CISC 380 - Exam 1 Topics

Note: The exam will cover **all** material from class and your assignments. This includes **but is not limited to** the following topics.

1. Algorithm Analysis Review
    - Best Case / Average Case / Worst Case
    - Big-O notation (O, Θ, Ω, o, ω)
        - Polynomial rule
        - Addition rule
        - Multiplication rule
        - Log rule
        - Exponential Rule
    - Limit Theorem
    - Ordering functions by asymptotic growth rate
    - Examples
        - Bubble Sort
        - Maximum Subarray
2. Brute Force Algorithms
3. Divide & Conquer
    - Solving recurrence relations
        - Master Theorem
        - Solving by unrolling
        - Recursion trees
    - Examples
        - Merge Sort
        - Multiplying n-bit numbers (Karatsuba multiplication)
        - Counting Inversions
        - Median Algorithm (randomized and deterministic)
4. Dynamic Programming
    - Subproblem Definitions & Recurrences
    - Iteration & Memoization
    - Recovering the solution (back-tracking through the table)
    - Examples
        - Domino Tilings & Fibonacci
        - Longest Increasing Subsequence
        - Longest Common Subsequence
        - Interleaving Problem (we may not get to this for Exam 1)

CISC 380 - Exam 1 Practice Problems

Note that Exam 1 covers ALL the material discussed in class up to and including the last day of class before the exam. These practice problems are representative of the type of problems you might get on the exam but do not cover all of the material you are responsible for.

1. Order the following functions by asymptotic growth from fastest to slowest. Clearly indicate any functions that grow at the same rate (i.e. $f(n) = \Theta(g(n))$).

$$n\log(n^2), \quad 2^n, \quad 2^{n/2}, \quad \sqrt{n}, \quad n\log^2 n, \quad 30, \quad n^{1.5}, \quad \log n, \quad n\log n$$

Solution:

$$30, \quad \log n, \quad \sqrt{n}, \quad n\log n, \quad n\log(n^2), \quad n\log^2 n, \quad n^{1.5}, \quad 2^{n/2}, \quad 2^n$$

Since, $n\log(n^2) = 2n\log n$ this function grows at the same rate as $n\log n$.

2. For each of the following program fragments, first calculate the running time $T(n)$ then give a big-Θ bound on the asymptotic running time in terms of n. Show your work - you must justify that your answer is tight (i.e., lower bound and upper bound)!

   (a)
```
Sum = 0;
for (i=0; i< n; i++)
   for(j=0; j < i; j++)
      for (k=0; k<j; k++)
         Sum = Sum + 1;
```

Solution: The inner-most for loop has running time $T_k(j) = 1 + 3j$. The middle loop has running time

$$\begin{aligned} T_j(i) &= 1 + 2i + \sum_{j=1}^{i} T_k(j) \\ &= 1 + 2i + \sum_{j=1}^{i} 1 + 3j \\ &= 1 + 3i + 3\sum_{j=1}^{i} j \\ &= 1 + 3i + i(i+1)/2 = i^2/2 + 3.5i + 1. \end{aligned}$$

The entire fragment has running time

$$\begin{aligned} T(n) &= 2 + 2n + \sum_{i=1}^{n} (T_j(i)) \\ &= 2 + 2n + \sum_{i=1}^{n} (i^2/2 + 3.5i + 1) \\ &= 2 + 2n + (1/2)\sum_{i=1}^{n} (i^2) + 3.5\sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 \\ &= 2 + 2n + (1/2)(n^3/3 + n^2/2 + n/6) + 3.5n(n+1)/2 + n \\ &= O(n^3), \end{aligned}$$

by the polynomial theorem from class.

(b)
```
Sum = 0;
for(j=0; j < n; j++)
    if(j % 2 == 0)
        for (k=0; k<j; k++)
            Sum = Sum + 1;
```

Solution: The inner loop has running time $T_k = 1+3j$. It executes when j is $0, 2, 4, 6, 8 \ldots n$. This results in the following runtime $T(n) = 1+2n+\sum_{i=0}^{n/2} T_k(2i) = 1+2n+\sum_{i=0}^{n/2}(1+6j) = O(n^2)$. For a simplier solution you could observe that the inner for loop executes $1/2$ as many times as it would without the if statement so adding that in only changes the overall running time by a constant factor.

3. Solve the following recurrence using the recursion tree method.

$$T(n) = 2T(n/3) + n^2.$$

Solution: (Note: Your solutions to this type of problem on the test should include a picture!)
There are 2^i calls per level each of size $n/3^i$, so we do $(n/3^i)^2$ work per call, and there are $\log_3 n$ levels.

