CSE6140 Fall2017 Homework 2 Solutions

September 2015

1 Greedy: Scheduling with Weights

a) This strategy is not optimal. We'll show it by a counter-example. We have two tasks:

   - E_1: $t_1 = 10$ and $w_1 = 1$
   - E_2: $t_2 = 20$ and $w_2 = 100$.

   The greedy algorithm which considers the emails in the order of increasing time will schedule E_1 and then E_2. Therefore, $C_1 = 10$ and $C_2 = 30$. The total cost will be $w_1C_1 + w_2C_2 = 10 + 3000 = 3010$.

   However, if we do E_2 and then E_1 the cost will be $20 \times 100 + 30 \times 1 = 2030$ which is clearly a better scheduling.

b) We'll show this strategy is not optimal by a counter-example too. Consider the following two emails:

   - E_1: $t_1 = 20$ and $w_1 = 100$
   - E_2: $t_2 = 10$ and $w_2 = 90$.

   The greedy algorithm which sorts the intervals by decreasing weight will schedule E_1 to be done before E_2. Therefore, $C_1 = 20$ and $C_2 = 30$. Therefore, the cost is $w_1C_1 + w_2C_2 = 2000 + 2700 = 4700$.

   However, if we do E_2 before E_1 we will get the cost equal to $10 \times 90 + 30 \times 100 = 3900$. This shows that greedy solution is not optimal.

c) Consider the order in which the jobs appear in our Greedy solution $G = g_1, ..., g_n$. We will use the notation $g(i)$ to refer to the index of email i in the Greedy solution. By our greedy rule applied here, we know that

   $$\frac{w_{g_1}}{t_{g_1}} \geq \frac{w_{g_2}}{t_{g_2}} \geq \ldots \geq \frac{w_{g_n}}{t_{g_n}}.$$

   We say that an email ordering A has an *inversion* if an email i is schedule before another email j and $g(i) > g(j)$. By our definition of inversion, the solution G produced by the greedy algorithm does not contain any inversions.

Theorem: There is an optimal schedule that has no inversions (i.e. is identical to Greedy).
Proof.

For the sake of contradiction, assume all optimal schedules have at least one inversion, and consider the optimal email ordering $O = \{o_1, o_2, ..., o_n\}$ with the fewest number of inversions. It is easy to show that if an ordering has at least one inversion then there is a pair of consecutive emails o_k and o_{k+1} that are also inverted, i.e. with $g(o_k) > g(o_{k+1})$ (by identical argument as the one made in theorem 4.9 in KT).

We will consider swapping the order of emails o_k and o_{k+1} to produce schedule $O' = \{o_1, o_2, ..., o_{k-1}, o_{k+1}, o_k, o_{k+2}, ..., o_n\}$. Let the cost of O be:

$$\mathit{Cost} = \sum_{i=1}^{n} w_{o_i} C_{o_i} = \sum_{i=1}^{k-1} w_{o_i} C_{o_i} + w_{o_k} C_{o_k} + w_{o_{k+1}} C_{o_{k+1}} + \sum_{i=k+2}^{n} w_{o_i} C_{o_i},$$

where $C_{o_k} = C_{o_{k-1}} + t_{o_k}$ and $C_{o_{k+1}} = C_{o_{k-1}} + t_{o_k} + t_{o_{k+1}}$. Thus,

$$\mathit{Cost} = \sum_{i=1}^{k-1} w_{o_i} C_{o_i} + w_{o_k}(C_{k-1} + t_{o_k}) + w_{o_{k+1}}(C_{o_{k-1}} + t_{o_k} + t_{o_{k+1}}) + \sum_{i=k+2}^{n} w_{o_i} C_{o_i}.$$

By exchanging emails k and $k+1$, we will get the following cost of O':

$$\mathit{Cost}' = \sum_{i=1}^{k-1} w_{o_i} C_{o_i} + w_{o_{k+1}}(C_{o_{k-1}} + t_{o_{k+1}}) + w_{o_k}(C_{k-1} + t_{o_{k+1}} + t_{o_k}) + \sum_{i=k+2}^{n} w_{o_i} C_{o_i}.$$

The difference in cost will be:

$$\mathit{Cost}' - \mathit{Cost} = w_{o_k} t_{o_{k+1}} - w_{o_{k+1}} t_{o_k}$$

Since $g(o_k) > g(o_{k+1})$ and $\frac{w_{g_1}}{t_{g_1}} \geq \frac{w_{g_2}}{t_{g_2}} \geq \ldots \geq \frac{w_{g_n}}{t_{g_n}}$, we can have two cases: 1) $\frac{w_{o_k}}{t_{o_k}} = \frac{w_{o_{k+1}}}{t_{o_{k+1}}}$, or 2) $\frac{w_{o_k}}{t_{o_k}} < \frac{w_{o_{k+1}}}{t_{o_{k+1}}}$.

