CS 157: Assignment 4

Douglas R. Lanman
10 April 2006

Problem 3: Maximum Subarrays

This write-up presents the design and analysis of several algorithms for determining the maximum sum of certain subsets of one-dimensional arrays. In the first section, we consider the Maximum Subsequence Sum (MSS) problem: given an array A with signed integer elements, find a contiguous subarray with the maximum possible sum. In Section 2, we extend our algorithm to handle the case of cyclic shifts of the array elements. Finally, in Section 3 we consider the Maximum Non-consecutive Sum (MNCS) problem. For this task, we find a subset of array elements (none of which are contiguous) which achieve the maximum possible sum.

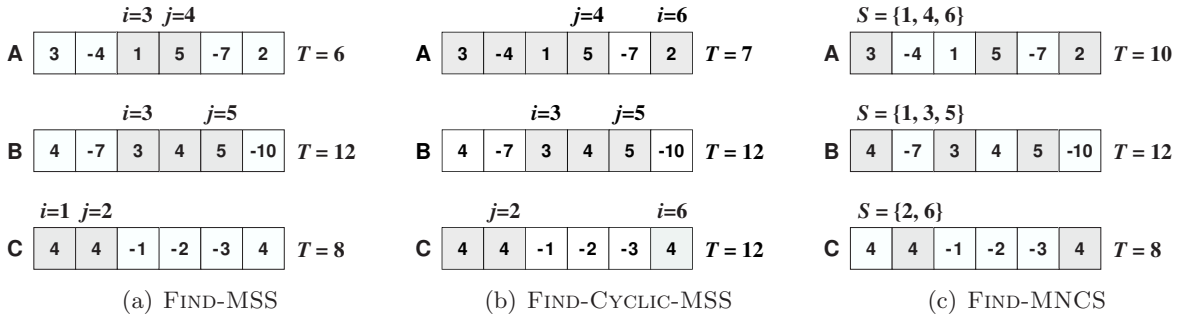
Part (a): Finding the Maximum Subsequence Sum

Given an array A of signed integers, design an algorithm that finds a subarray $A[i, \ldots, j]$ such that $A[i] + A[i+1] + \ldots + A[j]$ is maximum in time $O(n \log n)$. More specifically, find a pair (i, j) such that $\forall (x, y)\ \Sigma_{k=i}^{j} A[k] \geq \Sigma_{k'=x}^{y} A[k']$.

High-level Description: To obtain a running time of $O(n \log n)$, we can apply the divide-and-conquer method. Before we present the algorithm, note the following properties of the Maximum Subsequence Sum (MSS). First, if the array elements are all nonnegative, then the MSS is given by the sum from $i = 1$ to $j = n$. Second, if $A[i] < 0\ \forall i$, then the MSS can be defined as the element closest to zero. Note that, by definition, we disallow the empty set as a valid input (or solution to) the Maximum Subsequence Sum problem. In addition, note that the MSS is not necessarily unique; there could be more than one subarray which achieves the maximum sum.

```
FIND-MSS(A, p, r)
 1  if p = r
 2     then return {p, p, A[p]}
 3  q ← ⌊(p + r)/2⌋
 4  {i_l, j_l, T_l} ← FIND-MSS(A, p, q)
 5  {i_r, j_r, T_r} ← FIND-MSS(A, q + 1, r)
 6  {i_s, j_s, T_s} ← MAX-SPAN(A, p, q, r)
 7  T ← max(T_l, T_s, T_r)
 8  switch
 9     case T = T_l :
10           return {i_l, j_l, T}
11     case T = T_s :
12           return {i_s, j_s, T}
13     case T = T_r :
14           return {i_r, j_r, T}
```

To begin our analysis, let's consider FIND-MSS shown above. To initialize the recursion, we call FIND-MSS(A,1,length[A]). The output is a set of three elements $\{i, j, T\}$, where T is the Maximum Subsequence Sum and i and j are the starting and ending indices of the subarray, respectively. On lines 1-2 we handle the base case in which the input array is a single element. Lines 3-5 implement



(a) Find-MSS (b) Find-Cyclic-MSS (c) Find-MNCS

Figure 1: Maximum subarray examples (with the maximum sum T). (a) Maximum Subsequence Sum. (b) Maximum Subsequence Sum with cyclic shifts. (c) Maximum Non-consecutive Sum.

the basic divide-and-conquer strategy: partitioning the input array A into two nearly-equal length subarrays and evaluating the MSS recursively.

