Homework 1
Andre Achtar-Zadeh
Sfsu ID: 923051048
Analysis of Algorithms/CSC510
Ivan Corneillet
02/28 - Spring 2025

1. To turn the java method into pseudocode I looked back at the rules we went over in the slides from the second day of class. The java code uses 0-based indexing but pseudocode needs 1-based indexing. For 1-based indexing I changed the outerloop to i=1 to n-1 and the inner loop to j=i+1 to n. The minIndex starts at the current i and after finding the smallest the element is swapped. I figured it out because the outer loop in Java runs n=1 times to sort the array. With 1-based indexing, I set it to i=1 to n-1 which still gives n=1 passes. In inner loop checks el;ements after i. With 1 based indexing I adjusted it to j=i+1 to n, which works since the last spot is in. The minIndex gets set to i at the start of each round. The java swap uses a temp variables. Following the role I swapped it with swap A[i] and A[minIndex].

Pseudocode:

```
SelectionSort(A[1,..., n]) :
  for i = 1, ..., n - 1
    minIndex := i
    for j = i + 1, ..., n
      if A[j] < A[minIndex]
        minIndex := j
    swap A[i] and A[minIndex]
```

2. I worked through this problem by breaking it down into parts and using big-O ideas from pg14 from the second day of class. The plan was to figure out how many times each loop runs and then combine that to find the total runtime.

   i.

ALG1(n)	for e = d, ..., n	If d=1d = 1d=1, goes from 1 to nnn, so nnn times.	O(n)
ALG1(n)	for f = d, ..., e	For each eee, runs from ddd to eee. If d=1d = 1d=1, it's eee times per eee. Total is the sum from e=1e = 1e=1 to nnn, which is n(n+1)2\frac{n(n+1)}{2}2n(n+1), so O(n2n^2n2).	O(n^2)
ALG2(m, n)	for a = 5, ..., 5n	Goes from 5 to 5n5n5n, so	O(n)

		5n−5+1=5n−45n - 5 + 1 = 5n - 45n−5+1=5n−4 times, which is about nnn when scaled.	
ALG2(m, n)	count := count + ALG1(m)	Runs once per aaa, and ALG1(m) is O(m2m^2m2). With O(n)O(n)O(n) aaa loops, it's O(n⋅m2)O(n \cdot m^2)O(n⋅m2).	O(nm^2)
ALG2(m, n)	for b = 1, ..., m	Runs from 1 to mmm, so mmm times.	O(m)
ALG2(m, n)	`for c = 1, ...,	n/m	O(n/m)
ALG2(m, n)	while d < m	Starts at 1, doubles (1,2,4,...1, 2, 4, ...1,2,4,...) until past mmm. That's about log2m\log_2 mlog2m steps.	O(log m)
ALG2(m, n)	for e = 1, ..., m	Goes from 1 to mmm, so mmm times.	O(m)
ALG2(m, n)	for f = 1, ..., n	Goes from 1 to nnn, so nnn times.	O(n)
ALG2(m, n)	for g = 512, ..., 1024	Goes from 512 to 1024, which is 513 times, a fixed number.	O(1)

Part ii. Total Runtime Expression

ALG1(n): The main work is from the F loop inside e. The total is n(n+1)/2 sp

T1(n)=n(n+1)/2

ALG2(m, n):

The a loop with ALG1(m) runs O(n) times, and each ALG1(m) is O(m^2), so O(n*m^2). The b, c, and while nest is O(m)*O(n/m)*O(logm)=O(nlogm). The e, f, and g nest is O(m)*O(n)*O(1)=O(mn).

