U C L A

Computer Science Department

CS 180 **Algorithms & Complexity** **Fall 04-05**

Instructor: **Majid Sarrafzadeh**
3532C Boelter Hall
majid@cs.ucla.edu
www.cs.ucla.edu/~philip/cs180

Hwk 4: assigned 10/28, due 11/4 (in class; 10 am)

1. Problem 15.2-5 on Page 339. Show that a

Show that a full parenthesization of an n-element expression has exactly n – 1 pairs of parentheses.

We prove this by induction.

Induction Hypothesis: Suppose that for $n \leq k$, a full parenthesization of an n-element expression has exactly n – 1 pairs of parentheses.

Basis: Let n = 2. In other words, we multiply two matrices, A_1 and A_2: The (unparenthesized) product is A_1A_2. Since there is only one multiplication operation, there is only one parenthesization (A_1A_2).

Induction: We consider a product of n = k+1 matrices, $A_1, A_2, \ldots, A_k, A_{k+1}$. Now, suppose that the optimal parenthesization splits the product at the jth element. That is, the optimal solution has the recursive structure:

$$(\!(A_1A_2\ldots A_j)(A_{j+1}A_{j+2}\ldots A_{k+1})\!)$$

where the two sub-products $(A_1A_2\ldots A_j)$ and $(A_{j+1}A_{j+2}\ldots A_{k+1})$ are recursively solved optimally. By the induction hypothesis, the two sub-products require j – 1 and ((k+1) – j + 1) – 1 parenthesis respectively. Taken together, the induction hypothesis allows us to compute number of parentheses required to fully parenthesize the two sub-products is:

$$(j - 1) + (((k+1) - j + 1) - 1) = k - 1$$

Finally, we must add the outermost parenthesis (shown in bold above), which indicate that the two sub-products $(A_1A_2\ldots A_j)$ and $(A_{j+1}A_{j+2}\ldots A_{k+1})$ are to be multiplied as well. The total number of parentheses is $(k - 1) + 1 = k$. Therefore, the induction hypothesis holds for $n = k + 1$.

2. Problem 1 on Page 364, at the end of Dynamic Programming chapter (Bitonic Euclidean Traveling Salesman)

The Euclidean traveling-salesman problem is the problem of determining the shortest closed tour that connects a given set of n points in the plane. The general problem is NP-complete, and the solution is therefore believed to require more than polynomial time.

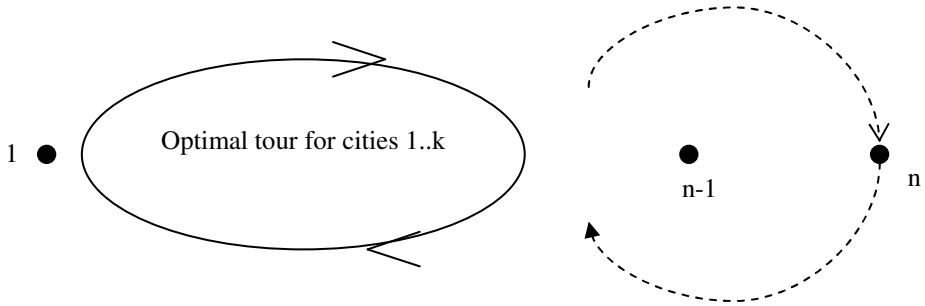
J. L. Bentley has suggested that we simplify the problem by restricting our attention to bitonic tours, that is, tours that start at the leftmost post, go strictly left to the right to the rightmost point, and then go strictly right to left back to the starting point. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate. (Hint: Scan left-to-right, maintaining optimal possibilities for the two parts of the tour).

Solution:

Begin by sorting the cities based on their x-coordinates in $O(n^2)$-time (or less).