CASE 1) $\frac{w_{o_k}}{t_{o_k}} = \frac{w_{o_{k+1}}}{t_{o_{k+1}}}$
This immediately implies that $\mathit{Cost}' - \mathit{Cost} = w_{o_k} t_{o_{k+1}} - w_{o_{k+1}} t_{o_k} = 0$. This means that O' is also an optimal ordering but has one less inversion than O. This is a contradiction with our assumption that O is an optimal ordering with at least one inversion.

CASE 2) $\frac{w_{o_k}}{t_{o_k}} < \frac{w_{o_{k+1}}}{t_{o_{k+1}}}$
This immediately implies that $\mathit{Cost}' - \mathit{Cost} = w_{o_k} t_{o_{k+1}} - w_{o_{k+1}} t_{o_k} < 0$. This means that O' is an ordering with smaller sum of weighted completion times. This is a contradiction with our assumption that O is an *optimal* ordering.

This means that our initial assumption that there is no optimal solution with zero inversions, and hence that that the optimal solution with fewest number of inversions will have at least one inversion was wrong. Since G contains no inversions, it must be an optimal solution.

2 Divide and Conquer

a) The algorithm should return the indices 4 and 7, which leads to a sum of 32.

b) We want to divide the array T into halves. Then, three cases are possible:

   i) The interval of maximum sum is included in the first half;

   ii) Then interval of maximum sum is included in the second half;

   iii) The interval of maximum sum contains elements from both halves.

   The first two cases are solved through simple recursive calls on the halves. For the third case, we compute the interval of maximum sum that starts at the first element of the second half(this interval is tehn included in the second half). Symmetrically, we compute the interval of maximum sum that end with the last elements of the first half. The union of these two interval is the interval of maximum sum that contains elements from both halves. Algorithm 2.1 presents a simple realization of this divide-

and-conquer principle.

Algorithm 2.1: $LS(b, e)$ - Find the maximum sum of contiguous elements of the array T between the indices b and e (included)

```
if $b = e$ then
    return $(b, b, A[b])$
end
$middle \leftarrow \lfloor \frac{e+b}{2} \rfloor$
$(start_1, end_1, sum_1) \leftarrow LS(b, middle)$
$(start_2, end_2, sum_2) \leftarrow LS(middle + 1, e)$
$LeftHalfSum \leftarrow A[middle]$
$LeftLimit \leftarrow middle$
$s \leftarrow A[middle]$
for $i = middle - 1$ to $b$ do
    $s \leftarrow s + A[i]$
    if $s > LeftHalfSum$ then
        $LeftHalfSum \leftarrow s$
        $LeftLimit \leftarrow i$
    end
end
$RightHalfSum \leftarrow A[middle + 1]$
$RightLimit \leftarrow middle + 1$
$s \leftarrow A[middle + 1]$
for $i = middle + 2$ to $e$ do
    $s \leftarrow s + A[i]$
    if $s > RightHalfSum$ then
        $RightHalfSum \leftarrow s$
        $RightLimit \leftarrow i$
    end
end
$sum_3 \leftarrow LeftHalfSum + RightHalfSum$
if $sum_1 = max\{sum_1, sum_2, sum_3\}$ then
    return $(start_1, end_1, sum_1)$
end
if $sum_2 = max\{sum_1, sum_2, sum_3\}$ then
    return $(start_2, end_2, sum_2)$
end
return $(LeftLimit, RightLimit, sum_3)$
```

