アルゴリズム論
Theory of Algorithms

第2回講義
アルゴリズムの設計と解析の基礎(2)

アルゴリズム論
Theory of Algorithms

Lecture #2

Foundation of Design and Analysis of Algorithms(2)

問題P3（最大区間和）: n個のデータが配列a[]に蓄えられているとき,区間[p,q]に対する和（区間和）sum(p, q)を，その区間内の要素a[p]～a[q]の和と定義する．このとき，区間和の最大値を求めよ．

すべての区間について対応する区間和を求めればよい．

アルゴリズムP3-A0:

```
maxsum=0;
for(p=0; p<n; p++)
  for(q=p; q<n; q++){
    // 区間[p,q]での和sumを求める
    sum=0;
    for(i=p; i<=q; i++)
      sum = sum + a[i];
    if(sum > maxsum) maxsum = sum;
  }
```

計算時間:

３重ループだから

O(n^3)時間

Problem P3（Largest Sum Interval）: Given n data in an array a[], a sum interval sum(p,q) for an interval [p,q] is defined as the sum of elements a[p]～a[q]. Find a largest sum interval for a given array.

It can be computed by computing the interval sum for every interval.

Algorithm P3-A0:

```
maxsum=0;
for(p=0; p<n; p++)
  for(q=p; q<n; q++){
    // find the interval sum in an interval [p,q]
    sum=0;
    for(i=p; i<=q; i++)
      sum = sum + a[i];
    if(sum > maxsum) maxsum = sum;
  }
```

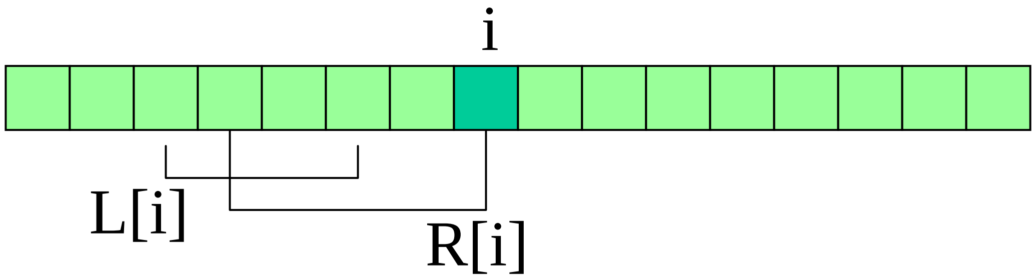
Computation time:
triple-loop=>
$O(n^3)$ time

動的計画法に基づくアルゴリズム

配列を左から右に順に調べていく.
a[i]を調べているとき, [0,i-1]の範囲における最大の区間和をL[i],
a[i]を右端とする区間の中での最大区間和をR[i]とする.
このとき,

$$L[i] = \begin{cases} L[i-1] & L[i-1] \geqq R[i-1] \text{のとき,} \\ R[i-1] & \text{それ以外のとき.} \end{cases}$$

$$R[i] = \begin{cases} a[i] & R[i-1]+a[i] < a[i] \text{のとき,} \\ R[i-1] + a[i] & \text{それ以外のとき.} \end{cases}$$

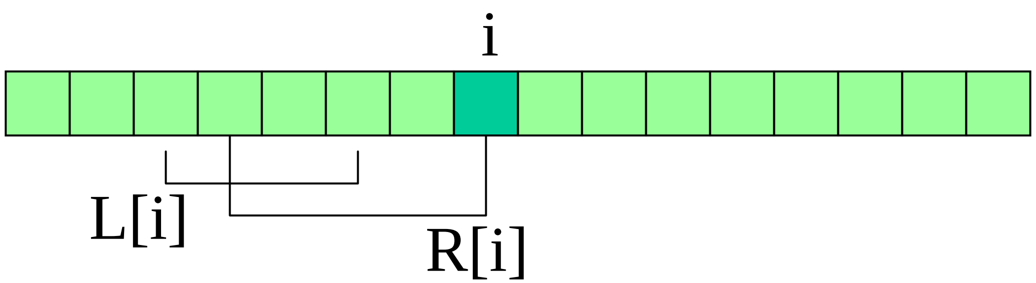


Algorithm based on dynamic programming

The array is checked from left to right.
Let L[i] be the largest sum interval in the interval [0, i-1] and
R[i] be the largest sum interval for interval with a[i] in its right end.
Then, we have

$$L[i] = \begin{cases} L[i-1] & \text{if } L[i-1] \geqq R[i-1], \\ R[i-1] & \text{otherwise.} \end{cases}$$

$$R[i] = \begin{cases} a[i] & \text{if } R[i-1]+a[i] < a[i], \\ R[i-1] + a[i] & \text{otherwise.} \end{cases}$$



$$L[i] = \begin{cases} L[i-1] & L[i-1] \geqq R[i-1]\text{のとき}, \\ R[i-1] & \text{それ以外のとき}. \end{cases}$$

$$R[i] = \begin{cases} a[i] & R[i-1]+a[i]<a[i]\text{のとき}, \\ R[i-1] + a[i] & \text{それ以外のとき}. \end{cases}$$

最後にL[n-1]とR[n-1]の大きい方が最大値.

アルゴリズムP3-A5:

```
L[0] = R[0] = a[0];
for(i=1; i<n; i++){
   if( L[i-1] >= R[i-1] ) L[i] = L[i-1]; else L[i] = R[i-1];
   if( R[i-1] + a[i] < a[i] ) R[i] = a[i]; else R[i] = R[i-1] + a[i];
}
if( L[n-1] > R[n-1] ) return L[n-1]; else return R[n-1];
```

計算時間はO(n).

作業用の配列をなくす事はできるか？

$$L[i] = \begin{cases} L[i-1] & \text{if } L[i-1] \geqq R[i-1], \\ R[i-1] & \text{otherwise.} \end{cases}$$

$$R[i] = \begin{cases} a[i] & \text{if } R[i-1]+a[i]<a[i], \\ R[i-1] + a[i] & \text{otherwise.} \end{cases}$$

Finally, we take the larger of L[n-1] and R[n-1] as the maximum.

Algorithm P3-A5:

```
L[0] = R[0] = a[0];
for(i=1; i<n; i++){
  if( L[i-1] >= R[i-1] ) L[i] = L[i-1]; else L[i] = R[i-1];
  if( R[i-1] + a[i] < a[i] ) R[i] = a[i]; else R[i] = R[i-1] + a[i];
}
if( L[n-1] > R[n-1] ) return L[n-1]; else return R[n-1];
```

Computation time is O(n).

Is it possible to do without any auxiliary array?

L[i]の値はL[i-1]とR[i-1]だけで決まる.
R[i]の値はR[i-1], a[i]だけで決まる.
よって，配列を使う必要はない.

アルゴリズムP3-A6:

```
L = R = a[0];
for(i=1; i<n; i++){
   if( L >= R ) L = L; else L = R;
   if( R + a[i] < a[i] ) R = a[i]; else R = R + a[i];
}
if( L > R) return L; else return R;
```

L[i] is determined only by L[i-1] and R[i-1].
R[i] is determined only by R[i-1] and a[i].
Therefore, no auxiliary array is required.

Algorithm P3-A6:

```
L = R = a[0];
for(i=1; i<n; i++){
   if( L >= R ) L = L; else L = R;
   if( R + a[i] < a[i] ) R = a[i]; else R = R + a[i];
}
if( L > R) return L; else return R;
```

問題P4: n個のデータが配列a[]に蓄えられているとする．区間[p,q]における平均値をave[p,q]とするとき，この平均値を最大にする区間を求めよ．

	0	1	2	3	4	5	6	7	8	9
a	10	-9	-5	12	-3	10	-8	11	-8	-2

ave[3,7] = 22/5=4.4　　ave[3,5]=19/3=6.33

実は，区間に制限がなければ，この問題は実に簡単．
配列内の最大値をa[i]とすると，平均値を最大にする区間は[i,i]ということになる．

質問：上記の事実を証明せよ．

では，区間の長さを２以上に限定した場合はどうだろう？

今度は単純な方法では解は求まらない．

Problem P4: Suppose n data are stored in an array a[]. By ave[p,q] we denote the average of elements in an interval [p,q]. Then, find an interval maximizing the average.

	0	1	2	3	4	5	6	7	8	9
a	10	-9	-5	12	-3	10	-8	11	-8	-2

ave[3,7] = 22/5=4.4   ave[3,5]=19/3=6.33

If there is no constraint in intervals, this problem is in fact quite easy. If a[i] is the maximum in the array, then the interval maximizing the average is obviously [i,i].

Exercise: Question: Prove the above fact.

Then, what about if the length of an interval must be at least two?

In this case there is no simple algorithm.

（腕力法）

すべての区間について平均値を求め，その最大値を求める

アルゴリズムP4−A0:

```
 maxave=(a[0]+a[1])/2;
 for(p=0; p<n-1; p++)
   for(q=p+1; q<n; q++){
     sum = 0;
     for(i=p; i<=q; i++)
       sum = sum + a[i];
     ave = sum/(q-p+1);
     if(ave > maxave) maxave = ave;
   }
 return maxave;
```

プログラムの構造が３重ループであるから，$O(n^3)$時間かかる.

(Brute-force algorithm)

computes average for every interval to find the largest value.

Algorithm P4−A0:

```
maxave=(a[0]+a[1])/2;
for(p=0; p<n-1; p++)
  for(q=p+1; q<n; q++){
    sum = 0;
    for(i=p; i<=q; i++)
      sum = sum + a[i];
    ave = sum/(q-p+1);
    if(ave > maxave) maxave = ave;
  }
return maxave;
```

Triple-loop structure => $O(n^3)$ time.

（改良1）

区間和を求める計算と組み合わせると無駄が省ける

アルゴリズムP3-A1:

```
maxsum=a[0];
for(p=0; p<n-1; p++){
    sum=0;
    for(q=p; q<n; q++){
       sum = sum + a[q];
       if( sum > maxsum) maxsum = sum;
    }
return maxsum;
```

上記のアルゴリズムを区間平均用に変形すればよい．
ただし，線形時間のアルゴリズムを応用することはできない．
（何が違うか？）

(Improvement 1)

Application of algorithm for largest sum interval leads efficiency.

```
Algorithm P3-A1:
maxsum=a[0];
for(p=0; p<n-1; p++){
    sum=0;
    for(q=p; q<n; q++){
        sum = sum + a[q];
        if( sum > maxsum) maxsum = sum;
    }
return maxsum;
```

It suffices to convert the above algorithm for average in intervals.
Note that we cannot apply the linear-time algorithm.
(What is different?)

アルゴリズムP4-A1:

```
maxave=(a[0]+a[1])/2;
for(p=0; p<n-1; p++){
    sum=a[p];
    for(q=p+1; q<n; q++){
       sum = sum + a[q];
       ave = sum/(q-p+1);
       if( ave > maxave) maxave = ave;
    }
}
return maxave;
```

区間平均の場合には，区間の長さが2以上でなければならないことに注意.

計算時間

今度は2重ループの構造なので，$O(n^2)$時間.

区間和問題のように線形時間アルゴリズムは存在するか？

Algorithm P4-A1:

```
maxave=(a[0]+a[1])/2;
for(p=0; p<n-1; p++){
    sum=a[p];
    for(q=p+1; q<n; q++){
       sum = sum + a[q];
       ave = sum/(q-p+1);
       if( ave > maxave) maxave = ave;
    }
}
return maxave;
```

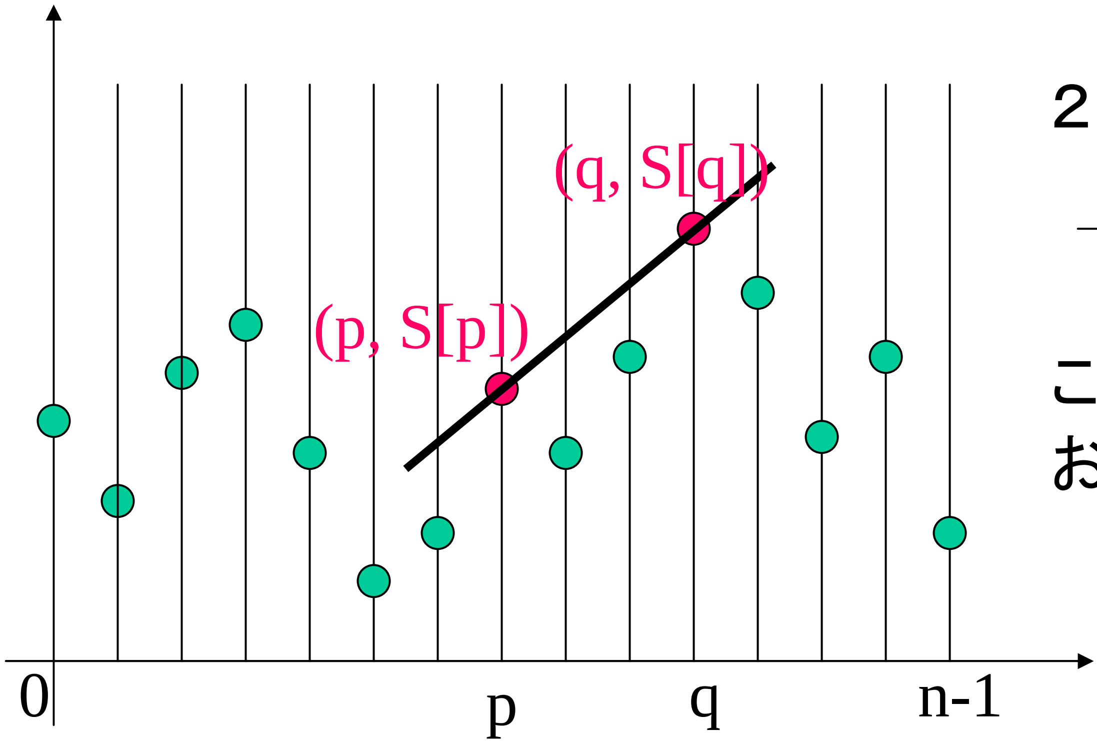
For interval average, note that the length of interval must be at least 2.

