Section Solutions #5

Problem 1: Big-O

a) $O(n^2)$. The outer loop will run n times. Each time through the outer loop, the inner loop will run i times, where i runs from 0 to n-1. A constant amount of work is done in the body of the inner loop. This leads to the series: 0 + 1 + 2 + ... n-1 which, as we know, equals n(n-1)/2. Keeping only the highest order term and throwing away any constant factors, we arrive at our answer of $O(n^2)$ computational complexity.

b) O(1). The outer loop executes 10 times, the inner loop i times where i runs from 0 to 9, so there are ~100 multiply/add operations, but it does that same amount of work for any value of n. Thus the computational complexity is *constant* with respect to `n`. The constant "1" in O(1) signifies this. Constant time doesn't necessarily mean that a function is trivial or computes its result instantly, but the function always does the same amount of work, regardless of the inputs.

c) O(lg n). Each time the function recurses, it divides n/2, which will create the following recurrence relation: T(n) = T(n / 2) + 1, and T(1) = 1. Solving the two equations, we see that we can repeatedly apply the first definition of T(n) until $n/2^k = 1$. Solving for k, we get that $k = \log_2 n$. Therefore, we know that Mystery3 should take $O(\log_2 n) = O(\lg n)$ time.

d) `Winky` is $O(2^N)$. A call to `Winky` turns into a call to `Pinky` on N. Each call to `Pinky` makes two recursive calls that are each one smaller. This is the same as Towers of Hanoi or knapsack. You might recognize the classic 2^N pattern or get the result by solving the recurrence relation of T(N) = 2*T(N-1) + 1 or drawing the recursion tree and counting the number of calls.

Problem 2: Searching and Sorting

For each case, we need to compare the cost of doing the sort and binary search with the cost of doing linear searches. If we let x be the number of searches done, we can write the following inequality:

cost of sort + (x * cost of binary search) < x * cost of linear search

If this inequality holds, then doing a sort first is the more efficient approach. So, we solve for x to determine how many searches make it worth our while to sort the data:

Selection Sort _ $O(n^2)$

a) $((2)^4)^2 + (x * \log_2 (2)^4) < x * (2)^4$
$256 + 4X < 16X$
$21 < x$

b) $((2)^{10})^2 + (x * \log_2(2)^{10}) < x * (2)^{10}$
$1048576 + 10x < 1024x$
$1034 < x$

From these numbers, we basically need to perform about N searches to make the selection sort worthwhile. This makes sense though. Since N linear searches is going to cost N^2, then we need to do N of these to break even on the cost of the sort.

And for the case of merge sort:

MergeSort _ O(n log n)

a) $((2)^4 * \log_2(2)^4) + (x * \log_2(2)^4) < x * (2)^4$
$64 + 4X < 16X$
$5 < X$

b) $((2)^{10} * \log_2(2)^{10}) + (x * \log_2(2)^{10}) < x * (2)^{10}$
$10240 + 10x < 1024x$
$10 < x$

From these numbers, it now appears that we have to do approximately log N searches before sorting becomes worthwile. The reasoning follows similarly, except now we only need log N searches to pay for the cost of the sort.

Overall then, using merge sort with an input size of 2^10 only requires about 10 searches to have made the sort beneficial, rather than over 1000 with selection sort. A cheaper sort clearly pays for itself in the end.

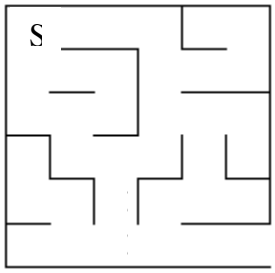
Problem 3: Those Big-O Constant Factors
a.) If we had 50 items to sort, then the time to selection sort them would be $10 * 50^2 =$ 25,000 while the time to merge sort them would be 100 * 50 * log 50 = 28,219. Thus, in this case the constant factors help make selection sort the better sorting algorithm.

b.) If we had 100 items to sort, then the time to selection sort them would be $10 * 100^2 =$ 100,000 while the time to merge sort them would be 100 * 100 * log 100 = 66,438, and therefore merge sort is the better choice. Our input size is now large enough to overcome merge sort's larger constant factor.

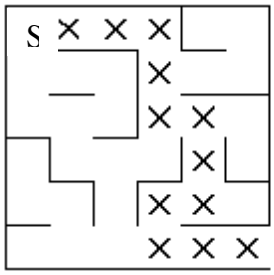
c.) As the above has shown, for small enough inputs, selection sort may actually perform better than merge sort. Thus, a hybrid strategy is the one to take. To generate the fastest possible code, we could first find the input size such that selection sort is just as fast as merge sort. Then, when doing our recursive merge sort algorithm, if the number of items we wish to sort is less than this threshold, we can switch to using selection sort, rather than continuing with merge sort, since for input sizes smaller than the given threshold, selection sort performs better.

Problem 4: Search Algorithms
Neither breadth-first nor depth-first search are necessarily faster than the other (each can be somewhat faster on different inputs). However, in the worst case they are the same since in the worst case they examine every square in the maze exactly once before finding the solution. They are therefore both O(l * w). The breadth-first algorithm uses more memory than the depth-first algorithm in general since it stores multiple paths at once whereas depth-first search only considers one path at a time.

