Prof. Sergio A. Alvarez
21 Campanella Way, room 569
Computer Science Department
Boston College
Chestnut Hill, MA 02467 USA

http://www.cs.bc.edu/∼alvarez/
alvarez@cs.bc.edu
voice: (617) 552-4333
fax: (617) 552-6790

CS383, Algorithms
Spring 2009
HW6 Solutions

1. The computational task of computing the maximum consecutive subsequence sum (MSS) of a numerical array of length n is described below, together with a simple algorithm (Algorithm 1) that performs this task. For example, the correct output for the MSS task for the input instance $a = \{-5, 3, 12, -8, 6, 4\}$ is $S = 17$, $\underline{i} = 2$, $\bar{i} = 6$. Your main goal will be to develop an efficient divide and conquer algorithm for the MSS task.

```
Algorithm 1: Simple Maximum Consecutive Subsequence Sum
Input: Nonempty array $a[1...n]$ of numbers.
Output: Maximum consecutive sum $S = \sum_{\underline{i} \leq i \leq \bar{i}} a[i]$, associated indices $\underline{i}$, $\bar{i}$.
SimpleMSS($a[1...n]$)
(1)   $S = a[1]$, $\underline{i} = 1$, $\bar{i} = 1$
(2)   foreach $i = 1$ to $n$
(3)       foreach $j = i$ to $n$
(4)           $S' = \sum_{i \leq k \leq j} a[k]$
(5)           if $S' > S$
(6)               $S = S'$, $\underline{i} = i$, $\bar{i} = j$
(7)   return $S, \underline{i}, \bar{i}$
```

(a) Find the worst-case asymptotic running time of Algorithm 1 in big Θ notation. Explain carefully.

Solution

Let n denote the length of the input array $a[1...n]$. Ignoring the time required for arithmetic operations (they are $\Theta(1)$ for operands that fit within a machine word), the asymptotic running time of Algorithm 1 is determined by the number of evaluations of $a[k]$ in line 4. This number is the number of elements in the following set of index triples:

$$\{(i, j, k) | 1 \leq i \leq n, \quad i \leq j \leq n, \quad i \leq k \leq j\}$$

Since the number of values of k between i and j is $j - 1 + 1$, the total number of index triples equals

$$\sum_{1 \leq i \leq n} \sum_{i \leq j \leq n} (j - i + 1) = \sum_{1 \leq i \leq n} \sum_{1 \leq j' \leq n-i+1} j' = \sum_{1 \leq i \leq n} (n - i + 1)(n - i + 2)/2$$

Summing over all values of i, the result is a polynomial in n with leading term $n^3/6$. It follows that Algorithm 1 runs in time $\Theta(n^3)$.

(b) Implement (in detailed pseudocode) two functions **MSLeft**$(a[1...n])$ and **MSRight**$(a[1...n])$ that run in time $\Theta(n)$ and that return the maximum consecutive subsequence sums (and corresponding index ranges) over ranges of the restricted forms $i...\lfloor n/2\rfloor$ and $\lfloor n/2\rfloor+1...i$, respectively (note that these ranges reach the midpoint of the array). For example, for the input array $a = \{-5, 3, 12, -8, 6, 4\}$, **MSLeft**$(a)$ and **MSRight**(a) would return the values $S = 15$, $\underline{i} = 2$, $\bar{i} = 3$ and $S = 2$, $\underline{i} = 4$, $\bar{i} = 6$, respectively.

Solution

Algorithm 2: Boundary Sum from Below
Input: Array $a[1...n]$.
Output: Solution S, $\underline{i}$, $\bar{i}$ to *restricted* MSS task for lower half $a[1...\lfloor n/2\rfloor]$, with $1 \le \underline{i} \le \bar{i} = \lfloor n/2\rfloor$.

```
MSSLeft(a[1...n])
(1)  $S = a[\lfloor n/2\rfloor]$, $\underline{i} = \bar{i} = \lfloor n/2\rfloor$
(2)  foreach $i = \lfloor n/2\rfloor - 1...1$
(3)      if $S + a[i] > S$
(4)          $S = S + a[i]$, $\underline{i} = i$
(5)  return $S, \underline{i}, \bar{i}$
```