Total: T2(m,n)=n*m^2+n*logm+m*n

Part iii. Simplified Big-O
For T1(n): n(n+1)/2=n^2+n/2 the n^2 is the biggest part and the contant ½ doesnt count so its O(n^2).
For T1(m,n):T2(m,n)= nm^2+n log m+ mn. I checked which term grows the fastest. nm^2 gets big if m or n grow a lot. N log m and mn are smaller compared to nm^2 when m is big. The tightest is O(nm^2)

3. A. 2n^2+14n-45 is O(n^2)
   I used C=7 and K=13. I picked a C a bit higher than the n^2 coefficient (2) and tried an odd K like 13.
   -First i checked if 2n^2+14n-45≤7n^2 when n≥13.
   -I moved terms around: 2n^2+14n-45-7n^2≤0 which becomes -5n^2+14n-45≤0
   -Multiplied by -1 and flipped the sign: 5n^2-14n+45≥0
   -Solved the quadratic: Discriminant is (-14)^2-4*5*45=196-900=-704
   -Since 5n^2-14n+45 is always greater than or equal to 0, in inequality 2n^2+14n-45≤7n^2 holds for n≥13 (tested n=13: 2(169)+14(13)-45=338+182-45=475, 7(169) = 1183, and 475 ≤ 1183).
   -so, C=7 and K=13 do work.

   B. 7n^5+18n^4+27n^3logn+2n+800 is O(n^5)
   I used C=11 and K.
   -I needed 7n^5+18n^4+27n^3logn+2n+800≤11n^ for n≥9
   -for n=9: left side is 7(9^5)+18(9^4)+27(9^3)log9+2(9)+800. So 9^5=59049 7*59,049=413,343, 9^4=6,561, 18*6,561=118,098, 9^3=729, log9=2.2, 27 729*2.2+43,346.2, + 18 + 800. total= 575,605.2. Right side is 11 * 59,049=649,539. Since 575,605.2≤649539, it holds
   For n=10, its even better as n^5 grows fast. So, c=11 and k=9 are fine.

   C. n^3+2n^2+n+1 is O(n^19)
   I used C=13 and K=3 to test it.
   -I checked if n^3+2n^2+n+1≤13n^19 for n≥3
   -For n=3: left side is 27+18+3+1=49, right side is 13*3^19. Since 3^19 is huge, 13*3^19 is way bigger than 49 so it holds.
   -For n=4: left side is 64+32+4+1=101, right side is 13*4^19 so it works because its even bigger.
   -with n^19 growing so fast, C=13 and K=3 should for for any n≥3.

4. proof : if f(n) is O(g(n)) then c*f(n) is O(g(n)) for all c>0.
   -I started with the fact that f(n) is O(g(n)), which means there are constants C1>0 and K1≥0 such that f(n)≤c1g(n) for all n≥k1
   -I set C=c*c1 and kept K=k1 since c>0 and c1>0, C is positive
   For n≥K i used the original inequality: f(n)≤c1g(n)

Multiplied both sides by this simplifies to c*f(n)≤Cg(n) where c=c*C1
Since this holds for all n≥k, by the big-0 definition, c*f(n) is o(g(n)).

5. This function merges two sorted parts of array A from 1 to m and from m+1 to n, into a new array B, then copies back to A.
   The incomplete part was how to pick the smallest element.

```
Merge(A[1,.., n], m):
   initialize B[1,.., n]
   i := 1          // Start of first subarray A[1, ..., m]
   j := m + 1      // Start of second subarray A[m+1, ..., n]
   k := 1          // Position in result array B
   while i <= m and j <= n
      if A[i] <= A[j]
         B[k] := A[i]
         i := i + 1
      else
         B[k] := A[j]
         j := j + 1
      k := k + 1
   while i <= m
      B[k] := A[i]
      i := i + 1
      k := k + 1
   while j <= n
      B[k] := A[j]
      j := j + 1
      k := k + 1
   for k = 1, …, n
      A[k] := B[k]
```

I put this together because i know merge sort combines two sorted pieces, so i set i to start at 1 and j to m+1. K tracks where to put results in B.
The first while loop compares A[i] and A[j], picks the smaller one for B[k], and moves forward, it stops when one part runs out.
The next while loop handles leftovers from either subarray copying them to B. The final for loop puts everything back into A. I adjusted J to m+1 since the second part starts there

Runtime complexity: O(n)
The first while loops runs until i>m or j>n. Each steps uses one element, and there are m from the first part +n-m from the second. Which means n elements max, O(n)
The second while loop copies any remaining from the first part, up to m-i+1 times which is O(n)
The third while loop does the same for the second part up to n-k+1 times so O(n)
The for loop copies B back to A running n times, so O(n)

Adding them up $O(n)+O(n)+O(n)+O(n)=O(n)$.