```
MinTotErr[point(x,y), applicablePtSet[p1 ... pn], TotErrSum]
PossibleKStepPts[KstepFunctionPts[Points(x,y)], TotalErrorSum]

main(set of points P, k)
{
    findStepFunctionError(P) and fill MinTotErr
    return findMinimizedErr(k,P) to get the final result
}

findStepFunctionError(point set P){
    if(P.size is zero) MintTotErr.add(p, point set[P], 0)
    for(point p : point set P)
    {
        findStepFunctionError((P - p) - P.tail)
        totalerror = previous element in MinTotErr arraylist + error[P.tail]
        MinTotErr.add(p, point set[P], totalerror)
    }
}

findMinimizedErr(int k,point set P, point set KStepPoints)
{
    temp pointset = P;
       //see if all points are covered in the union of set of points applicable of all k entries,
        by length or iterating over all pts (this current index's applicable point set + the recursive results point set)
    if(k == 0 && P.isEmpty == true) // all points covered, k steps used
    {
        retrieve total errors from MinTotErr for each K point and sum
        for(point p : KStepPoints)
        {
            TotalErrorSum += MinTotErr[p].TotErrSum
        }
        PossibleKStepPts.add[KStepPoints, TotalErrorSum];
    }
    else if (k == 0 && P.isEmpty == false) // k steps used, not all points covered
        return
    else if( k != 0 && P.isEmpty == true) // k steps not fully used, but all points covered
        return
    else // recurse
    {
        //substract applicable point set from given point set,
        temppointset = temppointset - MinTotErr[KStepPoints.last].applicablePtSet
        for(point p : temppointset)
        {
            findMinimizedErr(k-1, temppointset - p, KStepPoints + p)
        }
    }
    return Min(PossibleKStepPts.TotalErrorSum)
}
```

```
MinTotErr[point(x,y), applicablePtSet[p1 ... pn], TotErrSum]
PossibleKStepPts[TotalErrorSum, KstepFunctionPts[Points(x,y)]]

main(set of points P, k)
{
    findStepFunctionError(P) and fill MinTotErr
    return findMinimizedErr(k,P) to get the final result
}

findStepFunctionError(point set P){
    if(P.size is zero) MintTotErr.add(p, point set[P], 0)
    for(point p : point set P)
    {
        findStepFunctionError((P - p) - P.tail)
        totalerror = previous element in MinTotErr arraylist + error[P.tail]
        MinTotErr.add(p, point set[P], totalerror)
    }
}

findMinimizedErr(int k,point set P, point set KStepPoints)
{
    temp pointset = P;
       //see if all points are covered in the union of set of points applicable of all k entries,
        by length or iterating over all pts (this current index's applicable point set + the recursive results point set)
    if(k == 0 && P.isEmpty == true) // all points covered, k steps used
    {
        retrieve total errors from MinTotErr for each K point and sum
        for(point p : KStepPoints)
        {
            TotalErrorSum += MinTotErr[p].TotErrSum
        }
        PossibleKStepPts.add[KStepPoints, TotalErrorSum];
    }
    else if (k == 0 && P.isEmpty == false) // k steps used, not all points covered
        return
    else if( k != 0 && P.isEmpty == true) // k steps not fully used, but all points covered
        return
    else // recurse
    {
        //substract applicable point set from given point set,
        temppointset = temppointset - MinTotErr[KStepPoints.last].applicablePtSet
        for(point p : temppointset)
        {
            findMinimizedErr(k-1, temppointset - p, KStepPoints + p)
        }
    }
    // this returns KstepFunctionPts[Points(x,y)] which will be our K step function that has the minimum error.
    return PossibleKStepPts.get(Min(PossibleKStepPts.TotalErrorSum)
}
```

HW Solution

CS/ECE 374: Algorithms & Models of Computation, Spring 2019 Version: **1.0**

Submitted by:

- ≪**Alan Lee**≫: ≪**alanlee2**≫
- ≪**Joshua Burke**≫: ≪**joshuab3**≫
- ≪**Gerald Kozel**≫: ≪**gjkozel2**≫

(100 PTS.) Simplifying data.