Note that, in general, the MSS can either be included entirely within the left or right subarrays or it can span the center. If the MSS spans the center, then it must include elements q and $q+1$. As a result, the procedure Max-Span is used to find the maximum subarray from p to r which includes elements q and $q+1$.

```
Max-Span(A, p, q, r)
 1  ▷ Find maximum sum to the left, beginning at q
 2  i ← q, T_l ← −∞, S ← 0
 3  for k ← q downto p
 4      do S ← S + A[k]
 5         if S ≥ T_l
 6            then i ← k, T_l ← S
 7  ▷ Find maximum sum to the right, beginning at q + 1
 8  j ← q + 1, T_r ← −∞, S ← 0
 9  for k ← (q + 1) to r
10      do S ← S + A[k]
11         if S > T_r
12            then j ← k, T_r ← S
13  T ← T_l + T_r
14  return {i, j, T}
```

To complete our discussion of Find-MSS, note that on line 6 we evaluate the MSS spanning the center. Finally, on lines 7-14 we determine the maximum sum of either the left, right, or spanning subarrays and return the corresponding MSS subarray.

Proof of Correctness: Assuming that Max-Span is correct, the correctness of Find-MSS is immediate. That is, the comparison on line 7 will determine the best-possible subsequence among the three possibilities, i.e. the left, right, or spanning subarrays.

We can prove the correctness of Max-Span directly. Note that Max-Span can be separated into two parts; first, on lines 1-6 we determine the MSS beginning at q and possibly extending to p. Similarly, on lines 8-12 we determine the MSS beginning at $q+1$ and possibly extending to r. In either case, we can evaluate the MSS by looping over the array elements and storing the current MSS estimate. For example, on line 2 we initialize the left index to $i = q$, the MSS to $T_l = -\infty$, and the "accumulator" to $S = 0$. As we loop over the array elements from q down to

p, we increment the accumulator S. If the running sum surpasses the estimated MSS T_l, then we update our estimate of the endpoint i and T_l. A similar process is applied to the right side on lines 8-13 to determine j and T_r. In conclusion, the correct MSS for the "spanning" subarray is given by $T \leftarrow T_l + T_r$ on line 13.

Analysis of Running Time: The asymptotic worst-case running time of FIND-MSS is $O(n \log n)$. To prove this result, let's begin by examining the running time of MAX-SPAN. Since lines 3 and 9 involve a loop over all the array elements from p to r, and the remaining lines all involve constant time operations, then the asymptotic worst case running time of MAX-SPAN is $\Theta(n)$.

The recurrence relation for the worst-case running time of FIND-MSS is given by

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2 \end{cases} \tag{1}$$

The first term $2T(n/2)$ corresponds to lines 4 and 5 of FIND-MSS. The second term $\Theta(n)$ accounts for the call to MAX-SPAN on line 6 and the remaining $O(1)$ operations. We can solve this recursion using the methods outlined in [2] on page 66. First, we guess that the solution is $T(n) = O(n \log n)$. Substituting into Equation 1 gives

$$\begin{aligned} T(n) &\leq 2[c(n/2)\log(n/2)] + n \\ &\leq cn\log(n/2) + n \\ &= cn\log n - cn\log(2) + n \\ &\leq cn\log n, \text{ for } c \geq 1 \end{aligned}$$

In conclusion, we have shown that the asymptotic worst-case running time of FIND-MSS is $O(n \log n)$ by the substitution method. (QED)

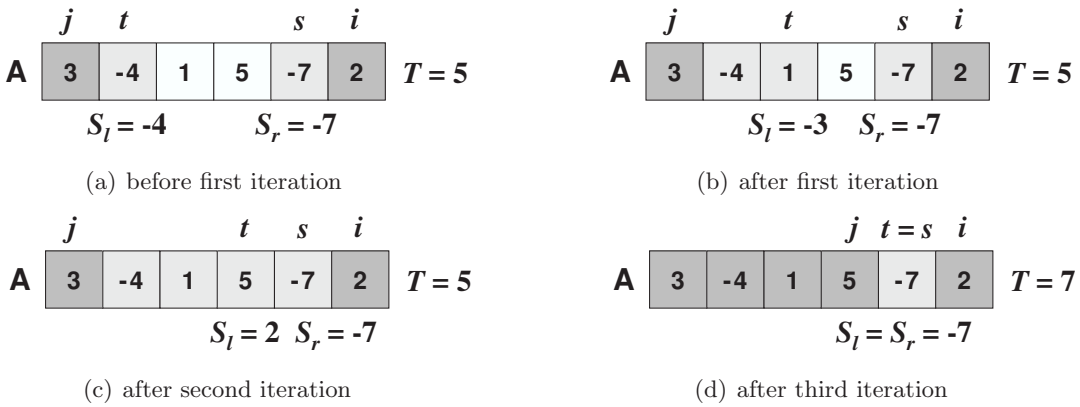
Part (b): Finding the Maximum Subsequence Sum with Cyclic Shifts

Now extend your algorithm (also in time $O(n \log n)$) to work with cyclic shifts: sequences of the form $A[i] + \ldots + A[n] + A[1] + A[2] + \ldots + A[j]$, where $j < i$.

