SAND2010-7479C

# Estimating the longest increasing sequence in polylogarithmic time

C. Seshadhri (Sandia National Labs,Livermore)
(work done in IBM Almaden)

Joint work with Michael Saks
(Rutgers University)

# The problem

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **4** | 24 | **10** | 9 | **15** | **17** | 20 | **18** | 4 | **19** | 3 |

- Given array f:[n] → N, find (length of) Longest Increasing Subsequence (LIS)
  - Rather self-explanatory

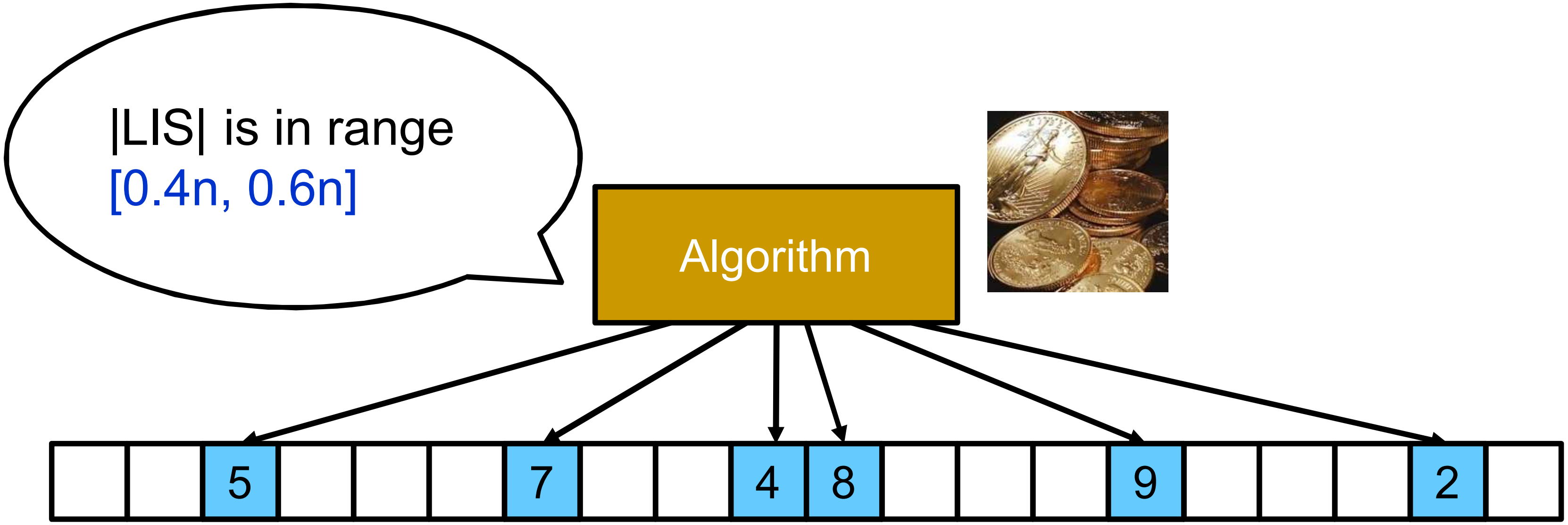
- By now, textbook dynamic programming problem
  - [CLRS 01] Chapter 15.4 (Longest Common Subsequence), Starred Problem 15.4-6
  - [Schensted, Fredman] O(n log n) algorithm

# Almost 50 years…

- [Schensted 61] [Fredman 75] [Apostolica Guerra 87] [Atshul et al 90] [Ramanan 97] [Goldreich Goldwasser Ron 97] [Baik Deift Johansson 99] [Delcher et al 99] [Dodis et al 99] [Aldous Diaconis 99] [Ergun et al 99] [Bespamyatnuikh Segal 00] [Fischer 01] [Liben-Nowell Vee Zhu 03] [Zhang 03] [Ailon et al 03] [Parnas Ron Rubinfeld 03] [Gal Gopalan 07] [Gopalan et al 07] [Sun Woodruff 07] [Ergun Jowhari 08]

- LIS is simple version of Longest Common Subsequence
  - Can’t list all references for LCS

# Too much to read



|LIS| is in range
[0.4n, 0.6n]

- Array f is extremely large, so can’t read all of it
  - What can we say about LIS length, if we see very little?
    - |LIS| = LIS length
  - Read only poly(log n) positions
    - For n = billion, (log n) = 21
  - Obviously randomized

# Too much to read

|LIS| is in range
[0.4n, 0.6n]

- Array
  - Wh ittle?
    - |L
  - Read only
  - Obviously ra

Mathematical question: How to locate small portion of array that tells about |LIS|?

Do such portions even exist?

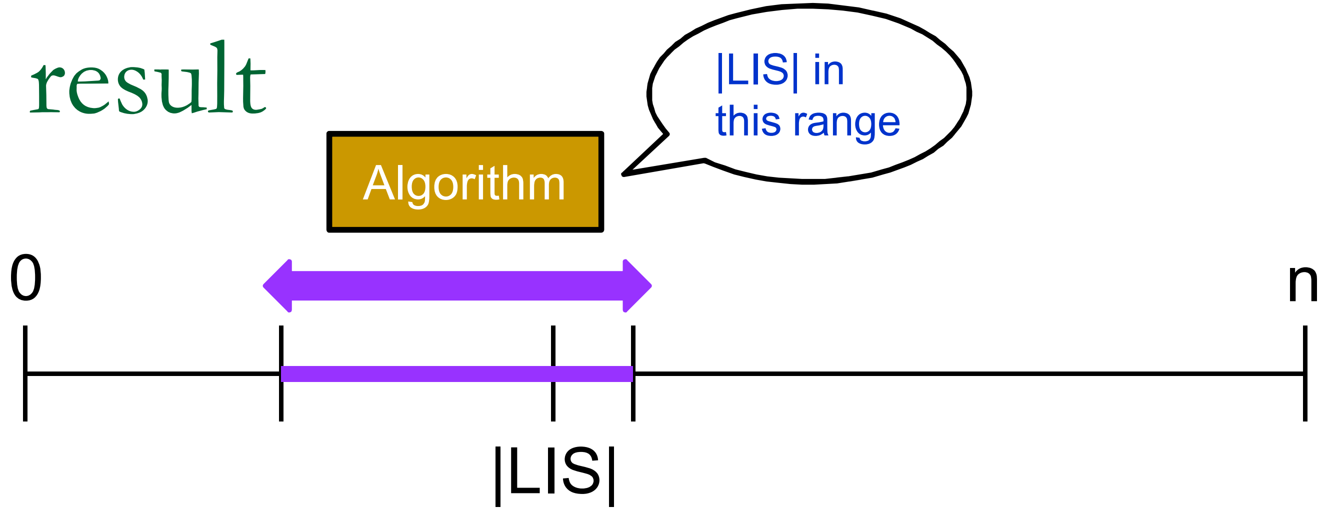
# Uniform sample says nothing

|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 3 | 6 | 5 | 8 | 7 | 10 | 9 | - - - - - - |

- Choose uniform random sample of poly(log n) size
- |LIS| = n/2, but random sample always increasing