Computation time

double loop structure => $O(n^2)$ time.

Is there a linear algorithm like in the largest sum interval problem?

区間和を最大にする問題P3と同様に，配列の0番目からの和 S[p] = a[0]+a[1]+...+a[p]をすべてのpについて求めて，(p, S[p])で定まる点を平面上にプロットしてみよう.



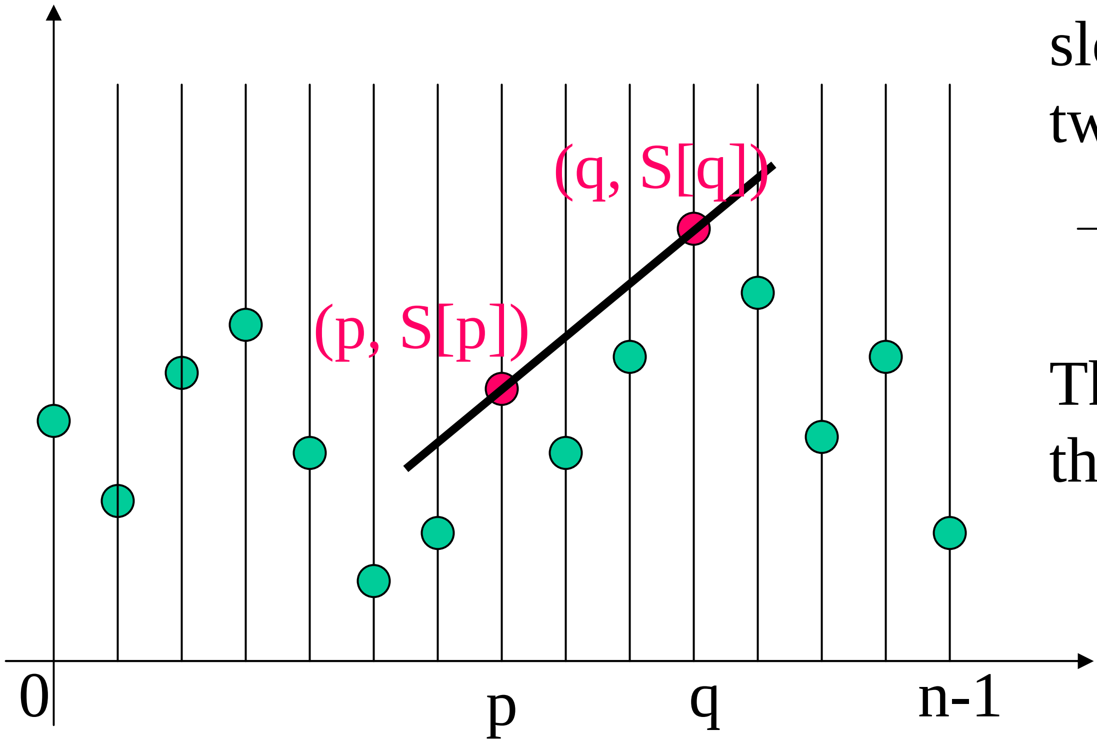
２点を通る直線の傾き

$$\frac{S[q]-S[p]}{q - p}$$

これは区間[p+1,q]における平均値に等しい

したがって，上記のn点の内，どの２点を通る直線を引けば傾きが最大になるかを求める問題となる．ただし，区間の長さを2以上に制限しているので，横座標は2以上離れていなければならない．

As in the largest sum interval problem P3, compute the sum S[p] = a[0]+a[1]+...+a[p] for each p and plot it at (p, S[p]) in the plane.



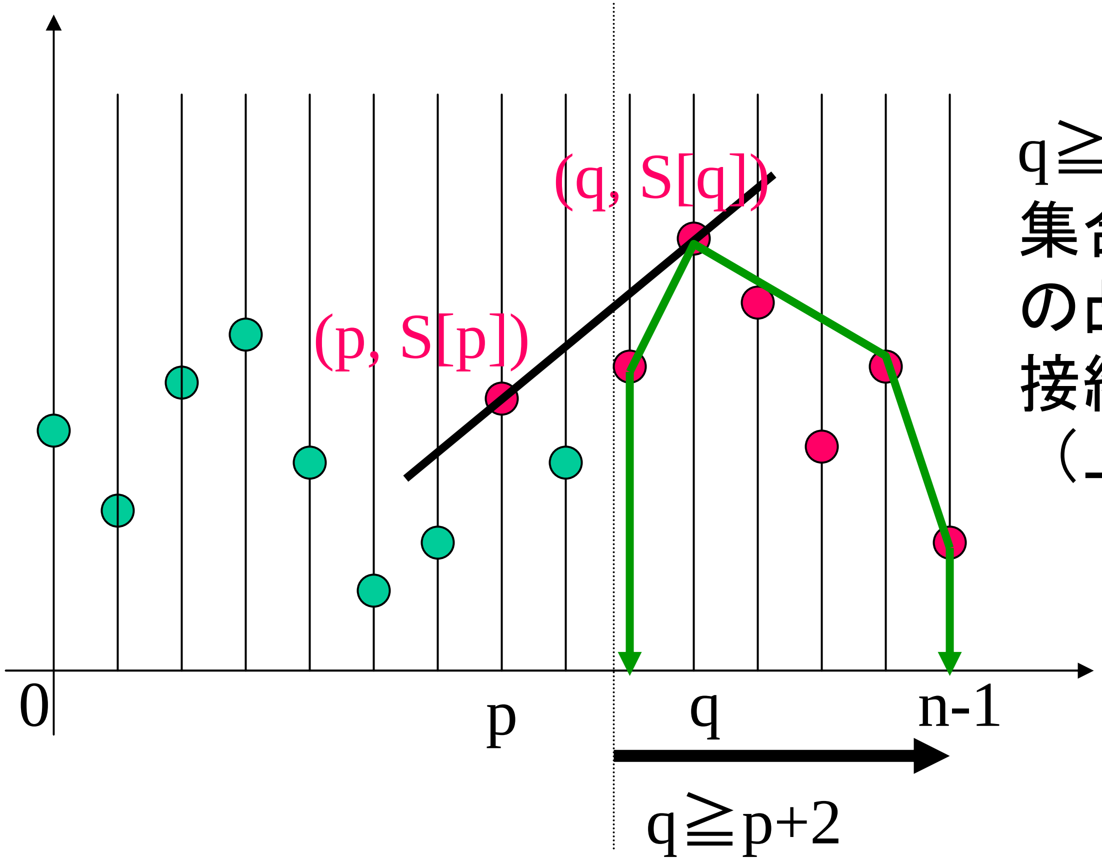
slope of line through the two points

$$\frac{S[q]-S[p]}{q - p}$$

This is the average in the interval [p+1,q].

Thus, the problem is to find two points among n points so that their associated line has a largest slope. Here, since the length of each interval must be at least 2, their x-coordinates must differ at least by 2.

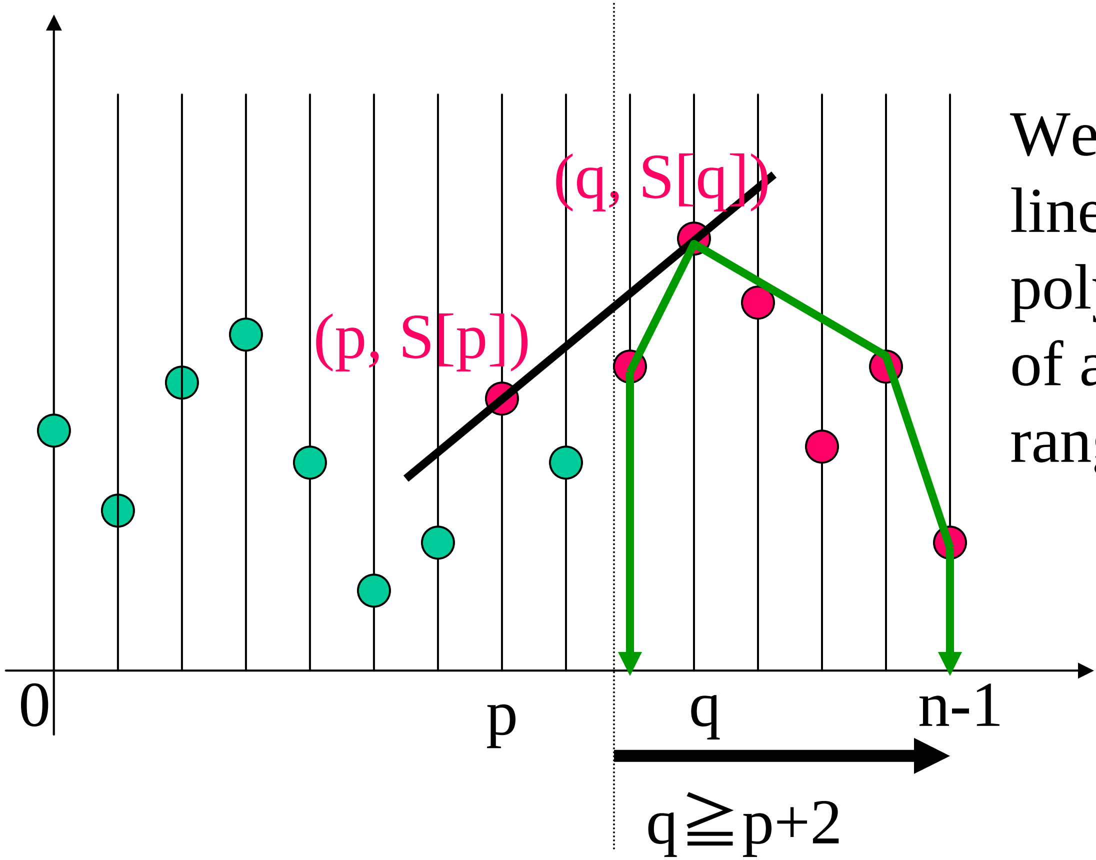
区間の左端pを固定したとき，
どの点(q, S[q])を選べば傾きが最大になるか？



