Asymptotic Complexity

Q1. Do worst-case analysis of the following INSERTION-SORT procedure and determine the time each statement takes and the number of times each statement is executed. Assume that the input is reverse sorted.

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)\,.$$

For worst-case analysis (where the input is reverse sorted) t_j=j

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

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

T(n) is in O(n^2)

Q2. Let A be an array of n elements. Assume array indices range from 1 to n. Consider the pseudocode below. What is the running time T(n) of `SomeFunction()`?

```
Sum(A, x) {
  Val = 0;
  for i= 1 to x
     Val += A[i]*A[i];
}
SomeFunction(A, n) {
  for (i= 1 to n/2) {
      for (j= n to (n/2)+1) {
           A[i] += A[j]+Sum(A,i);
      }
  }
}
```

Sum() takes O(i) time, where i is the variable in SomeFunction().

Analyzing the time for the statement `A[i] += A[j]+Sum(A,i);` in SomeFunction(), we can write:

$\sum_{i=1}^{n/2} (n/2).c.i$, where c is a constant.

$= cn/2 \sum_{i=1}^{n/2} i$

$= cn^2(n+2)/16$

After simplification: T(n) is in $O(n^3)$

Q3. Consider the following code.

```
int compute(int *array, int i, int j) {
  int k, out=1;
  for (k=i; k<=j; k++) {
    out *= array[k];
  }
  return (out);
}

int main() {
 int i, j, m, n;
 ...
 m = n/2;
  ...
  /* Assume that n is an even number. A is a
     single dimensional array size n integers and
     B is a 2-dimensional array of size n x n integers */
  for (i=0; i<n; i++) {       /* outer for-loop */
    for (j=i+1; j<m; j++) {    /* inner for-loop */
      B[i][j] = compute(A, i, j);
    }
  }
  ...
}
```

What is the asymptotic time complexity in Big-Oh notation of the function **compute()? Write your answer in terms of i and j.**

`O(j-i+1)`

____

Fill in the blank for the number of steps executed in the **inner for-loop.**

$\sum_{j=i+1}^{m}$ (___`c(j-i+1)`_____) // This is for the statement *inside* the inner for-loop

Simplify the expression you obtained above.

c (2+3+ … m-i+1) = c (m-i) (m-i+1)/2

____

What is the asymptotic time complexity in Big-Oh notation of the **inner for-loop? Write your answer in terms of m and i.** ______`O((m-i)`^{2}`)`___________________

Q4-i What is the asymptotic time complexity of the following program fragment. Show your working.

```
for (i=1; i<=n; i*=2) {
    j = i;
 }
```

We have to find the number of iterations of the for-loop.

The first iteration is i = 1 = 2^0

The second iteration is i = 2 = 2^1

The third iteration is i = 4 = 2^2

The last iteration is $i = n = 2^x$

$2^x = n$ hence $x = \log_2(n)$

The asymptotic complexity is O`(log(n))`

Q4-iia What is the asymptotic time complexity of the following program fragment. Give both the upper bound and lower bound for this fragment. Show your working.

```
int x, y, n;
...
/* P is a 1-D array of size n integers and
   W is a 2-D array of size n x n integers */
for (x=0; x<n; x++) {
  for (y=x+1; y<n; y++) {
    W[x][y] = func(P, x, y);
  }
}
...
```

The function func() called from the main program (above) is defined as follows:

```
int func(int *array, int i, int j)
{
  int ii, val=0;
  for (ii=i; ii<=j; ii++) {
    val += array[ii];
  }
  return (val);
}
```

Upper bound : $O(n^3)$. Lower bound : $\Omega(n^3)$

Q4-iib What is being stored in the 2-D W array in Q4-iia?

For `i<j`, `W[i][j]` contains the sum `P[i]+P[i+1]+ … + P[j]`

Q4-iic The program fragment in Q4-iia is not very efficient. Rewrite it to improve its time complexity.

We can use the value of W already computed in the previous iteration. This will improve the time complexity to O(n^2).

```
for (x=0; x<n; x++) {
      W[x][x] = P[x];
      for (y=x+1; y<n; y++) {
            W[x][y] = W[x][y-1] + P[y];
      }
}
```

Q5.

You are given the following two nxn-dimensional matrices. The first matrix called "Intern Preference Matrix" stores information about n interns that want to apply for jobs at n different companies. Each row corresponds to one intern and indicates his/her order of preference for the employers. For example: Intern-1's first preference is Employer-3, second preference is for Employer-1, third preference is Employer-5 and so-on.

The second matrix called "Employer Preference Matrix" stores preference information of each of the n employers for the intern applicants. For example: Employer-1's first preference is Intern-2, second preference is for Intern-4 and so-on.

Intern Preference Matrix

Intern 1 (I_1)	E_3	E_1	E_5	E_8	...	E_2
Intern 2 (I_2)	E_5	E_2	E_1	E_7	...	E_3
Intern 3 (I_3)	...	...	...	...	...	...
...	...	...	...	...	...	...
Intern n (I_n)	...	...	...	...	...	...

Employer Preference Matrix

Employer 1 (E_1)	I_2	I_4	I_5	I_1	...	I_3
Employer 2 (E_2)	I_6	I_2	I_3	I_4	...	I_1
Employer 3 (E_3)	...	...	...	...	...	...
...	...	...	...	...	...	...
Employer n (E_n)	...	...	...	...	...	...

We have a number of queries such as "Does E_x prefer I_a over I_b?" or "Does I_x prefer E_y over E_z?" and we want to answer **each** query in O(1) time.

How would you restructure/re-store the above information so that we can answer such queries in O(1) time? You are only allowed to use arrays. You cannot use other data structures such as hash tables, etc. **Clearly** explain your answer and the **space** required by your solution.

Create a new 2-D matrix, whose rows are numbered from I1 to In and columns are numbered from E1 to En. Using Intern Preference Matrix, fill in this new matrix with the ranks of employers. For example for Intern I1, rank of E3 is 1, rank of E1 is 2, rank of E5 is 3, etc. Create a similar ranking matrix from the employers' preference perspective. Now, you are able to answer the queries mentioned above in constant time (i.e. O(1)) by simply comparing the ranks in the corresponding cells.