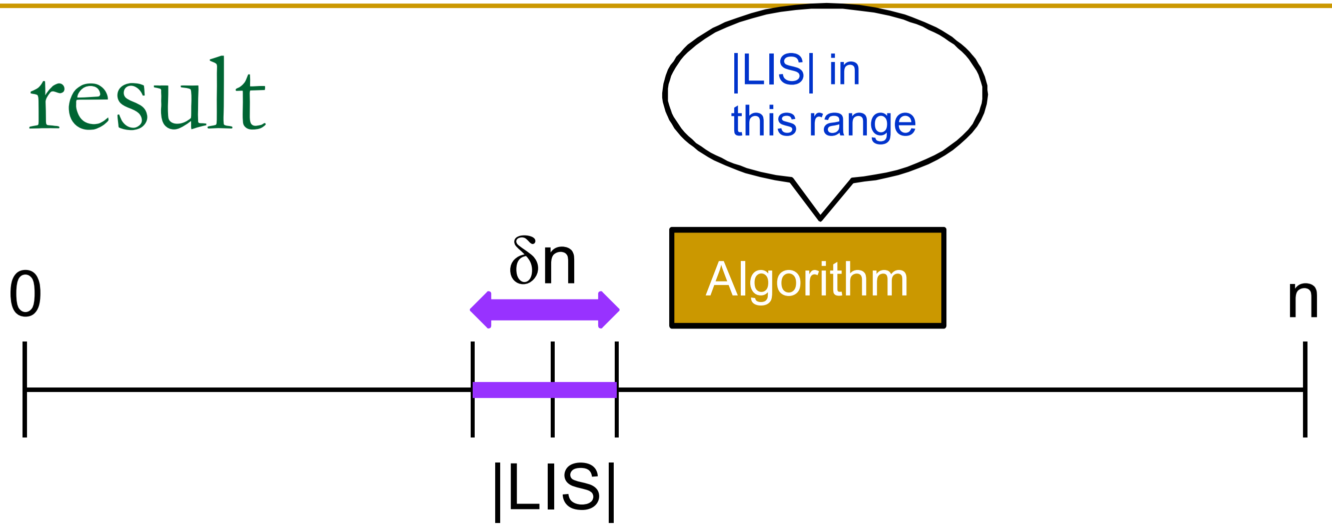
- So not really that easy to learn about |LIS|…

# Our result



- We want range to be small

# Our result

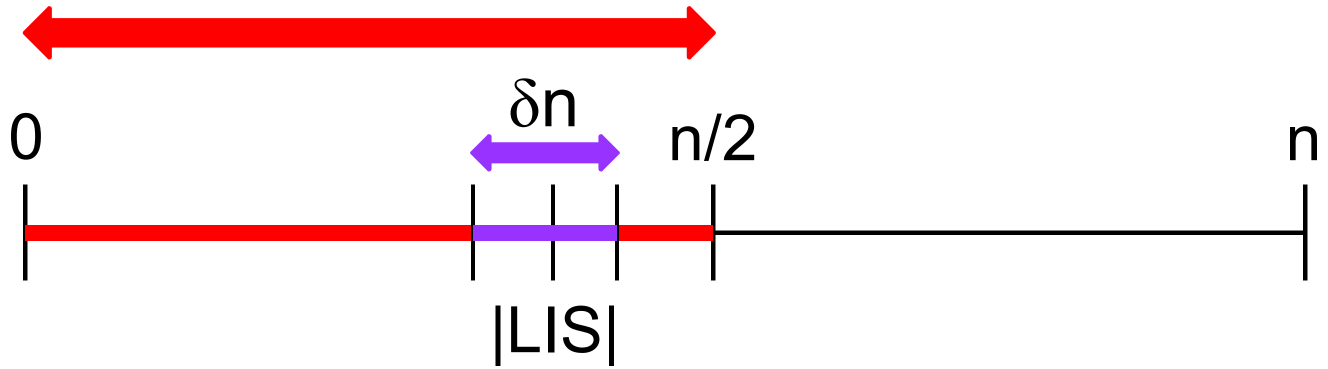


- We want range to be small
- [This work] For any (constant) $\delta > 0$

  Algorithm gives additive $\delta n$ approximation to |LIS|

  Running time is $2^{1/\delta} (\log n)^c$

# Our result



- We want range to be small
- [This work] For any (constant) $\delta > 0$

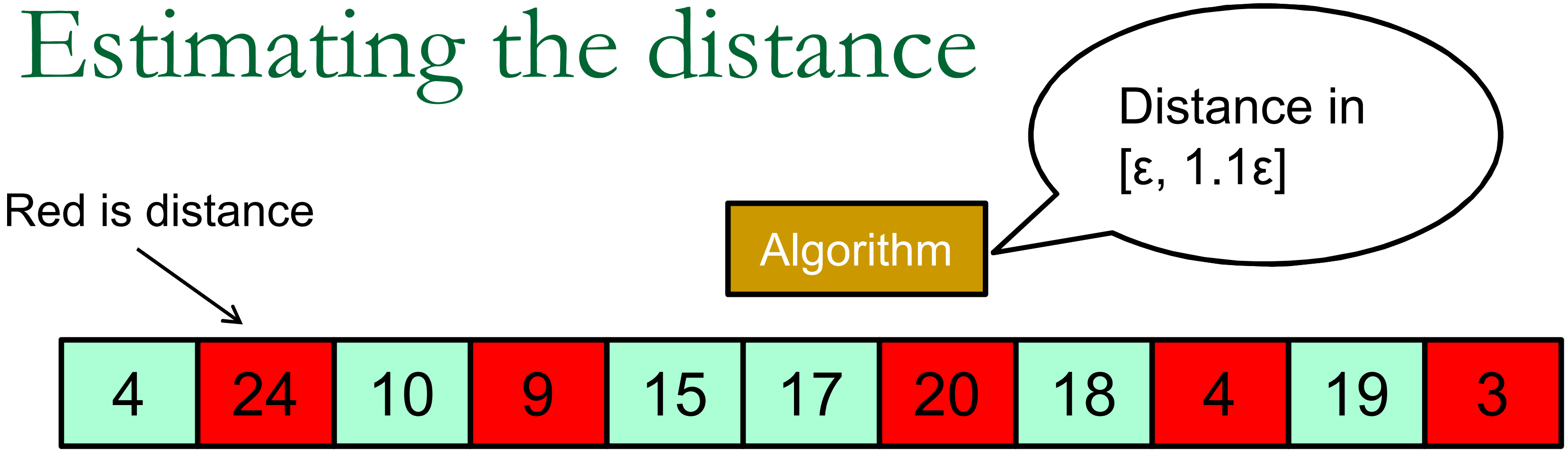
  Algorithm gives additive $\delta n$ approximation to |LIS|

  Running time is $2^{1/\delta} (\log n)^c$

- [Ailon Chazelle Liu S 03] [Parnas Ron Rubinfeld 03]

  Previous best: $\delta = ½$

# Estimating the distance



Red is distance

Algorithm

Distance in
[ε, 1.1ε]

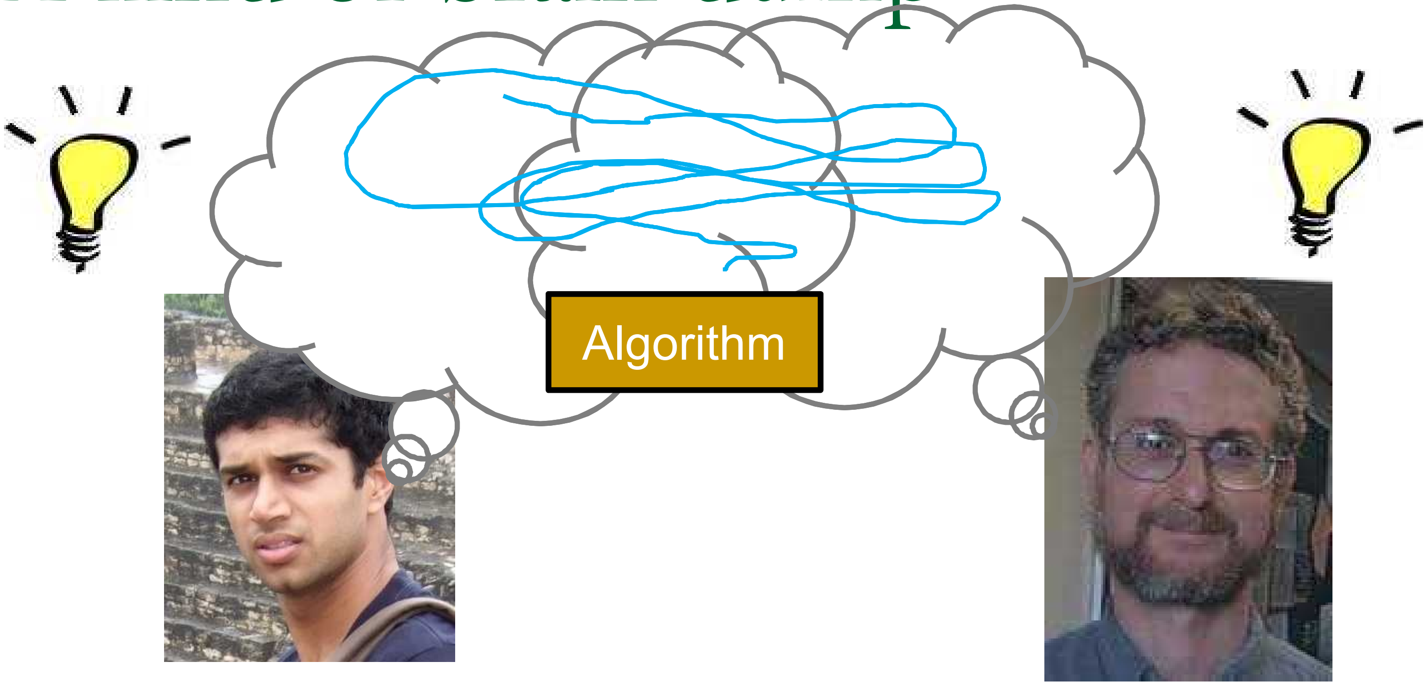
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 🟩 4 | 🟥 24 | 🟩 10 | 🟥 9 | 🟩 15 | 🟩 17 | 🟥 20 | 🟩 18 | 🟥 4 | 🟩 19 | 🟥 3 |