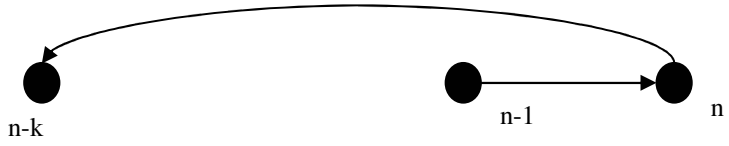
The best way to begin to solve this problem is to think about it inductively. In particular, assume that we have an optimal tour for cities 1 through n-1, and consider the possibility of extending this tour to an nth city. This is illustrated graphically as follows:



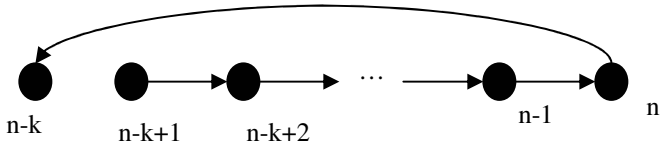
In particular, we must consider the incoming and outgoing edges to city n. Because the tour is bitonic, we know that the tour will contain either edge (n-1, n) or edge (n, n-1). Since the same tour may be observed in either clockwise our counter-clockwise direction, both cases are actually equivalent. Thus, it is sufficient to consider the case where edge (n-1, n) is included.

Where does our salesman go after he leaves n? We cannot know immediately. He may travel to city n-2... but on the other hand, may (n-2, n-1) is a link on the path from city 1 to city n-1.

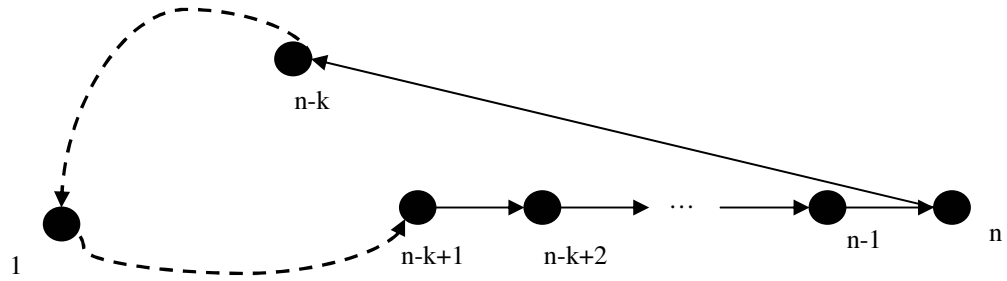
In general, we assume that our salesman travels from city n to city n – k, where k is an integer subject to $1 < k \leq n - 1$. What does this tell us about the tour? Consider the following picture:



Since the tour is bitonic, we know that we cannot double-back to cities n-k+1, n-k+2, …, n-2 after leaving city n. Consequently, all of these cities must precede city n-1 in order. This is illustrated as follows:



Now, we ask ourselves: what does the rest of the tour consist of? The answer is simple: the optimal path from city n – k to city n – k + 1 that passes through city 1. In other words, our optimal bitonic tour has the following structure:



Dashed lines represent optimal path from city n – k to n – k + 1 via city 1.

The next step is to determine how to select city n – k. There are a total of n – 2 possibilities, since any city other than n – 1 or n itself may be selected.

We define the following variables:

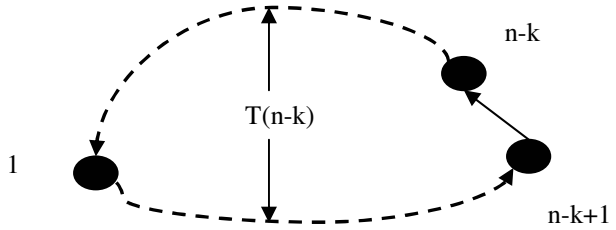
d(i, j) is the distance between two cities i and j.

T(i) is the total distance of the shortest path originating at vertex i, passing through vertex 1, and ending at vertex i+1.

The directed path in the above figure from city n – k + 1 through city n is given by the following formula:

$$d(n-k+1 \to n) = \sum_{i=n-k+1}^{n-1} d(i, i+1)$$

One more observation is necessary. Consider the following figure.



Optimal Bitonic Tour from city 1 to n-k+1

This figure demonstrates that the optimal path (from vertex n-k to n-k+1 via vertex 1) we are looking for can be constructed from the optimal bitonic tour for city n –k+1.

In general, let B(i) be the cost of the optimal bitonic tour for cities 1 through i. Then:

B(i) = T(i-1) + d(i, i-1), or equivalently
T(i-1) = B(i) – d(i, i-1).

