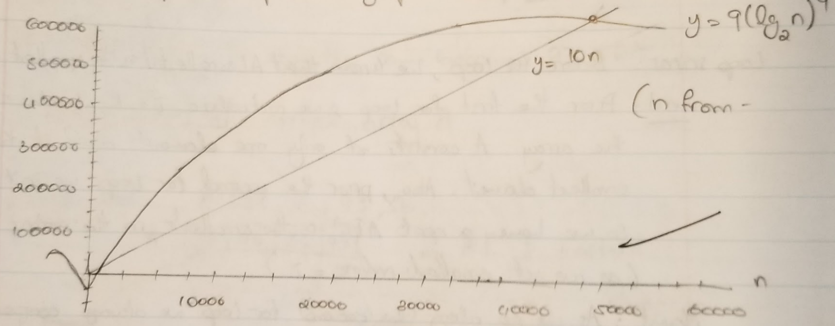
1 credit to Sovanmony Lim

① By plotting a graph of the two equations, the intersection of the lines can be located to see which one bounds the other. I used wolfram alpha to plot the graph (see the graph below)



The function $y = 10n$ and $y = 9(\log_2 n)^4$ intersect at $n \approx 55539.6$.
I assume that when the value of $n \approx 3.1$ to $n \approx 55539.6$ is when the running time $10n$ runs slower than an algorithm whose running time is $9(\log_2 n)^4$

2 credit to Jason Riendeau

*the following algorithm didn't have a name, and didn't use the psuedocode convention in textbook (we should!)
*the second line, and last two lines of code should be identated

Problem 2) Psuedocode for selection sort (A, n), where A is the array, and n is sizeof(A)

Psuedocode line	Number of times it's being run
for (i = 1; i < n; i++)	n
min = i	n-1
for (j = i+1; j < n+1; j++)	Sigma (i=1 -> n-1) n-i+2
if (A[j] < A[min])	Sigma (i=1 -> n-1) n-i+1
min = j	Max of Sigma (i=1 -> n-1) n-i
If min != i	n-1
swap(A[i], A[min])	Max of n-1

The loop invariant is that all elements in A with index < j are sorted. It starts out with j = 1, so no elements are sorted. At each step, we will add the minimum unsorted element to the end of the sorted section. When the algorithm is done, j = n+1, so all n elements of A are sorted.

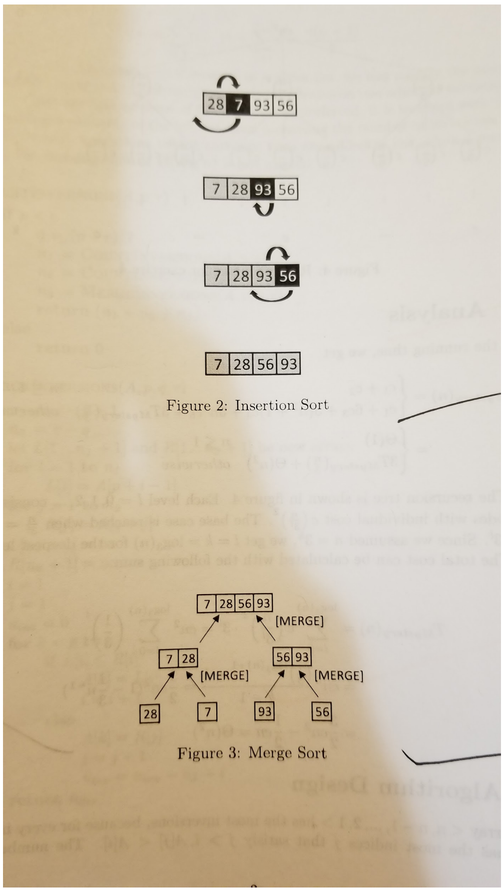
It only needs to run n-1 times. When it runs for A[n-1], it determines if A[n-1] < A[n] and swaps them. If A[n-1] < A[n], it leaves them and A[n] will be the greatest element in the array. If A[n-1] >= A[n], it will swap them, and the new A[n] will be the greatest element in the array. Since we know what A[n] will be in either case, we don't need to check the last element.

The best case is if it's sorted.
$T(n) = c1\ n + c2\ n\text{-}1 + c3\ ((n-1)(n) + (n-1)(2) - (n-1)(n)/2) + c4\ ((n-1)(n) + (n-1)(1) - (n-1)(n)/2) + c5\ (0) + c6\ (n-1) + c7(0)$
We'll drop the cx constants and the 0^{th} order polynomials, so we can simplify to:
$= n + n + n^2 - n + 2n - n^2/2 + n/2 + n^2 - n + n - n^2/2 + n/2 + n$
$= n^2 + 5n = \Theta(n^2)$

The worst case is the best case, except with c5 and c7 needing to be run every time.
Worst case $T(n) =$ Best case $+ c5\ ((n-1)(n) - (n-1)(n)/2) + c7\ (n-1)$
$\Rightarrow$ Best case $+ n^2 - n - n^2/2 + n/2 + n$
$= (3n^2)/2 + 11n/2$
$= \Theta(n^2)$
Best case and worst case both grow at an n^2 rate.

