CS360 - Assignment 1

1.2-2 Note that $lg()$ represents $log_2()$

$$\begin{aligned} 8n^2 &= 64nlg(n) \\ \Rightarrow n &= 8lg(n) \end{aligned}$$

This equation cannot be solved analytically, instead we will tablulate values to approximate the answer:

$$\begin{aligned} n = 16 &\quad 8lg(16) = 32 \\ n = 32 &\quad 8lg(32) = 40 \\ n = 64 &\quad 8lg(64) = 48 \\ n = 40 &\quad 8lg(40) = 42.6 \\ n = 44 &\quad 8lg(44) = 43.7 \end{aligned}$$

Hence for $n < 44$ insertion sort is faster than merge sort.

1.2-3 Again the equation $2^n = 100n^2$ cannot be solved analytically so tabulate values:

$$\begin{aligned} n = 10 &\quad 2^{10} = 1024 < (100)(10)^2 = 10000 \\ n = 20 &\quad 2^{20} = 1048576 > (100)(20)^2 = 40000 \\ n = 15 &\quad 2^{15} = 32768 > (100)(15)^2 = 22500 \\ n = 14 &\quad 2^{14} = 16384 < (100)(14)^2 = 19600 \end{aligned}$$

Hence for $n > 15$ the exponential algorithm will take longer, i.e. the polynomial algorithm is faster.

2.2-3 On average, half of the elements will have to be checked (assuming the element being searched for is equally likely to be any element in the array). This can be shown since each element will have $1/n$ probability of being the desired element and for the i^{th} element will have had to have searched i elements. Thus the average over all the elements can be written as

$$\begin{aligned} \frac{1}{n}\sum_{i=1}^{n} i &= \frac{1}{n}\left(\frac{n(n+1)}{2}\right) \\ &= \frac{n+1}{2} \end{aligned}$$

$\Rightarrow T(n) \approx c\frac{n+1}{2} = \Theta(n)$ on average.

The worst case is when the element is *not in* the array resulting in every element being checked.
$\Rightarrow T(n) \approx c(n+1) = \Theta(n)$ worst case (since we must also check that there are no more elements).

In both the average and worst case, the running time of linear search will grow linearly, i.e. $\Theta(n)$.

2-2 (a) For correctness we need to show not only that the algorithm terminates with $A'[1] \leq A'[2] \leq A'[3] \leq ... \leq A'[n]$, but additionally that the elements of A' are simply a permutation of he elements of A, i.e. that A' contains *all* the elements of A.

(b) The loop invariant for the loop 2-4 is that $A[j]$ is the smallest element in $A[j...n]$ and that $A[j...n]$ is a permutation of those elements.
Initialization - For $j = n$, $A[j...n] = A[n]$ is a single element which trivially satisfies the loop invariant.
Maintenance - By the loop invariant, at any iteration $A[j]$ is the smallest element in $A[j...n]$, so the only action that can occur within the loop is $A[j-1]$ is exhanged with $A[j]$ if $A[j] < A[j-1]$ (line 4). Hence $A[j-1]$ will be the smallest element in $A[j-1...n]$ and since $A[j...n]$ was a permutation, $A[j-1...n]$ must also be. Therefore after the iteration $A[j-1]$ is the smallest element of $A[j-1...n]$ and $A[j-1...n]$ is a permutation.
Termination - The loop terminates when $j = i$ which by the loop invariant gives $A[i]$ is the smallest element of $A[i...n]$ and $A[i...n]$ is a permutation.

(c) The loop invariant for the outer loop is that $A[1...i-1]$ are the smallest elements from $A[1...n]$ in sorted order with $A[i...n]$ containing the remaining elements.
Initialization - For $i = 1$ $A[1...i-1] = \emptyset$ thus no elements trivially satisfy the loop invariant.
Maintenance - For a given iteration i, $A[1...i-1]$ are the smallest (sorted) elements. From part (b), after the inner loop executes, $A[i]$ is the smallet element from $A[i...n]$ and $A[i...n]$ is a permutation of those elements. Since by the assumption $A[1..i-1]$ are the smallest elements from $A[1..n]$, $A[i]$ is greater than all $A[1..i-1]$ (but smaller than $A[i+1..n]$). Therefore $A[1...i]$ contains the smallest (sorted) elements and $A[i+1...n]$ is a permutation of the remaining elements.
Termination - The loop terminates when $i = n$ which by the loop invariant states $A[1...i-1] = A[1...n-1]$ are sorted and the smallest elements from $A[1..n]$. However, this means that $A[n]$ must be the *largest* element from $A[1..n]$ and hence $A[1..n]$ is sorted (and a permutation of the original elements). Therefore, the entire array is sorted.

(d) For this version of bubble sort (fixed iteration loops), define t_i as the number of times the if *condition* is **true** for outer loop iteration i (**Note:** this is **not** the same as the number of times the if *statement* executes as the statement will always execute for every iteration of the inner loop).

