```
Ramon Rovirosa
CS130B
Homework 2

//problem 1
//Coin Change


    //returns array: number_of[quarters,dimes,nickes,pennies]
    int* coinChange(int change){
        //Quarter == $25
        //Dimes   == $10
        //Nickels == $5
        //Pennies == $1

        int numberCoins[4];


        while(change >= 25){
            numberCoins[0]++;
            change-=25;
        }
        while(change >= 10){
            numberCoins[1]++;
            change-=10;
        }
        while(change >= 5){
            numberCoins[2]++;
            change-=5;
        }
        while(change >= 1){
            numberCoins[0]++;
            change-=25;
        }
        return numberCoins;
    }

    The greedy algorithm for the coin change problem first gives the max number
    of quarters possible, then the max number of dimes possible, and then the
    max number of nickels and then pennies.

    Since it is always true that if you have one coin of higher value e.g. a
    quarter you need more than one coins of lower values, e.g. 2 dimes + 1 nickel to
    make the same amount of change. The greedy algorithms makes sure that there are no
    more than the necessary number of coins, and is therefore optimal.



//problem 2
//matching dance couples

    struct Dancer
    {
        string gender;
        int height;
    };

    struct Pair{
        Dancer male;
        Dancer female;
    };

    Pair *danceCouples(Dancer dancers[]){

        //first sort all dancers based on height;
        //shortest to tallest;
        Dancer *sortedDancers=quickSort(dancers,height);

        Dancer maleDancers[];
        Dancer femaleDancers[];

        int j=0;
        int k=0;
        int l=0;

        for(i:0=>sortedDancers.length){
            if(sortedDancers[i].gender == "Male")
                maleDancers[j++] = sortedDancers[i];
            else
                femaleDancers[k++] = sortedDancers[i];
        }

        Pair matchedCouples[];

        //find the max, weather the max is males or females.
        int max = (j>k)?j:k;

        j=0;
        k=0;
        while(max>j || max>k){
            if( femaleDancers[k].height <= maleDancers[j].height + 3){
              matchedCouples[l].male = maleDancers[j++];
              matchedCouples[l++].female = femaleDancers[k++];
            }
            else if(femaleDancers[k].height > maleDancers[j].height )
                j++; //move on to next male
```

```
            else
                k++; //no match move on to next female

        }

        return matchedCouples;

    }

    //Complexity
    Quicksort: O(n*log(n))
    Split array into male & female: O(n)
    Pair  dance couples: O(n)

    Overall Complexity:
    T(n) = O(n*log(n)) + O(n) + O(n)

    Complexity: O(n*log(n))


    Optimal Solution: Yes

    The algorithm involves first sorting the men & women by height shortest to tallest.
    Then the idea is to match the couples beginning with the two shortest in the group and moving
    along to the tallest pairs.

    By sorting the couples first & then matching them in order, we find that all couples within the expected
    range are matched. This gives us an optimal number of pair matches between couples.



//problem 3
//N intervals on the X axis.else

    Greedy strategy interval selection technique:

    1. their starting x locations (smaller ones first)

        No. Will not generate optimal results.

        Given the (x_start, x_end) list:
        { (0,100), (10,20), (30,40) , (70,90)  }

        1. XXXXXXXXXXXXXXXXXXXXXXXXXX
        2. XXX 3. XXXXX 4.XXXX 5.XXXX

        This strategy of selecting intervals would return 1 (the first) interval that overlaps
        all the other intervals, when we know that the optimal result would involve returning
        the 3 intervals: (10,20), (30,40) , (70,90)



    2. their end x locations (smaller ones first)

        Yes, will generate optimal results.

        If you have 1 interval that overlaps 2 intervals in the same region it will pick the
        two intervals since they the 1st of two interval will finish before the one overlapping interval.


    3. their lengths (shorter ones first).

        No. Will not generate the optimal results.

        Given the (x_start, x_end) list:
        { (0,50), (45,55), (50,100)  }

        1.XXXXXXXXX  2.XXXXXXXXX
                3.XXXXX

        This algorithm would pick the interval (45,55) first and return 1 total interval,
        were the optimal solution would be to pick the two largest intervals first instead of the
        3rd smallest one.