- Distance = (n - |LIS|)/n
- Plethora of algorithms: approximate distance
- [ACLS,PRR] 2-multiplicative approx to distance
- [This result] (1+δ)-approx to distance, for any constant δ

# What I’m supposed to do

- First, the algorithm (for LIS)
- Then, a proof of why it works
- I won’t do that

# A kind of brain dump

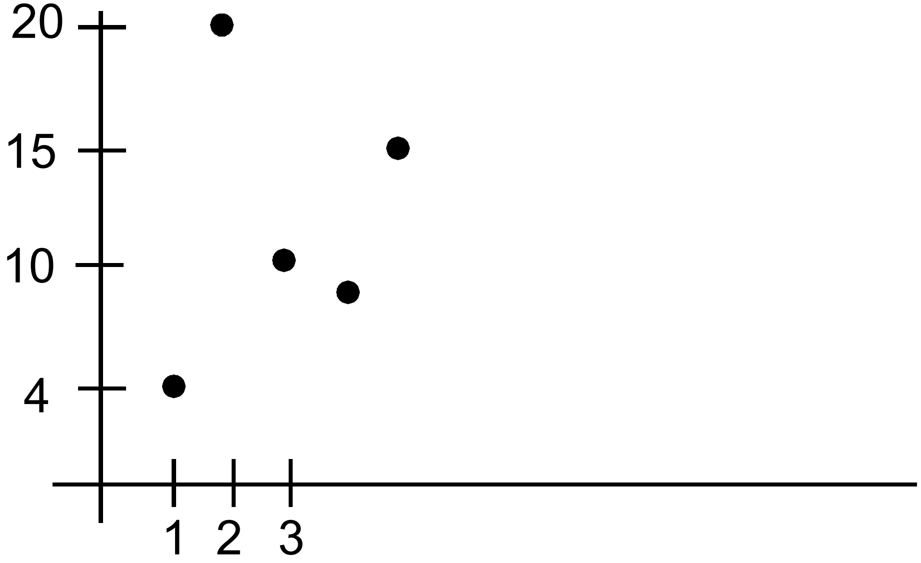


Algorithm

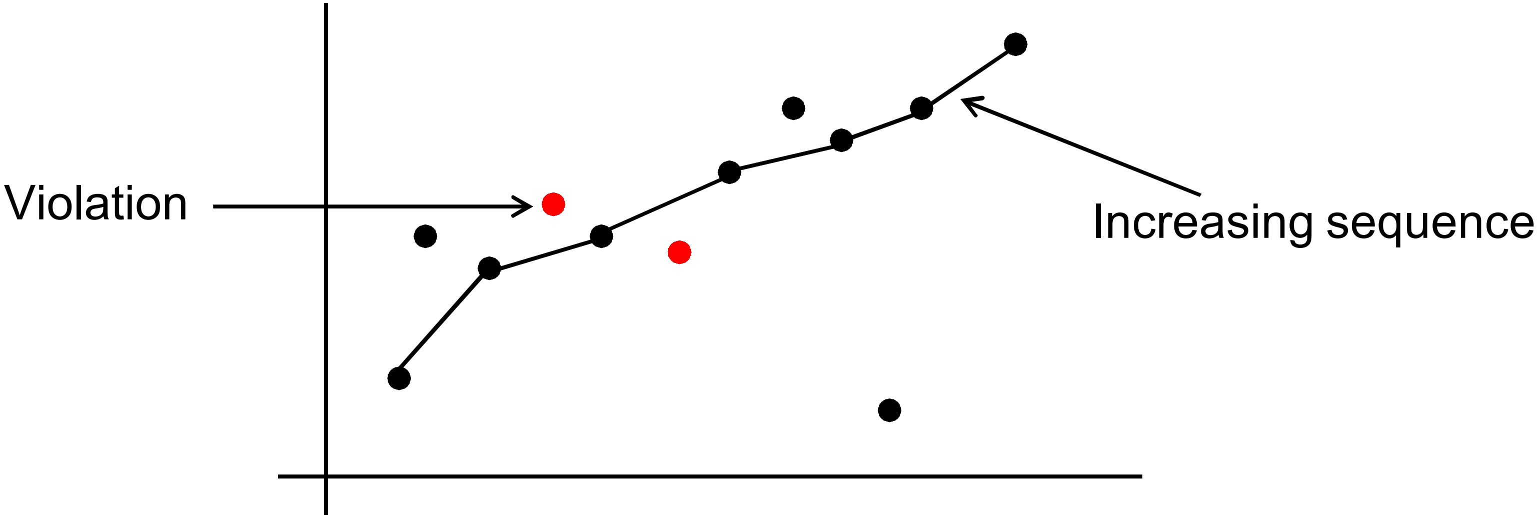
- Why is this problem hard?
- How did we arrive at this algorithm?
  - If there’s time, you might even see it

# Prelims: the array in space

| | | | | |
|---|---|---|---|---|
| 4 | 20 | 10 | 9 | 15 |

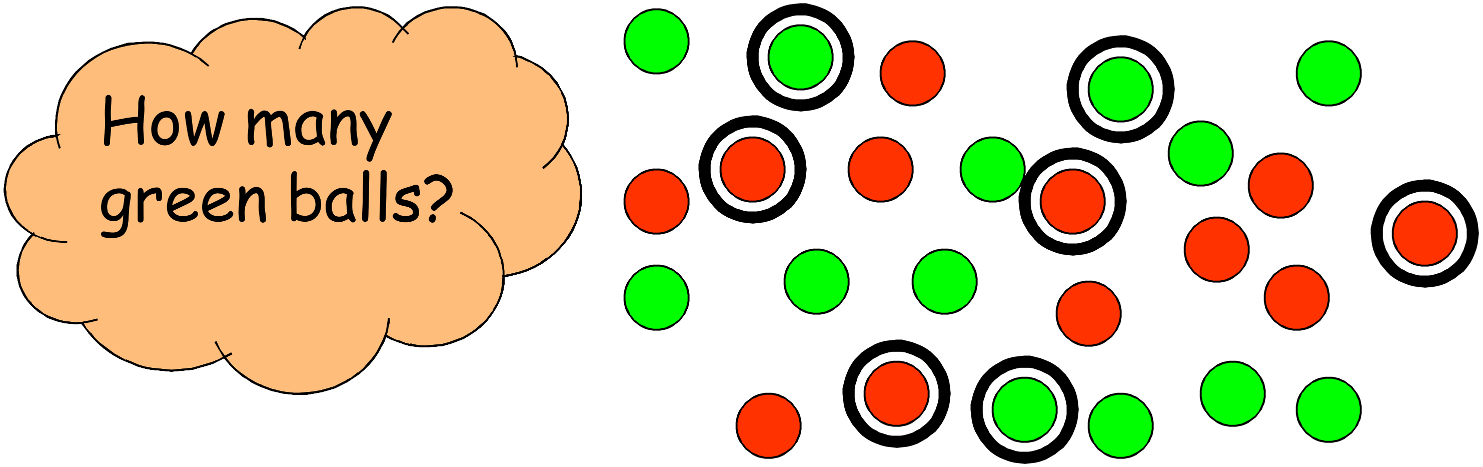


# Prelims: the array in space



- Input is points in plane, given as array
- (LIS is longest chain in partial order)