We must select a city n - k must be selected according to the following formula, which minimizes B(n):

$$B(n) = \min_{2 \le k \le n-1} \{T(n-k) + d(n-k+1 \to n) + d(n, n-k)\}$$

Substituting the above formula, we recursively define B(n) as follows:

$$B(n) = \min_{2 \le k \le n-1} \{B(n-k+1) - d(n-k+1, n-k) + d(n-k+1 \to n) + d(n, n-k)\}$$

Algorithm: Bitonic-Euclidean-TSP

Input: C[1..n] – a list of cities
C[i].x – the x-coordinate of the ith city
C[i].y – the y-coordinate of the ith city

Output: B[2..n] – length of the bitonic Euclidean tour for the nth city

Variables: P[i, j] – length of the path i → i + 1→ … → j.

Auxiliary Functions:

Distance(C, i, j)

Return $\sqrt{(C[i].x - C[i+1].x)^2 + (C[i].y - C[i+1].y)^2}$

Compute-Path-Lengths(C)

```
1.  For i ← 1 to n
2.      P[i, i+1] = Distance(C, i, i+1)
3.      For j ← i+2 to n
4.          P[i, j] = P[i, j-1] + Distance(C, i, j)
5.  Return P
```

Pseudocode:

Bitonic-Euclidean-TSP(C)

```
1.  Sort the cities using an O(nlgn) sort based on their x-coordinates (C[i].x)
2.  P ← Compute-Path-Lengths(C)
3.  B[2] = Distance(C, 1, 2)
4.  B[3] = Distance(C, 1, 2) + Distance(C, 2, 3) + Distance (C, 3, 1)
5.  For i ← 4 to n
6.      q ← ∞
7.      For k ← n-1 downto 2
8.          q'← B[i-k+1] – Distance(C, i-k+1, i–k) +
                P[i–k+1, i] + Distance(C, i, i – k)
9.          If q' < q
10.             q ← q'
11.     B[i] ← q
12. Return B[n]
```

This algorithm minimizes the total distance of the bitonic Euclidean tour; it does not, however, output the ordering of cities which comprises the tour. For an exercise, you should modify the algorithm to print out the cities in order.

It should be trivial to show that the time complexity of this algorithm is $O(n^2)$.

Note: Because the Traveling Salesman Problem is NP-complete, this yields only an approximate solution. This algorithm is not guaranteed to find the best solution.

3. Problem 2 on Page 364 at the end of Dynamic Programming chapter (Printing Neatly)

Let $l_1, l_2, \ldots, l_n$ be the lengths of the n words. We solve the problem by first defining two help functions. The first function, *cost(i, j),* is the extra free space if the words i through j are put on the same line. *cost(i, j)* will be negative if the words i through j don't fit on one line. The function is defined as

$$\cos t(i, j) = M - j + i + \sum_{k=i}^{j} l_k$$

The other help function is *linecost(i, j)* that give the cost of placing the words i through j on one line. We define *linecost* as infinite if the words don't fit. We define *linecost* as

$$line\cos t(i, j) = \begin{cases} \infty & if\ \cos t(i, j) < 0 \\ 0 & if\ (j = n) lastline \\ \cos t(i, j)^3 & otherwise \end{cases}$$

We can now solve the problem by dynamic programming. Let *t(i)* be the cost of an optimal arrangement of words *1, 2, . . . , i.* The optimal solution will have the property that if a line-break is placed after word k, then both the words *1, . . . , k* as well as the words k + 1, . . . , n are placed optimally. Therefore we can build up the solution array starting from 1 and moving up to n as

$$t(j) = \min_{j=1}^{i} \{t(j-1) + line\cos t(j, i)\}$$

using

$$t(0) = 0$$

as base case.

