Design and Analysis of Computer Algorithms
CS 6363.005: Homework #2

Due on Monday September 26, 2016 at 11:59pm

Professor Benjamin Raichel

Hanlin He **(hxh160630)**
hanlin.he@utdallas.edu

Contents

Problem 1 Inversions

Consider MERGESORT:

> The procedure first divides the array into two same size sub-array, recursively sort each sub-array, and then merge two sorted sub-array.

So, when counting the inversions, we can also adopt similar method:

1. divide the array into two same size sub-arrays;
2. recursively count inversions within each two sub-arrays;
3. count inversions which one index comes from first sub-array and the other index comes second sub-array;
4. combine the result.

In the merge step of the MERGESORT, the function combine the first half and the second half sub-array into one sorted array. Now we need to count the inversions which one index comes from first sub-array, and the other index come from second sub-array. Consider the length of two arrays are n and m. According to the definition of inversion, if the jth element in the second array is smaller than the ith element of the first array, then there are $n - i$ inversions.

According to the observation, the modified MERGE is shown as COUNT_INVERSIONS in algorithm 1.

Algorithm 1 Modified Merge of two arrays.

```
procedure COUNT_INVERSIONS(A[1...n+m], B[1...n], C[1...m])
    InversionsCount = 0
    Bindex = 1; Cindex = 1
    for Aindex = 1 to n+m do
        if Cindex > m then                                  ▷ C is exhausted. ◁
            A[Aindex] = B[Bindex]
            Bindex++
        else if Bindex > n then                             ▷ B is exhausted. ◁
            A[Aindex] = C[Cindex]
            Cindex++
        else if B[Bindex] < C[Cindex] then                  ▷ B is smaller than C. ◁
            A[Aindex] = B[Bindex]
            Bindex++
        else                            ▷ C is smaller than B, count inversions. ◁
            A[Aindex] = C[Cindex]
            Cindex++
            InversionsCount = InversionsCount + (n − Bindex)
        end if
    end for
    return InversionsCount
end procedure
```

Problem 1 continued on next page...

In algorithm 1, there is one For-Loop with only constant time operation in each loop. Thus the running time of COUNT_INVERSION is $\mathcal{O}(n)$.

Based on algorithm 1, if we recursively sort and count inversions in two sub-array, we will get the result of inversions sum. Therefore, the modified MERGESORT is shown as SORT_AND_COUNT in algorithm 2.

Algorithm 2 Count inversion of an array.

procedure SORT_AND_COUNT($A[1 \dots n]$)
  Define $\overline{A}[1 \dots n]$ as working buffer for merge.
  if $n = 1$ **then** ▷ Only one element in the array, no inversions exist. ◁
    return 0
  else ▷ if $n > 1$ ◁
    $left\_inversion$ =SORT_AND_COUNT($A[1 \dots \lfloor n/2 \rfloor]$)
      ▷ Sort and count inversions within the first half subarray. ◁

    $right\_inversion$ =SORT_AND_COUNT($A[\lfloor n/2 \rfloor + 1 \dots n]$)
      ▷ Sort and count inversions within the second half subarray. ◁

    $split\_inversion$ =COUNT_INVERSIONS($\overline{A}[1 \dots n]$, $A[1 \dots \lfloor n/2 \rfloor$, $A[\lfloor n/2 \rfloor + 1 \dots n]$)
      ▷ Merge two sorted subarray and count inversions between two subarrays. ◁

    $A[1 \dots n] \leftarrow \overline{A}[1 \dots n]$ ▷ Copy buffer array to the original array. ◁
  end if
  return $left\_inversion + right\_inversion + split\_inversion$
end procedure

Claim: SORT_AND_COUNT($A[1 \dots n]$) returns the number of inversions of array $A[1 \dots n]$.

Proof: If $n = 1$, the length of the array is 1, obviously there is no inversion. Otherwise, divide the array from the middle into two sub-array. The total inversions of the array is the combination of the inversions number of the first half and second half and those inversions whose indices split into two halves.

Thus, sort and count the inversions within each sub-array respectively. Then merge two sub-array and count inversions which two indices split into to sub-array. Together, we can easily prove using induction that the combination of three result is the total number of inversions in the array. □

The SORT_AND_COUNT($A[1 \dots n]$) is recursive, according to the algorithm, its running time can be expressed by a recurrence.