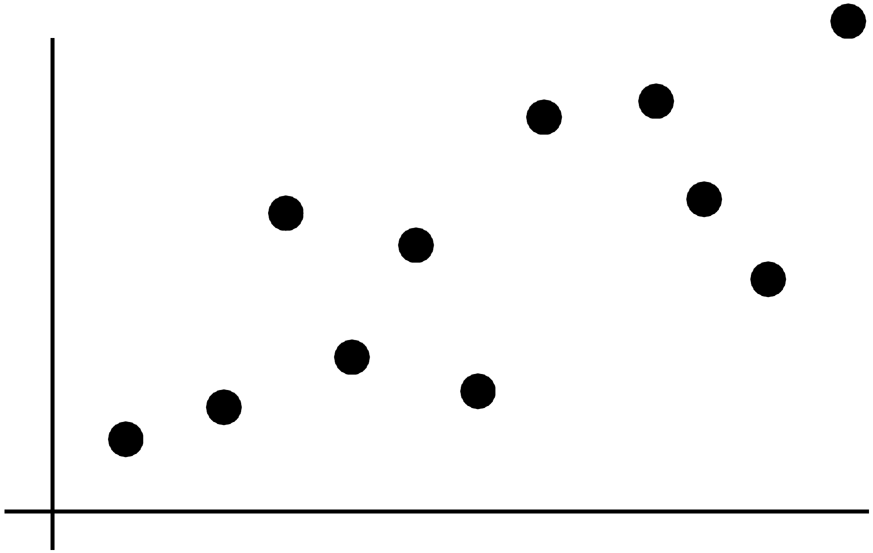
# The use of randomness



- Find fraction of green
- Randomized - in constant time
  - [Chernoff-Hoeffding] (log n)/$\alpha^2$ samples for error $\alpha$
- From now on, assume we can do this

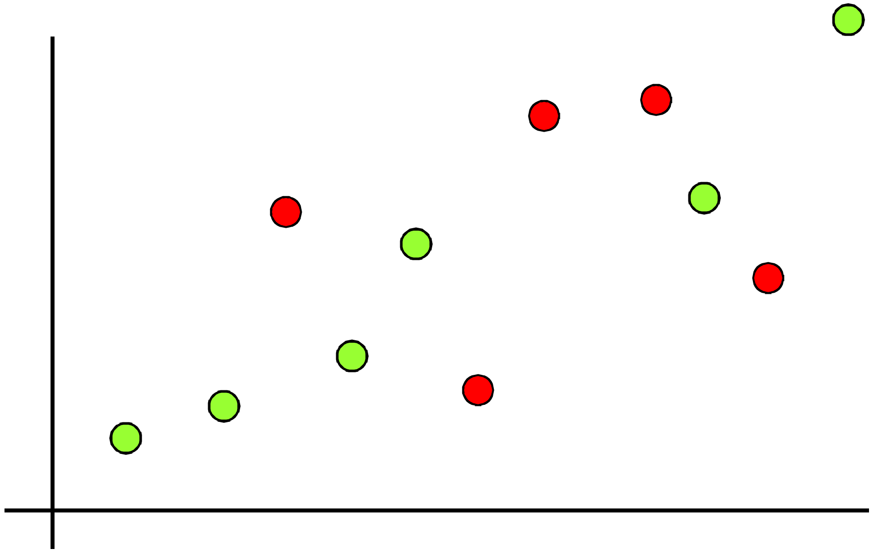
# Let’s look at the history

# Coloring points



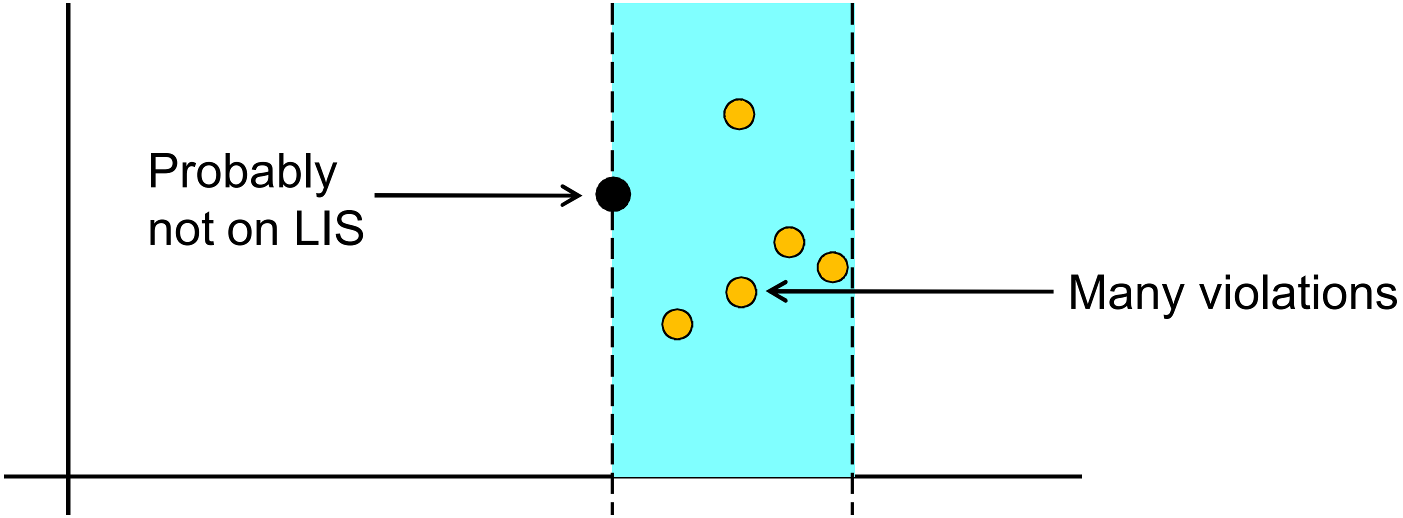
- Characterize points as good (green) or bad (red)

# Coloring points

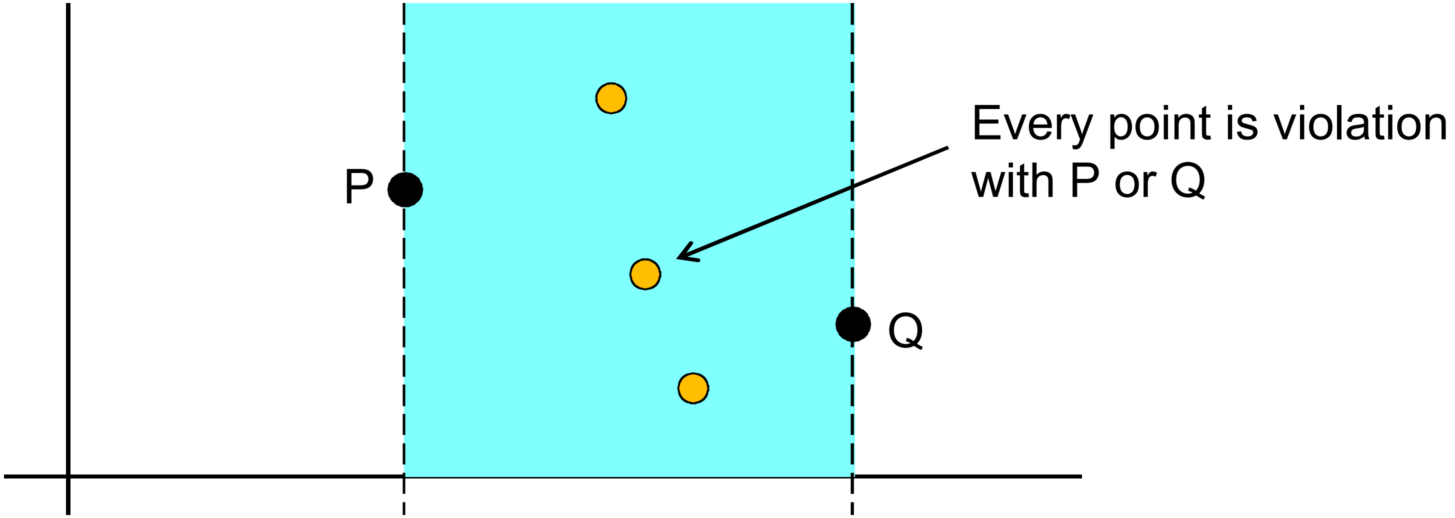


- Characterize points as good (green) or bad (red)
- Good points are increasing seq.
- Not too many bad points (compared to distance)
- There is (log n) algo to tell if point is good/bad