q≧p+2の範囲の点集合を包含する最小の凸多角形への接線を求めればよい．
（上部凸包）

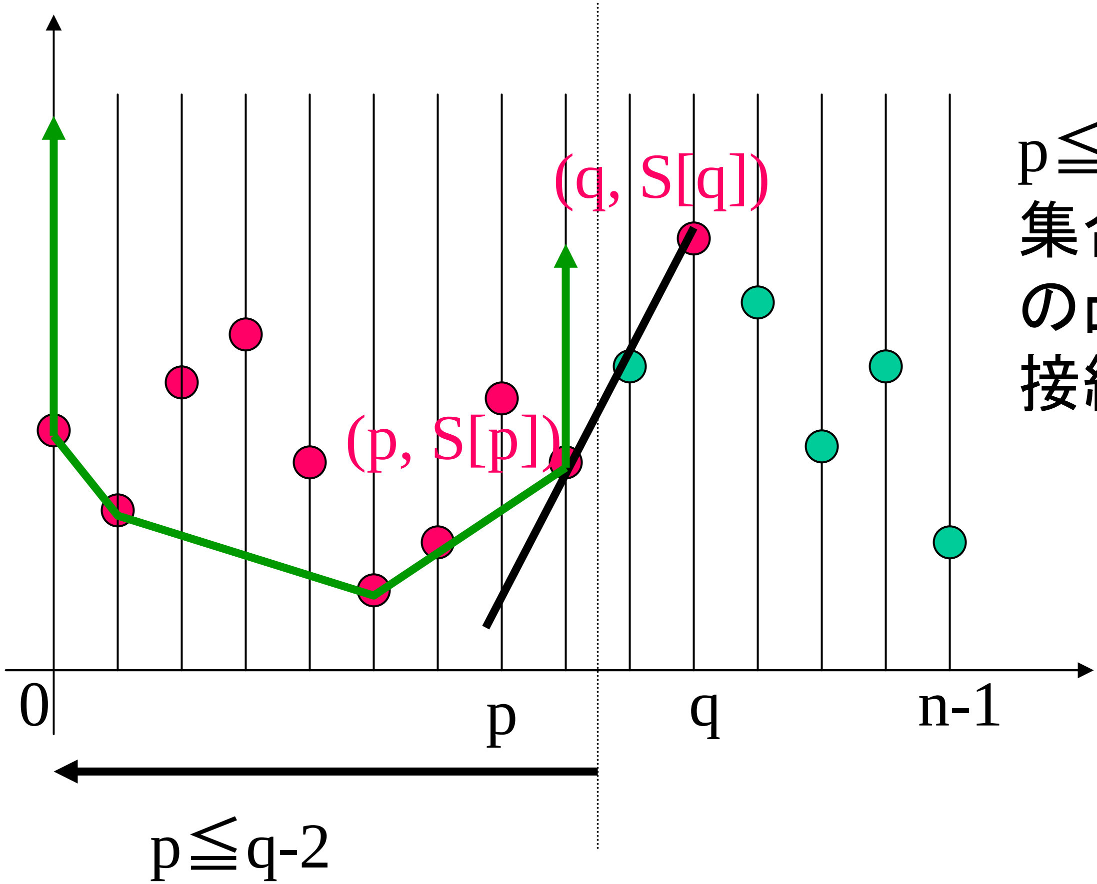
Fixing the left endpoint p,

which point (q, S[q]) should be selected to maximize the slope?



We should find a tangent line to the smallest convex polygon (upper convex hull) of a set of point in the range q≧p+2.

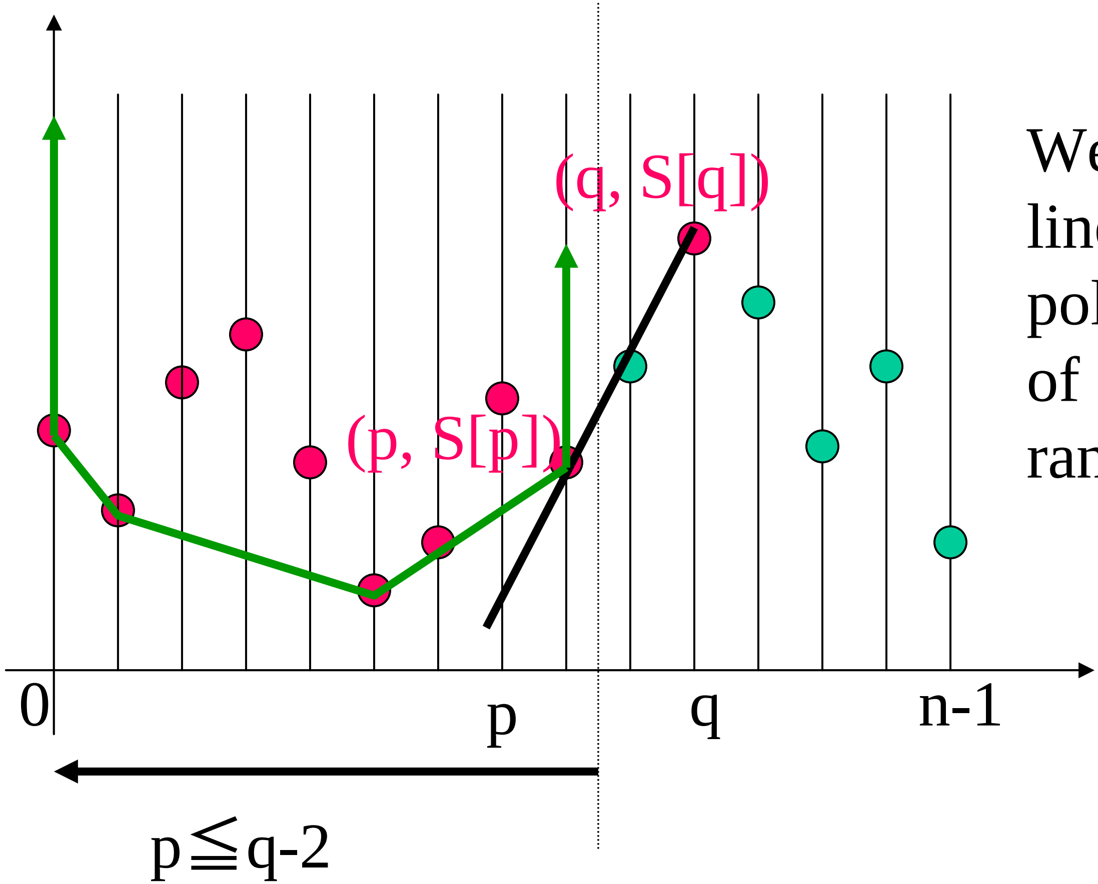
区間の右端qを固定したとき,
どの点(p, S[p])を選べば傾きが最大になるか？



p≦q-2の範囲の点集合を包含する最小の凸多角形への接線を求めればよい.
（下部凸包）

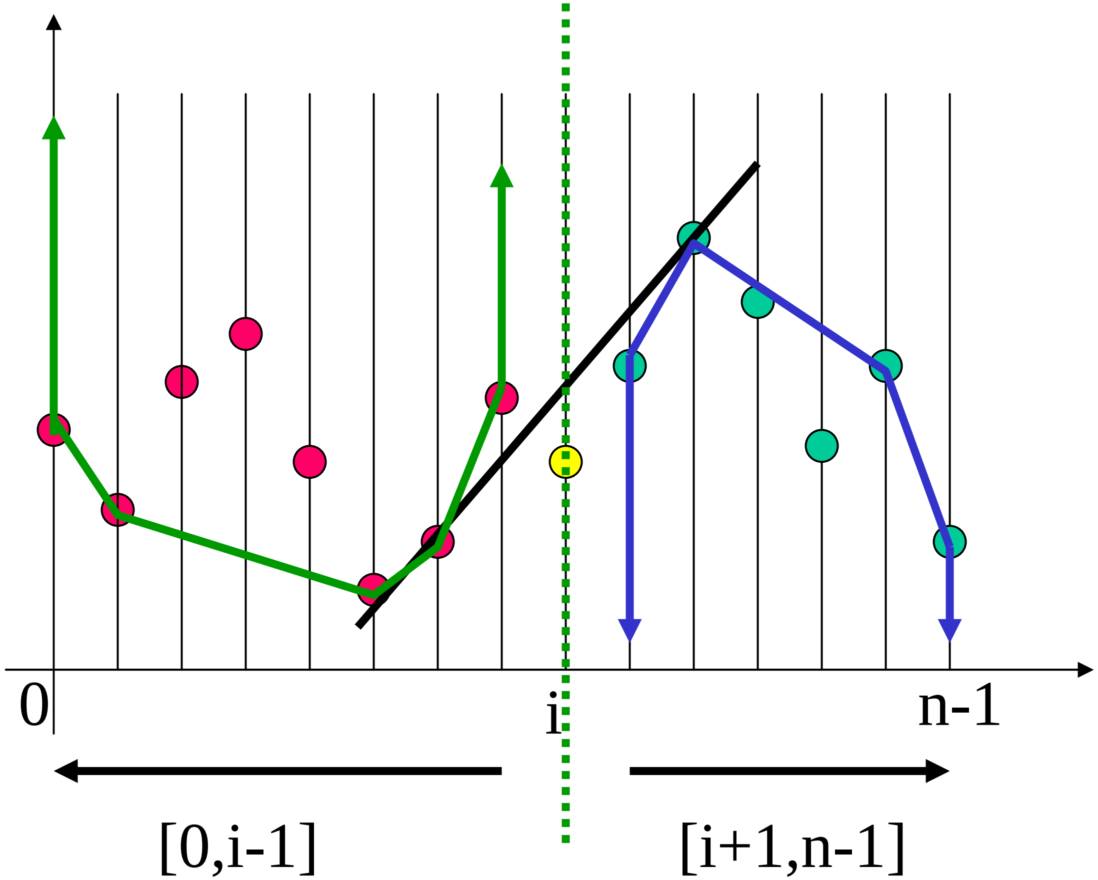
Fixing the right endpoint p,

which point (p, S[p]) should be selected to maximize the slope?

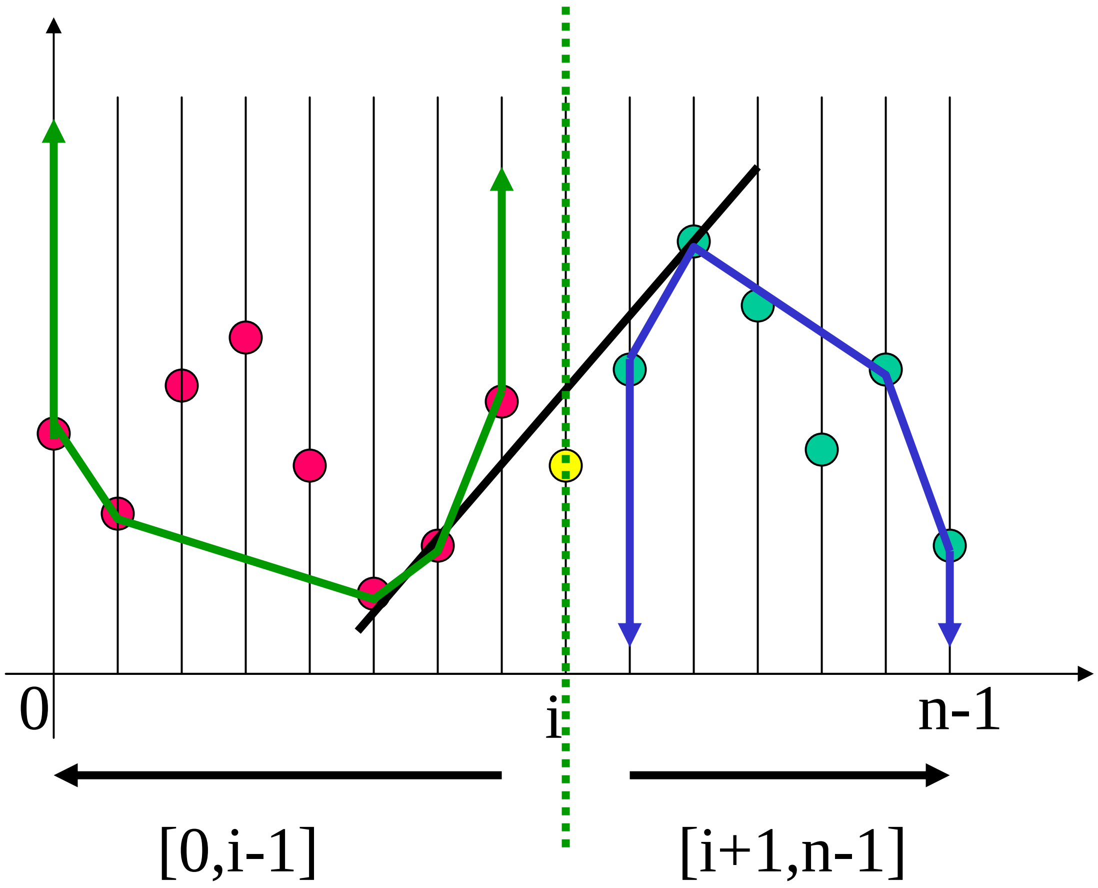


We should find a tangent line to the smallest convex polygon (lower convex hull) of a set of point in the range p≦q-2.

