1i)

```
Sort(A,n)
        For i in range of n-1
                smallest = i
                For j in range of i to n-1
                        if A[j] < A[smallest]: smallest = j
                A[i], A[smallest] = A[smallest], A[i]
```

1ii) Let T(n) denote the running time of the sorting algorithm above. Then the worst case analysis of the algorithm is shown below:

$$T(n) = c_1(n) + c_2(n-1) + c_3\left(\sum_{i=1}^{n-1} t_i\right) + c_4\left(\sum_{i=1}^{n-1} t_i - 1\right) + c_5(n-1)$$

$\sum_{i=1}^{n-1} t_i \leq 1+2+3+4+\ldots+n$ which gives $\sum_{i=1}^{n-1} t_i \leq \frac{n(n+1)}{2}$

Thus, $\sum_{i=1}^{n-1} t_i$ is $O(n^2)$. If $t_i = i$, then $\sum_{i=1}^{n-1} t_i$ is $\Omega(n^2)$.

Since the algorithm is both $O(n^2)$ and $\Omega(n^2)$, it is also $\Theta(n^2)$. This makes sense as the algorithm will always have the same worst case time complexity, because the original order of the n elements will not affect how many executions are done.

2)

Algorithm description in plain English:

First, initialize an answer variable as 1. While y is greater than 0, multiply the answer variable by x only if y's last bit is 1, then multiply x by x and shift y's bits to the right once. Perform the while loop until y is no longer greater than 0 due to the bits shifting, then return the answer variable.

Proof of Correctness:

- Claim (Loop invariant): At every 1-bit encountered through right shifting, the answer variable will be equal to x to the power of the equivalent of all the 1-bits processed up to that point (for example, if the bits "101" have been processed, the answer variable will be equal to $x^5$).
- Initialization: At the beginning of the algorithm, the answer is set to 1 and therefore the answer is true for $x^0$ since 0 1-bits have been processed.
- Maintenance: The while loop works by increasing 1 (the initial answer variable) once for each 1-bit in the binary representation of y, rather than increasing x using y-1 steps. If a 1-bit is encountered, the answer will be raised to the power of whatever value that 1-bit represents in the original binary representation, due to the fact that x is repeatedly being multiplied by itself for each bit. Thus the invariant holds from iteration to iteration.

- Termination: The while loop ends when y is no longer greater than 0 due to the bits shifting. When this occurs, x must have grown from $x^0$ initially to $x^y$ since all of y's 1-bits have been "used".

Complexity Analysis:

The variable y has n bits, and each bit must be iterated through. One multiplication is performed for every 0-bit and two multiplications are performed for every 1-bit which occurs in constant time. Therefore the total time complexity is O(n).

3)

If $c < 1$, $g(n)$ is $\Theta(1)$ since there exists constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 * 1 \leq g(n) \leq c_2 * 1$ for all $n \geq n_0$.
If $c = 1$, $g(n)$ is $\Theta(n)$ since there exists constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 * n \leq g(n) \leq c_2 * n$ for all $n \geq n_0$.
If $c > 1$, $g(n)$ is $\Theta(c^n)$ since there exists constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 * c^n \leq g(n) \leq c_2 * c^n$ for all $n \geq n_0$.

4a) The algorithm's upper bound would be $O(n^3)$ since the two for loops require $n^2$ iterations, the adding of array entries A[i] through A[j] requires at most n iterations, and the storing of results in B[i, j] requires constant iterations. Therefore the algorithm is $O(n^2(n + 1))$, or $O(n^3)$.
4b) Let T(n) denote the running time of the algorithm. Then the worst case analysis of the algorithm is shown below:

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

If $t_i = i$, then $\left(n\sum_{i=1}^{n} t_i\right)$ is $\Omega(n^3)$.

4c)

Algorithm description in plain English:

First, initialize a current_sum variable as 0. For i in the range of 1 to n, set the sum to A[i]. Then, begin a nested loop for j in the range of i+1 to n. In the nested loop, increment the sum by A[j] and store the result in B[i, j].

Proof of Correctness:

- Claim (Loop invariant): At the start of each iteration of the outer for loop (indexed by i), the array B will have entries in which B[i, j] (for i < j) contain the sum of array entries A[i] through A[j].
- Initialization: Just before the first iteration, none of the elements of A have been processed and therefore the current_sum is 0, and array B is empty. Before the first

iteration of the nested loop, the current_sum is equal to A[i]. This current_sum is not entered into array B since i ≥ j.

- Maintenance: The two for loops work by keeping track of the current sum and incrementing over time, rather than performing a recalculation $n^2$ different times. For any j, the current_sum will be equal to the sum of all entries of A[i] + A[...] + A[j]. Results will be stored in B for every iteration of j since i < j. Thus the invariant holds from iteration to iteration.
- Termination: Both for loops end after i and j have performed all their iterations. When this occurs, array B will have been filled with all of its required entries.

Complexity Analysis:

For every iteration of i, j will iterate through the following elements of A. One addition is performed for every j iteration which occurs in constant time. Therefore the total time complexity is O(n).