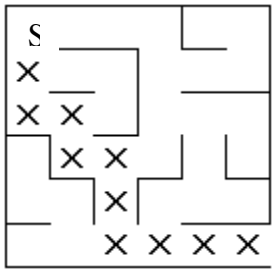
Interestingly, the two algorithms are not guaranteed to return the same paths. Depth-first search returns the first path it finds to the goal (which is not necessarily the shortest path), whereas breadth-first search is guaranteed to return the shortest path (which is by design also the first path it finds). The following is an example of a maze where the two algorithms return different results:



Depth-first search will return this path (length 13):



whereas breadth-first search will return the path below (length 11).



Problem 5: Algorithmic Problem Solving
At the end of each of these algorithms, MaxSoFar will contain the maximum sum, if applicable.

a)

```
MaxSoFar = 0
```

```
for L = 1 to N
      for U = L to N
            Sum = 0
            for I = L to U
                  Sum = Sum + vec[I]
            /* Sum now contains the sum of vec[L...U] */
            MaxSoFar = max(MaxSoFar, sum)
```

This algorithm is O(N^3) because the outer loop is executed exactly N times. the first inner loop is executed at most N times in an iteration of the outer loop, and the innermost loop is executed at most N times in an iteration of the first inner loop.

b)

```
MaxSoFar = 0
for L = 1 to N
      Sum = 0
      for U = L to N
            Sum = Sum + vec[U]
            /* Sum now contains the sum of vec[L...U] */
            MaxSoFar = max(MaxSoFar, Sum)
```

This algorithm is O(N^2) because the outer loop is executed exactly N times and the inner loop is executed at most N times for an iteration of the inner loop.

c) This recursive solution takes advantage of the following insight: if a vector is divided into two halves, the maximum subvector is either the maximum subector in the left half, the right half, or the maximum subvector which crosses the border between the two halves. Furthermore, the maximum subvector crossing the border is the maximum subvector which touches the border from the left plus the maximum subvector which touches the border from the right.

```
MaxSum(L, U, vec)
{
      /* Zero-element vector */
      if L > U
            return 0
      /* One-element vector */
      if L = U
            return max(0, vec[L])

      /* The left half of the recusion is vec[L...M], the right is
            vec[M+1...U] */
      M = (L+U)/2
      /* Find max touching the border on the left */
      SumLeft = 0
      MaxToLeft = 0
      for I = M to L /* M is more than L, so I decreases from M to L */
            SumLeft = SumLeft + vec[I]
            MaxToLeft = max(MaxToLeft, SumLeft)
      /* Find max touching the border on the right */
      SumRight = 0
      MaxToRight = 0
      for I = M+1 to U
```

```
            SumRight = SumRight +vec[I]
            MaxToRight = max(MaxToRight, SumRight)
      /* MaxCrossing is the maximum subvector sum which crosses the
            border */
      MaxCrossing = MaxToLeft + MaxToRight

      MaxInLeftHalf = MaxSum(L, M, vec)
      MaxInRightHalf = MaxSum(M+1, U, vec)
      return max(MaxCrossing, MaxInLeftHalf, MaxInRightHalf)
}
```

This function is fairly complex, so don't worry if it takes a few reads to grok it. It is O(NlogN) because each recursive call does O(N) work since finding the maximum subvector which crosses the border executes statements a maximum of N/2 times. Furthermore, since the recursion divides the vector by 2 every call, there will be logN calls, for a total of NlogN.

d)

```
MaxSoFar = 0
MaxEndingHere = 0
for I = 1 to N
      /* MaxEndingHere and MaxSoFar are accurate for X[1...I-1] */
      MaxEndingHere = max(MaxEndingHere+X[I], 0)
      MaxSoFar = max(MaxSoFar, MaxEndingHere)
```

Like many such solutions, the O(N) algorithm is relatively short, but dense. It iterates through the vector from left to right, keeping track of the sum of the vector so far as well as the maximum sum found so far. If the sum so far drops below zero, it instead becomes zero, effectively starting to count over again beginning after the element which made the sum drop below zero. To understand why this works, consider what must be true about the maximum subvector. Consider the element on the left side of the subvector. Either it must be nonexistent because the maximum subvector starts at the beginning of the vector, or the element must be negative. If this were not so, the element would be included in the maximum subvector because it would increase the sum. Furthermore, every subvector starting at this element and extending to the left must sum to zero or a negative number for the same reason. Therefore, the sum so far must be made zero at the start of the maximum subvector, so we are sure to count it correctly. It may also be helpful to make some example subvectors and run this algorithm to gain insight into why it works. As is true of this algorithm, the best solutions to problems frequently involve a precise understanding of the problem's properties and insight into what must be calculated and what need not be.

e)

Algorithm	O(N^3)	O(N^2)	O(NlogN)	O(N)
Size of N				
100	3 seconds	.1 seconds	.03 seconds	.003 seconds
200	24 seconds	.4 seconds	.06 seconds	.006 seconds
1000	50 minutes	10 seconds	1 second	.03 seconds
10000	35 days	17 minutes	20 seconds	.3 seconds