Data Structures 1 - Homework 3 - Abraham Murciano

1) a)
```
bubbleSort(list L){
    Node* i = NULL;          // i starts at end of L, and goes down
    while(i ≠ L->next){      // continue until i goes down to the second element
        Node* j = L;         // j starts at beginning of L, and goes up
        while(j->next ≠ i){  // continue until j goes up to the 2nd last element
            if(j->data > j->next->data){   // if j^th element > (j+1)^th element
                swap(j, j->next);          // swap j^th & (j+1)^th elements
            }
            j = j->next;     // increment j
        }
        i = j;               // decrement i
    }
}
```

b) yes. they're both $O(n^2)$. The algorithm is the same for both, and the complexity of each step is the same.

2)
```
function(List L1, List L2){
    List L3;
    for(int k = 1; k ≤ L1.size; k++){
        Node* ck = L3.insertEnd(0);
        for(int i = 1; i ≤ k; i++){
            Node* a = L1.start;
            for(int j=1; j≤ i; j++){
                a = a->next;
            }
            Node* b = L2.start;
            for(int j = 1; j≤ k-i+1; j++){
                b = b->next;
            }
            ck->data += a->data * b->data;
        }
    }
    return L3;
}
```

complexity is $O(n^3)$

3) a) Singly linked list with a pointer to the first frenzied task, a pointer to the last frenzied task and a pointer to the last regular task

b)
```
void insert(Task t){
    node* n = new node;
    n->data = t;
    if(t.code == 1){
        n->next = Manager.head;
        Manager.head = Manager.firstFrenzied = n;
    } else if(t.code == 2){
        n->next = Manager.lastFrenzied->next;
        Manager.lastFrenzied->next = n;
    } else if(t.code == 3){
        n->next = NULL;
        Manager.lastRegular = Manager.lastRegular->next = n;
    }
}
```

3) c)
```
if (Manager.head ≠ NULL){
    Task t = Manager.head->data;
    Manager.head = Manager.firstFrenzied = Manager.head->next;
    return t
} else {
    return NULL;
}
```

4)
```
Node*& operator[](const Matrix& M, int x){   // complexity is O(1)
    return M.arrayX[x];
}

Node  operator[](Node* head, int y){   // complexity is O(n)
    while (head->y < y){
        head = head->nextY;
    }
    return (head->y == y)? *head : Node(0);
}

Matrix addMatrix(const Matrix& A, const Matrix& B){
    Matrix c(A.m, A.n);
    for(int i=0; i<c.m; i++){          // O(n)
        for(int j=0; j<c.n; j++){      // O(n²)
            Node n = A[i][j] + B[i][j];   // O(n³)
            C.insert(n, i, j);
        }
    }
    return c;
}
```

complexity is $O(n^3)$