**CSCI-GA.1170-001/002 Fundamental Algorithms** November 15, 2019

# Solutions to Problem 1 of Homework 7 ((23+5))

*Name: Name (NetID)* *Due: Thursday, November 5*

*Collaborators: NetID1, NetID2*

You are a student of a course "Not-So-Fundamental Algorithms". You have been diligently submitting the solutions to various homework questions and have been assigned scores for each of them. Let us assume that there are $n$ such questions and you are given the scores as an array $A[1 \dots n]$. These scores can be positive or negative. (The grading has been a bit harsh, probably done by the recitation leader). As a result, you are being allowed to pick the homework questions you want to count towards your homework component of the final grade. Clearly, your goal is to maximize the sum of the scores you pick. However, there are some additional conditions imposed on how you pick this subset.

(a) (1 point) Warm-up Question: You are not told about the values in the array $A$. You are just asked to pick any or none of the scores. You have to provide the indices $i_1, i_2, \dots i_k$ and the final score would be $\sum_{j=1}^{k} A[i_j]$. You may also choose to pick none of the scores. (When does this happen?) Give me a $\Theta(n)$ solution to return the set of indices so that you maximize the final score.

**Solution:**
Since we could take any $i_j$ out of $k$ from $n$ scores, to maximize the sum of $A[i_j]$, we need find as many as positive $A[i_j]$ possible. So $k$ is the number of positive scores from n. When $k$ is zero, it implicates that all n scores are negative. For the algorithm, since we only traverse the array once and sum all positive scores, the running time is $\Theta(n)$

```
Grade(A,n)
    final = 0
    for i=1 to n
        if (A[i] > 0)
            final = final + A[i]
    return final
```

□

For parts (b)-(e), you are constrained to pick only continuous scores, i.e pick a subarray from the array $A$. You have to provide two indices $i, j$ and the final score will be $\sum_{k=i}^{j} A[k]$.

(b) (1 point) You are told that $\forall i,\ A[i] \geq 0$. What will your strategy be? Give me a $\Theta(1)$ solution to return the two indices $i, j$. Justify your strategy briefly.

**Solution:**
Since all scores are positive, the maximized final score is the sum of all n scores.
Thus, the solution is $i = 1, j = n$. Because we only return those two numbers where $n$ from the input, the running time is constant $\Theta(1)$.

□

(c) (3 points) You have no information about the values in the array $A$. Fill in the blanks to complete the following iterative algorithm that returns the maximum possible score. It is easy to see that this algorithm will have a run-time of $O(n^2)$.

```
GetMVCS($A, n$)
    $maxsum = 0$
    **for** $i = 1$ **to** $n$
        $current = maxsum$
        **for** $j = i$ **to** $n$
            $current += A[j]$
            **if** $current > maxsum$ **then** $maxsum = current$
    **return** $maxsum$
```

**Solution:** INSERT YOUR SOLUTION HERE □

(d) (5 points) Consider the problem as defined in Part (c). Construct an $O(n)$ time and $O(n)$ space algorithm using dynamic programming. Justify the runtime of your algorithm. Briefly argue the correctness of your algorithm. The algorithm will utilize a one-dimensional array $M$ of length $n$. Clearly define $M[i]$ to help formulate the ***bottom-up*** algorithm.

(**Hint**: Note that you have a running window. What does it mean to extend this window by one?)

**Solution:**

```
Bottom-Up-score(A,n)
    let M[1,...,n] be new array
    M[1] = A[1]
    result = max(M[1],0)
    for j=2 to n
        M[j] = max(M[j-1],0)+A[j]
        result = max(result, M[j])
    return result
```

Since we only iterate array A once with constant cost comparisons and adding value at each step, the total cost is linear $O(n)$.
Also, we only have memory operation on array M which is a n length array. So the storage

cost is also $O(n)$.
For n=1 case, we choose the bigger one from A[1] and 0 as the result, which is the max score we can have.
Let we assume n=k case gives us the correct maximized score result and the previous positive sub-array sum from array M.
For n = k+1 case, if M[k] is positive, it implies the previous array has a positive sum which is benefit to the max score and we need to add A[k+1] on it to get the new M[k+1].
If M[k] is negative, it implies the A[k] is so small that makes the previous sum to be negative and we should not consider it. So we choose A[k+1] as M[k+1].
Then, we compare the current M[k+1] with result to check whether it is a new global max sum.