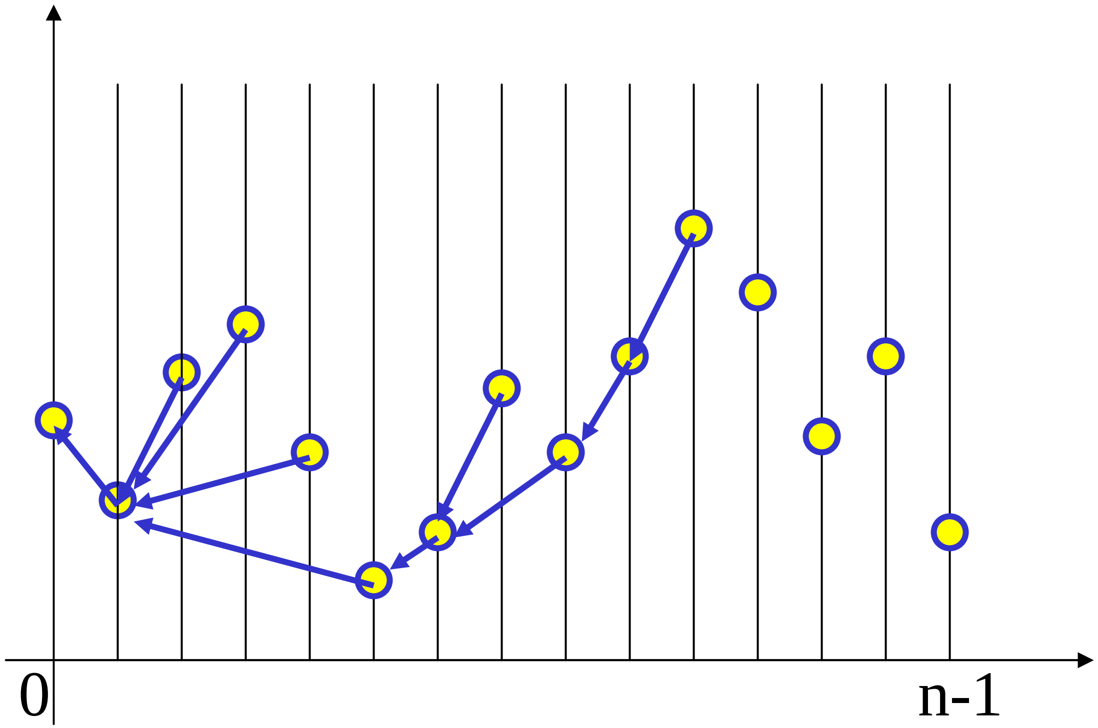
結局，各iについて，区間[0,i-1]の点の下部凸包LCH(0,i-1)と区間[i+1,n-1]の点の上部凸包UCH(i+1,n-1)を求めておき，両者の共通接線を求めればよい.



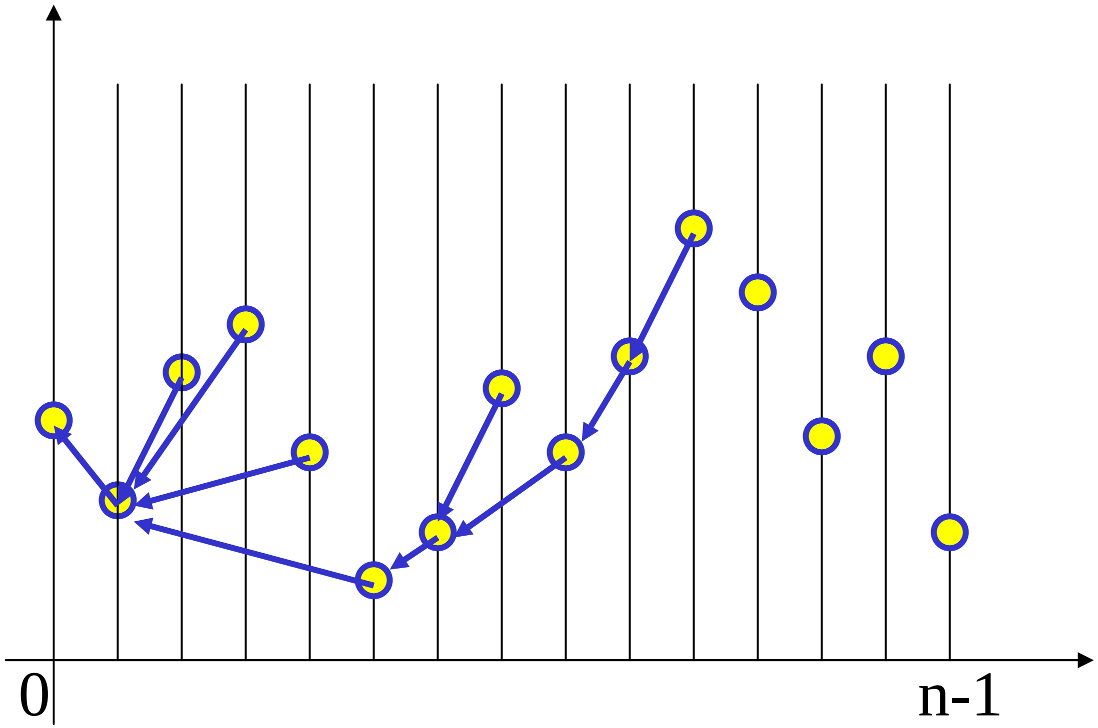
After all we should compute for each i the lower convex hull LCH(0,i-1) for an interval [0,i-1] and the upper convex hull UCH(i+1, n-1) for an interval [i+1, n-1] and then find their common tangent.



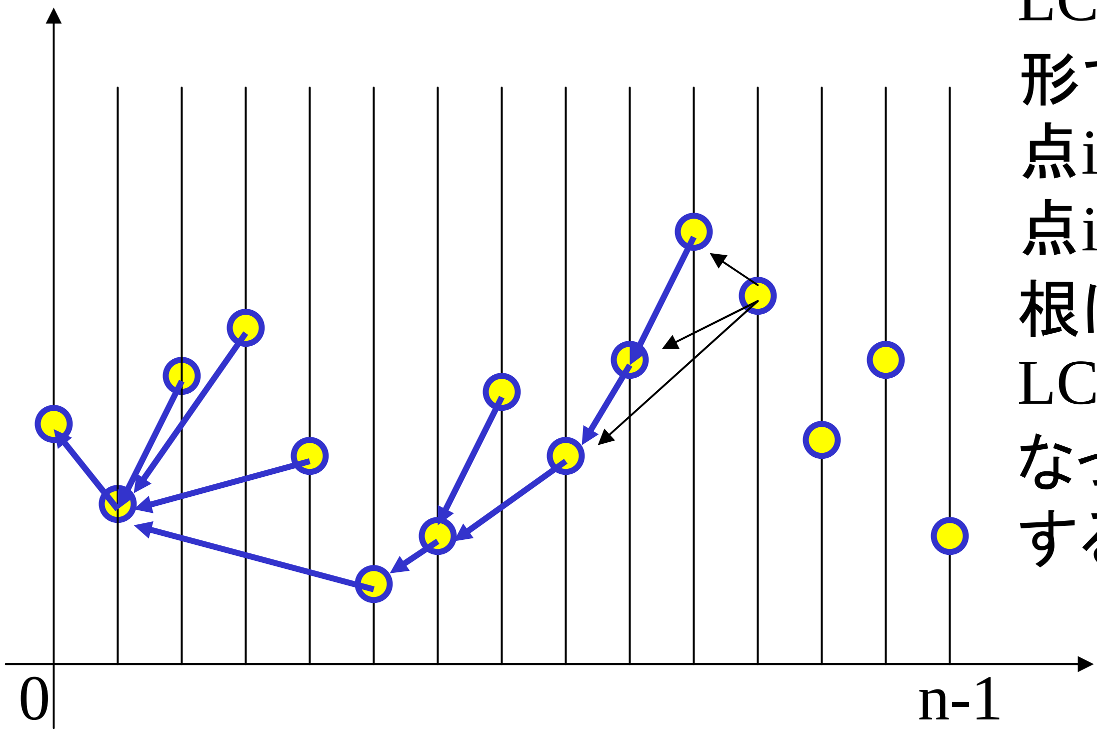
区間[0,i-1]の点の下部凸包LCH(0,i-1)の計算
iを順に増やしながらLCH(0,i)を順に更新



Compute the lower convex hull LCH(0,i-1) for an interval [0,i-1].
Update LCH(0,i) while increasing i.



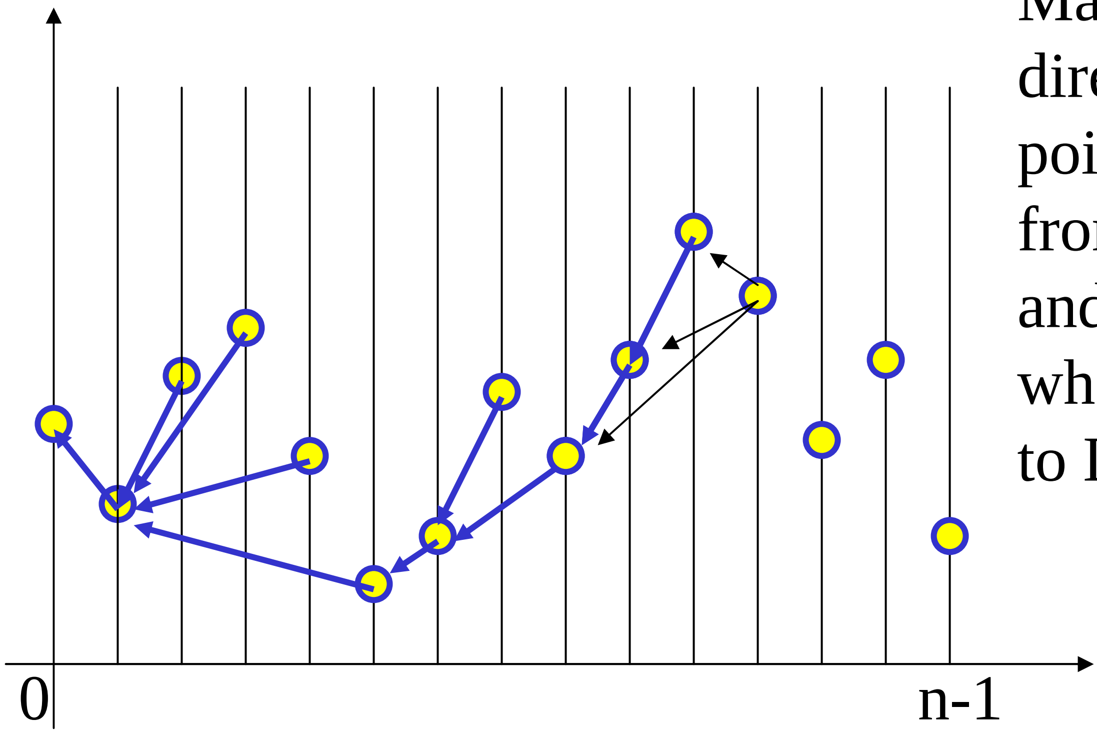
区間[0,i-1]の点の下部凸包LCH(0,i-1)の計算
iを順に増やしながらLCH(0,i)を順に更新



LCH(0,i-1)を有向木の形で管理しておく.
点iを加えるときは,
点i-1から始めて，木を根にむけて辿り,
LCH(0,i-1)への接線になったところで辺を確定する.

演習問題:上記の計算は線形時間O(n)でできることを示せ.

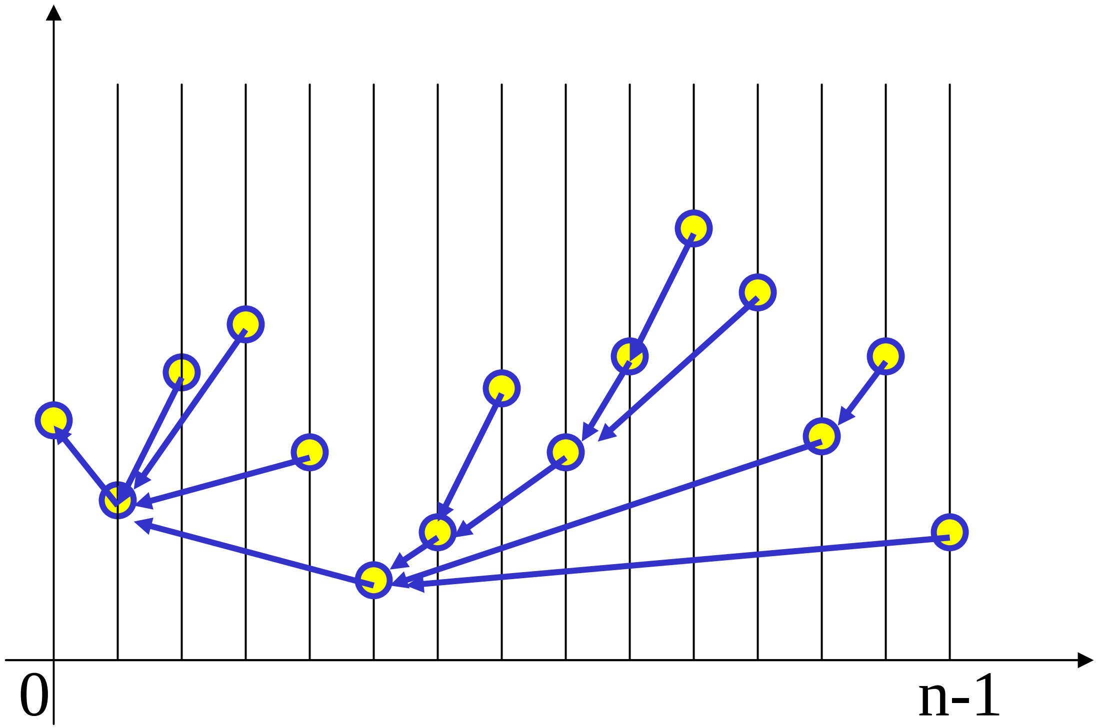
Compute the lower convex hull LCH(0,i-1) for an interval [0,i-1].
Update LCH(0,i) while increasing i.