The most frequently run line is the inner for loop (c3). The time is $c3\ ((n-1)(n) + (n-1)(2) - (n-1)(n)/2) = c3\ (n^2/2 + 3n/2 - 2)$.

3 Etienne Buhrle



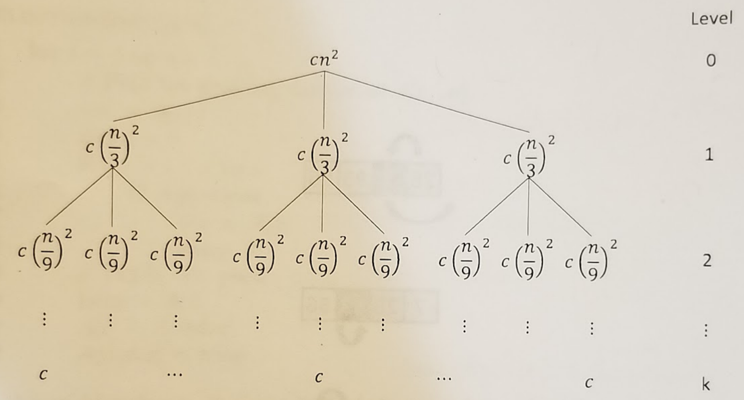


Figure 4: Recursion tree for exercise 4

4 Analysis

For the running time, we get

$$T_{Mystery}(n) = \begin{cases} c_1 + c_2 & n \leq 1 \\ c_1 + 6c_3 + 5(n^2+1)c_4 + 5n^2c_5 + 3T_{Mystery}(\frac{n}{3}) & otherwise \end{cases}$$
$$= \begin{cases} \Theta(1) & n \leq 1 \\ 3T_{Mystery}(\frac{n}{3}) + \Theta(n^2) & otherwise \end{cases}$$

The recursion tree is shown in figure 4. Each level $l = 0, 1, 2, ...$ consists of 3^l nodes with individual cost $c\left(\frac{n}{3^l}\right)^2$. The base case is reached when $\frac{n}{3^l} = 1 \Leftrightarrow n = 3^l$. Since we assumed $n = 3^k$, we get $l = k = \log_3(n)$ for the deepest level.

The total cost can be calculated with the following sum

$$T_{Mystery}(n) = \sum_{l=0}^{\log_3(n)} c\left(\frac{n}{3^l}\right)^2 \cdot 3^l = cn^2 \sum_{l=0}^{\log_3(n)} \left(\frac{1}{3}\right)^l$$
$$= cn^2 \frac{\left(\frac{1}{3}\right)^{\log_3(n)+1} - 1}{\frac{1}{3} - 1} = \frac{3}{2}cn^2(1 - \frac{1}{3}n^{-1})$$
$$= \frac{3}{2}cn^2 - \frac{1}{2}cn = \Theta(n^2)$$

5a, 5b Saara Luna

b. Implementation

```
Merge_sort_inv_count(A, p, r)
//Returns total inversions in A, sorts A as side effect
If p < r
   q = [(p+r)/2]
   //Recursively calculate inversions in each half
   num_inv1 = Merge_sort_inv_count(A, p, q)
   num_inv2 = Merge_sort_inv_count(A, q + 1, r)
   num_inv = num_inv1 + num_inv2
   return Inversion_count_merge(A, p, q, r, num_inv)

Inversion_count_merge(A, p, q, r, num_inversions)
n1 = q - p + 1
n2 = r - q
Let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
For i = 1 to n1
   L[i] = A[p + i - 1]
For j = i to n2
   R[j] = A[q + j]
L[n1 + 1] = infinity
R[n2 + 1] = infinity
i = 1
j = 1
for k = p to r
   if L[i] <= R[j]
      A[k] = L[i]
      i = i + 1
   else A[k] = R[j]
      num_inversions = num_inversions + (n1 + 1 - i)
      j = j + 1
return num_inversions
```

c. Reasoning: Whenever merge_sort places an element from the right side into the next sorted level of the array instead of one from the left side, L[i] > R[j]; additionally, because the original list was split between higher and lower indices, all elements in L have lower original indices than elements of R, making L[i] and R[j] an inversion. Since L[i+1] > L[i] for all i in $1..n_1$, the

inversion conditions hold for all elements of L at and above L[i]; therefore, when finding L[i] > R[j], we know that there are $n_1 + 1 - i$ inversions with R[j]. Since each sublist is sorted afterwards, no extraneous inversions will be found.

d. Runtime reasoning: num_inv = num_inv1 + num_inv2 is constant time and thus is dwarfed by the original complexity, causing no change for the main function. For the merge function, the added statement under the else loop will run n/2 times, and the return statement is of course constant time, giving the modified merge function the same Θ(Cn) runtime as the original. Since both functions' complexity matches that of the original, the new algorithm will have the same worst-case complexity: Θ(nlgn)