□

(e) (3 points) Typically, one uses a helper value to help reconstruct the optimal solution for a dynamic programming problem. However, for this question, one does not need to do that. Indeed, it is not hard to modify the problem from Part (d) to also print the optimal solution. For this question, you will construct an algorithm PRINT-SOLUTION($M$) that takes as input the array $M$ from before and prints the solution, i.e, give the starting and ending indices of the contiguous subarray that yields the maximum value. This will have to be a $O(n)$ algorithm.

**Solution:**

```
Print-Solution(M)
    j = 1
    result = 0
    for k=1 to n
        if (M[k]>result)
            result = M[k]
            j = k
    i = j
    while i>1 and M[i-1]>0
        i = i-1
    return i,j
```

Since we only iterate array M twice with constant cost operation, the running time is $O(n)$.

□

(f) (5 points) For this question, you are given a constant $k < n$. You have to choose $k$ scores such that you maximize your sum, i.e, find a subset of $A$ of size $k$ such that it has the maximum sum across all possible subsets of size $k$. These scores need *not be continuous.* Construct an $O(nk)$ space and time algorithm using dynamic programming to return the maximum sum.

Justify the runtime of your algorithm. Briefly argue the correctness of your algorithm. The algorithm will utilize a two-dimensional array $M$. You might find it helpful to index this array starting at 0. Clearly define $M[i][j]$ to help formulate the *bottom-up* algorithm.

**Solution:**

□

(g) (5 points) For this question, you are constrained to pick your scores in such a way that you *you cannot pick two adjacent scores* i.e, if index $i$ is picked, indices $i+1$ and $i-1$ cannot be picked. *You need to pick at least one score.* Construct an $O(n)$ time and $O(n)$ space algorithm using dynamic programming. Justify the runtime of your algorithm. Briefly argue the correctness of your algorithm. The algorithm will utilize a one-dimensional array $M$ of length $n$. It might be useful to define $M[0]$ as a base case. Clearly define $M[i]$ to help formulate the ***top-down*** algorithm.

**Solution:**

```
def GetMS(A,n)
    let M[0,...,n] be new array
    M[0] = 0
    GetMS2dis(A,M,n)
    return M[n]
def GetMS2dis(A,M,n)
    if (n==1) M[1] = A[1]
    GetMS2dis(A,M,n-1)
    M[n] = max(A[n]+M[n-2], M[n-1])
```

The running time is T(n) = T(n-1)+O(1) with T(1) = O(1), so the running time is O(n)
For the memory cost, because we only operate on the n length array, the memory cost is O(n)
For n=1 base case, we can only choose this score as the max sum of score, so it is correct.
Let we assume that for 1,...,k the algorithm can give us correct max sum of score M[k].
For n = k+1, we have two possible choices such that not choosing A[k] or not choosing A[k+1]. So the max sum could be M[k-1]+A[k+1] or M[k]. Then, we choose the larger one which is the max sum of the array $A_{k+1}$.

□

(h) (**Extra Credit**)(5 points) For this question, you are constrained to pick your scores in such a way that you *you cannot pick three adjacent scores* i.e, if index $i$ is picked then you cannot pick both of the in the indices in the following pairs of indices $(i-1, i+1), (i-2, i-1), (i+1, i+2)$. *You need to pick at least one score.* Construct an $O(n)$ time and $O(n)$ space algorithm using dynamic programming. Justify the runtime of your algorithm. Briefly argue the correctness of your algorithm. The algorithm will utilize a one-dimensional array $M$ of length $n$. It might be easier to define $M[0]$ as a base case. Clearly define $M[i]$ to help formulate a ***top-down*** algorithm.