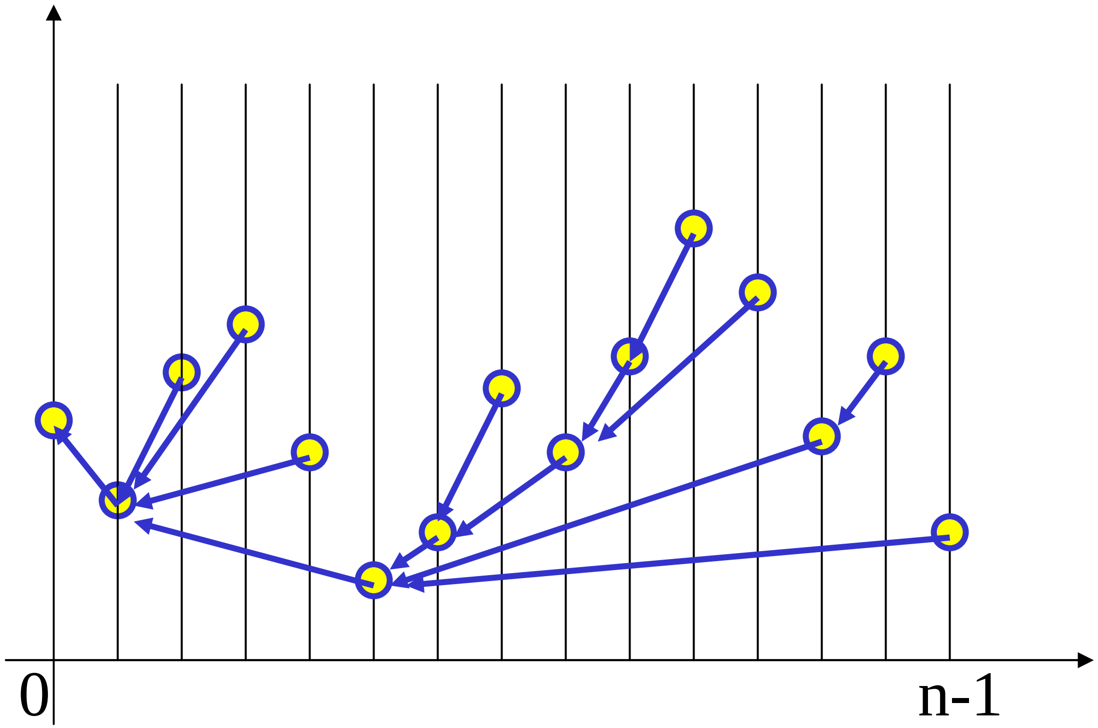
Maintain LCH(0,i-1) as a directed tree. To insert a point i, traverse the tree from point i-1 to the root and determine the edge when it becomes a tangent to LCH(0, i-1).

Exercise : Show that the above computation is done in O(n) time.

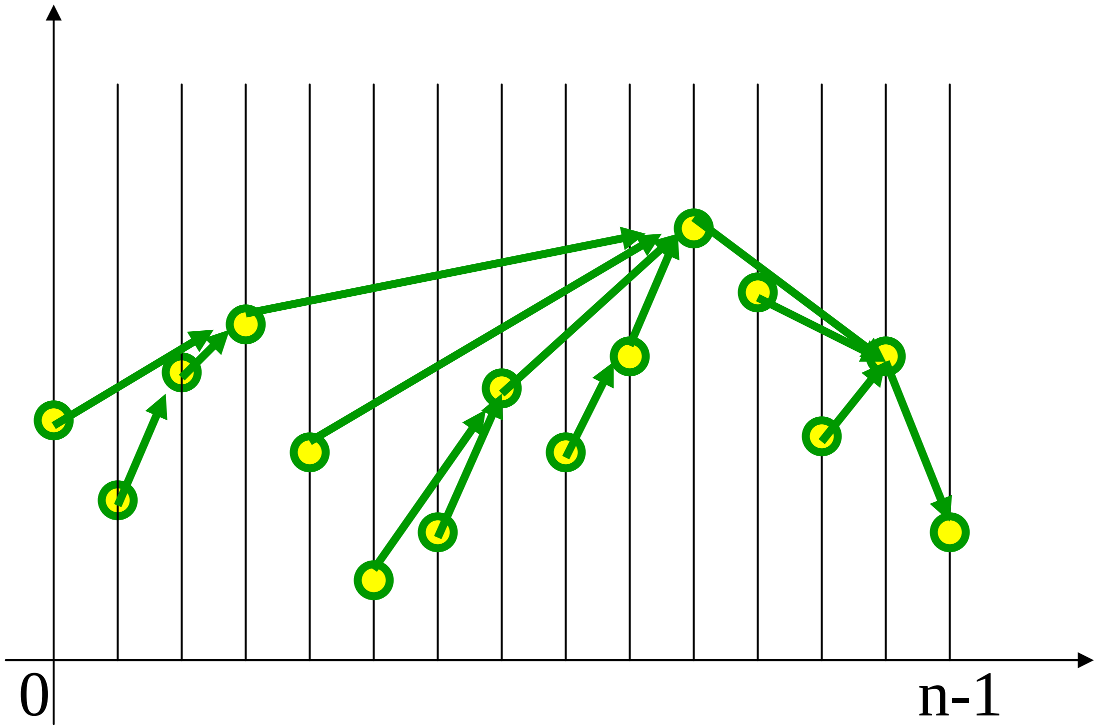
区間[0,i-1]の点の下部凸包LCH(0,i-1)の計算



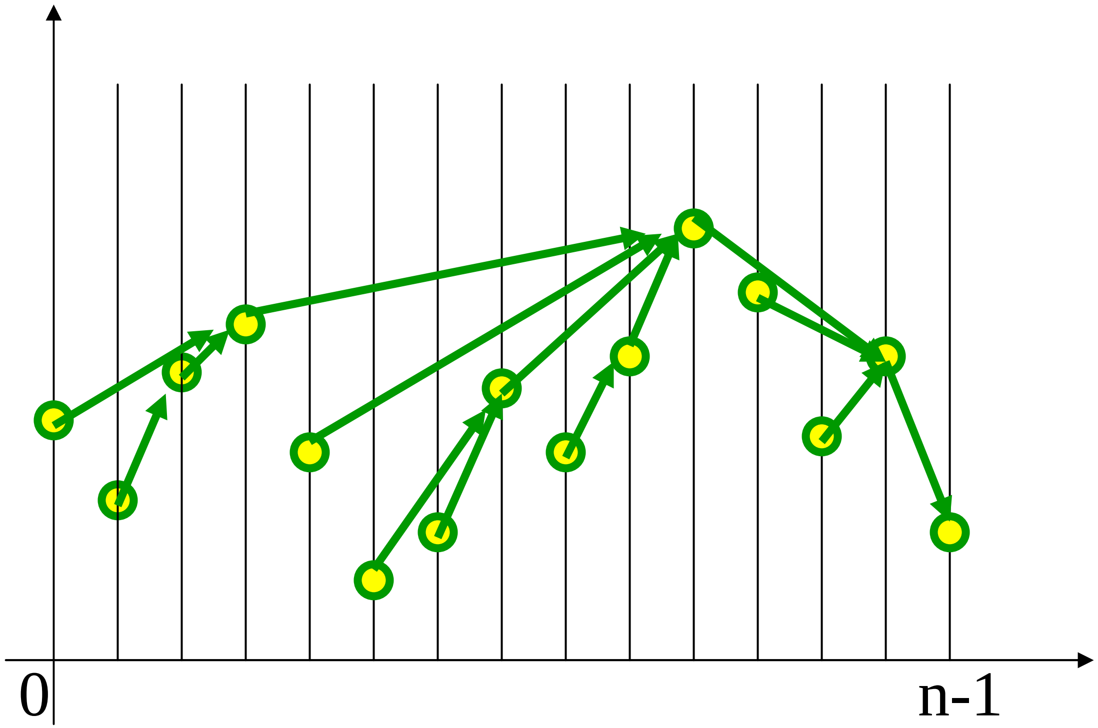
Compute the lower convex hull LCH(0,i-1) for an interval [0,i-1].



区間[i+1,n-1]の点の上部凸包UCH(i+1, n-1)の計算
iを順に減らしながらUCH(i+1,n-1)を順に更新

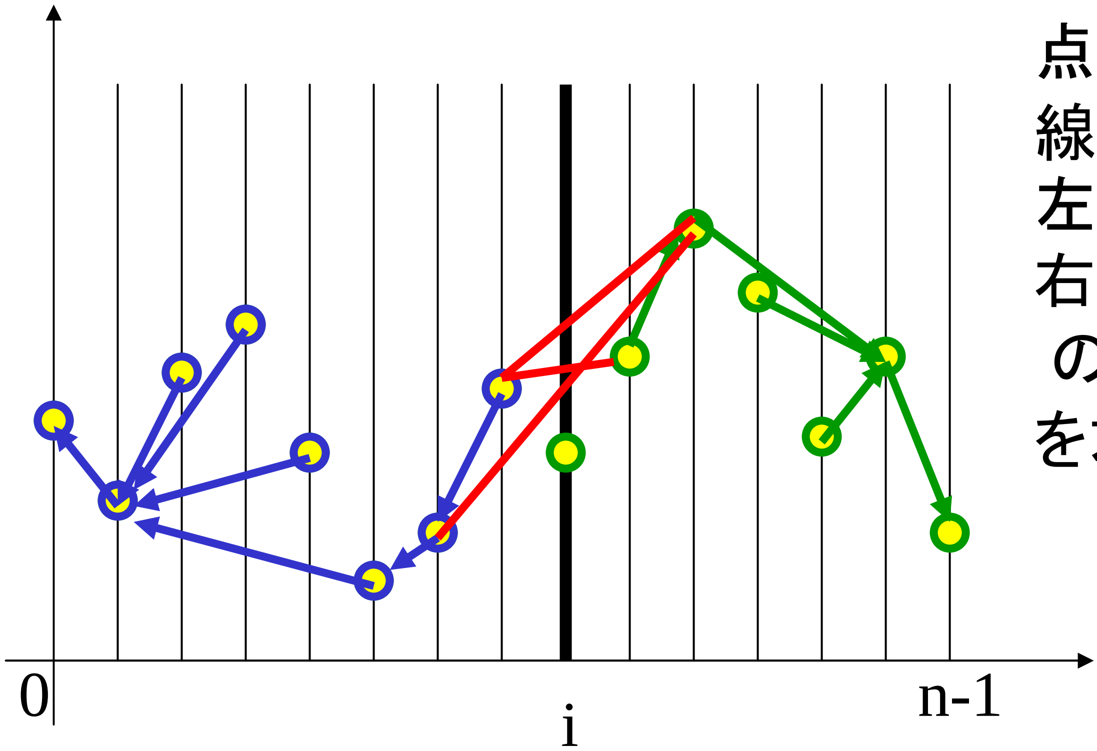


Compute the upper convex hull UCH(i+1,n-1) for an interval [i+1,n-1].
Update UCH(i+1,n-1) while decreasing i.



iに関して区間平均の最大値を求める.

