CSE 551: Foundations of Algorithms
Midterm Solutions

Set 1

Problem 1: Suppose that the dimensions of the matrices A,B,C and D are 20*22, 22*25, 25*50 and 50*5. Find the fewest number of multiplications that will be necessary to compute the final matrix and the order in which the matrix chain be multiplied so that it will require the fewest number of multiplications. Show all your work.

Solution: 11200. A*(B*(C*D))

Problem 1*: Suppose that the dimensions of the matrices A,B,C and D are 20*12, 12*5, 5*60 and 60*4. Find the fewest number of multiplications that will be necessary to compute the final matrix and the order in which the matrix chain be multiplied so that it will require the fewest number of multiplications. Show all your work.

Solution: 2400. A*(B*(C*D))

Problem 1:** Suppose that the dimensions of the matrices A,B,C and D are 20*2, 2*15, 15*40 and 40*4. Find the fewest number of multiplications that will be necessary to compute the final matrix and the order in which the matrix chain be multiplied so that it will require the fewest number of multiplications. Show all your work.

Solution: 1680. (A*(B*C)*D))

Problem 2: Let A_1,....A_n be matrices where the dimensions of the matrices A_i, 1<= i <= n are d_{i-1} * d_i. The following algorithm is proposed to determine the best order in which to perform the matrix multiplications to compute A_1*A_2*....*A_n.

At each step, choose the largest remaining dimensions, and multiply two adjacent matrices that share that dimension. What is the complexity of this algorithm? Is this algorithm always going to find the order in which to multiply the matrix chain that will require the fewest number of multiplications? If your answer is **yes**, then provide arguments to support the assertion that this algorithm will always find the best order. If your answer is **no**, then provide an example where the algorithm fails to find the best order.

Solution: No, this algorithm will not always find the fewest number of multiplications.
Let us consider the following example, A * B * C, with dimensions d_0 * d_1, d_1 * d_2, d_2 * d_3.
Let us assume that $d_1 > d_2$. (You could also consider $d_2 > d_1$).

Following our assumption, we will multiply A with B first and the resultant matrix with C. The number of multiplications required is,

$d_0d_1d_2 + d_0d_2d_3$

Now, this should be lesser than the number of multiplications required if we multiply B and C first and then the resultant with A. Therefore,

$d_0d_1d_2 + d_0d_2d_3 < d_1d_2d_3 + d_0d_1d_3$, or,

$d_0d_1d_2 + d_0d_3d_2 < d,d_1d_2 + d_0d_3d_1$, rearranging the terms.

Now, $d_0d_3d_2 < d_0d_3d_1$, as $d_1 > d_2$.
But, $d_0d_1d_2$ need not be less than $d_3d_1d_2$, as d_0 and d_3 are independent numbers.

Let us look at the following example,

100*2, 2*1, 1*1. Going by our approach, we multiply 100*2 and 2*1 first, resulting in 200 multiplications. We then multiply 100*1 with 1*1, resulting in 100 multiplications. Therefore the total number of multiplications required is 200 + 100 = 300.
But, we can do this is in fewer number of multiplications,
Multiply 2*1 with 1*1, resulting in a 2*1 matrix, with 2 multiplications. Then multiply this with 100*2, resulting in a 100*1 matrix, with 200 multiplications. Thus, the total number of multiplications required is 2 + 200 = 202.

Problem 3: Let S be an ordered sequence of n distinct integers. Develop an algorithm to find a longest increasing subsequence of the entries of S. The subsequence is not required to be contiguous in the original sequence. For example, if S = {11, 17, 5, 8, 6, 4, 7, 12, 3}, a longest increasing subsequence of S is {5, 6, 7, 12}. Analyze your algorithm to establish its correctness and also its worst-case running time and space requirement.

Solution:
Say the given list is S. Create another list T which is the sorted version of S.
Now compute LCS(S,T) in $O(n^2)$ time.

Problem 4: Suppose that you are choosing between the following three algorithms:

1. A solves the problem by dividing it into 4 subproblems of half the size, recursively solving each subproblem and then combining the solutions in linear time.
2. B solves the problem of size n by recursively solving 4 subproblems of size n - 1 and then combining the solutions in constant time.
3. C solves the problem of size n by dividing them into 8 subproblems of size n/3, recursively solving each subproblem, and then combining the solution in $O(n^2)$ time.

What are the running time of each of the algorithms and which would you choose?

Solution:
For 1, $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. Solving $4T(n/2)$ would result in $\Theta(n^2)$, which is greater than $O(n)$, therefore the complexity is $\Theta(n^2)$.

For 2, $T(n) = 4T(n - 1) + k$, $T(1) = 1$. Solving $4T(n - 1)$, would result in an exponential function (2^n), hence the complexity is $O(2^n)$.

For 3, $T(n) = 8T(n/3) + O(n^2)$. Solving $8T(n/3)$, would result in $\Theta(n^{1.89})$, but $O(n^2)$ would dominate and hence, the complexity is $O(n^2)$.

You can choose either 1 or 3 to receive full points.

Problem 4*: Suppose that you are choosing between the following three algorithms:

4. A solves the problem by dividing it into 5 subproblems of half the size, recursively solving each subproblem and then combining the solutions in linear time.
5. B solves the problem of size n by recursively solving 2 subproblems of size n - 1 and then combining the solutions in constant time.
6. C solves the problem of size n by dividing them into 9 subproblems of size n/3, recursively solving each subproblem, and then combining the solution in $O(n^2)$ time.

What are the running time of each of the algorithms and which would you choose?