High-level Description: This problem can be solved in $O(n \log n)$ using a simple extension of FIND-MSS. Note that, in general, the maximum subsequence will either contain a cyclic shift, or it will not. In the former case $j < i$, whereas in the later case $i \leq j$ (corresponding to the solution from Part (a)). As a result, we can apply the divide-and-conquer method to solve the Maximum Subsequence Sum problem with cyclic shifts using the procedure FIND-CYCLIC-MSS shown below.

```
FIND-CYCLIC-MSS(A)
1  {i_l, j_l, T_l} ← FIND-MSS(A, 1, length[A])
2  {i_s, j_s, T_s} ← MAX-CYCLIC-SUBARRAY(A)
3  if T_l ≥ T_s
4     then return {i_l, j_l, T_l}
5     else  return {i_s, j_s, T_s}
```

On line 1 we determine the MSS which does not contain a cyclic shift (i.e., $i \leq j$). One line 2 we use the procedure MAX-CYCLIC-SUBARRAY (shown below) to find the maximum subarray which includes a cyclic shift (i.e., $j < i$). Finally, on lines 3-5 we compare the two possible subarrays and return the one with the larger sum (breaking ties in favor of sequences without cyclic shifts).



(a) before first iteration

(b) after first iteration

(c) after second iteration

(d) after third iteration

Figure 2: Operation of MAX-CYCLIC-SUBARRAY. The state of the maximum cyclic subarray estimate is shown in (a) through (d), where the shading corresponds to the the state of the array element. White indicates that the element has not yet been considered. Light gray indicates that the element has been added to either the S_l or S_r accumulators. Dark gray indicates elements that are part of the current MSS estimate. Note that the final output is found after three iterations.

```
MAX-CYCLIC-SUBARRAY(A)
 1  n ← length([A])
 2  if n ≤ 2
 3     then return {0, 0, −∞}
 4  i ← n, j ← 1
 5  T ← A[1] + A[n]
 6  s ← n − 1, t ← 2
 7  S_l ← A[t], S_r ← A[s]
 8  while s > t
 9     do if S_r ≥ S_l
10           then s ← s − 1
11                if S_r ≥ 0
12                   then T ← T + S_r
13                        i ← s + 1
14                        S_r ← A[s]
15                   else S_r ← S_r + A[s]
16           else t ← t + 1
17                if S_l > 0
18                   then T ← T + S_l
19                        j ← t − 1
20                        S_l ← A[t]
21                   else S_l ← S_l + A[t]
22  return {i, j, T}
```

Proof of Correctness: Assuming that MAX-CYCLIC-SUBARRAY is correct, the correctness of FIND-CYCLIC-MSS is immediate. That is, the comparison on line 3 will determine the best-possible subsequence among the two possibilities, i.e. either one with or without cyclic shifts. We can prove the correctness of MAX-CYCLIC-SUBARRAY directly. First, note that when $length[A] \leq 2$ there is always a non-cyclic solution equivalent to every cyclic one. As a result, we return the default values of $\{i, j, T\} = \{0, 0, -\infty\}$ on lines 2 and 3 of MAX-CYCLIC-SUBARRAY. This will ensure that

the non-cyclic solution is selected in these simple cases.

For $length[A] \geq 3$ the MSS can potentially include a cyclic shift. Consider the example shown in Figure 2 for $A = \{3, -4, 1, 5, -7, 2\}$. Since the MSS with a cyclic shift must include the elements $A[1]$ and $A[n]$, we initialize the MSS as $T \leftarrow A[1] + A[n]$ on line 5. Recall that we want to determine the maximum cyclic subarray such that $j < i$. We can determine the optimal values of i and j by an incremental method. First, we compare the elements $A[2]$ and $A[n-1]$ (as implemented on lines 6-9). We select the larger of the two. So long as the larger element is positive, then we increment our estimate of the MSS (and the array indices). This is achieved on lines 12-14 and 18-20. As shown in Figure 2(d), we only update our estimate of the MSS if the left or right accumulators (S_l and S_r, respectively) are greater than zero. That is, the MSS estimate T must be monotonically increasing.