        So in the case were a smaller interval overlaps two larger intervals, then this algorithm would not be optimal.

    4. the number of other intervals that they overlap with (fewer overlaps first)

        No. Will not generate optimal results.

        Given the (x_start, x_end) list:
        { (0,20), (50,70), (100,120), (140, 160)
               (10, 60),(69,110),  (110,150),
               (10,60),            (110,150),
               (10,60),            (110,150)
                                              }
        }
        1. XXXXX   2.XXXXX  3.XXXXXXX  4.XXXXX
             5.XXXXXX  6.XXXXXXX 7.XXXXXXX
             8.XXXXXX            9.XXXXXXX
            10.XXXXXX           11.XXXXXXX

        This algorithm will pick 6 first since it has two overlaps then it will pick 1 & then 4.
        Which will generate 3 intervals, instead of the optimal solution which should be 4 intervals returned.



//Problem 4.
```

```
//programs on tape

    Assume n programs of length l1, l2, · · · , ln are to be stored on a tape. Program i is to be
    retrieved with frequency fi. If the program are stored in the order of i1, i2, · · · , in,
    the expected retrieval time (ERT)

    a. Show that storing programs in nondecreasing order of L(i) does not necessarily minimize ERT.

        Given the case where you have n=5 programs of increasing length:

        [f=2.5%,l=1][f=2.5%,l=2][f=2.5%,l=3][f=2.5%,l=4][f=90%,l=5]

        & you try to retrieve the program with length l=5 with f=90% frequency, and
        the other 4 frequencies for the programs add up to 10% (2.5% each)
        the expected retrieval time will be a lost larger than

        the case where you have the n programs sorted in non increasing order and
        choose the largest element length l=5 with 90% frequency, nd
        the other 4 frequencies for the programs add up to 10% (2.5% each)
        in which this would be the most optimal algorithm for minimizing ERT. Since the largest program,
        l=5 would always be chosen always & only first.

        [f=90%,n=1][f=2.5%,n=2][f=2.5%,n=3][f=2.5%,n=4][f=2.5%,n=5] ==> minimum ERT.


    b. Show that storing programs in non-increasing order of fi does not necessarily minimize ERT.

        Given the case were you have all your programs retrieved with the same frequency (which is still non-increasing) but increasing le
        were the largest program is at the end, this strategy would not necessarily minimize ERT.

        * all n programs have same frequency f = 1.
        * increasing length as you go through the tape.
        [f=20%,l=1][f=20%,l=2][f=20%,l=3][f=20%,l=4][f=20%,l=5]

        because the optimal runtime will be the strategy where the largest element is at the beginning of the list will have a more minima

        [f=20%,l=5][f=20%,l=4][f=20%,l=3][f=20%,l=2][f=20%,l=1]

    c. Show that storing programs in non-increasing order of fi/li does minimize ERT. Note: You can assume that the tape is long enough to

        Given the greedy solution fi/li is represented by:
        Greedy solution fi/li:i1,i2,i3,...,ia,...,ib,...,in

        -Only differing in one pair of programs at position a & b,

        The retrieval time before & after ia-ib is the same.
        So we consider the retrieval time change between ia-ib
        Given that the ERT is minimized when f(i=a)/l(i=a) <= f(i=b)/l(i=b)

        Lets assume that we now swap the order of a&b and have b going before a:
        fi/li:i1,i2,i3,...,ib,...,ia,...,in
        Thus we now have that the ERT of: f(i=a)/l(i=a)>f(i=b)/l(i=b)
        But we see that the ERT of accessing a increases by l(i=b), & the ERT of accessing b decreases by l(i=a).
        Thus the total ERT between ia-ib is: l(i=b)f(i=b) - l(i=a)f(i=b).
        But this change is an improvement since: f(i=a)/l(i=a)>f(i=b)/l(i=b)
        Thus: l(i=b)f(i=b) - l(i=a)f(i=b) < 0.
        Which means that if two adjacent programs are out of order (b,a) instead of (a,b) based on the algorithm f/a,
        we can improve the ERT by simply swapping them to be in ERT order.
```