# The violation counting trick



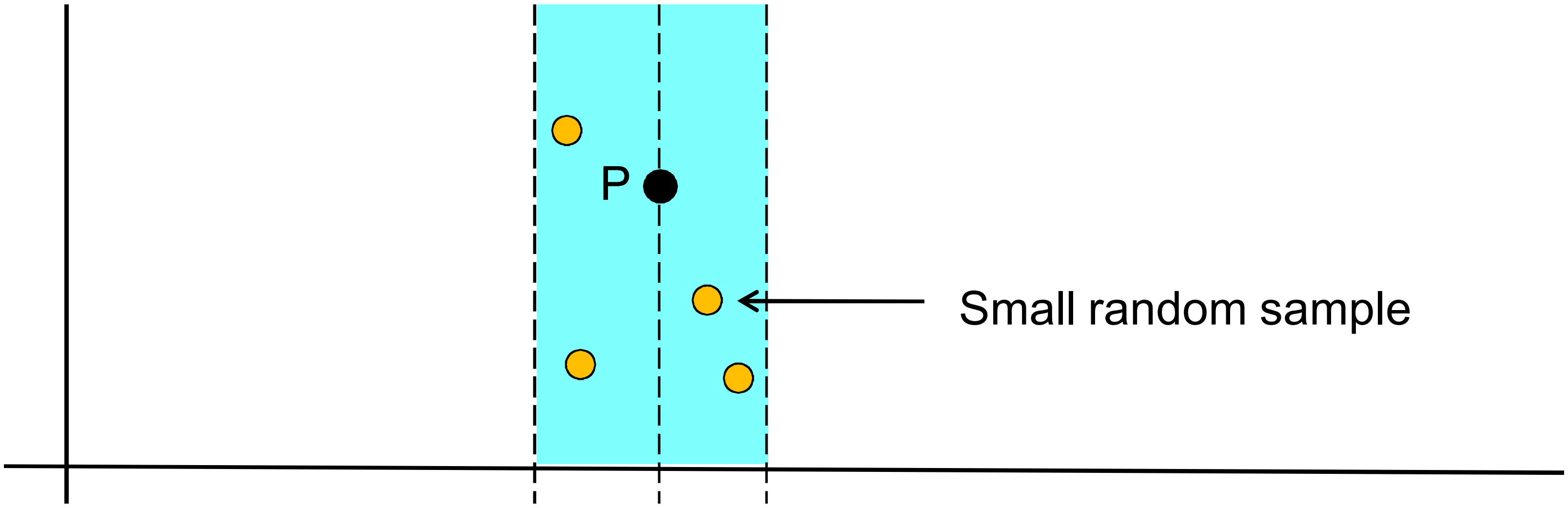
# The violation counting trick



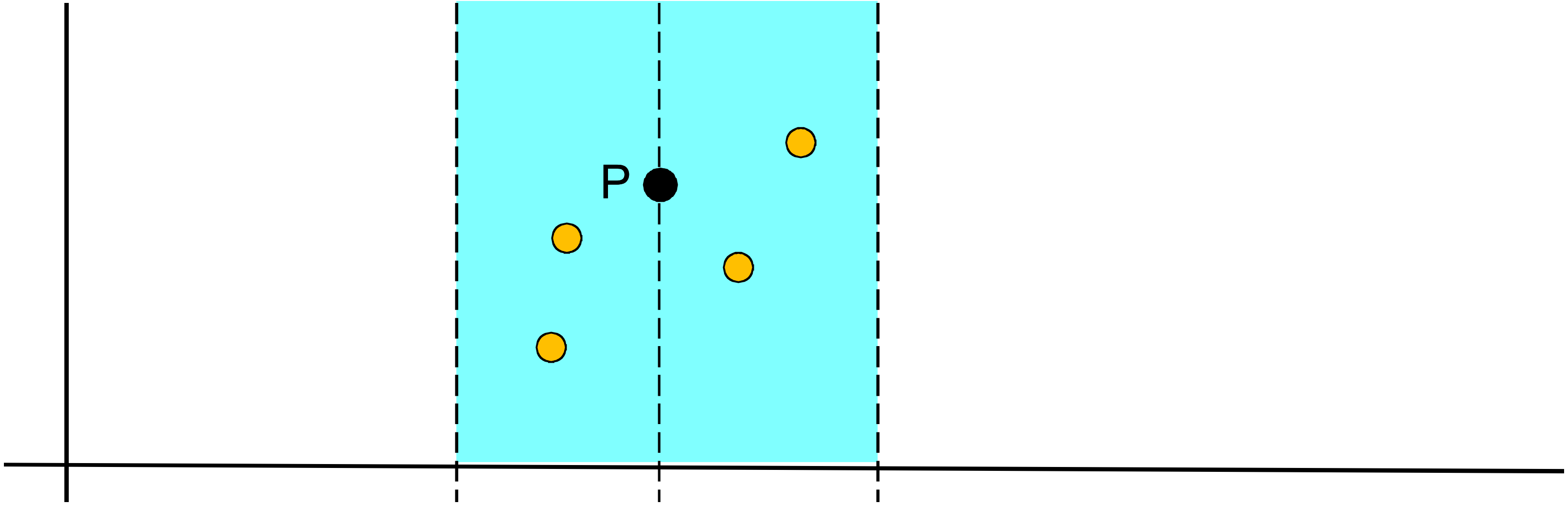
- Every point in [P,Q] is violation with either P or Q
- [Ergun Kannan Kumar Rubinfeld Viswanathan 99]