The program looks as follows. Note that we only compute the values of cost and linecost that we actually need in order not to spend more time than necessary.

```
Printing Neatly(l)
(1)     c(0) = 0
(2)             for each i = 1, . . . , n
(3)                     c(i) = ∞
(4)                     j = i
(5)                     cont = true
(6)                     cost(i, i) = M − l(i)
(7)                     while cont
(8)                             if cost(j, i) < 0
(9)                                     linecost(j, i) = ∞
(10)                            else if i = n
(11)                                    linecost(j, i) = 0
(12)                            else
(13)                                    linecost(j, i) = cost(j, i)^3
```

```
(14)                    c(i) = min(c(i), linecost(j, i)+c(j−1))
(15)                    cost(j − 1, i) = cost(j, i) − l(j) − 1
(16)                    j = j − 1
(17)                    cont = cost(j, i) >= 0
(18)        return c(n)
```

The main loop will run n iterations. In each iteration we will at most go $\lceil M / 2 \rceil$ steps since that is the maximum number of words that can fit into one line. The total complexity is *O(Mn)*.

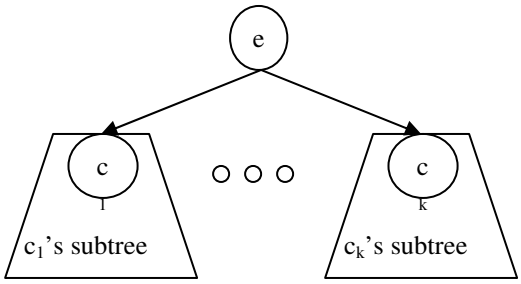
4. Problem 4 on Page 367 at the end of Dynamic Programming chapter (Planning a Company Picnic)

Professor Stewart is consulting for the president of a corporation that is planning a company party The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

Solution:

The key to understanding this problem is to take advantage of the tree structure of the organization. We observe that each employee is the root of a sub-tree of the organization chart.



Let e be an employee rooted at a sub-tree s. c_1, …, c_k be e's children, each of which is the root of its own subtree. With respect to the inclusion of e, there are two possibilities:

(1) e is invited to the party in the optimal solution for e's subtree.
(2) e is not invited to the party in the optimal solution for e's subtree.

Let $R_{(1)}[e]$ be the optimal conviviality rating of e's subtree under (1).
Let $R_{(2)}[e]$ be the optimal conviviality rating of e's subtree under (2).
Let conv[e] be the conviviality rating of employee e.

We develop recurrence relations for each of these cases as follows:

Case (1):

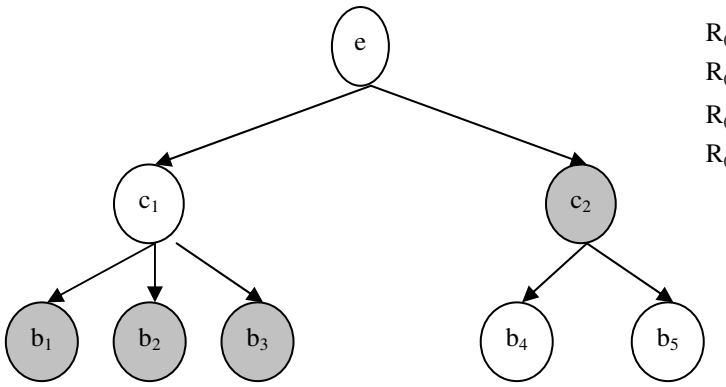
Since e is invited the party, we know that none of c_1, …, c_k can be invited. Then the best we can do for the respective subtree of child c_i is $R_{(2)}[c_i]$, which is the optimal solution for ci's subtree under assumption that ci is not included.

$$R_{(1)}[e] = conv[e] + \sum_{i=1}^{k} R_{(2)}[c_i]$$

If e is a leaf node, then $R_{(1)}[e] = conv[e]$.

Case (2):

Since e is not invited to the party, we may or may not invite any of e's children. We only invite child c_i if c_i is invited under the optimal solution of the subtree rooted at c_i.



$R_{(1)}[c_1] = 2$ $R_{(2)}[c_1] = 5$ $R_{(1)}[c_2] = 7$ $R_{(2)}[c_2] = 6$

$conv[e] = 1$ $conv[c_1] = 2$ $conv[c_2] = 7$ $conv[b_1] = 1$ $conv[b_2] = 2$ $conv[b_3] = 2$ $conv[b_4] = 4$ $conv[b_5] = 2$

When e is not included in a solution, we may or may not want to include its children.