Algorithm 3: Boundary Sum from Above
Input: Array $a[1...n]$.
Output: Solution S, $\underline{i}$, $\bar{i}$ to MSS task for upper half $a[\lfloor n/2\rfloor + 1...n]$, with $\lfloor n/2\rfloor + 1 = \underline{i} \le \bar{i} \le n$.

```
MSSRight(a[1...n])
(1)  $S = a[\lfloor n/2\rfloor + 1]$, $\underline{i} = \bar{i} = \lfloor n/2\rfloor + 1$
(2)  foreach $i = \lfloor n/2\rfloor + 1...n$
(3)      if $S + a[i] > S$
(4)          $S = S + a[i]$, $\bar{i} = i$
(5)  return $S, \underline{i}, \bar{i}$
```

(c) Design a $\Theta(n)$ gluing function for a divide and conquer algorithm for the MSS task. Provide detailed pseudocode. Gluing inputs include the MSS solutions S_L, $\underline{i}_L$, $\bar{i}_L$ and S_H, $\underline{i}_H$, $\bar{i}_H$ for the lower and upper halves $a[1...\lfloor n/2\rfloor]$ and $a[\lfloor n/2\rfloor + 1...n]$, respectively, as well as the full array $a[1...n]$. For example, for $a = \{-5, 3, 12, -8, 6, 4\}$, one would have $S_L = 15$, $\underline{i}_L = 2$, $\bar{i}_L = 3$ and $S_H = 10$, $\underline{i}_H = 5$, $\bar{i}_H = 6$. The output of the gluing function should be a solution S, $\underline{i}$, $\bar{i}$ to the MSS task on the full array $a[1...n]$. Your gluing function may call the functions defined in 1b.

Solution

See Algorithm 4.

(d) Write detailed pseudocode for a divide and conquer algorithm that solves the MSS task using the gluing function from 1c. What is your algorithm's running time? Explain.

Algorithm 4: Gluing Step
Input: Array $a[1...n]$; MSS half-solutions S_L, $\underline{i}_L$, $\bar{i}_L$, S_H, $\underline{i}_H$, $\bar{i}_H$ (see above).
Output: Global solution S, $\underline{i}$, $\bar{i}$ to MSS task for full array $a[1...n]$.

```
GLUE($a[1...n], S_L, \underline{i}_L, \bar{i}_L, S_H, \underline{i}_H, \bar{i}_H.$)
(1)   $(S_{ML}, \underline{i}_{ML}, \bar{i}_{ML}) = \text{MSL}\textsc{eft}(a[1...n])$
(2)   $(S_{MH}, \underline{i}_{MH}, \bar{i}_{MH}) = \text{MSR}\textsc{ight}(a[1...n])$
(3)   $S_M = S_{ML} + S_{MH}$
(4)   if $S_M \geq S_L$ and $S_M \geq S_H$
(5)       return $(S_M, \underline{i}_{ML}, \bar{i}_{MR})$
(6)   else if $S_L \geq S_M$ and $S_L \geq S_H$
(7)       return $(S_L, \underline{i}_L, \bar{i}_L)$
(8)   else
(9)       return $(S_H, \underline{i}_H, \bar{i}_H)$
```

Solution

See Algorithm 5. Since the gluing step (Algorithm 4) runs in time $\Theta(n)$, the running time of Algorithm 5, $T(n)$, satisfies the recurrence relation

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$$

By the Master Theorem (critical gluing time $a = 2$, $b = 2$, $p = 1$), we see that the asymptotic running time of Algorithm 5 is

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

Algorithm 5: Divide and Conquer MSS
Input: Maximum consecutive sum $S = \sum_{\underline{i} \leq i \leq \bar{i}} a[i]$, associated indices $\underline{i}$, $\bar{i}$.
Output: Array of numbers $a[1...n]$.

```
DCMSS($a[1...n]$)
(1)   if $n = 0$
(2)       return 0
(3)   if $n = 1$
(4)       return $a[1]$
(5)   $(S_L, \underline{i}_L, \bar{i}_L) = \text{DCMSS}(a[1...\lfloor n/2 \rfloor])$
(6)   $(S_R, \underline{i}_R, \bar{i}_R) = \text{DCMSS}(a[\lfloor n/2 \rfloor + 1...n])$
(7)   return  $\text{GLUE}(a[1...n], (S_L, \underline{i}_L, \bar{i}_L), (S_R, \underline{i}_R, \bar{i}_R))$
```