A *k-step function* is a function of the form

$$f(x) = b_i \qquad \text{if } a_i \le x < a_{i+1} \quad (i = 0, \ldots, k-1)$$

for some $-\infty = a_0 < a_1 < \cdots < a_{k-1} < a_k = \infty$ and some $b_0, b_1, \ldots, b_{k-1}$.

We are given n data points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ and a number k between 1 and n. Our objective is to find a k-step function f such that $f(x_i) \ge y_i$ for all $i \in \{1, \ldots, n\}$, while minimizing the total "error" $\sum_{i=1}^{n}(f(x_i) - y_i)$ (this is the total length of the red vertical segments in the figure below).

1 (70 PTS.) Describe an algorithm, as fast as possible, that computes the minimum total error of the optimal k-step function. Bound the running time of your algorithm as a function of n and k.

[Note: in dynamic programming questions such as this, first give a clear English description of the function you are trying to evaluate, and how to call your function to get the final answer, then provide a recursive formula for evaluating the function (including base cases). If a correct evaluation order is specified clearly, iterative pseudocode is not required.]

2 (30 PTS.) Describe how to modify your algorithm in (A) so that it computes the optimal k-step function itself.

16 Solution:

1. 1. Formulating the problem Recursively, See page 1 for part A algorithm and page 2 for part B algorithm

   1.A Specification:

   The optimal step function is function that minimizes the sum of all e(p,f). What we want to calculate is the total error from all possible step function combinations for all point combinations so that we know the minimum sum of e(p,f) for some subset.

   1.B Solution:

   A brute force approach that we can take is to set a k step function value to a given point's y value for all points p in the given point set. We calculate each points error relative to all other points if it were the step function and sum them to get the total error then memoize it. We then recursively call the same function to find the total error sum on a smaller subset of 1 point less than the current point until the subset is size 1. After calculating the base case of size 1, all previous parent calls will build upon the memoized value its child call stored. After recursing fully, our memoizing DS MinTotErr[point(x,y), applicablePtSet[p1 ... pn], TotErrSum] will be filled.

   We will then do post processing and store potential point set solutions mapped to a total error sum value in the DS PossibleKStepPts[KstepFunctionPts[Points(x,y)], TotalErrorSum].We will recursively find all valid point set solutions that cover the entire given points set and fill the DS above. Then find the set of k points that gives us minimum total error value that also cover all points over its step function. We return the minimum PossibleKStepPts.TotalErrorSum for part A, and return PossibleKStepPts.KstepFunctionPts point set that has the minimum PossibleKStepPts.TotalErrorSum for part B.

   2. Building the solution to the recurrence from the bottom up.

   2.A Identify Subproblems:

   For the first part, each recursive sub problem will take in a set of points size of n-1 than its parent until the set is size 1. We calculate each total error of every point from the point's y value for every possible subset from the bottom up and memoize the smaller subproblem results so that the parent call may use its value to save work.

   For the second part, each recursive sub problem will take in a set of points size of n - MinTotErr.applicablePtSet.size. This will be called K times for every point n^2 times.

   2.B Analyzing runtime

   Calculating the total error for every point recursively would normally take $O(n^3)$. Because we memoize the smaller subproblem's total error, it will take $O(n^2)$. Post processing takes $O(kn^2)$. Overall runtime should be $O(kn^2)$.

   2.C Choosing a memoizing data structure.

   MinTotErr[point(x,y), applicablePtSet[p1 ... pn], TotErrSum]

   2.D identifying dependencies.

   Every subproblem of calculating total error for a smaller set of points for a given point will depend on the subproblem result that its current call produces with a set of size 1 less than the current argument.This recurses down until the set is only 1 point large.

   For the postprocessing, every subproblem will depend on a point set from the parent call size n - MinTotErr.applicablePtSet.

   2.E evaluation order

   We will start total error calculation with the base case as all other values depend on it. For post processing, we start with the given point set and recursively subtract applicablePtSet's k times until the entire given set is covered.