Problem 1
Find an optimal parenthesization of a matrix chain product whose sequence of dimensions is

< 5, 10, 3, 12, 5 >

Answer:
We can use the Dynamic Programming algorithm described in textbook pp273 to solve this problem. Let \( m[i,j] \) be the minimum number of scalar multiplications needed to compute the matrix \( A_{i...j} \).

\[
m[i,j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + d_{i-1}d_sd_j \} & \text{if } i < j
\end{cases}
\]

And we use another array \( s[i,j] \) to keep the value of \( k \) at which we can split the product \( A_iA_{i+1}...A_j \) to obtain an optimal parenthesization. The two tables \( m[i,j] \) and \( s[i,j] \) for this problem are as follows.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>150</td>
<td>330</td>
<td>405</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>360</td>
<td>330</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the table \( s[i,j] \), we know we can split the product \( A_1A_2A_3A_4 \) at \( d_2 \) with the cheapest cost. So the optimal parenthesization is \((A_1A_2)(A_3A_4))\).

Problem 2 (Problem from Brassard and Bratley)
There are \( n \) trading posts along a river. At any of the posts you can rent a canoe to be returned at any other post downstream. (It is next to impossible to paddle against the current.) For each possible departure point \( i \) and each possible arrival point \( j \) the cost of a rental from \( i \) to \( j \) is known. However, it can happen that the cost of renting from \( i \) to \( j \) is higher than the total cost of a series of shorter rentals. In this case you can return the first canoe at some post \( k \) between \( i \) and \( j \) and continue your journey in a second canoe. There is no extra charge for changing canoes in this way.
Give an efficient algorithm to determine the minimum cost of a trip by canoe from each possible departure point \( i \) to each possible arrival point \( j \). In terms of \( n \), how much time is needed by your algorithm?

**Answer:**

Let \( A[i,j] \) represent the cost for renting a canoe at post \( i \) and returning it at post \( j \). There are \( n \) subproblems: Let \( T[1,k] \) be the cheapest cost of traveling from post 1 to post \( k \), for \( k = 1 \ldots n \). Our goal is to find \( T[1,k], 1 < k \leq n \). Here is the dynamic program:

\[
T[1,1] = 0 \\
\text{for } k = 2 \text{ to } n \text{ do} \\
T[1,k] = \min_{j=1 \ldots k-1} \{T[1,j] + A[j,k]\}
\]

The running time is \( O(n^2) \). Similarly, we can find \( T[2,k], 2 < k \leq n, T[3,k], 3 < k \leq n, \ldots, T[n-1,n] \). Totally, the running time is \( O(n^3) \).

**Problem 3**

Suppose you are driving around Chicago. Your goal is to drive on every single street. (Ignore that there are 1-way streets.) Can you drive on every street at most twice? If so, how? If not, why not?

**Answer:**

Yes We can use Depth First Search (DFS) to drive each street at most twice. Let us consider each street to be an edge and each intersection to be a vertex. Then a DFS moves along streets crossing intersections until we reach an intersection we have encountered before. At this point we back track down streets and explore any streets we have not taken before in exactly the same manner. If we proceed in this manner we will have explored every street exactly twice, one straight and one backtrack. Finally we will end up at our starting point.

**Problem 4**

You are standing on a street looking for a particular house. But you do not know if you should go east or west. The house is at distance \( L \) from you, but you do not know what \( L \) is.

(a) Give an algorithm to find the house. The algorithm should run in time \( O(L) \).

(b) Draw a picture to illustrate your idea. (If you have a very good picture, then you do not even need to write words for part (a).)

**Answer:**

Use technique of repeated doubling. Walk distance \( 2^i \) in each direction, where \( i = 1, 2, \ldots \). After the first \( i \) such that \( 2^i > L \), we have found the house. Make sure you understand why the running time is \( O(L) \) !!!
Problem 5
Suppose you are given an integer A (usually say 32/64 bit integer). And you are only allowed the operations

< statement > ≡ return {True | False}
< statement > ≡ z = { AND x, y | OR x, y | NOT x | z + 1 | z − 1}
< statement > ≡ if(x <= y) then < statement > else < statement >
on A and a constant number of other variables. How would you determine if A has only 1 bit on without using a loop. For instance A = 100000 returns true whereas A = 101000 returns false. Note: This is an O(1) operation. You can write the solution in three statements! And if you know C/C++, one line of C should suffice in writing this whole function!

Answer:

#define IsPowerOf2(x) (!(x & (x-1))

Problem 6
Consider the knapsack problem discussed in class: For set \( S = \{s_1, s_2, \ldots, s_n\} \) of integers and target integer K, is there a subset \( T \subseteq S \), such that

\[
\sum_{t_i \in T} t_i = K?
\]

(1) Recall the algorithm from class. Provide the subproblems, then present pseudocode from class.
(2) The algorithm from (1) only answers the question with a yes or no. What part of the algorithm would we modify so that we can compute the set T? What are the modifications?