2. Apply Dijkstra's algorithm to the weighted graph G that has the vertices A, B, C, D, E, F and the edge weights given below in adjacency list format (edges not listed have weight ∞). Let A be the start vertex.

$$\begin{aligned}
&w(A,B) = 20,\ w(A,C) = 30,\ w(A,F) = 100\\
&w(B,C) = 30,\ w(B,D) = 20,\ w(B,F) = 100\\
&w(C,D) = 40,\ w(C,E) = 15,\ w(C,F) = 60\\
&w(D,E) = 50,\ w(D,F) = 5\\
&w(E,F) = 60
\end{aligned}$$

Provide a complete iteration table that shows the contents of the distance and predecessor arrays at each stage, and that indicates what vertex is selected for expansion in each case.

Solution

Each entry in the following table shows the distance and predecessor values for the vertex in the corresponding column. The rightmost column shows what vertex is expanded as a result of the distance updates in that row.

A	B	C	D	E	F	expand
0, null	∞, null	∞, null	∞, null	∞, null	∞, null	A
0, null	$20, A$	$30, A$	∞, null	∞, null	$100, A$	B
0, null	$20, A$	$30, A$	$40, B$	∞, null	$100, A$	C
0, null	$20, A$	$30, A$	$40, B$	$45, C$	$90, C$	D
0, null	$20, A$	$30, A$	$40, B$	$45, C$	$45, D$	E
0, null	$20, A$	$30, A$	$40, B$	$45, C$	$45, D$	F

3. You're starting a new business that requires delivering orders from your warehouse to customers in various rural towns. You have a fleet of delivery trucks and a known grid of roads that they can travel on. There is a single road between every pair of towns. Road conditions sometimes shut down travel along a stretch of road. Such events are unpredictable, but you have for each road an estimate of the probability that the road will be closed at any given time. You are interested in determining routes that will maximize the probability that a delivery attempt will not run into a closed road anywhere along the route. Assuming that road closings occur independently of one another, this "all clear" probability along a route $t_0, t_1, ...t_k$, where the t_i are the towns along the route, is the *product* of the individual probabilities $p_{t_i,t_{i+1}}$ that each of the roads between consecutive towns t_i and t_{i+1} will be open.

   (a) You wish to compute, for each of the n towns $1, ...n$, the route from the warehouse to that town that maximizes the all clear probability described above. The probability $p_{i,j}$ that the road from town i to town j will be clear (open) is known in advance for every pair of towns. The warehouse is considered to be town 0. Cast this as a computational task involving a graph. Precisely describe the graph involved, and give a detailed input-output specification of the computational task. Make explicit note of any constraints on the inputs.

Solution

The relevant graph here is a directed weighted graph with vertices $0...n$ and an edge from i to j with edge weight $p_{i,j}$ for every pair of vertices (i, j). The target computational task involving this graph is described below. I'll use the term *multiplicative value* for the product of the individual edge weights along a path in a weighted graph. A *path of greatest multiplicative value* between a given pair of vertices is one that has a multiplicative value that is not less than that of any other path between the same vertices.

Algorithm 6: Greatest Multiplicative Value Paths
Input: Weighted graph G with vertices $0...n$ and edge weights $p_{i,j}$ with $0 \le p_{i,j} \le 1$.
Output: Array pred[1...n] such that for each vertex $v \neq 0$ of G, there is a path from 0 to v of greatest multiplicative value that visits pred[v] immediately before v.

(b) Describe an algorithm that solves the computational task described in 3a. Provide detailed pseudocode with explanations.