A call to Algorithm 2.1 with an array of size n leads to two recursive calls on arrays of size $n/2$. Dividing the array has a constant cost. However, computing the overall results from the subresults requires scanning the whole array and, hence, has a cost of $\Theta(n)$. Therefore, the complexity is given by:

$$C(n) = 2C(\frac{n}{2}) + \Theta(n)$$

Using the master theorem, we have $a = 2, b = 2$, and $d = 1$, and thus

$$C(n) = \Theta(n \log n)$$

In order to lower the complexity of the divide-and-conquer solution, we do not want to scan the entire array to compute the interval of maximum sum that contains elements of both halves. In other words, we want to be able to compute such an interval in constant time. To solve this problem, we just have to remark that the maximum-sum interval starting with the first element of an array either is fully contained in the first half of this array or includes all of this first half and then is equal to the whole first half plus the maximum-sum interval starting with the first element of the second half of the array. If the recursive calls on the two halves identify:

- The maximum-sum interval starting with the first element
- The maximum-sum interval ending with the last element
- The sum of all the elements in the array

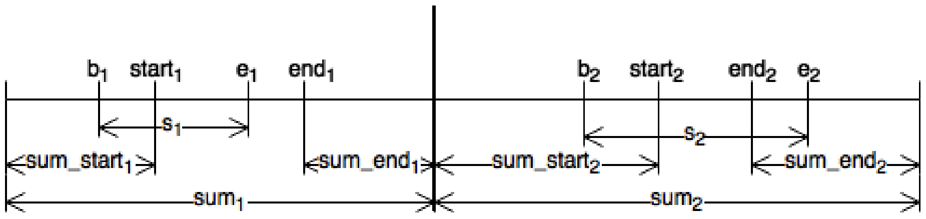
we will be able to compute these values in constant time for the original array. Algorithm 2.2 exactly realizes this scheme.

A call to Algorithm 2.2 with an array of size n leads to two recursive calls on arrays of size n/2. Dividing the array and computing all the needed results from the subresults has a constant cost. Therefore, the complexity is:

$$C(n) = 2C(\frac{n}{2}) + \Theta(1)$$

Using the master theorem, we have $a = 2, b = 2$, and $d = 0$, and thus

$$C(n) = \Theta(n)$$



Algorithm 2.2: $LSR(b, e)$ - Find the maximum sum of contiguous elements of the array T between the indices b and e (included)

```
if $b = e$ then
    return $(b, T[b], b, T[b], T[b], b, b, T[b])$
end
$middle \leftarrow \lfloor \frac{e+b}{2} \rfloor$
$(start_1, sum\_start_1, end_1, sum\_end_1, sum_1, b_1, e_1, s_1) \leftarrow LSR(b, middle)$
$(start_2, sum\_start_2, end_2, sum\_end_2, sum_2, b_2, e_2, s_2) \leftarrow$
  $LSR(middle + 1, e)$
if $sum\_start_1 \geq sum_1 + sum\_start_2$ then
    $sum\_start \leftarrow sum\_start_1$
    $start \leftarrow start_1$
else
    $sum\_start \leftarrow sum_1 + sum\_start_2$
    $start \leftarrow start_2$
end
if $sum\_end_2 \geq sum_2 + sum\_end_1$ then
    $sum\_end \leftarrow sum\_end_2$
    $end \leftarrow end_2$
else
    $sum\_end \leftarrow sum_2 + sum\_end_1$
    $end \leftarrow end_1$
end
$sum \leftarrow sum_1 + sum_2$
$b_3 \leftarrow end_1$
$e_3 \leftarrow start_2$
$s_3 \leftarrow sum\_end_1 + sum\_start_2$
if $s_3 = max\{s_1, s_2, s_3\}$ then
    return $(start, sum\_start, end, sum\_end, sum, b_3, e_3, s_3)$
end
if $s_2 = max\{s_1, s_2, s_3\}$ then
    return $(start, sum\_start, end, sum\_end, sum, b_2, e_2, s_2)$
end
return $(start, sum\_start, end, sum\_end, sum, b_1, e_1, s_1)$
```

c) **Solution1**

This new solution to the maximum-sum problem relies on several properties:

– If none of the elements in the array is positive, then the interval of maximum sum is reduced to a single element, the maximum array element.