So at least half the points in [P,Q] are violation with P or violation with Q

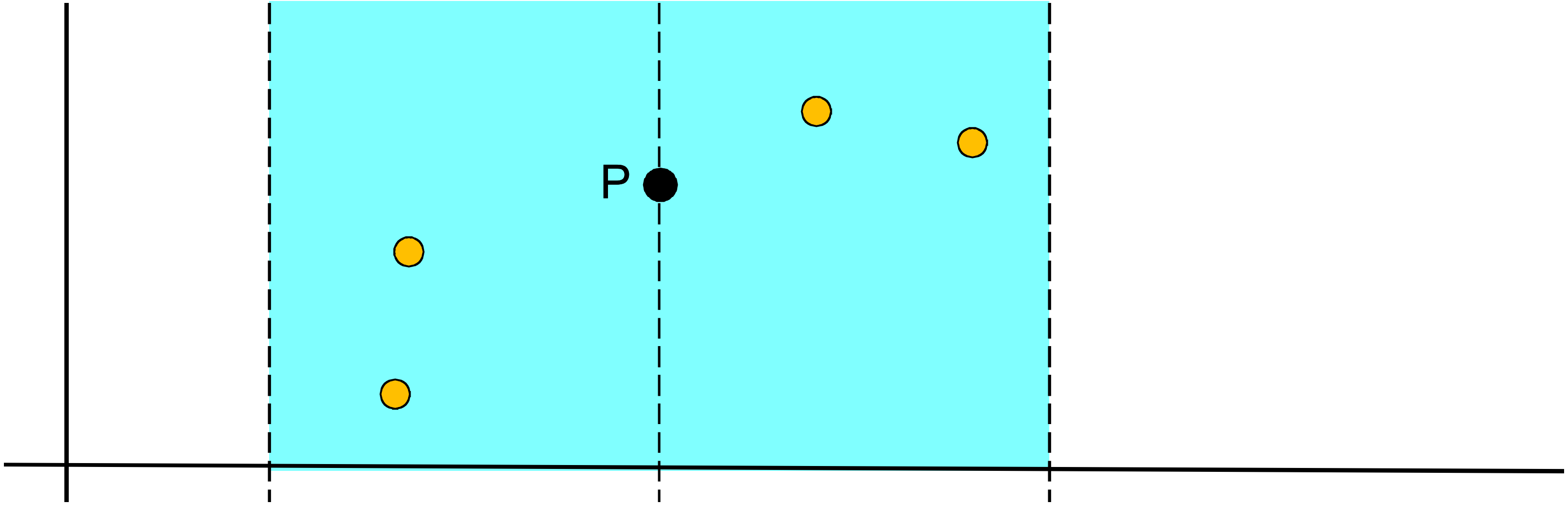
# The generic algorithm



# The generic algorithm

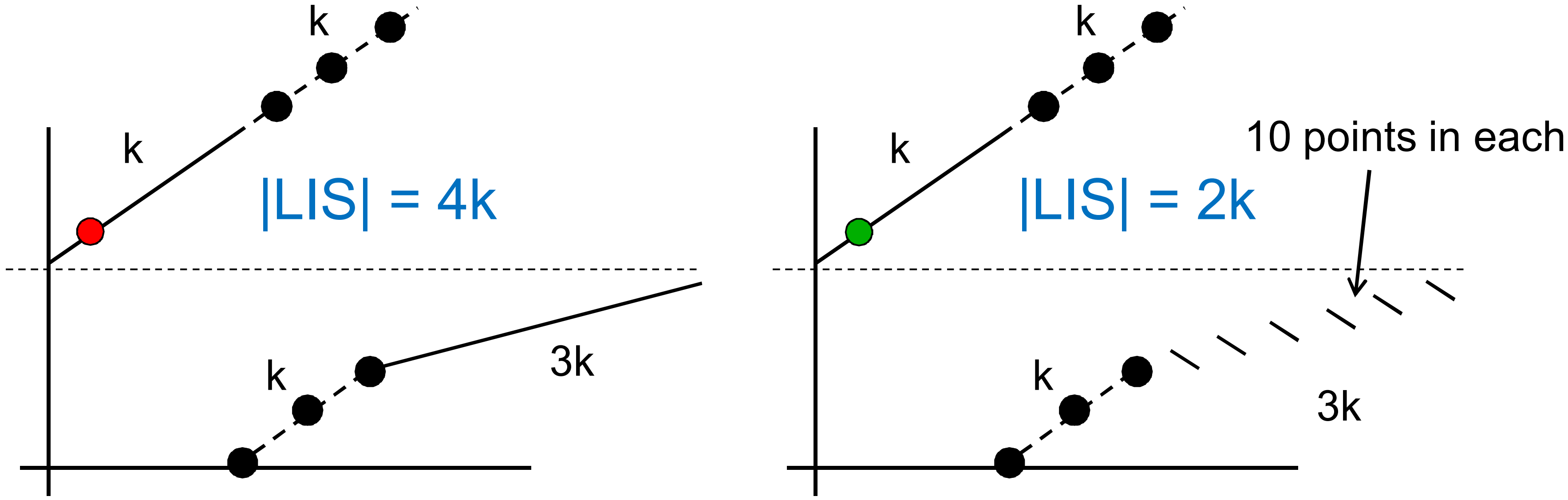


# The generic algorithm



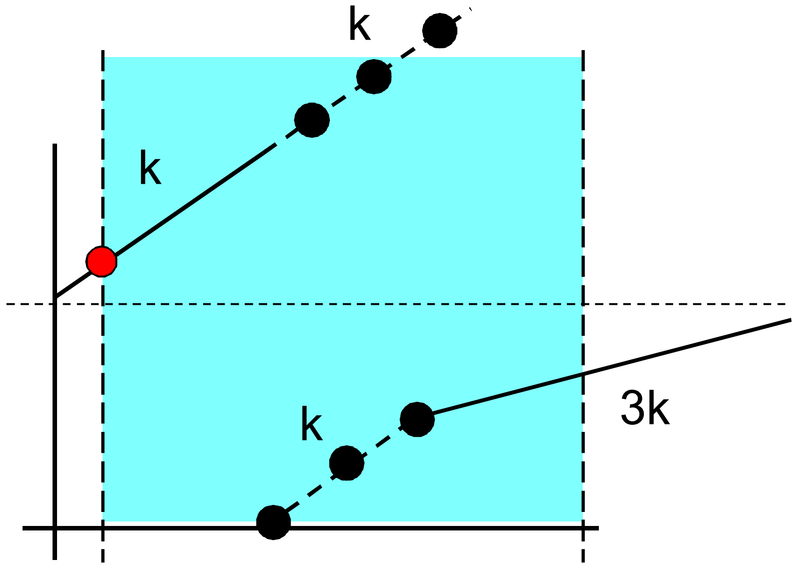
- Study samples in all “neighborhoods” of P
  - Decide whether P is good or bad
- Uses neighborhoods of size $2^k$, so (log n) time overall
- Very nice, clean approach

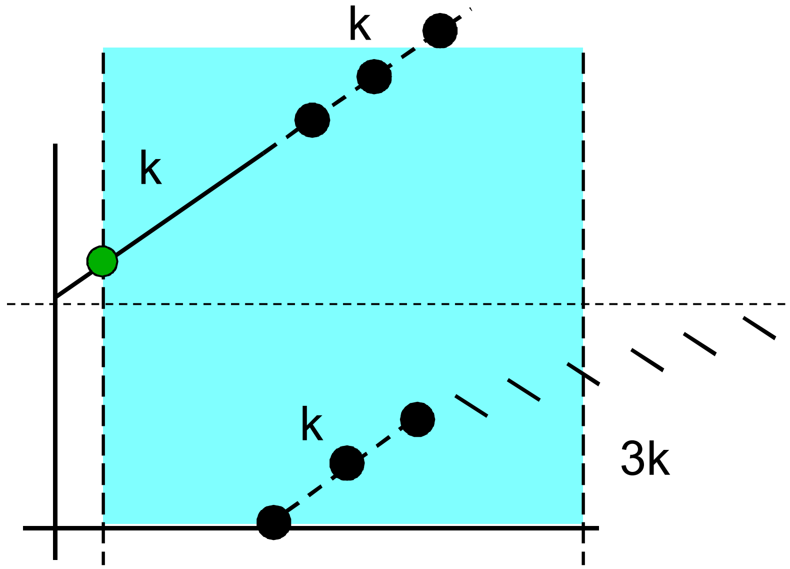
# A major obstruction



- The decision of good/bad for a point depends on small scale properties of “far away” portions