Solution:
For 1, $T(n) = 5T(n/2) + O(n)$, $T(1) = 1$. Solving $5T(n/2)$ would result in $\Theta(n^{2.32})$, which is greater than $O(n)$, therefore the complexity is $\Theta(n^{2.32})$.

For 2, $T(n) = 2T(n - 1) + k$, $T(1) = 1$. Solving $2T(n - 1)$, would result in an exponential function (2^n), hence the complexity is $O(2^n)$, where k is a constant.

For 3, $T(n) = 9T(n/3) + O(n^2)$. Solving $9T(n/3)$, would result in $\Theta(n^2)$, which is same as the recombination, hence the complexity is $\Theta(n^2)$.

Select 3 to receive full points.

Problem 4:** Suppose that you are choosing between the following three algorithms:

7. A solves the problem by dividing it into 6 subproblems of half the size, recursively solving each subproblem and then combining the solutions in linear time.
8. B solves the problem of size n by recursively solving 3 subproblems of size n - 1 and then combining the solutions in constant time.
9. C solves the problem of size n by dividing them into 10 subproblems of size n/3, recursively solving each subproblem, and then combining the solution in $O(n^2)$ time.

What are the running time of each of the algorithms and which would you choose?

Solution:
For 1, $T(n) = 6T(n/2) + O(n)$, $T(1) = 1$. Solving $6T(n/2)$ would result in $\Theta(n^{2.58})$, which is greater than $O(n)$, therefore the complexity is $\Theta(n^{2.58})$.

For 2, $T(n) = 3T(n - 1) + k$, $T(1) = 1$. Solving $3T(n - 1)$, would result in an exponential function (2^n), hence the complexity is $O(2^n)$, where k is a constant.

For 3, $T(n) = 10T(n/3) + O(n^2)$. Solving $10T(n/3)$, would result in $\Theta(n^{2.09})$, which is greater than the recombination, hence the complexity is $\Theta(n^{2.09})$.

Select 3 to receive full points.

Set 2

Problem: Let u and v be two n bit numbers where n is a power of 2. the traditional multiplication algorithm requires O(n2) operations. A Divide-and-Conquer based algorithm splits the numbers into two equal parts, computing the product as

$$uv = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^{n/2} + (ad + bc)2^{n/2} + bd$$

The multiplication of ac, ad, bc and bd are done using this algorithm recursively.
(i) Determine the computing time of the above algorithm.
(ii) What is the computing time if $ad + bc$ is computed as $(a + b)(c + d) - ac - bd$?

Solution:

(i) The expression makes recursive calls to four subproblems of half the size each, and then evaluates the preceding expression in $O(n)$ time.

$$T(n) = 4T(n/2) + O(n).$$

Using Master's Theorem, we solve the recurrence to obtain $O(n^2)$.

(ii) Note that this method gives the recurrence

$$T(n) = 3T(n/2) + O(n)$$

Since the transition from 4 to 3 occurs at every level of the recursion, we get lower time bound of $O(n^{1.59})$ (again, use Master's Theorem to prove this).

Problem: The n-th Fibonacci number is defined by the recurrence relation F(n) = F(n −1) + F(n − 2), with initial conditions F(0) = 0 and F(1) = 1. F(n) can easily be computed with an algorithm with complexity Θ(n). Develop an algorithm to compute F(n) with complexity Θ(logn). Explain the idea of your algorithm and analyze the algorithm to show that the complexity is indeed Θ(logn). Show all your work.

Solution:

Discussed in class.
Check the class notes for the exact solution.

Problem: $T(n) = k, n = 1$
$T(n) = 3T(n/2) + kn, n > 1.$

Solution:

$T(n) = 3T(n/2) + kn$
$T(n/2) = 3T(n/4) + kn/2$
.
.
.
$T(n/2^{m-1}) = 3T(1) + kn/2^{m-1}$
Substituting,

$T(n) = 3^m k + (3/2)^{m-1}kn + (3/2)^{m-2}kn + \ldots\ldots + (3/2)^0 kn$
Where $n = 2^m$

$= 3^{\log 2\,(n)}k + kn\,[\,(3/2)^{m-1} + \ldots\ldots. + (3/2)^0]$
$= 3^{\log 2\,(n)}k + 2kn[(3/2)^m - 1]$
$= 3^{\log 2\,(n)}k + 2kn[(3^{\log 2\,(n)}/\,2^{\log 2\,(n)}) - 1]$
$= 3^{\log 2\,(n)}k + 2kn[3^{\log 2\,(n)} / n - 1]$
$= 3^{\log 2\,(n)}k + 2k[3^{\log 2\,(n)} - n]$
$= 3^{\log 2\,(n)}k + 2k3^{\log 2\,(n)} - 2kn$
$= 3k3^{\log 2\,(n)} - 2kn$
$= 3kn^{\log 2\,(3)} - 2kn$ $\quad [3^{\log 2\,(n)} = n^{\log 2\,(3)}]$

Problem: While discussing the LCS problem, we noted that there may be more than one LCS between two given LCSs S and Y. The algorithm discussed in class produces only one LCS. Develop an algorithm to find all the possible LCSs between two given sequences X and Y.

Solution:

In the LCS-length matrix, in the final row, if multiple instances of the same maxima, say *m*, exists, then instead of just following one route(sequence of arrows, i.e. "↑" and "←") from one of the instances of *m*, follow the arrows from all the instances of it from the last row. This will give you all the possible LCSs for a pair of strings. In your solution, you have to demonstrate the full algorithm and not just the changes from the single LCS algorithm.