– No strict prefix or suffix of the interval of maximum sum has a negative sum. We prove this result by contradiction, assuming that the interval of maximum sum starts at index i, ends at $k(k > i)$, and admits a prefix of negative sum that ends at $j(i \leq j < k)$. Then $\sum_{l=i}^{k} T_l = (\sum_{l=i}^{j} T_l) + (\sum_{l=j+1}^{k} T_l) < \sum_{l=j+1}^{k} T_l$. As a consequence, if at least one element in the array is positive, there exists an interval of maximum sum whose first and last elements are positive.

– If (j, k) is an interval of maximum sum, then, whatever the index $i < j$, the interval $(i, j-1)$ has a nonpositive sum. To prove this,we remark that as (j, k) is an interval of maximum sum, its sum is greater than or equal to that of the interval (i, k). Therefore:

$$\sum_{l=i}^{k} T[l] \leq \sum_{l=j}^{k} T[l] \Leftrightarrow \sum_{l=1}^{j-1} T[l] \leq 0.$$

Algorithm Algorithm 2.3 computes an interval of maximum sum that is based on the above properties. First, it looks for the first positive element in the array. If none is found, the solution is reduced to the largest element in the array. Otherwise, starting at the first positive element, it scans the array computing the sum of the current interval and updating the maximum sum if needed. Once an interval of nonpositive sum is encountered, using the last of the above properties, the algorithm skips the elements so far. Indeed, if a suffix of the current interval was a prefix of an interval of maximum sum, then the current interval would have a nontrivial, nonpositive prefix and the algorithm would have skipped it.

Algorithm 2.3: Find the maximum sum of contiguous elements of an array through a single array scan

```
$start \leftarrow 1$
$sum\_max \leftarrow T[1]$
$max\_elem \leftarrow 1$
while $T[start] < 0$ and $start < n$ do
    $start \leftarrow start + 1$
    if $T[start] > T[max\_elem]$ then
        $max\_elem \leftarrow start$
    end
end
if $start = n$ then
    return $(max\_elem, max\_elem, T[max\_elem])$
end
$sum\_max \leftarrow T[start]$
$end \leftarrow start$
$local\_max \leftarrow sum\_max$
$local\_start \leftarrow start$
for $i = start + 1$ to $n$ do
    if $local\_max + T[i] < 0$ then
        $local\_max \leftarrow 0$
        $local\_start \leftarrow i + 1$
    else
        $local\_max \leftarrow local\_max + T[i]$
        if $local\_max > sum\_max$ then
            $sum\_max \leftarrow local\_max$
            $start \leftarrow local\_start$
            $end \leftarrow i$
        end
    end
end
return $(start, end, sum\_max)$
```

Solution2

Algorithm 2.4: Kadane(T): Find the maximum sum of contiguous elements of an array through a single array scan

```
$max\_current = max\_global = T[1]$
$local\_start = local\_end = global\_start = global\_end = 1$
for $i = 2$ to $n$ do
    if $local\_max \leq 0$ then
        $local\_max = T[i]$
        $local\_start = local\_end = i$
    else
        $local\_max = local\_max + T[i]$
        $local\_end = i$
    end
    if $local\_max > global\_max$ then
        $global\_max = local\_max$
        $global\_start = local\_start$
        $global\_end = local\_end$
    end
end
return $(global\_start, global\_end, max\_global)$
```

Solution3

Please check Algorithm 2.2

3 Master Theorem

a) $a = 49, b = 7, d = 2, a = b^d, T(n) \in \Theta(n^2 \log n)$.

b) $a = \frac{1}{4} < 1$, which means there is less than one subproblem. MT is not applicable.

c) $f(n) = -n^2 - 2n$ is negative. MT is not applicable.

d) $a = 2, b = 4, d = 0.6, a < b^d, T(n) \in \Theta(n^{0.6})$

e) $a = 3, b = 2, d = 0, a > b^d, T(n) \in \Theta(n^{\log_2 3})$

4 DP: String Segmentation

a) Let $P(i)$ be the segmentation plausibility of string $y_1 ... y_i$. Observe that if the problem is nontrivial, i.e. $i \geq 1$, then to compute the segmentation plausibility $P(i)$ of string $y_1 ... y_i$, we must compute the segmentation plausibility $P(k)$ of the prefix substring $y_1 ... y_k$, and add it to the plausibility of substring $y_{k+1} ... y_i$, for some k in the range $0 \leq k < i$.
The optimal substructure of this problem is as follows. Suppose that to