That means the total work done is

$$\sum_{i=0}^{\log_3 n} 2^i (n/3^i)^2 = n^2 \sum_{i=0}^{\log_3 n} (2/9)^i$$

$\sum_{i=0}^{\log_3 n}(2/9)^i$ is $\Theta(1)$ (it's upper-bounded by $\sum_{i=0}^{\infty}(2/9)^i = 1/(1-2/9) = 9/7$ and lower-bounded by 1) so $T(n) = \Theta(n^2)$.

4. You are given an array $A[1 \ldots n]$ of n distinct numbers. You are told the sequence of values $A[1], A[2], \ldots, A[n]$ is unimodal. That means there is an index p where the values in the array are increasing from position 1 to p, and then decreasing from p to n. For example, the sequence: -9, -8, 1, 2, 3, 0, -5 has its peak at position $p = 5$ where $A[5] = 3$. Give an $O(\log n)$ divide and conquer algorithm for finding the "peak entry" p.

   (a) Explain your algorithm in words.

   Solution: Again, we do a modification of binary search.
   If the array is small (size ≤ 2), then just return the larger (or only) element. Otherwise, choose two elements in the exact middle of the array, say at positions $\frac{n}{2} - 1, \frac{n}{2}$. There are two cases:

   - If $A[\frac{n}{2} - 1] > A[\frac{n}{2}]$, then that means that the array has stopped increasing, so the the peak has a index lower than $\frac{n}{2}$. So, we can recurse in the first half of the array.
   - If $A[\frac{n}{2} - 1] < A[\frac{n}{2}]$, then that means that the array is still increasing, so the the peak has a higher index than $\frac{n}{2}$. So, we can recurse in the second half of the array.

   (b) Analyze your algorithm including stating and solving the relevant recurrence relation.

   Solution: To solve the problem on an input of size n, we do a single comparison, and then solve one subproblem of size $n/2$. The recurrence relation is $T(n) = T(n/2) + O(1)$, which has the solution $T(n) = O(\log n)$ by the master theorem.

5. You are consulting for a small investment company. They give you a price of Google's shares for the last n days. Let $p(i)$ represent the price for day i. During this time period, the company wanted to buy a fixed number of shares on some day and sell all these shares on some later day. The company wants to know when they should have bought and when they should have sold the shares in order to maximize the profit. If there is no way to make money during the n days, you should report this instead.

   For example, suppose $n = 3, p(1) = 9, p(2) = 1, p(3) = 5$. Then you should return buy on day 2, sell on day 3. Your goal is to design a divide-and-conquer algorithm for this problem that runs in time $O(n)$.

   (a) Given an $O(n^2)$ time algorithm for this problem. It is expected that your solution to this part might not use divide-and-conquer. Explain your algorithm in words and justify its running time.

   Solution: Simply test all possible pairs of days to buy and days to sell, and choose the best one. This is correct because it checks exhaustively every possible day to buy and sell. Its runtime is $O(n^2)$ because there are at most n days to choose from to buy, n days to choose from to sell, and $O(1)$ work done to calculate the potential profit.

   (b) Suppose you want to buy in the first $n/2$ days and you want to sell in the last $n/2$ days. Given an $O(n)$ time algorithm for finding the best pair of days for buying/selling under

this restriction. Explain your algorithm in words and justify its running time.

Solution: We should buy at the lowest possible cost in those first set of days, and sell at the highest possible value in the second set of days. Finding the minimum of the first $n/2$ numbers takes $O(n)$ time, as does finding the maximum of the second $n/2$ numbers. We then return these two indices if the difference is positive, or nothing if it is negative.

(c) Give a divide and conquer algorithm with running time $O(n \log n)$ for finding the best pair of days for buying/selling. (There are no restrictions on days, except that you need to buy at an earlier date than you sell.) Describe your algorithm in words and analyze your algorithm, including stating and solving the relevant recurrence relation. Hint: your algorithm should use your solution to part b.

Solution: If the array is of size < 2, then just return the two given indices if the difference is positive, or return nothing.
Split the problem into two subregions of size n/2. There are three cases to consider - the optimal pair of days could fall into any one of these cases.

- Case one: The best day to buy is in the first set of n/2 days, and the best day to sell is in the second set of n/2 days.

  In this case, we can use the algorithm in part b) to find the best set of days in $O(n)$ time among all cases where you buy in the first set of n/2 days, and sell in the second set of n/2 days.

- Case two: The best day to buy and sell are both in the first half of the days.

  Then, we can recurse on the first half of the array.

- Case three: The best day to buy and sell are both in the second half of the days.

  Then, we can recurse on the second half of the array.

