i, w'))$ 
      elseif  $w_i > w$  and  $w_i \leq w'$  then
         $V(i, w, w') := \max(V(i - 1, w, w'), v_i + V(i - 1, w, w' - w_i))$ 
      else
         $V(i, w, w') := \max(V(i - 1, w, w'), v_i + V(i - 1, w - w_i, w'), v_i + V(i - 1, w, w' - w_i))$ 

```

The running time of this algorithm is clearly  $O(n \cdot W \cdot W')$ .

For any  $i, w, w'$  such that  $0 \leq i \leq n$ ,  $0 \leq w \leq W$ , and  $0 \leq w' \leq W'$ , the procedure  $\text{FINDOPT}(i, w, w')$  below uses the computed values of array  $V$  to return  $S(i, w, w')$ . To obtain the optimal knapsack we seek, we call  $\text{FINDOPT}(n, W, W')$ .

```

 $\text{FINDOPT}(i, w, w')$ 
if  $i = 0$  then return  $\emptyset$ 
elseif  $V(i, w, w') = V(i - 1, w, w')$  then return  $\text{FINDOPT}(i - 1, w, w')$ 
elseif  $w_i \leq w$  and  $V(i, w, w') = v_i + V(i - 1, w - w_i, w')$  then
   $(S, S') := \text{FINDOPT}(i - 1, w - w_i, w')$ 
  return  $(S \cup \{i\}, S')$ 
else
   $(S, S') := \text{FINDOPT}(i - 1, w, w' - w_i)$ 
  return  $(S, S' \cup \{i\})$ 

```

A call to  $\text{FINDOPT}$  takes constant time plus the time for a recursive call with a smaller  $i$ . So the time for  $\text{FINDOPT}(n, W, W')$  is  $O(n)$ .

Therefore, the total time for the algorithm to find an optimal knapsack is  $O(n \cdot W \cdot W')$ .