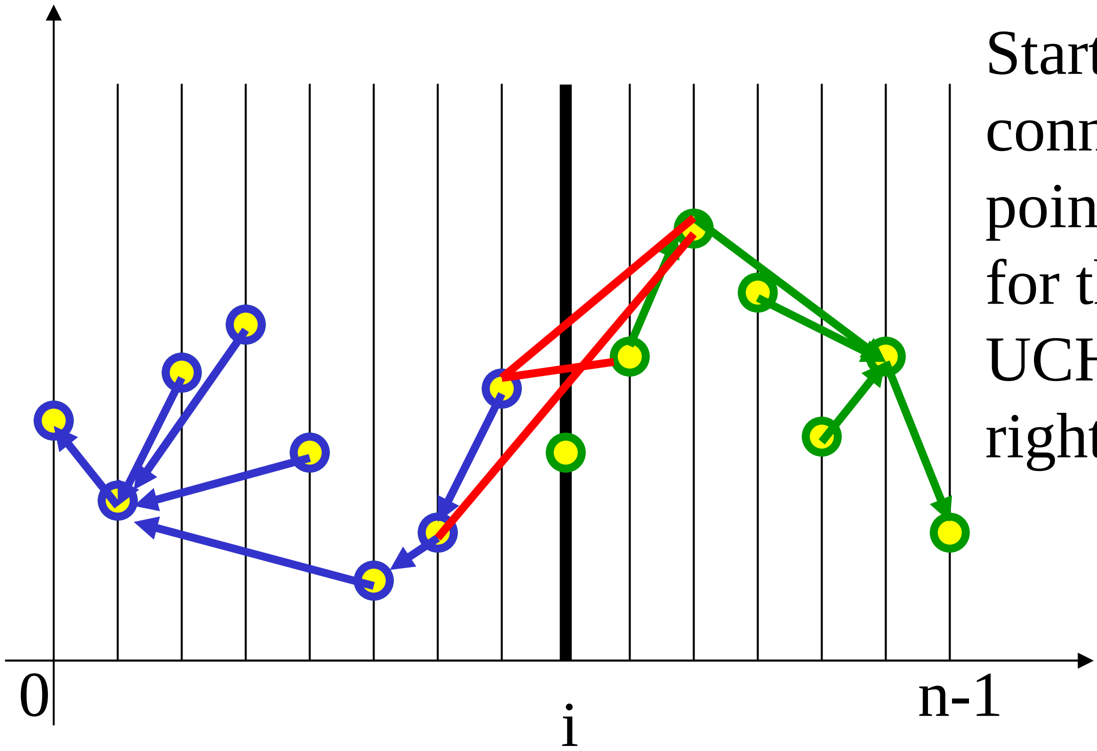
[0,i-1]の点と[i+1,n-1]の点を結ぶ傾き最大の線分を求める.



点i-1と点i+1を結ぶ線分から始めて,
左の点はLCH(0,i-1)
右の点はUCH(i+1,n-1)
の辺を辿り, 共通接線を求める.

Find an interval maximizing the average for each i.

Find a largest-slope line connecting points in [0,i-1] and [i+1,n-1].



Starting from the line connecting point i-1 and point i+1, trace LCH(0,i-1) for the left points and UCH(i+1,n-1) for the right points.

具体的な手続きは次の通り

```
p=i-1; q=i+1;
do{
  change=false;
  while((p,q,UCH(i+1,n-1)におけるqの親)が反時計回り){
    q=UCH(i+1,n-1)におけるqの親; change=true;
  }
  while((LCH(0,i-1)におけるpの親,p,q,)が時計回り){
    p=LCH(0,i-1)におけるpの親; change=true;
  }
}
```

上記の手続きはO(n)時間でできる.
これをn回繰り返すと全体では$O(n^2)$になってしまう.
線形時間に改善できるか？
あるいは，本当に$O(n^2)$時間かかるのか？

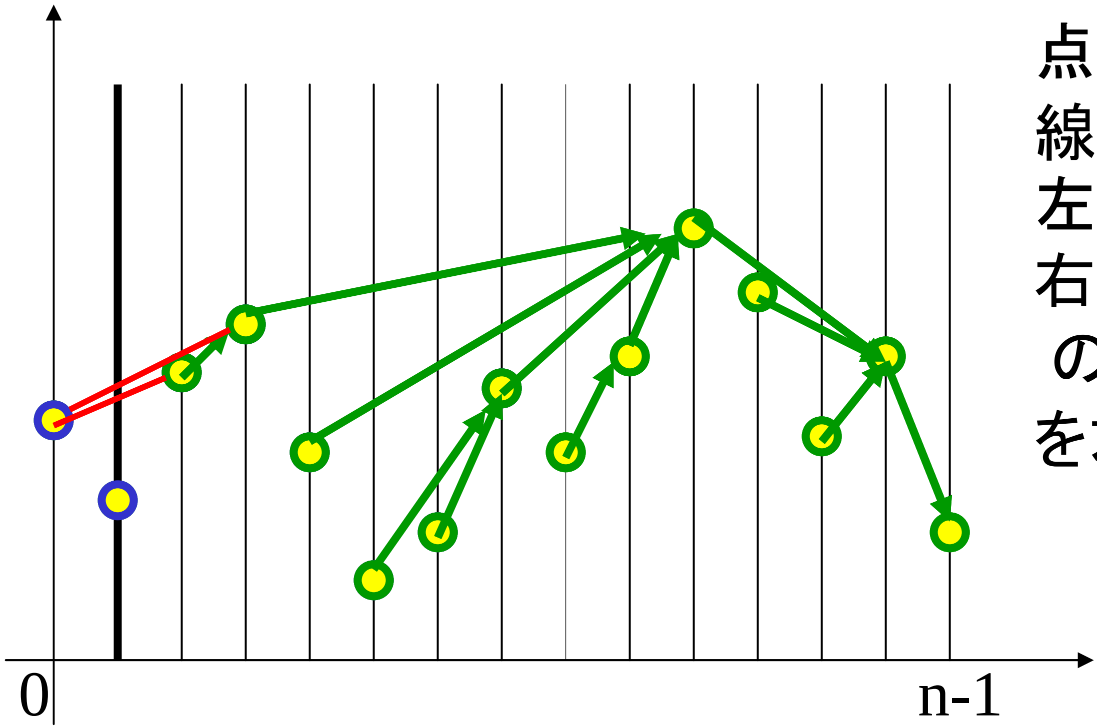
The concrete procedure is as follows:

```
p=i-1; q=i+1;
do{
  change=false;                 ccw: counter-clockwise
  while((p,q,parent of q in UCH(i+1,n-1)) is ccw){
    q=parent of q in UCH(i+1,n-1); change=true;
  }
  while((parent of p in LCH(0,i-1),p,q,) is cw){
    p=parent of p in LCH(0,i-1); change=true;
  }
}
```

The above procedure is done in $O(n)$ time.
If we iterate it n times, the overall time becomes $O(n^2)$.
Can we improve it into linear time?
Or, does it really take $O(n^2)$ time?

iに関して区間平均の最大値を求める.

[0,i-1]の点と[i+1,n-1]の点を結ぶ傾き最大の線分を求める.

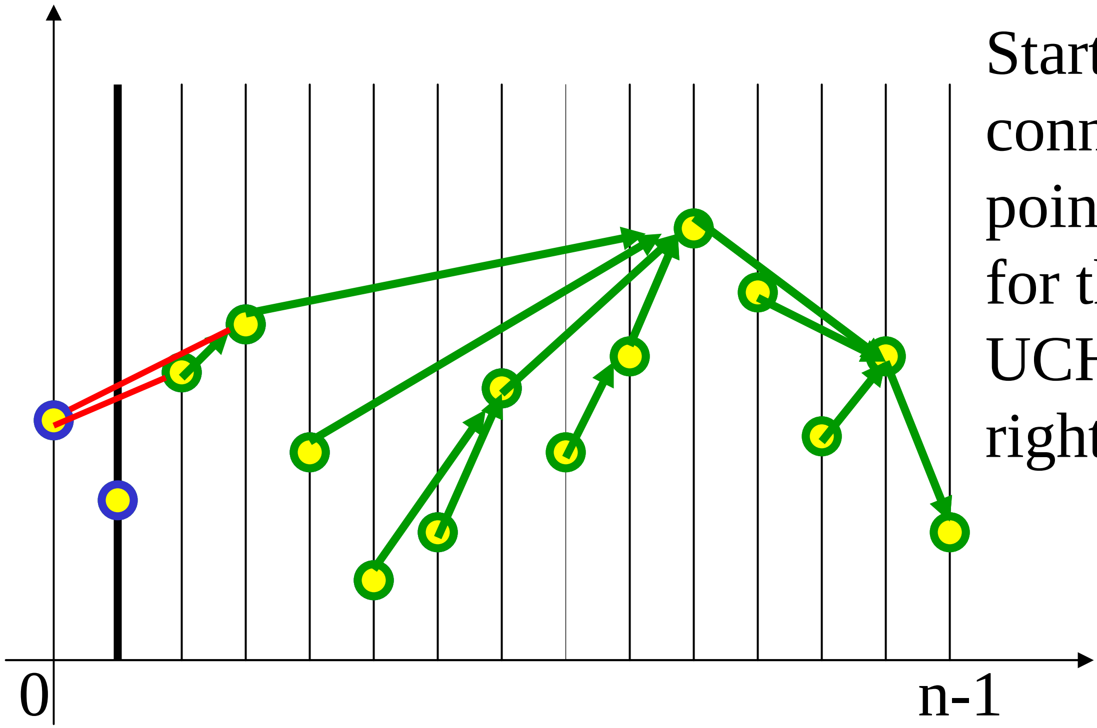


点i-1と点i+1を結ぶ
線分から始めて,
左の点はLCH(0,i-1)
右の点はUCH(i+1,n-1)
の辺を辿り，共通接線
を求める.

i=1

Find an interval maximizing the average for each i.

Find a largest-slope line connecting points in [0,i-1] and [i+1,n-1].

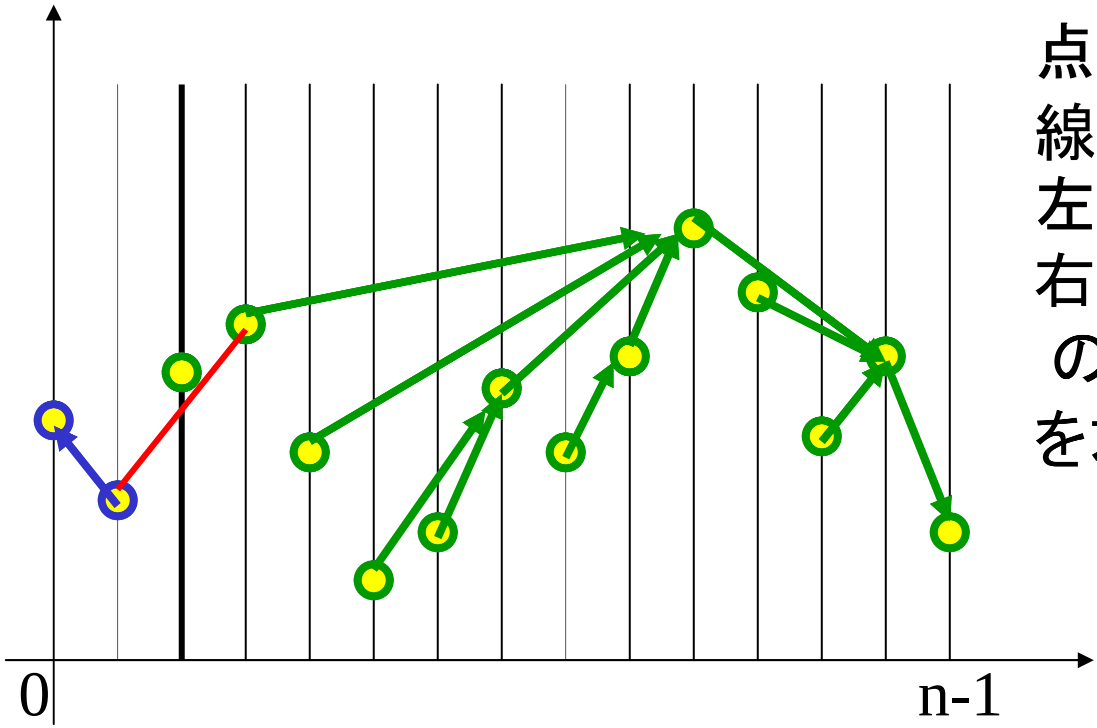


Starting from the line connecting point i-1 and point i+1, trace LCH(0,i-1) for the left points and UCH(i+1,n-1) for the right points.

i=1

iに関して区間平均の最大値を求める.

[0,i-1]の点と[i+1,n-1]の点を結ぶ傾き最大の線分を求める.