optimally segment $y_1...y_i$, we segment the prefix $y_1...y_k$, and combine the resulting segmentation with the rest of the string, i.e. $y_{k+1}...y_i$. Then the way we segment the prefix $y_1...y_k$ must result in an optimal segmentation plausibility for $y_1...y_k$. Assume that there exists a different segmentation of $y_1...y_k$ with superior segmentation plausibility. Then, we can substitute the plausibility of that segmentation in the optimal segmentation plausibility of $y_1...y_i$, to produce another segmentation of $y_1...y_i$ whose segmentation plausibility is higher than the optimum, a contradiction.

b) Recurrence:

$$P(i) = \begin{cases} 0 & i = 0 \\ \max\limits_{0 \leq k < i} \{P(k) + \mathit{plausibility}(y_{k+1}...y_i)\} & \text{otherwise} \end{cases}$$

The final answer is stored in $P[n]$.

c) Recursive algorithm with memoization: call $\mathit{SegPlausibilityTD}(n)$

Algorithm 4.1: *SegPlausibilityTD*

```
Input: Index $i$ into string $y_1...y_n$
Output: Optimal segmentation plausibility $Q[n]$
1 if $i = 0$ then
2     return 0
3 end
4 if $Q[i]$ is not empty then
5     return $Q[i]$
6 else
7     $Q[i] = \max_{0 \leq k < i} \{SegPlausibilityTD(k) + plausibility(y_{k+1}...y_i)\}$
8     return $Q[i]$
9 end
```

Space complexity: $O(n)$
The array Q stores exactly n values.

Time complexity: $O(n^2)$
Each subproblem corresponds to a possible prefix, of which there are n. For each such prefix, we iterate over k to find the optimal segmentation which requires $O(n)$ operations, resulting in $O(n^2)$ time complexity.

d) Bottom-up algorithm: call $\mathit{SegPlausibilityBU}(n)$

Algorithm 4.2: *SegPlausibilityBU*

```
Input: String $y_1...y_n$
Output: Optimal segmentation plausibility $Q[n]$
1 $Q[0] = 0$
2 for $i = 1 to\ n$ do
3     $Q[i] = \max_{0 \leq k < i} \{Q[k] + plausibility(y_{k+1}...y_i)\}$
4 end
5 return $Q[n]$
```

An alternative DP formulation for the problem, with time and space complexities $O(n^3)$ and $O(n^2)$, respectively, has the following recurrence relation:

$$P(i,j) = \begin{cases} 0 & j > i \text{ (empty string)} \\ plausibility(y_i) & i = j \\ \max\limits_{i \le k < j} \{plausibility(y_i ... y_j), P(i,k) + P(k+1,j)\} & \text{otherwise} \end{cases}$$

The corresponding recursive algorithm with memoization: call $SlowSegPlausibilityTD(1, n)$

Algorithm 4.3: *SlowSegPlausibilityTD*

```
Input: Indices $i, j, i \geq j$ into string $y_1 ... y_n$
Output: Optimal segmentation Plausibility $Q[1][n]$
if $j > i$ then
    return 0
end
if $i = j$ then
    return $plausibility(y_i)$
end
if $Q[i][j]$ is not empty then
    return $Q[i][j]$
end
else
    $Q[i][j] = \max_{i \le k < j} \{plausibility(y_i ... y_j), SlowSegPlausibilityTD(i,k) +$
      $SlowSegPlausibilityTD(k+1, j)\}$
    return $Q[i][j]$
end
```

The other bottom-up algorithm: call $SlowSegPlausibilityBU(n)$

Algorithm 4.4: *SlowSegPlausibilityBU*

```
Input: String $y_1 ... y_n$
Output: Optimal segmentation Plausibility $Q[1][n]$
$Q[1][0] = 0$
for $i = 1$ to $n$ do
    $Q[i][i] = plausibility(y_i);$
end
for $d = 1$ to $n - 1$ do
    for $i = 1$ to $n - d$ do
        $j = i + d$
        $Q[i][j] = \max_{i \le k < j} \{plausibility(y_i ... y_j), plausibility(y_i ... y_k) +$
          $plausibility(y_{k+1} ... y_j)\}$
    end
end
return $Q[1][n]$
```