**Solution Sketch:** Define $M[i]$ to be the maximum sum possible from the subset of 1 to $i$ numbers without selecting two continuous numbers. Also, $M[0] = 0$

$$M[i] = \begin{cases} \max(M[i-3] + A[i] + A[i-1], A[i] + M[i-2], M[i-1]) & i > 2 \\ A[1] & i = 1 \\ \max(A[1], A[2]) & i = 2 \end{cases}$$

Note to Graders: If they give a bottom-up algorithm, give at most 2 points because the question asks for top-down algorithm. They also need to define $M$ clearly.

□

**Solution:**

```
def MS(A,n)
    let M[0,...,n] be new array
    M[0] = 0
    GetMS3dis(A,M,n)
    return M[n]
def GetMS3dis(A,M,n)
    if (n==1) {M[1] = A[1]}
    if (n==2) {M[2] = max(A[1], A[2], A[2]+A[1]); M[1] = A[1]}
    if (n>2)
        GetMS3dis(A,M,n-1)
        M[n] = max(M[n-3]+A[n]+A[n-1], A[n]+M[n-2], M[n-1])
```

The running time is T(n) = T(n-1)+O(1) with T(2)=T(1) = O(1) which is O(n).
For the memory cost, since we only operate on the array n length array M, the memory cost is O(n).
For n = 1 case, the algorithm has to choose this only score as max sum which is correct.
For n = 2 case, we choose the sum of these two numbers or the larger one of them where they are not both positive.
Let we assume that for n = k case, we have the correct max sum array M from 1 to k.
For n = k+1 case, we have three possible choices such that not choosing A[k-1], not choosing A[k] or not choosing A[k+1] from those 3 elements A[k-1], A[k], A[k+1]. So the max sum for these three choices are M[k-2]+A[k]+A[k+1] or M[k-1]+A[k+1] or M[k]. Thus, we choose the max value from these 3 choices as our max score.

□

**CSCI-GA.1170-001/002 Fundamental Algorithms** November 15, 2019

# Solutions to Problem 2 of Homework 7 (7+3 points)

*Name: Name (NetID)* *Due: Thursday, November 5*

*Collaborators: NetID1, NetID2*

Recall, Fibonacci number $F_0, F_1, F_2, ...$ are defined by setting $F_0 = 0,\ F_1 = 1$, and $F_i = F_{i-1}+F_{i-2}$, for $i \geq 2$. Ignoring the issue of $F_n$ being exponentially large, one can easily compute $F_n$ in linear time:

```
$F_0 = 0,\ F_1 = 1$
for $i = 2$ to $n$
    $F[i] := F[i-1] + F[i-2]$
```

(a) (1 point) Complete the blanks to provide a *recursive* algorithm that computes $F_n$.

```
FIB($n$):
    if n==1 return 0
    else if n==2 return 0+1
    else return FIB($n-1$)+FIB($n-2$)
```

**Solution:** INSERT YOUR SOLUTION HERE □

(b) (3 points) Let $T(n)$ be the running time of FIB($n$). What is the recurrence relation for the running time of this recursive procedure? Do not forget to write the base case(s) of your recurrence relation. You may assume that addition and the **if** statement can be accomplished in constant time.

Prove by induction that $T(n) \geq c^n$, for some constant $c > 1$. What is the largest value of $c$ that you can use in your induction?

**Solution:**
we have $T(n) = T(n-1) + T(n-2) + O(n)$ with $T(2) = T(1) = O(1)$.
To find the largest c, let we assume $T(n) \geq c^n$ for $n = k$.
For $n = k+1$, we have $T(k+1) \geq c^k + c^{k-1} + O(1)$. To get $T(k+1) \geq c^{k+1}$, $c^k + c^{k-1} + O(1) \geq c^{k+1}$. Because $c^k$ is much larger than constant cost, so $c - 1 - c^{-1} \leq 0$. we have $\frac{1-\sqrt{5}}{2} \leq c \geq \frac{1+\sqrt{5}}{2}$. Thus, the largest value of c is $\frac{1+\sqrt{5}}{2}$.
For n = 1 and n=2 base case, we have $T(1) = T(2) = O(1)$ which is larger or equal than constant cost $c^1, c^2$.
Let we assume for 1,...,k case, $T(k) \geq c^k$.
For n = k+1 case, $T(k+1) \geq c^k + c^{k-1} + O(1)$. Since we have $1 < c \leq \frac{1+\sqrt{5}}{2}$, we know that $c^k + c^{k-1} + O(1) \geq c^{k+1}$.
Thus, $T(n) \geq c^n$.