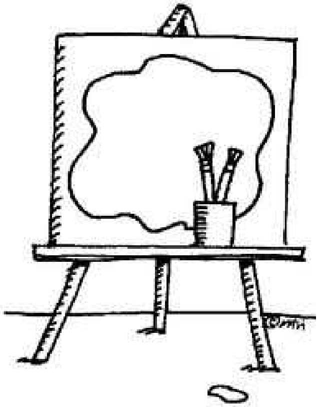
# A major obstruction



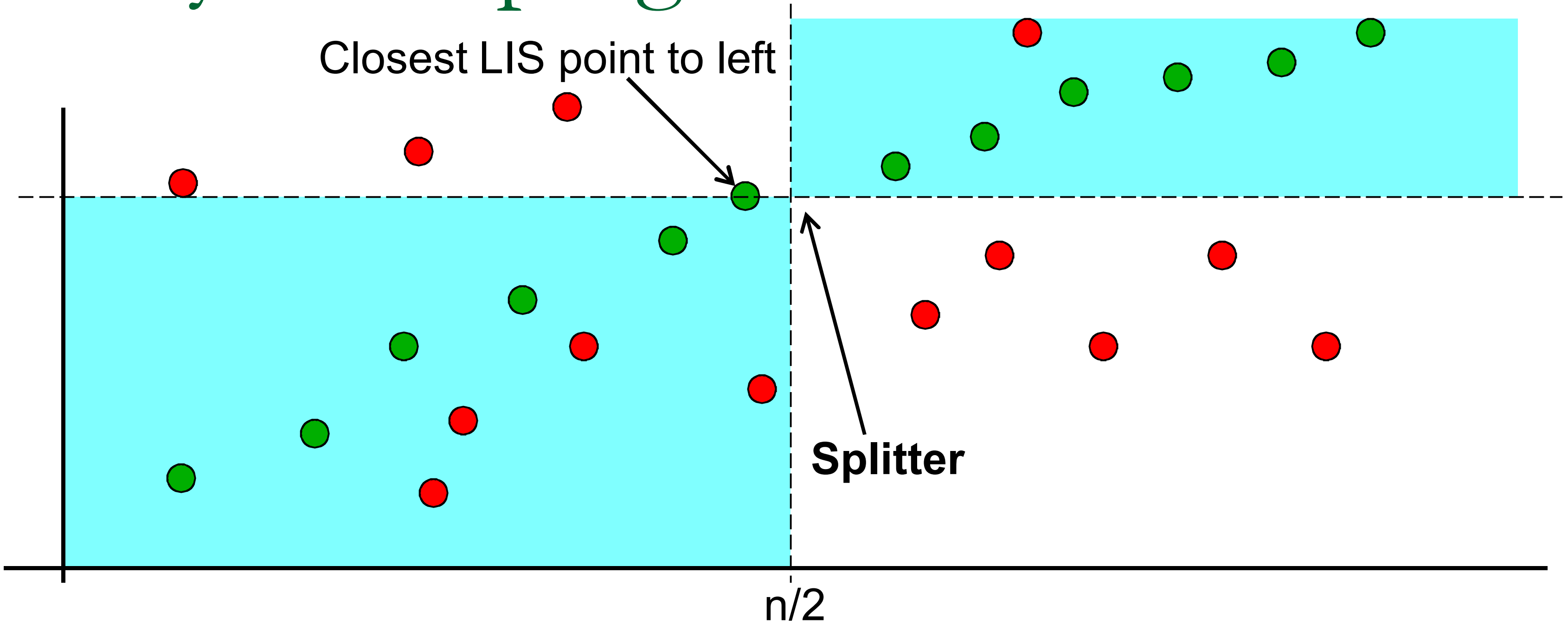


- Random samples in neighborhoods of points are identical!
- “Can we really estimate LIS in polylog time?”
- We need to rethink our approach

# Back to the drawing board

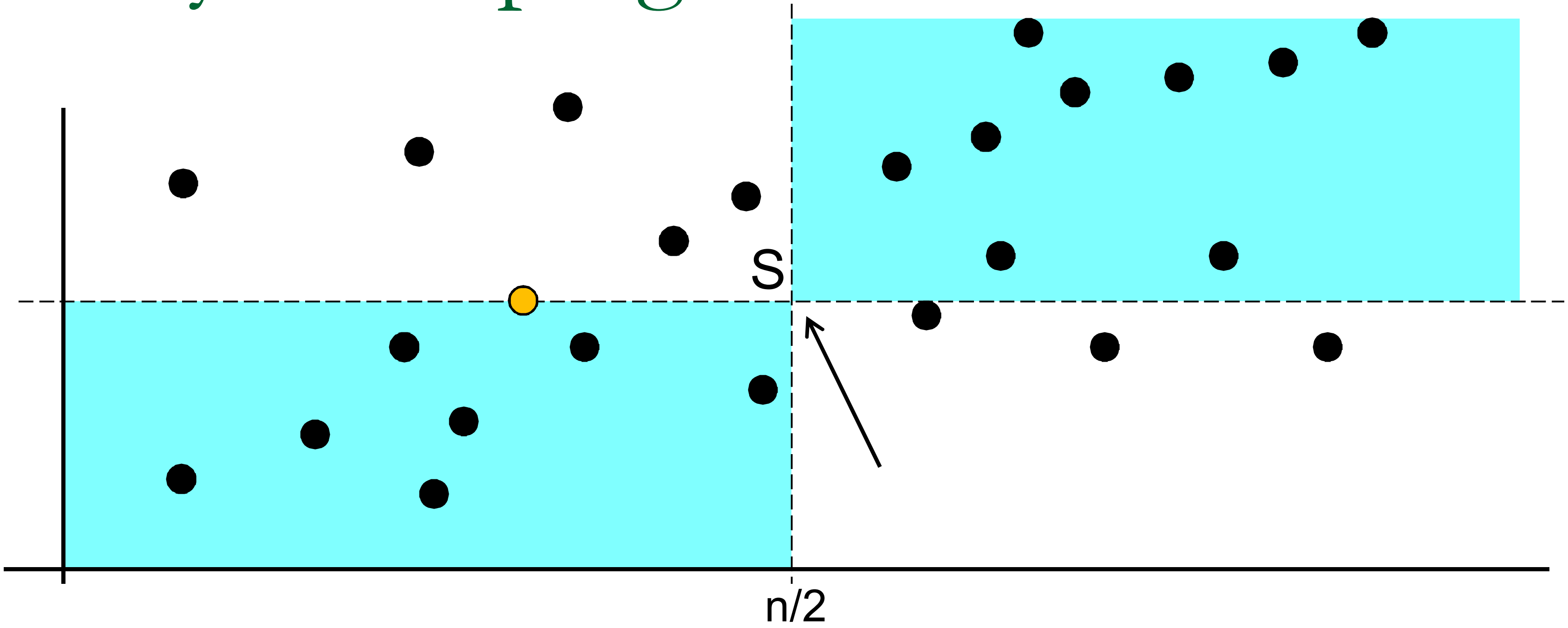


# The dynamic program



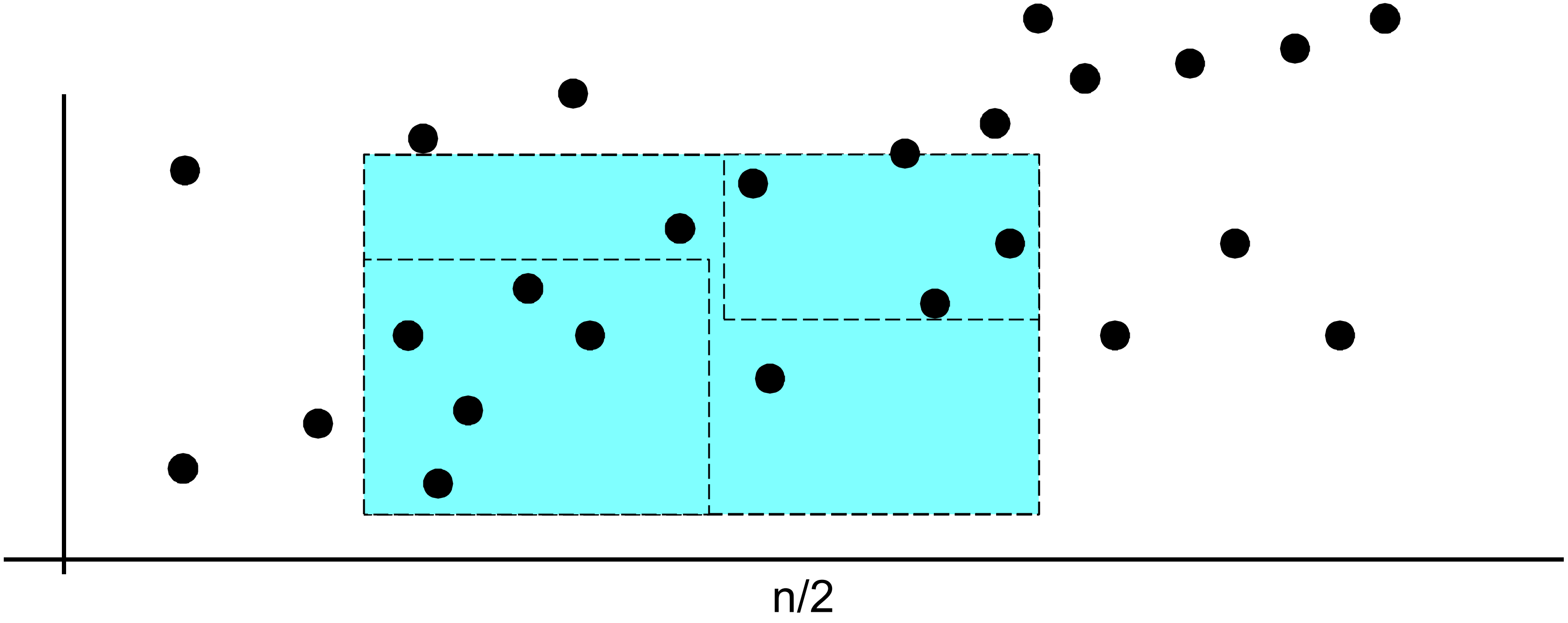
- Closest LIS point to left gives “splitter”
- Find LIS is each blue region. Piece together!
  - So we break up original problem into subproblems