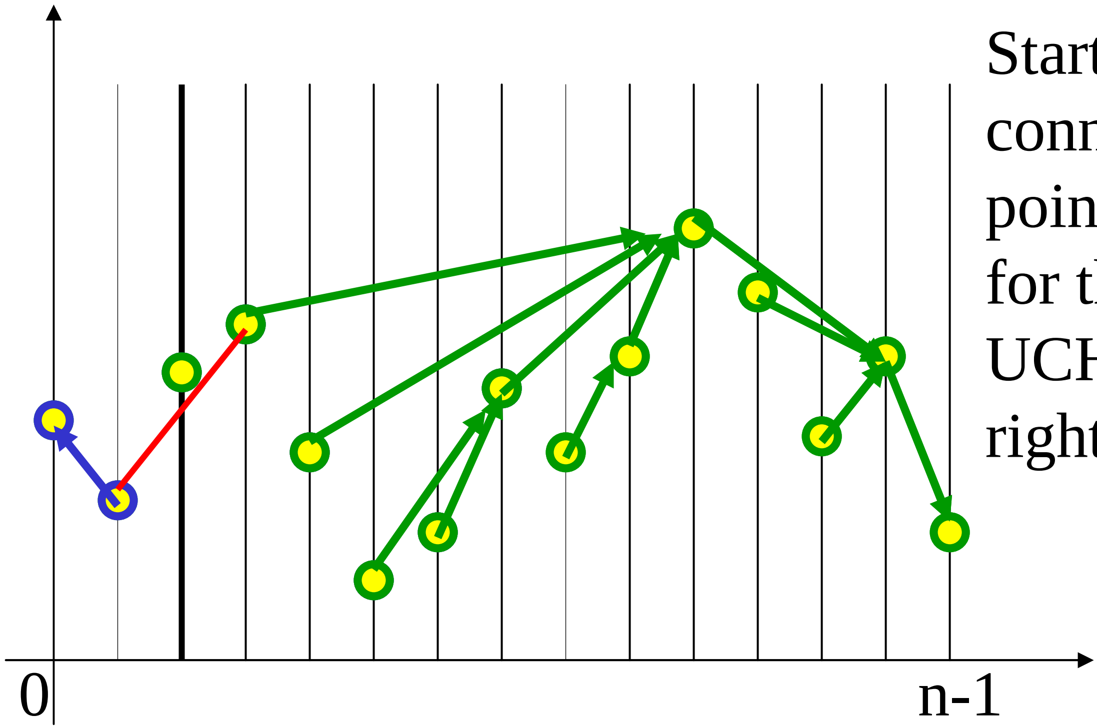


点i-1と点i+1を結ぶ線分から始めて,
左の点はLCH(0,i-1)
右の点はUCH(i+1,n-1)
の辺を辿り, 共通接線を求める.

i=2

Find an interval maximizing the average for each i.

Find a largest-slope line connecting points in [0,i-1] and [i+1,n-1].

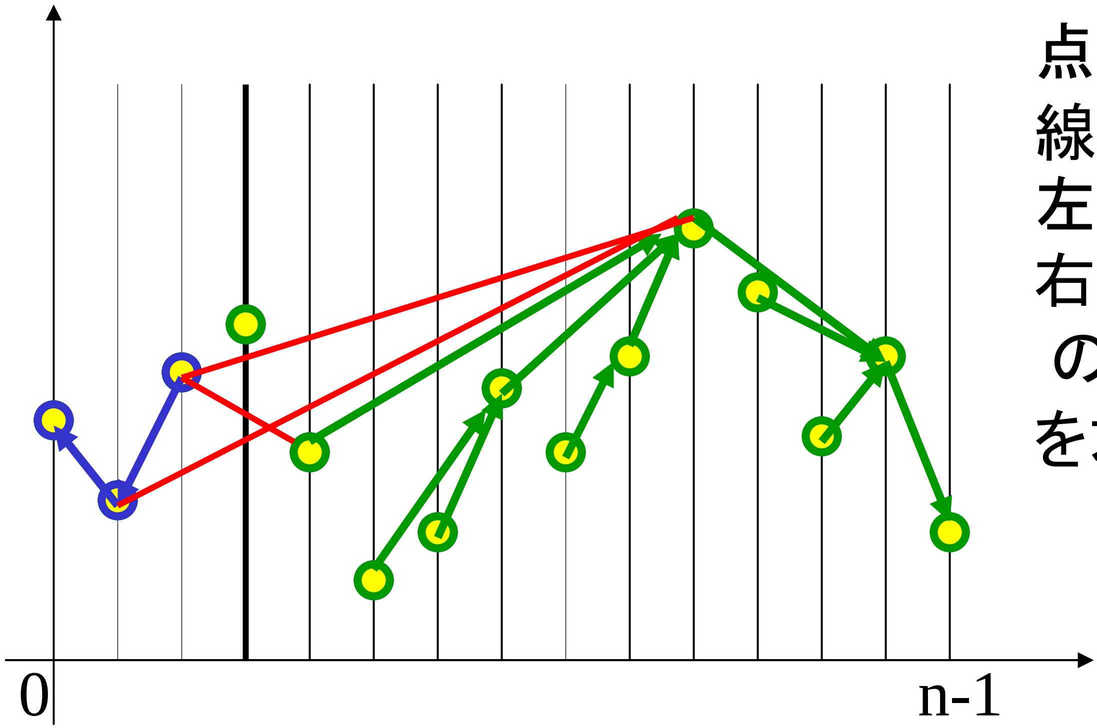


Starting from the line connecting point i-1 and point i+1, trace LCH(0,i-1) for the left points and UCH(i+1,n-1) for the right points.

i=2

iに関して区間平均の最大値を求める.

[0,i-1]の点と[i+1,n-1]の点を結ぶ傾き最大の線分を求める.

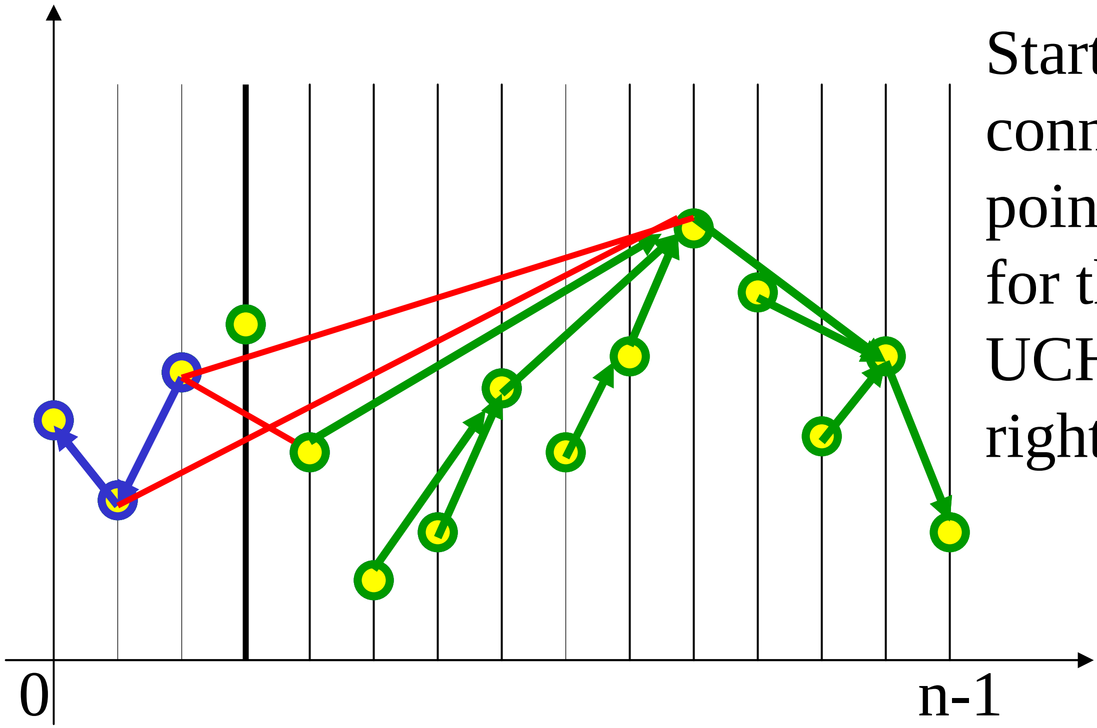


点i-1と点i+1を結ぶ
線分から始めて,
左の点はLCH(0,i-1)
右の点はUCH(i+1,n-1)
の辺を辿り, 共通接線
を求める.

i=3

Find an interval maximizing the average for each i.

Find a largest-slope line connecting points in [0,i-1] and [i+1,n-1].

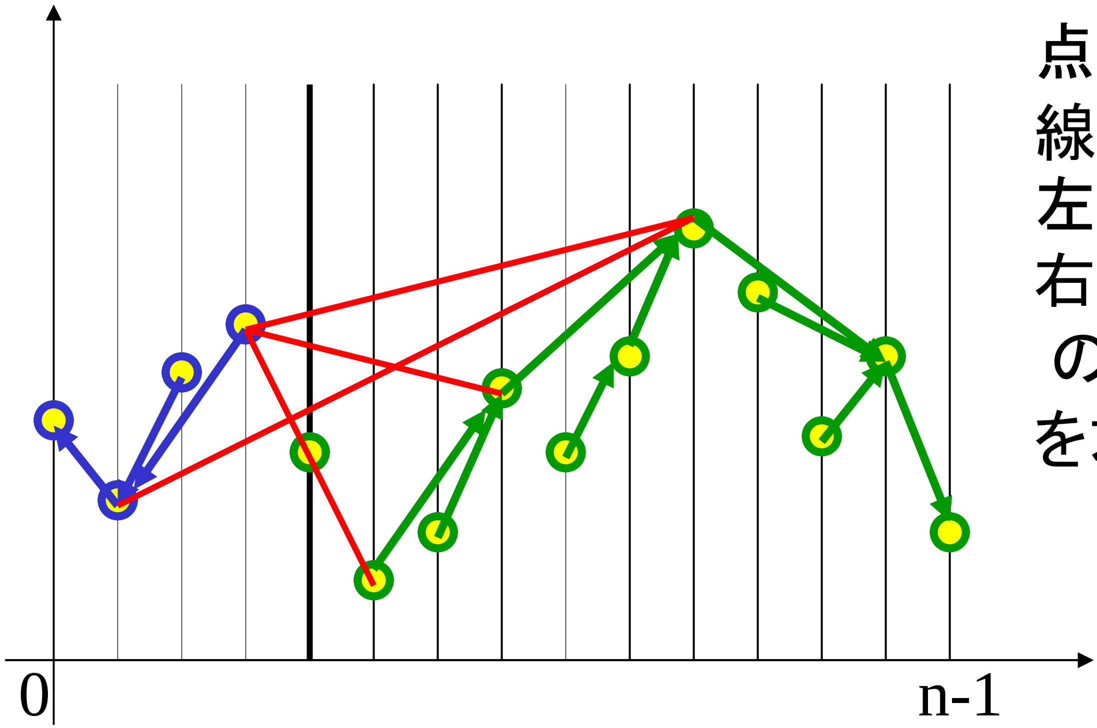


Starting from the line connecting point i-1 and point i+1, trace LCH(0,i-1) for the left points and UCH(i+1,n-1) for the right points.

i=3

iに関して区間平均の最大値を求める.

[0,i-1]の点と[i+1,n-1]の点を結ぶ傾き最大の線分を求める.

