1. Let us assume by contradiction that there are two different permutations of size n, \( a = \{a_1...a_n\} \) and \( b = \{b_1...b_n\} \), that have the same set of inversions. Since these permutations are different, there must be at least one position (as a matter of fact two...) where they differ. Let us denote by \( i \) the first index that’s different between them. In other words, \( a_i \neq b_i \) but \( a_1 = b_1 \) all the way up to \( i - 1 \) (unless \( i = 1 \), which is possible). Since \( a \) and \( b \) are permutations on the same set of numbers, we know that the number corresponding to \( a_i \) appears in \( b \), but at a different position, say \( j \). We know \( j > i \) (since we said \( i \) was the first different position). Also, we know that the number corresponding to \( b_i \) appears in \( a \), but at a different position, say \( k \), s.t. \( k > i \). Put together - it means that the pair \( a_i, b_i \) are swapped w.r.t. one another in \( a \) and \( b \), so they must be an inversion either in \( a \) or \( b \), but not both. This contradicts our initial assumption that \( a \) and \( b \) have the same set of inversions.

2. (a) True, since \( 2^{n+1} = 2 \cdot 2^n \)
(b) \( f(n) = O(g(n)) \) implies \( 2f(n) = O(2g(n)) \) – False. See for example \( f=2n \) and \( g=n \).

3. This follows simply from the log change base rule. We know that \( \log_a x = \log_a b \cdot \log_b x \) for every two positive constants \( a, b \) and every \( x > 0 \). Therefore \( \log_a b \) is our constant and any positive number can be our \( n_0 \)

4. If \( f = O(g) \) then there are two constants \( c \) and \( n_0 \) s.t. \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \). If \( g = O(h) \) then there are two constants \( d \) and \( n_1 \) s.t. \( g(n) \leq d \cdot h(n) \) for all \( n \geq n_1 \). Combining the two, we get \( f(n) \leq c \cdot d \cdot h(n) \) for all \( n \geq \max(n_0, n_1) \), so we can use \( c \cdot d \) as our constant. Notice that you can’t assume the same constant for both steps.

5. (a) \( T(n) = 2T(n/2) + n^3 \). We can apply the master theorem here. We have \( n^{\log_2 2} = n \) and \( f(n) = n^3 \). This is case 3 if we can show that there is a constant \( 0 < c < 1 \) and a constant \( n_0 \) such that for every \( n > n_0 \), \( a \cdot f(n/b) \leq c \cdot f(n) \), or in other words - \( 2 \cdot (\frac{2}{3})^3 = n^3/4 \leq c \cdot n^3 \). This is true for every \( n \geq 1 \) and \( c = 1/4 \), for example. Therefore \( T(n) = O(n^3) \).
(b) \( T(n) = T(8n/11) + n \). This falls into case 1 of the master theorem since \( n^{\log_{11} 8} = 1 = \Omega(n) \), then \( T(n) = O(n) \).
(c) \( T(n) = 16T(n/4) + n^2 \) This is case 2 of the master theorem since \( n^{\log_4 16} = n^2 \) and \( f(n) = n^2 \). Therefore, \( T(n) = n^2 \log n \).
(d) \( T(n) = 7T(n/2) + n^2 \log n \) We have \( a = 7, b = 2 \) and \( f(n) = n^2 \log n \). \( \log_2 7 = 2.8074 \), so this is case 1 and \( T(n) = O(n^{\log_2 7}) \)
(e) \( T(n) = 2T(n/4) + \sqrt{n} \). This is very similar to (c) above - hence case 2 applies and \( T(n) = O(\sqrt{n} \log n) \).
6. \[ T(n) = \sum_{j=2}^{n} (a + (j - 1)c) = \sum_{j=2}^{n} (a) + c \sum_{j=2}^{n} (j - 1) = a(n - 1) + c \sum_{j=1}^{n-1} j = a(n - 1) + c\frac{n(n-1)}{2}. \]

Rearranging the terms we get \[ an - a + n^2 \frac{c}{2} - n \frac{c}{2}. \] So, if we set \[ A = \frac{c}{2}, B = a - \frac{c}{2} \] and \[ C = -a \] we get a term of the form \[ T(n) = An^2 + Bn + C. \] Since \( c > 0 \), then \( A > 0 \) as well, so it’s a quadratic term.

7. See drawing. I can be quite flexible with any handling of terminal nodes where \( T(n/4) < 1 \).

8. (a) To implement a stack, use a Max-HEAP priority queue. Initiate an index \( i = 0 \). To implement push, assign the new item the priority \( i \) and increment \( i \) by 1. To pop, just extract-max (notice that the operations are logarithmic and not constant).

(b) To implement a queue, use a Min-HEAP priority queue. Initiate an index \( i = 0 \). To implement push, assign the new item the priority \( i \) and increment \( i \) by 1. To pop, just extract-min (notice that here too, the operations are logarithmic and not constant).

9. Split3-sort (known as Stooge sort) is an example of an inefficient, yet correct, sorting algorithm.

(a) Proof of correctness (by induction):

- Base case: Notice that the recursion only takes place if the array is of at least size 3 \((p + 1 > r\), per line 4\). Therefore there are two base cases: If the array is of size 1, it is trivially sorted. If the array is of size 2, then the if statement in lines 1-2 takes care of the sorting.
- Inductive hypothesis: By induction, Split3-sort\((A,i,j)\) sorts correctly any array of size \(1..n-1\).
- Proof of correctness for \( n \): By inductive hypothesis, line 6 sorts \( A[p..r-k] \) - the first two thirds. Therefore, we know that the middle third (the current \( A[p+k..r-k] \)) does not contain the smallest third of the \( A[p..r-k] \) segment. Line 7 sorts the last two thirds (\( A[p+k..r] \)). Since the middle third took part in the previous sorting, we know it does not contain the smallest third of its segment. Hence, the one thing we can be sure about is that after line 7 is executed, positions \( A[r-k..r] \) – that is, the last third of the array, contain the largest one third of the numbers, sorted (can you see why?). With this part out of the way, line 8 sorts the remaining numbers, the lower two-thirds, with respect to themselves and in the end the entire array is sorted.

(b) The recurrence formula is: \[ T(N) = 3T\left(\frac{2N}{3}\right) + O(1). \] Let’s use the master theorem.

This formula falls into case I, so the recursive part dominates. Here \( a = 3 \) and \( b = \frac{2}{3} \).

According to the master theorem, it means that the asymptotic runtime is \( O(N^{\log_b a}) = O(N^{\log_3 3}) = O(N^2) \approx O(N^{2.709}) \)

(c) Obviously, it is much less efficient than our \( O(N \log N) \) and even \( O(N^2) \) sorting algorithms.