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \mathcal{O}(n) \\ &= 2T(n/2) + \mathcal{O}(n) \end{aligned} \tag{1.1}$$

Thus, the running time of algorithm 2 is $T(n) = \Theta(n \log n)$

Problem 2 Selection in sorted arrays

Algorithm 3 Merge two sorted arrays and return the kth element.

procedure SELECTK($A[1 \ldots n]$, $B[1 \ldots m]$, k)
    Define $C[1 \ldots n + m]$
    $Aindex = 1$; $Bindex = 1$
    for $Cindex = 1$ to $n + m$ **do**
        if $Bindex > m$ **then** ▷ B is exhausted. ◁
            $C[Cindex] = A[Aindex]$
            $Aindex{+}{+}$
        else if $Aindex > n$ **then** ▷ A is exhausted. ◁
            $C[Cindex] = B[Bindex]$
            $Bindex{+}{+}$
        else if $A[Aindex] < B[Bindex]$ **then** ▷ A is smaller than B. ◁
            $C[Cindex] = A[Aindex]$
            $Aindex{+}{+}$
        else ▷ B is smaller than A. ◁
            $C[Cindex] = B[Bindex]$
            $Bindex{+}{+}$
        end if
    end for
    return $C[k]$
end procedure

Claim: SELECTK($A[1 \ldots n], B[1 \ldots m], k$) returns the kth ranked value in the union of the two arrays.

Proof: Merge two array into one, sort this new array, then the kth element in this array is the kth ranked value in the union of the two arrays. The two array is sorted in the first place. Thus, in the For-Loop shown in algorithm 3, the smaller element in two array is put into the new array. Therefore, when the For-Loop finishes, the new array would be a sorted array.

Thus, Return the kth element in the new array will give the kth ranked value in the union of the two arrays. □

In algorithm 3, there is one For-Loop with only constant time operation in each loop. Thus the running time of SELECTK is $\mathcal{O}(n + m)$.

Problem 3 Maximum Subarray Sum

Let $maxSum(i, j)$ be the maximum sub-array sum in $A[i \dots j]$, there are two cases:

- $maxSum(i, j)$ does not include $A[j]$, then

$$maxSum(i, j) = maxSum(i, j-1)$$

- $maxSum(i, j)$ does include $A[j]$, then

$$maxSum(i, j) = maxEndAt(i, j)$$

  in which $maxEndAt(i, j)$ is the max sub-array sum in $A[i \dots j]$ restricted to include $A[j]$.

There are again two cases for $maxEndAt(i, j)$:

- $maxEndAt(i, j)$ is the element $A[j]$ itself, i.e.

$$maxEndAt(i, j) = A[j]$$

- $maxEndAt(i, j)$ is the element $A[j]$ plus $maxEndAt(i, j-1)$, i.e.

$$maxEndAt(i, j) = A[j] + maxEndAt(i, j-1)$$

Thus, the recursive algorithm solving the maximum sub-array sum can be described as algorithm 4.

Algorithm 4 Recursive Solution to Maximum Sub-array Sum

procedure MaxSum($A[i \dots j]$)
    if $i > j$ **then**
        return 0
    end if
    return $\max\{$MaxEndAt$(A[i \dots j])$, MaxSum$(A[i \dots j-1])\}$
end procedure

---

procedure MaxEndAt($A[i \dots j]$)
    if $i > j$ **then**
        return 0
    end if
    return $\max\{A[j],\ A[j] +$ MaxEndAt$(A[i \dots j-1])\}$
end procedure

To find the maximum sub-array sum of $A[1 \dots n]$, we can call MaxSum($A[1 \dots n]$).

Observing algorithm 4, we can find that:

- In MaxEndAt, each recursion is strictly based on the previous call, i.e. MaxEndAt(i, j) is strictly based on MaxEndAt$(i, j - 1)$.
- In MaxSum, each recursion is strictly based on the previous call and MaxEndAt, i.e. MaxSum(i, j) is strictly based on MaxSum$(i, j - 1)$ and MaxEndAt(i, j).

Therefore, we can memorize MaxSum and MaxEndAt in each recursion. The memoized algorithm is shown in algorithm 5.

Globally defined array $R[1 \dots n]$ and $M[i \dots n]$.

Algorithm 5 Memoized Solution to Maximum Sub-array Sum

procedure MemMaxSum$(A[i \dots j])$
    if $i > j$ **then**
        return 0
    end if
    if $R[j]$ undefined **then**
        $R[j] = \max \{$MemMaxEndAt$(A[i \dots j])$, MemMaxSum$(A[i \dots j - 1])\}$
    end if
    return $R[j]$
end procedure

---

procedure MemMaxEndAt$(A[i \dots j])$
    if $i > j$ **then**
        return 0
    end if
    if $M[j]$ undefined **then**
        $M[j] = \max \{A[j],\ A[j] +$ MemMaxEndAt$(A[i \dots j - 1])\}$
    end if
    return $M[j]$
end procedure

To find the maximum sub-array sum of $A[1 \dots n]$, we can call MemMaxSum$(A[1 \dots n])$.

Applying DP:

According to the previous observation, MemMaxSum($A[1 \ldots n]$) depends on two array $R[1 \ldots n]$ and $M[1 \ldots n]$, each ranging over $\mathcal{O}(n)$ values, and is strictly based on previous call. Hence the above recursive algorithm can be turned into a DP algorithm using this two array. The array can be filled starting at 1 and going up to n. First the $M[1 \ldots n]$, then the $R[1 \ldots n]$.

The dynamic programming algorithm is shown in algorithm 6.