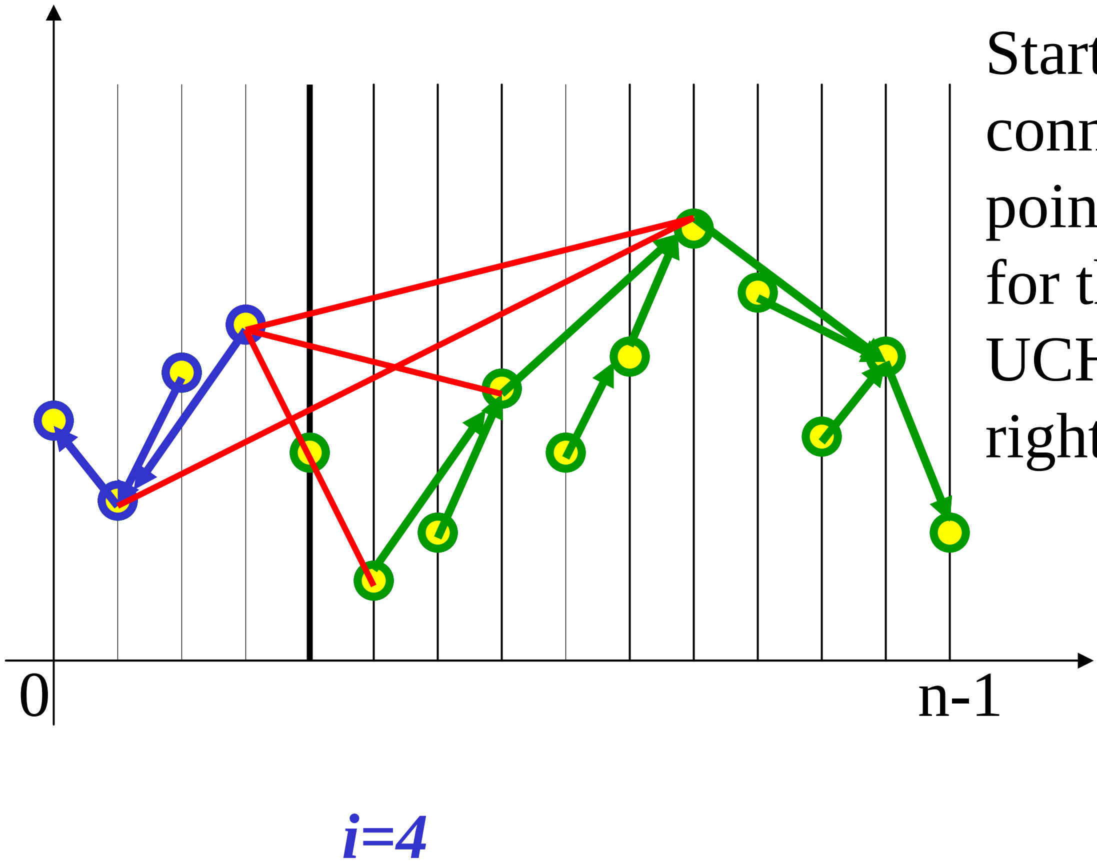


点i-1と点i+1を結ぶ線分から始めて,
左の点はLCH(0,i-1)
右の点はUCH(i+1,n-1)
の辺を辿り，共通接線を求める.

i=4

Find an interval maximizing the average for each i.

Find a largest-slope line connecting points in [0,i-1] and [i+1,n-1].

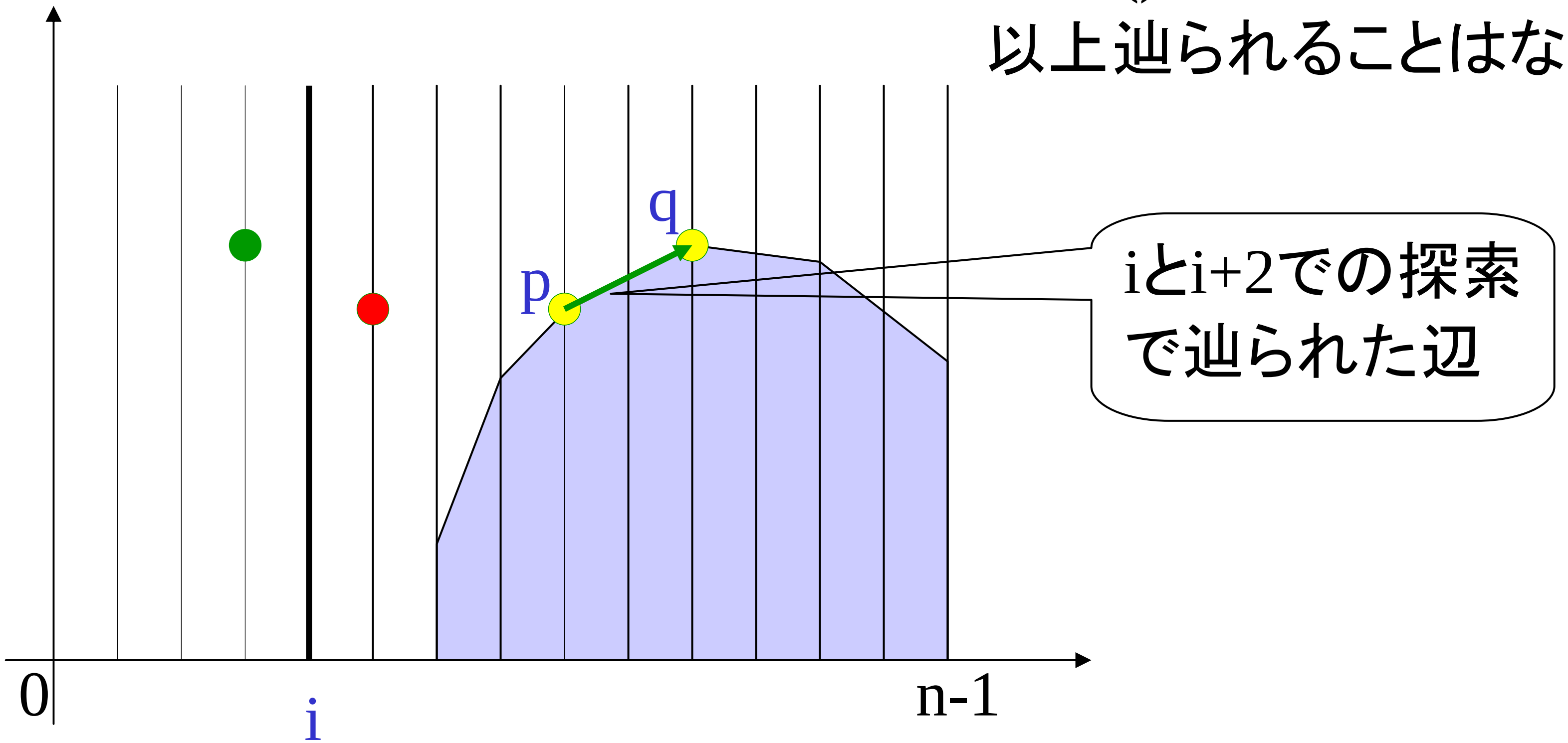


Starting from the line connecting point i-1 and point i+1, trace LCH(0,i-1) for the left points and UCH(i+1,n-1) for the right points.

この方法では同じ辺を何度も辿ってしまうことがあり，
計算時間がO(n)であることを保証できない．
$O(n^2)$時間かかってしまう例はあるか？

線形時間の保証

UCH()のどの辺も3度
以上辿られることはない．

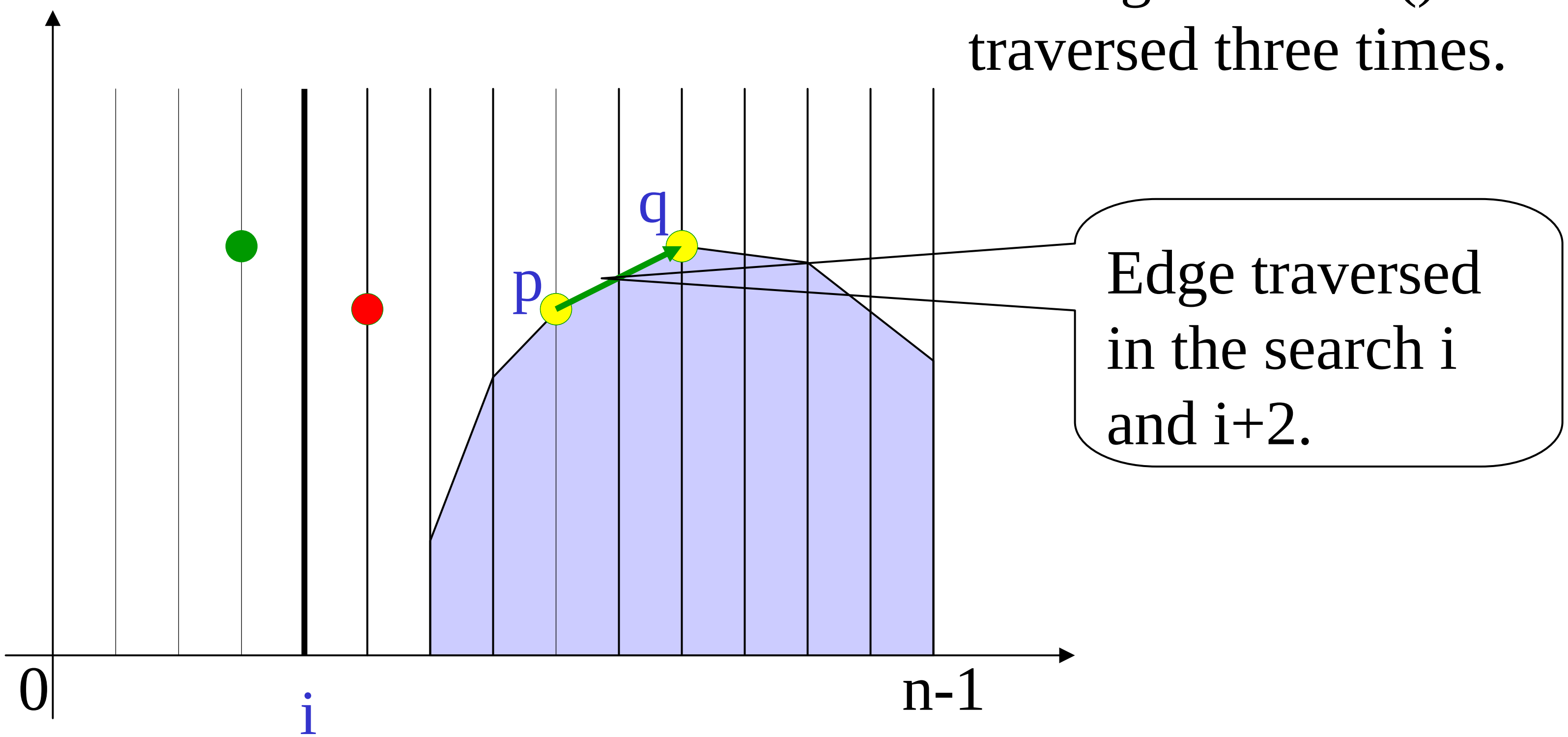


2点i-1, i+1とも直線pqより上になければならない．しかし，そう
すると，iでの探索で辺pqを辿ることはないから矛盾（何故か？）

The same edge can be traversed more than once in this algorithm and thus it cannot guarantee linear computation time.
Is there any example requiring $O(n^2)$ time?

Guarantee of linear time

No edge of UCH() is traversed three times.



The two points i-1 and i+1 must lie above the line pq. But, then the edge is never traversed in the search at i, a contradiction (Why?).

同様に，k≧2に対して，点i-1からの探索と点i+kからの探索でUCHの同じ辺が辿られることはない.

注意: 点i-1からの探索と点iからの探索でUCHの同じ辺が辿られることはある.

結局，UCHのどの辺も3度以上辿られることはないことが証明された. LCHについても同様である.

Similarly, for k$\geqq$2, the same edge in UCH is never traversed in the search from point i-1 and from i+k.

Remark：It happens that the same edge in UCH is traversed in the search from point i-1 and point i.

At last, it has been proven that no edge in UCH is traversed three times. Same for LCH.

最終的にアルゴリズムは次のようになる.

アルゴリズムP4-A2:

```
最初に, S[i]=a[0]+a[1]+...+a[i]を求める(i=0, 1, ... , n-1)
次に点集合(i, S[i]), i=0, ..., n-1を構成.
LCH(0, i-1)とUCH(i+1, n-1)を順に構成.
maxslope=(a[0]+a[1])/2;
for(i=1; i<n-1; i++){
  点i-1と点i+1を結ぶ線分から始めて,
  左の点はLCH(0,i-1), 右の点はUCH(i+1,n-1) の辺を辿り,
  共通接線(p, S[p])-(q, S[q])を求める.
   if( 共通接線の傾き>maxslope) maxslope=共通接線の傾き;
}
maxslopeを出力;
```

演習問題:このアルゴリズムを実装せよ.

Finally we have the following algorithm.

Algorithm P4-A2:
First, we compute S[i]=a[0]+a[1]+...+a[i] (i=0, 1, ... , n-1).
Then, construct a set of points (i, S[i]), i=0, ..., n-1.
Construct LCH(0, i-1) and UCH(i+1, n-1) in order.

```
maxslope=(a[0]+a[1])/2;
for(i=1; i<n-1; i++){
  Starting from the line connecting point i-1 and point i+1,
  traverse edges in LCH(0,i-1) for the left point and in UCH(i+1,n-1)
  for the right point to find the common tangent (p, S[p])-(q, S[q]).
   if( slope of tangent line>maxslope) maxslope=slope of tangent line;
}
output maxslope;
```

Exercise: Implement this algorithm.