The exit condition for the **while** loop on line 8 is $s \leq t$. This will occur once we have traversed each element in the array. Since we always advance the left and right endpoints so that the increase in the MSS is *as large as possible*, MAX-CYCLIC-SUBARRAY will correctly estimate the Maximum Subsequence Sum which contains a cyclic shift. Note that, for the example array A in Figure 1, the subarray containing the cyclic shift has a greater sum than the one without a cyclic shift.

Analysis of Running Time: The asymptotic worst-case running time of FIND-CYCLIC-MSS is $O(n \log n)$. To prove this result, let's begin by examining the running time of MAX-CYCLIC-SUBARRAY. Lines 1-7 and lines 9-22 all involve $\Theta(1)$ operations. As a result, the worst-case running time is determined by the number of iterations required to exit the **while** loop on line 8 (with exit condition $s \leq t$). Since the initial values of $\{s, t\} = \{n-1, 2\}$ (from line 6) and each iteration of the **while** loop either decrements s or increments t, then the worst-case running time of MAX-CYCLIC-SUBARRAY is $\Theta(n)$. That is, in the worst case, s will be decremented (or t will be incremented) every iteration until the exit condition is met. Since there are n elements in the array and s and t start adjacent to one endpoint of the array, then $\Theta(n)$ iterations are required to exit.

Given that the running time of MAX-CYCLIC-SUBARRAY is $\Theta(n)$, then the asymptotic worst-case running time of FIND-CYCLIC-MSS is given by $O(n \log n) + \Theta(n) = O(n \log n)$. The first logarithmic term is due to the call to FIND-MSS on line 1, whereas the $\Theta(n)$ term results from calling MAX-CYCLIC-SUBARRAY on line 2. (QED)

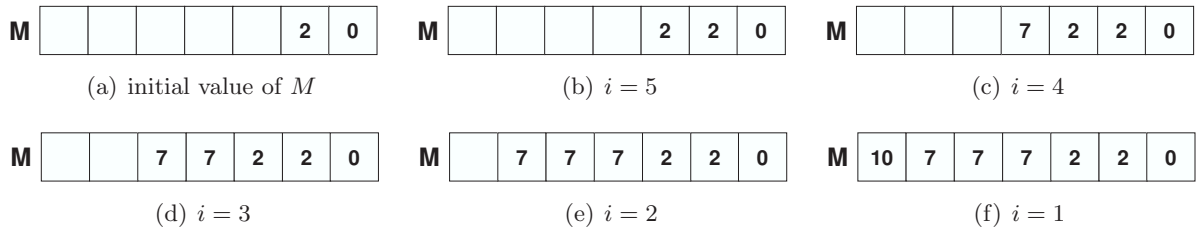
Part (c): Finding the Maximum Sum of Non-consecutive Elements

Given an array A of integers, design an algorithm to find a subset $S \subseteq \{1, 2, \cdots, n\}$ which maximizes the sum of $A[i]$ for all $i \in S$ such that if $i \in S$ then $(i+1) \notin S$.

High-level Description: This problem can be solved in $O(n)$ using dynamic programming. As is typical for developing a dynamic-programming algorithm, we begin by (1) characterizing the structure of an optimal solution and (2) defining a recursive solution. To begin our design, note that the first element $A[1]$ is either included in S or it is not. If it is included, then we cannot include $A[2]$, however we must include the Maximum Non-consecutive Sum (MNCS) for $A[3, \ldots, n]$. If $A[1]$ is not in the MNCS, then we must include the MNCS for $A[2, \ldots, n]$.

By induction, a simple recursive algorithm for computing the MNCS is apparent. Beginning at the first element of A, we can recursively compute the MNCS for $A[2, \ldots, n]$ and $A[3, \ldots, n]$. If $A[1]$ plus the MNCS for $A[3, \ldots, n]$ is greater than or equal to the MNCS for $A[2, \ldots, n]$, then the first element must be included in S, otherwise it is not. We can proceed through the array elements, performing similar comparisons, to determine the elements of the subset S and the MNCS.