Algorithm 6 Dynamic Programming Solution to Maximum Sub-array Sum

procedure DPMaxSum($A[1 \ldots n]$)
  Let $R[0 \ldots n]$ and $M[0 \ldots n]$ be an array.
  $M[0] = 0$, $R[0] = 0$
  for $i = 1$ to n **do**
    $M[i] = \max\{A[i],\ A[i] + M[i-1])\}$
  end for
  for $i = 1$ to n **do**
    $R[j] = \max\{M[i],\ R[i-1])\}$
  end for
  return $R[n]$
end procedure

With two for loop of n iteration and constant time operations in each iteration, the running time for algorithm 6 is $\mathcal{O}(n)$.

Problem 4 Starting A Jewelery Collection

Consider how to handle the very first element (jewelery) of $J[1\dots n]$ with the bag size S.

- If A is empty, there is no element to be taken, return 0.
- If the size of the first element is bigger than the bag size, i.e. $V[1] > S$, the first element cannot be taken.
$$JCV^{[1]}(J[1\dots n], S) = JCV(J[2\dots n], S)$$
- If the size of the first element is smaller than the bag size, i.e. $V[1] < S$, the first element can be either taken or thrown out.
$$JCV(J[1\dots n], S) = \max\{JCV(J[2\dots n], S), JCV(J[2\dots n], S - V[1])\}$$

Now to true this in a recursive algorithm, define JCollection($CurN, CurB$) be the maximum value of the set $J[CurN\dots n]$, with value $V[CurN\dots n]$, size $S[CurN\dots n]$ and bag size $CurB$. Based on the above observation, we have the following.

Assume $V[1\dots n]$ and $S[1\dots n]$ defined globally, and $CurN, CurB > 0$.

Algorithm 7 Recursive Solution to Jewelery Collection

procedure JCollection($CurN, CurB$)
    if $CurN > n$ or $CurB < 1$ **then**
        return 0
    end if
    $ignore$ = JCollection($CurN + 1, CurB$)
    $best = ignore$
    if $S[CurN] < CurB$ **then**
        $include = V[CurN]$ + JCollection($CurN + 1, CurB - V[CurN]$)
        if $include > ignore$ **then**
            $best = include$
        end if
    end if
    return $best$
end procedure

To find the maximum value of $J[1\dots n]$ with b size bag, we can call JCollection($1, b$).

The correctness of algorithm 7 can be proved by previous observation.

Applying DP: JCollection($CurN, CurB$) depends on two parameters, the first ranging over $\mathcal{O}(n)$ values and the second over $\mathcal{O}(b)$ values, since they are indices into $J[1\dots n]$ and the size downgraded from b respectively. Hence the above recursive algorithm can be turned into a DP algorithm using a 2D array, of total size $\mathcal{O}(nb)$.

[1] $JCV(J[1\dots n], S)$ is the max value to achieve in jewelery set $J[1\dots n]$ with S size bag.

Problem 4 continued on next page...

JCollection($CurN, CurB$) makes at most two recursive calls which are JCollection($CurN + 1, CurB$) and JCollection($CurN + 1, CurB - V[CurN]$). In each case, at least one of the two parameters increases and the other does not decrease. Therefore, the 2D array can be filled in using a pair of nested for loops, the outer one ranging over the first parameter and starting at n going down to 1, and the inner one ranging over the second parameter and start at 1 going up to b. Ignoreing the time for computing recursive calls, the above algorithm runs in $\mathcal{O}(1)$ time. Therefore, if processed in the right order, each table entry takes $\mathcal{O}(1)$ time to compute and so the total running time is $\mathcal{O}(nb)$.

To simplify the process, add the $n+1$ item with 0 size and 0 value to the element set. Thus the 2D array is expanded from $C[1 \ldots n][1 \ldots b]$ to $C[1 \ldots n+1][0 \ldots b]$. Therefore, each cell $C[x][y]$ represent the maximum value when considering the x-th element with current bag size y, i.e.

$$C[x][y] = \text{JCollection}(x, y)$$

The dynamic programming solution to the problem is shown in algorithm 7.

Algorithm 8 Dynamic Programming Solution to Jewelery Collection

```
procedure DPJCollection(V[1...n], S[1...n], b)
    Define C[1...n+1][0...b]
    for i = 1 to n+1 do
        C[i][0] = 0
    end for
    for i = 0 to b do
        C[n+1][i] = 0
    end for
    for CurN = n to 1 do
        for CurB = 1 to b do
            ignore = C[CurN+1][CurB]
            best = ignore
            if S[CurN] < CurB then
                include = V[CurN] + C[CurN+1][CurB - S[CurN]]
                if include > ignore then
                    best = include
                end if
            end if
            C[CurN][CurB] = best
        end for
    end for
    return C[1][b]
end procedure
```

With two for loop of n and b iterations and constant time operation in each iteration, the running time of algorithm 7 is obviously $\mathcal{O}(nb)$.