□

(c) (3 points) You will now write a variant of the recursive procedure. Let us call it SMART-FIB which will compute $F_n$ in time $O(n)$, like we expect a good procedure should. You will use memoization to achieve the speed-up. More formally, you *will use* an array $A[1 \ldots n]$ to memoize.

**Solution:**

```
def SMART-FIB(A,n)
    if (n==0) {A[0] = 0; return A[1]}
    if (n==1) {A[1] = 0+1; A[1] = 0; return A[2]}
    if (n¿1)
        SMART-FIB(A,n-1)
        A[n] = A[n-1]+A[n-2]
        return A[n]
```

□

(d) **(Extra Credit)** (3 points) The above algorithm runs in $O(n)$ time but also uses $O(n)$ space. Provide an iterative variant of the above recursive procedure which will run in $O(n)$ time but will only use constant additional memory. Let us call it algorithm SUPER-SMART-FIB.

**Solution:**

```
def SUPER-SMART-FIB(n)
    pre1, pre2 = 1,0
    if (n==0) return pre2
    if (n==1) return pre1
    if (n¿1)
        for i=2 to n
            current = pre1+pre2
            pre2 = pre1
            pre1 = current
        return current
```

□

**CSCI-GA.1170-001/002 Fundamental Algorithms** November 15, 2019

# Solutions to Problem 3 of Homework 7 (8 points)

*Name: Name (NetID)* *Due: Thursday, November 5*

*Collaborators: NetID1, NetID2*

Let us recall the definition of *binary search tree property*: Let $x$ be a node in a binary search tree. If $y$ is a node in the left subtree of $x$, then $y.key \leq x.key$. If $y$ is a node in the right subtree of $x$, then $y.key \geq x.key$.

The goal of this question is to count the number of possible BSTs with $n$ distinct nodes. For simplicity you may assume that the possible labels are $[1 \ldots n]$.

For example, when $n = 1$, you know that the only possibility is just a root which is also the leaf. However, for $n = 2$, we have two choices for the root: We can make 2 the root and 1 its left child or make 1 the root and 2 its right child.

(a) (2 points) Illustrate all the possible BSTs for $n = 3$.

**Solution:**
Suppose $a_1 < a_2 < a_3$ Let $a_1$ be the root, we have $a_2$ being the right child or $a_3$ being the right child.
Let $a_2$ be the root, we have only $a_1$ being the left child and $a_3$ being the right child.
Let $a_3$ be the root, we have $a_1$ being the left child or $a_2$ being the left child.
Thus the number of total possible trees are 5.

□

There are totally 5 possible BSTs.

(b) (3 points) Let $F_n$ denote the number of possible BSTs with $n$ elements. How many trees are possible with $i$ as the root? Use this to write a recursive formulation for computing $F_n$. (**Hint**: If $i$ is the root, what are the possible elements that can occur in the left child and what about the ones on the right child?)

**Solution:**

```
def BSTN(n)
    if (n==1) return 1
    if (n==2) return 2
    if (n==3) return 5
    if n>3
        sum = 0
        for i = 1 to n-1
            sum+=BSTN(i)*BSTN(n-i)
        return sum
```

□

(c) (3 points) Use the above formulation to write a *recursive* algorithm (top-down) based on Dynamic Programming to compute the value of $F_n$. You may find it useful to declare a global array $M[1 \dots n]$. Assume that the array can store the large values of $F$.

**Solution:**

```
def BSTN(n)
    let M[1,...,n] be new array
    return BSTNDP(M,n)
def BSTNDP(M,n)
    if (n==1) {M[1]=1; return M[1]}
    if (n==2) {M[1]=1; M[2]=2; return M[2]}
    if (n==3) {M[1]=1; M[2]=2; M[3]=5; return M[3]}
    if (n>3)
        BSTNDP(M,n-1)
        for i=1 to n-1
            M[n]+=M[i]*M[n-i]
        return M[n]
```

□