We can greatly improve upon the naive solution by applying the dynamic programming method. That is, we can build a "maximum sum" lookup table M, whose elements $M[i]$ store the MNCS



(a) initial value of M (b) $i = 5$ (c) $i = 4$

(d) $i = 3$ (e) $i = 2$ (f) $i = 1$

Figure 3: Creation of "maximum sum" lookup table M, as implemented by FIND-MNCS on lines 2-12. The input array is $A = \{3, -4, 1, 5, -7, 2\}$. The state of M after each iteration of the **for** loop (on line 7) is shown in (b)-(f). Note that $M[1]$ is equal to the Maximum Non-consecutive Sum.

for the subarray $A[i, \ldots, n]$. The construction of M is implemented on lines 2-12 of FIND-MNCS shown below (the correctness of M is described in the following section). By the definition of M, the Maximum Non-consecutive Sum for the array $A[1, \ldots, n]$ is given by $M[1]$ (as assigned on line 13). Note that, since we also want to know the elements of S in addition to the value of the MNCS, lines 15-21 traverse M and construct the optimal solution for S. Finally, the subset S and MNCS T are returned on line 22.

```
FIND-MNCS(A)
 1  ▷ Compute the lookup table M
 2  n ← length[A]
 3  M[n + 1] ← 0
 4  if A[n] > 0
 5     then M[n] ← A[n]
 6     else  M[n] ← 0
 7  for i ← (n − 1) downto 1
 8       do if A[i] > 0
 9             then S_1 ← A[i] + M[i + 2]
10             else  S_1 ← 0
11          S_2 ← M[i + 1]
12          M[i] ← max(S_1, S_2)
13  T ← M[1]
14  ▷ Determine elements in subset S
15  S ← ∅
16  i ← 1
17  while i ≤ n
18       do if M[i] > M[i + 1]
19             then S ← S ∪ i
20                  i ← i + 2
21             else  i ← i + 1
22  return S and T
```

Proof of Correctness: In order to prove the correctness of FIND-MNCS, we must prove: (1) that lines 1-12 correctly construct M and (2) that lines 13-22 correctly construct the subset S and MNCS T from M. We begin our analysis by verifying the estimate of S – assuming M correctly contains the MNCS for the subarray $A[i, \ldots, n]$. For example, consider $A = \{3, -4, 1, 5, -7, 2\}$ and $M = \{10, 7, 7, 7, 2, 2, 0\}$, as shown in Figure 3. We initialize $S \leftarrow \emptyset$ on line 15. Next, we check whether $M[1] > M[2]$; if it is, then $A[1]$ must be included in the MNCS and, as a result, $A[2]$ cannot be included. If the first element is included, then we skip the second and search the

remaining elements of M. As a result, it is clear that lines 13-22 correctly construct S and T, the MNCS, given that M is correct.

To complete our analysis, we must prove that lines 1-12 correctly construct M. Once again, consider the example in Figure 3. We initialize our estimate of M on lines 2-6. Notice that we must include the element $M[n+1] \leftarrow 0$, since including the last array element does not exclude any elements other than its left neighbor. In general, we note that only nonnegative elements of A will be included in S, since there is no benefit to including negative values (like there was in Parts (a) and (b)). As a result, we will only add $A[n]$ to $M[n]$ if $A[n] > 0$, otherwise the MNCS will not contain $A[n]$ and can be set equal to 0.

Given the initial value of M, shown in Figure 3(a), lines 7-12 build M in a bottom-up fashion. At each step, we use the previously-computed MNCS values in M to determine the MNCS for the subarray $A[i, \ldots, n]$ – just as was done in the naive solution. As a result, M is correct and, consequently, so is FIND-MNCS.

Analysis of Running Time: The asymptotic worst-case running time of FIND-MNCS is $O(n)$. We can prove this result by direct inspection. First, note that lines 1-6, 13-16, and 22 all involve constant time operations. The **for** loop on lines 7-12 requires $O(n)$ iterations, each of which can be completed in linear time. Similarly, the **while** loop on lines 17-21 also requires a maximum of $O(n)$ iterations (each of which can be completed in linear time), since in the worst case i is incremented by one each iteration (on line 21) until the exit condition $i > n$ is satisfied. In conclusion, the asymptotic worst-case running time of FIND-MNCS is $O(n)$.

Conclusion

This write-up has presented several algorithms for determining the maximum sum of certain subsets of one-dimensional arrays. As discussed in [1], the Maximum Subsequence Sum problem was originally motivated by a pattern-matching method proposed by Ulf Grenander from Brown University. While his interest was in solving the related Maximum Subarray problem in two-dimensions, the one-dimensional version presented in this write-up successfully illustrated several algorithm design techniques. In contrast to [1], this write-up extended the divide-and-conquer solution to cyclic arrays in Part (b). In addition, Part (c) demonstrated the application of dynamic programming to achieve a linear-time solution to the Maximum Non-consecutive Subset problem. For more information on maximum subarray problems, and a linear-time solution to Part (a), see [1].

References

[1] Jon Bentley. Programming pearls: algorithm design techniques. *Commun. ACM*, 27(9):865–873, 1984.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill, 2001.