We define $R_{opt}[e]$ to be the optimal conviviality rating of employee e, which is the maximum of $R_{(1)}[e]$, and $R_{(2)}[e]$:

$$R_{opt}[e] = \max\{R_{(1)}[e], R_{(2)}[e]\}$$

The recurrence relation is as follows:

$$R_{(2)}[e] = \sum_{i=1}^{k} R_{opt}[c_i]$$

If e is a leaf node, then $R_{(2)}[e] = 0$.

Algorithm: Company-Party

Inputs: n – number of employees.
conv[1..n] – the conviviality rating of each employee
supervisor[1..n] – the supervisor vertex of each vertex
(the supervisor of the root is NIL)
child[1..n][1..k] – the children of each vertex (k is the number of children and varies from vertex to vertex).

Vars: $R_{(1)}[1..n]$, $R_{(2)}[1..n]$ – defined above
num-children[1..n] – the number of children of each vertex.
mark[1..n] – a Boolean variable used to mark each vertex true or false.
name[1..n] – the name of each employee
Q – a queue of vertices.

Output: $R_{opt} = \max\{R_{(1)}[r], R_{(2)}[r]\}$, where r is the root of the organization.
invited[1..n] = true if employee is invited; false otherwise.

```
1.  For i ← 1 to n
2.      mark[i] = false
3.  For i ← 1 to n
4.      if num-children[i] = 0
5.          R(1)[i] = conv[i]
6.          R(2)[i] = 0
7.          If mark[supervisor[i]] = false
8.              Q.enqueue(supervisor[i])
9.              mark[supervisor[i]] ← true
10. While Q is not empty
11.     root = Q.dequeue()
12.     r1← conv[root]
13.     r2← 0
14.     For j ← 1 to num-children[root]
15.         c← child[root][j]
16.         r1← r1+ R(2)[c]
17.         r2← r2+max{R(1)[c], R(2)[c]}
18.     R(1)[root] ← r1
19.     R(2)[root] ← r2
20.     If mark[supervisor[root]] = false and supervisor[root] not NIL
21.         Q.enqueue(supervisor[root])
22.         mark[supervisor[root]] ← true
```

```
// Output routine
1.  R_opt ← max{R_(1)[root], R_(2)[root]}
2.  For i ← 1 to n
3.      mark[i] = false
4.  Q.enqueue(root)
5.  While Q is not empty
6.      e ← Q.dequeue
7.      m ← max{R_(1)[e], R_(2)[e]}
8.      If m = R_(1)[e] and mark[e] = false
9.          invited[e] = true
10.         For j ← 1 to num-children[e]
11.             c ← child[e][j]
12.             invited[c] ← false
13.             mark[c] ← true
14.             Q.enqueue(c)
15.     Else
16.         invited[e] = false
17.         For j ← 1 to num-children[e]
18.             Q.enqueue(c)
```

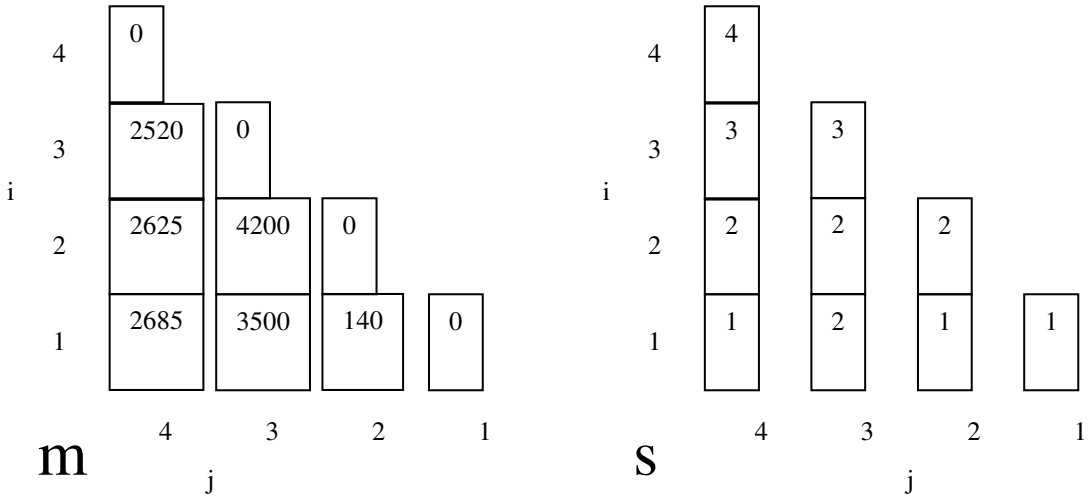
The time complexity of this algorithm is O(n).