Our algorithm tests these three possibilities, and chooses the one among the three that gives the most profit.
To solve one problem of size n, we solve two subproblems of size $n/2$, run the $O(n)$ algorithm from part b, and do a constant number of comparisons. Our final recurrence relation is $T(n) = 2T(n/2) + O(n)$, which gives us an $O(n \log n)$ running time.

Solution: There's a better algorithm than either the brute force or the divide and conquer algorithm. At any day i, the best profit I could make if I sold that day is if I bought it at the *cheapest that it's been at any point up to i*. In otherwords, my profit is the difference between the current value and the running minimum. You could calculate this in a single pass algorithm as follows:

$RunningMin \leftarrow 0$
$RunningMinIndex \leftarrow 0$
$BestSellDAY \leftarrow 0$
$BestProfit \leftarrow 0$

```
BestBuyDAY ← 0
for i = 1 to n do
    if P(i) < RunningMin then
        RunningMin ← P(i)
        RunningMinIndex ← i
    else if BestProfit < P(i) − RunningMin then
        BestProfit ← P(i) − RunningMin
        BestSellDay ← i
        BestBuyDay ← RunningMinIndex
return BestBuyDay, BestSellDay
```

This is an $O(n)$ algorithm because it does a single pass through an array, and does a constant amount of work at each element.

6. Modify the algorithm given in class for the longest common subsequence (LCS) to return the actual longest common subsequence in addition to the length without changing the running time. Write your own pseudocode and analyze the runtime to show it hasn't changed.

Solution: Recall $L(i, j)$ = length of LCS of $x_1x_2 \dots x_i$ with $y_1y_2 \dots y_j$
We'll write a recursive function `getLCS` to backtrack through the table and print the LCS.

```
function LCS(X = x_1 x_2 ... x_n, Y = y_1 y_2 ... y_m )
    for i = 0 → n do                                   ▷ initialize first row 0
        L(i, 0) = 0
    for j = 0 → m do                                   ▷ initialize first column 0
        L(0, j) = 0
    for i = 1 → n do                                   ▷ each row
        for j = 1 → m do                               ▷ each column
            if x_i == y_j then
                L(i, j) = 1 + L(i − 1, j − 1)
            else
                L(i, j) = max{L(i − 1, j), L(i, j − 1)}
    getLCS(n, m)
function getLCS(i, j)
    if i == 0 or j == 0 then Return
    if x_i == y_j then
        output(x_i)
        getLCS(i − 1)j − 1
    else
        if L(i − 1, j) > L(i, j − 1) then
            getLCS(i − 1, j)
        else
            getLCS(i, j − 1)
```

7. You are given an $n \times m$ array $A[i, j]$ filled with integers. A *right/down* path in this array is a sequence of elements in the array, beginning with position $(1, 1)$ (the top/left), and ending at

position (n, m) (the bottom/right), where each element is to the right of or below the previous element. In other words, each element is greater in either the x or y coordinate by 1. The *weight* of a right/down path is the sum of the array values of that path.

Write a dynamic programming algorithm that finds the right/down path in this array of maximum weight.

3	4	1	10	2	1	4
7	2	11	7	3	-1	2
3	1	12	-3	14	1	5
2	1	3	2	3	4	1
4	6	2	4	10	-1	3

This array has a right/down path (shown in the array using missing borders) with weight 61. (Note that this isn't the maximum one in the array.)

(a) Describe your subproblems in words.

Solution: Let $MAXPATH(i, j)$ be the weight of a maximum path from $(1, 1)$ to (i, j).

(b) State the recurrence for your subproblems, and explain in words why it's true.

Solution: The very last entry in the path ending at (i, j) must have come from either $(i-1, j)$ or $(j, i-1)$. We're trying to maximize the total path weight, so we should maximize the paths to these prvious steps as well. The total weight is then going to be $A[i, j] + \max MAXPATH(i-1, j), MAXPATH(i, j-1)$. If $i = 1$ or $j = 1$, then we only have one option - the other is invalid. There are lots of ways to handle this, we will sort it out in a special loop before the main recursive part.

(c) Write pseudocode for your algorithm.

Solution:

```
function MAXPATH(Array A[1...n, 1...m])
    Array MP[1...n, 1...m]
    MP[1,1] ← A[1,1]
    for i = 2 to n do
        MP[i,1] ← A[i,1] + MP[i-1,1]
    for j = 2 to m do
        MP[1,j] ← A[1,j] + MP[1,j-1]
    for i = 2 to n do
        for j = 2 to m do
            MP[i,j] ← A[i,j] + max MP[i,j-1], MP[i-1,j]
    return MP[n,m]
```

(d) Analyze the running time of your algorithm. (use O notation)

Solution: We do $O(1)$ work per entry of our array, and there are $O(nm)$ entries, for a total runtime of $O(nm)$.