Answer:
(1) Suppose we use Boolean \( ExistSubSet(i, K) \) represent whether or not there exists a subset \( T \subseteq s_1, s_2, \ldots, s_i \) such that \( \sum_{t_i \in T} t_i = K \). Then

\[
ExistSubSet(i, K) = \begin{cases} 
  TRUE & \text{if } i \text{ or } K = 0 \\
  ExistSubSet(i - 1, K - s_i) \lor ExistSubSet(i - 1, K) & \text{if } K \neq 0 
\end{cases}
\]

algorithm as follows:

function Boolean ExistSubSet(n,K)
  1: for i = 0 . . . n do ExistSubSet(i, 0) = TRUE;
  2: for j = 1 . . . K do ExistSubSet(0, j) = FALSE;
  3: for i = 1 . . . n do
  4:     for j = 1 . . . K do
5: \[ \text{ExistSubSet}(i, j) = \text{ExistSubSet}(i - 1, K - s_i) \cup \text{ExistSubSet}(i - 1, K); \]

6: \textbf{end for}

7: \textbf{end for}

8: return \text{ExistSubSet}(n, K);

(2) In fact, from the algorithm for (1), we can get an array \text{ExistSubSet}[0 \ldots n, 0 \ldots K]. Suppose we use a table to describe this array, then begin at \text{ExistSubSet}[n, K] and trace through the table following the elements which equal TRUE, we can compute the set \( T \) easily. For example, suppose we want to know whether \( s_i \) belongs to set \( T \) from the condition \( \text{ExistSubSet}[i, j] = \text{TRUE} \), we discuss in three cases.

Case 1: \( \text{ExistSubSet}[i - 1, j - s_i] = \text{TRUE} \land \text{ExistSubSet}[i - 1, j] = \text{FALSE} \), then \( s_i \in T \).

Then we repeat this process on \( \text{ExistSubSet}[i - 1, j - s_i] = \text{TRUE} \) to decide whether or not \( s_{i-1} \in T \).

Case 2: \( \text{ExistSubSet}[i - 1, j - s_i] = \text{FALSE} \land \text{ExistSubSet}[i - 1, j] = \text{TRUE} \), then \( s_i \notin T \).

Then we repeat this process on \( \text{ExistSubSet}[i - 1, j] = \text{TRUE} \) to decide whether or not \( s_{i-1} \in T \).

Case 3: \( \text{ExistSubSet}[i - 1, j - s_i] = \text{TRUE} \land \text{ExistSubSet}[i - 1, j] = \text{TRUE} \), then \( s_i \in T \). But it can \( \notin T \) too. We select one possibility randomly, then continue our trace as case 1 or case 2.

Note: From case 3, we can conclude: trace different path, maybe we can get different set \( T \).

Problem 7

Professor Greedy suggests the following alternative algorithm for determining the minimum cost multiplication of a chain of matrices, \( A_1, A_2, \ldots, A_n \). Recall (from class) that we are given a sequence \( d_0, d_1, \ldots, d_n \) of dimensions where matrix \( A_i \) is a \( d_{i-1} \times d_i \) matrix. The problem is the same as we did in class, to determine the order in which to multiply these matrices so that the the number of multiplications (or unit operations) is least. Professor Greedy suggest a greedy algorithm: find the index \( k \) such that \( 1 \leq k \leq n - 1 \) where \( d_k \) is maximum (You may assume that all \( d_i \)'s are distinct). The algorithm replaces the two matrices \( A_k \) and \( A_{k+1} \) with a single matrix which is their product, \( A_k A_{k+1} \). This is repeated on the remaining \( n - 1 \) matrices until only a single matrix remains. (The intuition is that you want to get rid of large dimensions as quickly as possible, since they result in higher multiplication cost).

Let's do an example of this algorithm: \( d = < 5, 3, 7, 2, 6 > \). Since 7 is the largest *inner* dimension we first multiply \( C = A_2 A_3 \) whose multiplication cost is \( 3 \times 7 \times 2 \). The new resulting dimension vector is \( < 5, 3, 2, 6 > \). We now multiply \( D = A_1 C \) with a cost of \( 5 \times 3 \times 2 \) and then finally we multiply \( DA_4 \) with a cost of \( 5 \times 2 \times 6 \). The total cost is \( 42 + 30 + 60 = 132 \).

Show that this algorithm is not optimal. In particular give a sequence of matrix di-
dimensions; show the result of the execution of the above algorithm and show that a better multiplication order exists. (One example is all that is needed to show that the algorithm is not correct). Shorter counterexamples are better than longer ones. This can be done with 4 matrices whose dimensions are in the range from 1 to 3.

**Answer:**
Counterexample: We can give a sequence of dimensions as follows

\[<1, 3, 5, 4, 2>\]

If use the method of Professor Greedy, we get the multiplication order \((A_1((A_2A_3)A_4))\), the total cost is 60+24+6=90. However, if the product \(A_1A_2A_3A_4\) is parenthesized in this way \(((A_1A_2)A_3)A_4\), the total cost is only 15+20+8=43. So this algorithm is not optimal.