1. The for-loops comprising lines 1-9 count from 1 to n.
2. The while-loop in lines 10-22 iterates as long as there are employees in the queue. Each employee e is enqueued only once, which is ensured by setting mark[e] to true when e is enqueued the first time. The internal for-loop visits all the children of each employee. Since each employee has one supervisor, each employee (except for the root) will be encounted exactly once during the while loop. Consequently, this while loop has O(n) time.
3. The for-loop in lines 24-25 counts from 1 to n.
4. The while-loop in lines 10-22 is analogous to the while-loop described in 2. It runs in O(n) time as well.

Taken together, 1, 2, 3, and 4 show that the algorithm runs in O(n) time.

5. (a). Use Dynamic Programming to find the best way to multiply the following matrices (only the dimensions are shown): (4x5)(5x7)(7x120)(120x3).

We use the dynamic programming algorithm to build the tables m and s. Note that the right-to-left ordering of the bottom subscript.



By examining m[1, 1], we see that the optimal matrix chain multiplication for this problem requires 2685 scalar multiplication operations. By examining s[1, 1], we see that the optimal parenthesization of $A_{1..4}$ has the form $(A_1(A_{2..4}))$. By examining s[2, 4], we see that the optimal parenthesization of $A_{2..4}$ has the form: $(A_2(A_{3..4}))$. By examining s[3, 4], we see that the optimal parenthesization of $A_{3..4}$ has the form: (A_3A_4). Therefore, the optimal parenthesization of the entire chain multiplication is $(A_1(A_2(A_3A_4)))$.

(b). Prove that the following algorithm is optimal or give a counterexample to its optimality. Take the two matrices whose "middle rank" is maximum (in the above example take ((7x120) and (120x3) whose middle rank is 120). Multiply those two matrices. Repeat the process until you are left with

We give a counterexample to prove that the algorithm described is sub-optimal.

Example: $A_1 = (4x7)$ $A_2 = (7x5)$ $A_3 = (5x120)$ $A_4 = (120x3)$

i. Sub-optimal solution using the algorithm above.

Since 120 is the largest middle rank, we multiply A_3 and A_4. This requires 5 x 120 x 3 = 1800 scalar multiplication operations. The list of partially multiplied matrices is updated as follows:

$A_1 = (4x7)$ $A_2 = (7x5)$ $A_{3..4} = (5x3)$

Since 7 is now the largest middle rank, we multiply A_1 and A_2. This requires 4 x 7 x 5 = 140 scalar multiplication operations. The list of partially multiplied matrices is updated as follows:

$A_{1..2} = (4x5)$ $A_{3..4} = (5x3)$

Since 5 is the only choice for middle rank, we multiply $A_{1..2}$ and $A_{3..4}$. This requires 4 x 5 x 3 = 60 scalar multiplication operations. We are left with a single matrix $A_{1..4} = (4x3)$.

The total number of scalar multiplication operations is 1800 + 140 + 60 = 2000 operations. The parenthesized chain is as follows: $((A_1A_2)(A_3A_4))$.

ii. Optimal solution using dynamic programming.

Solving the problem using dynamic programming yields the parenthesized solution $(A_1(A_2(A_3A_4)))$. Multiplying A_3 x A_4 requires 5 x 120 x 3 = 1800 scalar multiplication operations. Multiplying A_2 x $A_{3..4}$ requires 7 x 5 x 3 = 105 scalar multiplication operations. And multiplying A_1 x $A_{2..4}$ requires 4 x 7 x 3 = 84 scalar multiplication operations.

The total number of scalar multiplication operations required for this parenthesization is 1800 + 105 + 84 = 1989 < 2000. Therefore, the algorithm described above is not an optimal solution.