Solution

The pseudocode appears below. We transform the graph G slightly and apply the standard Dijkstra single-source shortest paths algorithm to the modified graph G'. The transformation simply replaces each edge weight by the negative of itslogarithm (which results in a non-negative value since the original edge weights are all less than or equal to 1). Since the log of a product is the sum of the logs of the factors, this transformation has the property that the multiplicative value of a path in the original graph may be obtained as the inverse logarithm of the standard additive path cost in the transformed graph. Furthermore, because of the negative sign, a path has greatest multiplicative value in the original graph G if and only if the path has *least* cost in the transformed graph G'. Hence, the standard Dijkstra algorithm applied to the transformed graph finds the desired paths of greatest multiplicative value in the original graph.

Algorithm 7: Dijkstra for Paths of Greatest Multiplicative Value
Input: Weighted graph G with vertices $0...n$ and edge weights $p_{i,j}$ with $0 \le p_{i,j} \le 1$.
Output: Array pred[1...n] such that for each vertex $v \neq 0$ of G, there is a path from 0 to v of greatest multiplicative value that visits pred[v] immediately before v.

```
productDijsktra($G$)
(1)    Define $G'$ to be a graph with vertices $0...n$ and edge weights $-\log p_{i,j}$, $i, j = 0...n$.
(2)    return standardDijkstra($G'$, 0)
```

4. Suppose that a store offers that for each item purchased at full price, you can get another item of the same price or less for free. You would like to buy a total of $2n$ items, in a way that maximizes the money saved through this offer. You know the price of each item, and you wish to pair them in the form (P_1, F_1), (P_2, F_2), $\cdots (P_n, F_n)$, where you pay for item P_i and get item F_i for free based on the purchase of P_i (so F_i costs no more than P_i).

   (a) Describe an algorithm that takes a list of $2n$ items and their prices as inputs, and returns an optimal pairing (P_1, F_1), (P_2, F_2), $\cdots (P_n, F_n)$. Optimality here means that the total amount paid should be minimized (equivalently, the total amount saved should be maximized). Give detailed pseudocode.

Solution

Algorithm 8 describes a simple greedy solution for this task: simply sort the items in decreasing order of price, and pair every two consecutive items, starting with the two highest priced items.

Algorithm 8: Maximum Savings on Pay One Item, Get Second Item Free

Input: Even-length array $I[1...2n]$ of pairs $(x, \text{price}(x))$ representing items x with their prices.

Output: Pairs$[1...n]$ with Pairs$[i] = (P_i, F_i)$, where P_i is to be purchased at full price and F_i ($\text{price}(F_i) \leq \text{price}(P_i)$) is free, such that the savings $\sum_{i=1}^{n} \text{price}(F_i)$ is maximized by this pairing.

```
maxSavings($I[1...2n]$)
(1)    sort $I$ in decreasing order of price
(2)    foreach $i = 1...n$
(3)        Pairs$[i] = (I[2i-1], I[2i])$
(4)    return Pairs
```

(b) Prove that your algorithm from 4a returns an optimal solution. Give a carefully reasoned and complete explanation.

Solution

It is clear that the total discount depends only on which items are discounted. Thus, we focus our attention on the indices of the discounted items in the sorted array. We call such indices the *discounted indices*, and we list them in increasing order as $i_1 < i_2 < ... < i_n$. Each pairing of the items produces a set of discounted indices. The pairing produced by Algorithm 8 satisfies $i_k = 2k$ for all $k = 1, .., n$. We claim that for *any* admissible pairing, one must have $i_k >= 2k$ for all k. We will prove this below. Observe that this claim implies that the proposed pairing is optimal, since the value of the i-th item is a decreasing function of the index i, so that the savings associated with any admissible pairing will be less than the savings associated with the proposed pairing, which achieves the minimum allowable values of the discounted indices.

Proving that $i_k >= 2k$ for any admissible pairing is quite easy. Admissibility reduces to the constraint that each discounted index be paired with a nondiscounted index corresponding to an item of equal or greater value, i.e. with a nondiscounted index which is less than or equal to the given discounted index. Thus, since there are precisely k discounted indices which are less than or equal to the k-th discounted index i_k, there must also be at least k associated nondiscounted indices less than or equal to i_k. But the total number of nondiscounted indices in the range $1, ..., i_k$ is precisely $i_k - k$. We must therefore have $i_k - k >= k$, or $i_k >= 2k$.