BUBBLESORT(A, n)	**Cost**	**# Iterations**
1 **for** $i = 1$ **to** $n - 1$	c_1	$\displaystyle\sum_{i=1}^{n} 1$
2 **for** $j = n$ **downto** $i + 1$	c_2	$\displaystyle\sum_{i=1}^{n-1}\left(\sum_{j=i}^{n} 1\right)$
3 **if** $A[j] < A[j-1]$	c_3	$\displaystyle\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 1\right)$
4 exchange $A[j]$ with $A[j-1]$	c_4	$\displaystyle\sum_{i=1}^{n-1} t_i$

Hence

$$T(n) = c_1\left(\sum_{i=1}^{n} 1\right) + c_2\left(\sum_{i=1}^{n-1}\left(\sum_{j=i}^{n} 1\right)\right) + c_3\left(\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 1\right)\right) + c_4\left(\sum_{i=1}^{n-1} t_i\right)$$

Worst Case

In the *worst case*, the `if` condition will be **true** for *every* iteration of the inner loop when the array is sorted in *decreasing* order giving $t_i = \left(\sum_{j=i+1}^{n} 1\right)$ for all i (thus whenever line 3 executes, line 4 will also execute) giving

$$T_W(n) = c_1\left(\sum_{i=1}^{n} 1\right) + c_2\left(\sum_{i=1}^{n-1}\left(\sum_{j=i}^{n} 1\right)\right) + c_3\left(\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 1\right)\right) + c_4\left(\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 1\right)\right)$$

Evaluating the first summation gives

$$\sum_{i=1}^{n}(1) = n$$

Evaluating the second summation (where we let $k = n - i + 1$)

$$\sum_{i=1}^{n-1}\left(\sum_{j=i}^{n} 1\right) = \sum_{i=1}^{n-1}(n-i+1) = \sum_{k=2}^{n} k = \left(\sum_{k=1}^{n} k\right) - 1 = \frac{n(n+1)}{2} - 1$$

Evaluating the third summation (where we let $k = n - i$)

$$\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 1\right) = \sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$$

Hence worst case,

$$\begin{aligned} T_W(n) &= c_1 n + c_2\left(\frac{n(n+1)}{2} - 1\right) + c_3\frac{(n-1)n}{2} + c_4\frac{(n-1)n}{2} \\ &= \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}\right)n^2 + \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2} - \frac{c_4}{2}\right)n - c_2 \\ &= \Theta(n^2) \end{aligned}$$

Therefore worst case running time is asymptotically the same as insertion sort - $\Theta(n^2)$.

Note: For the *best case* the `if` condition will be **false** for every iteration of the inner loop (when the input array is already sorted in *increasing* order). However, this only eliminates teh c_4 term still resulting in $\Theta(n^2)$. Thus, bubblesort will *always* run asymptotically quadratic.

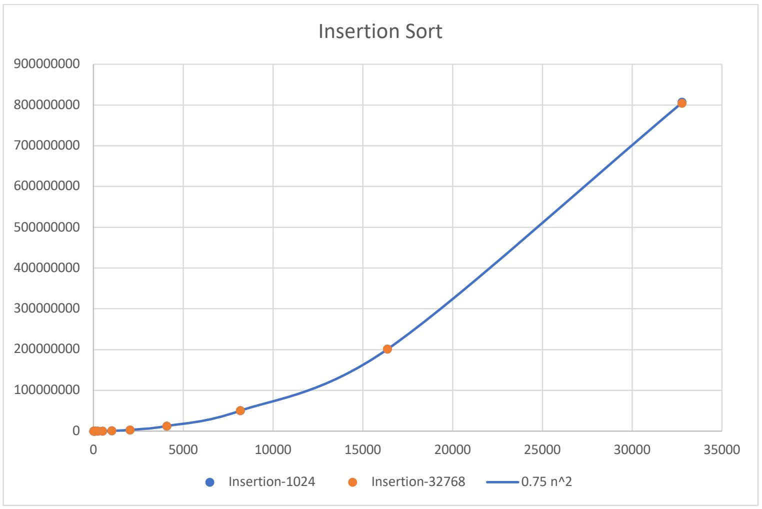
Implementation:

Is there a table showing the empirical data for both element ranges (but it should *not* contain the calculated trend values)?

Is there a graph with:

1. The empirical data for both element ranges plotted as **only** *points*
2. The calculated trend for *each* element range plotted as **only** a *curve*
3. A legend listing each data set and showing the trend curve function with the approximated *constant*
4. Properly labeled graph axes with a linear scale x-axis

Insertion Sort

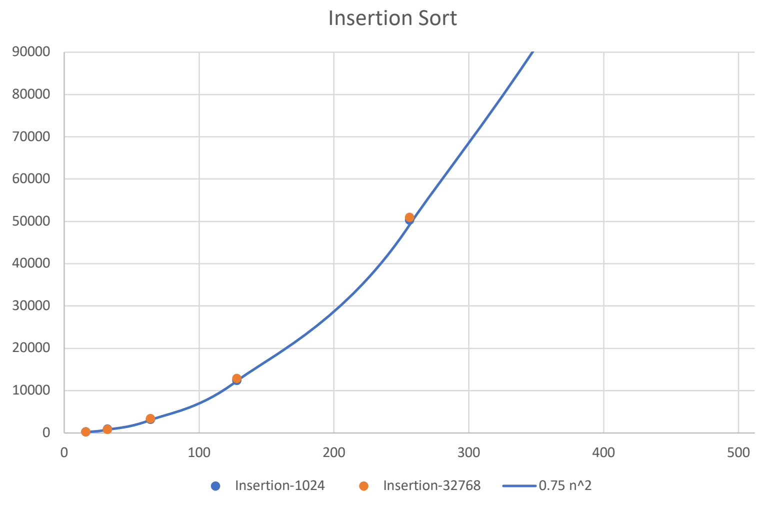


Note: The last data point was ignored due to numerical precision overflow.

Constants

- **InsertionSort:** $1024 \approx .75n^2$, $32768 \approx .75n^2$

Small Data Set



The graph above illustrates that the asymptotic curve fits the empirical data relatively well even for small data set sizes.