# The dynamic program



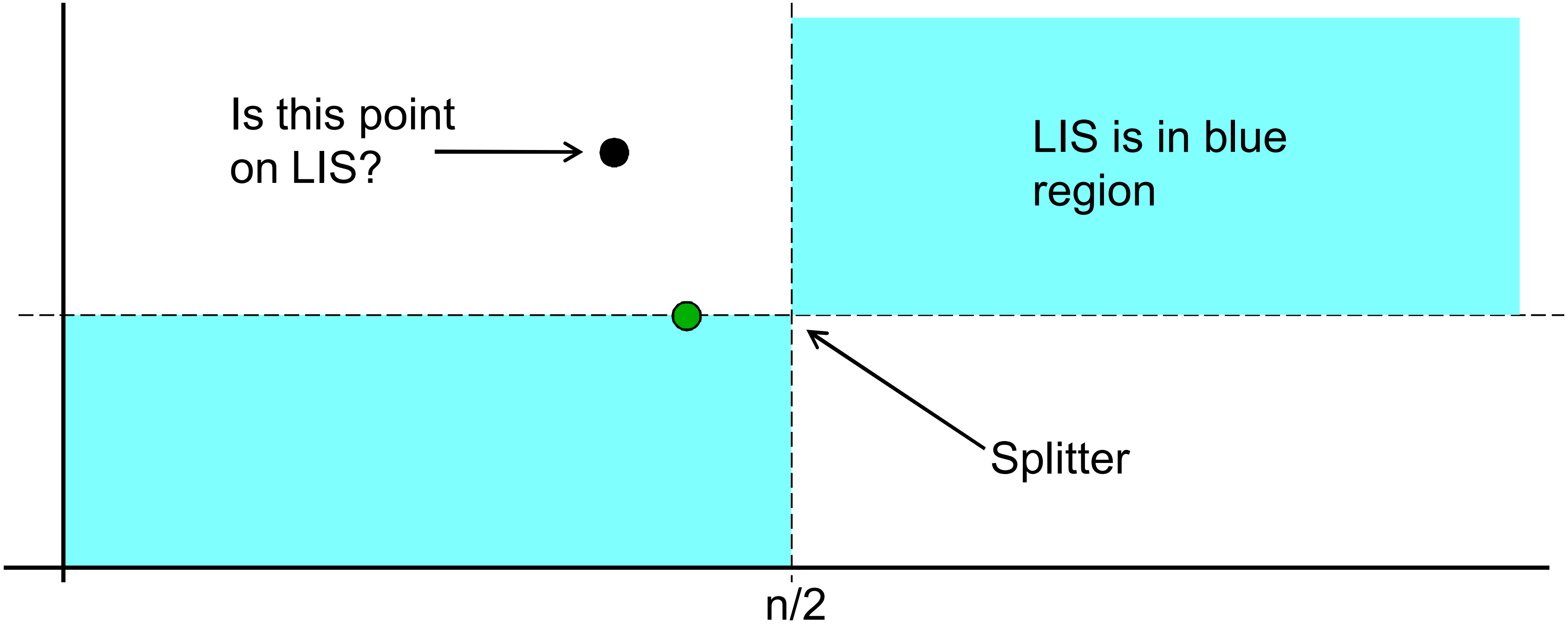
- But we don't know right splitter.
  - So try all possible! Only n different choices
- Choose the one that gives the largest sum of LIS's
  - $\text{Max}_S$ (|LIS-below-S| + |LIS-above-S|)

# The dynamic program



- If you LIS in all small boxes, you can build LIS for bigger boxes
- Not the most efficient DP…
- So our sublinear algo has to mimic this process

# The IP



Is this point on LIS?

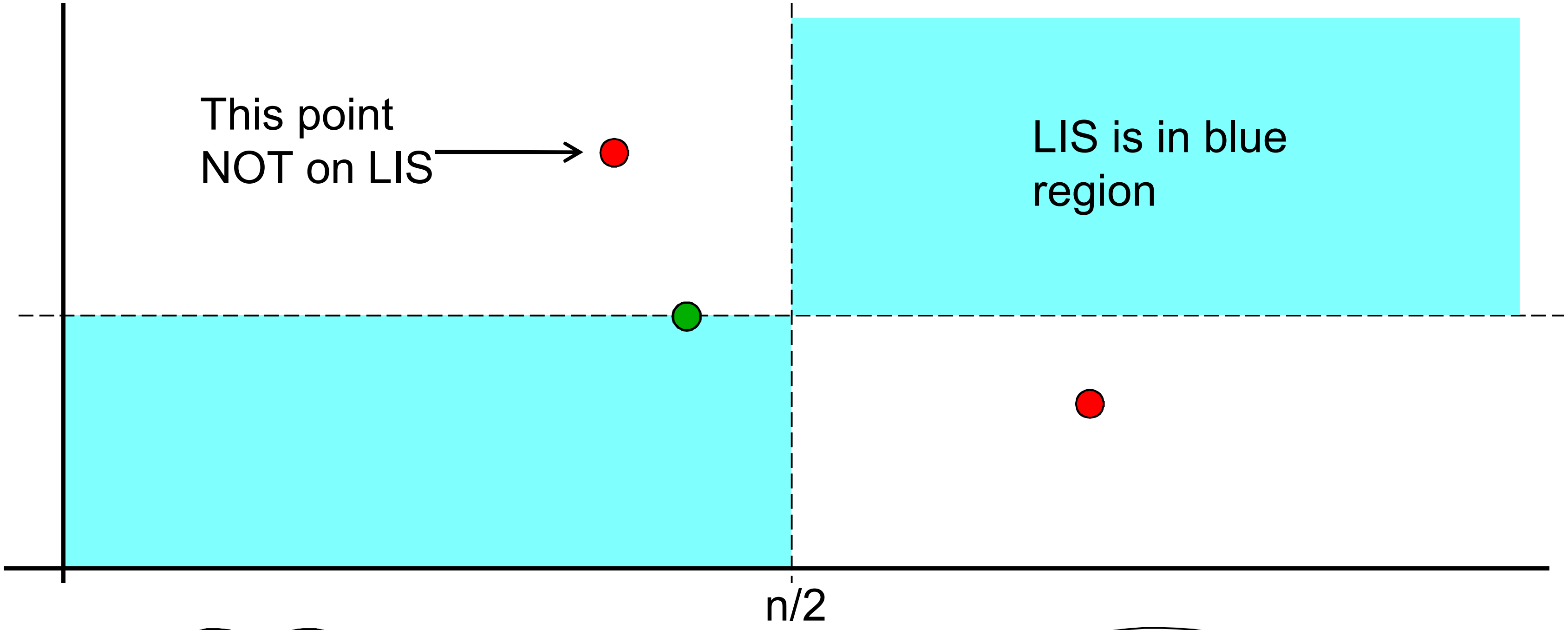
LIS is in blue region

Splitter

Where is the splitter?

It is there.

# The IP



This point NOT on LIS

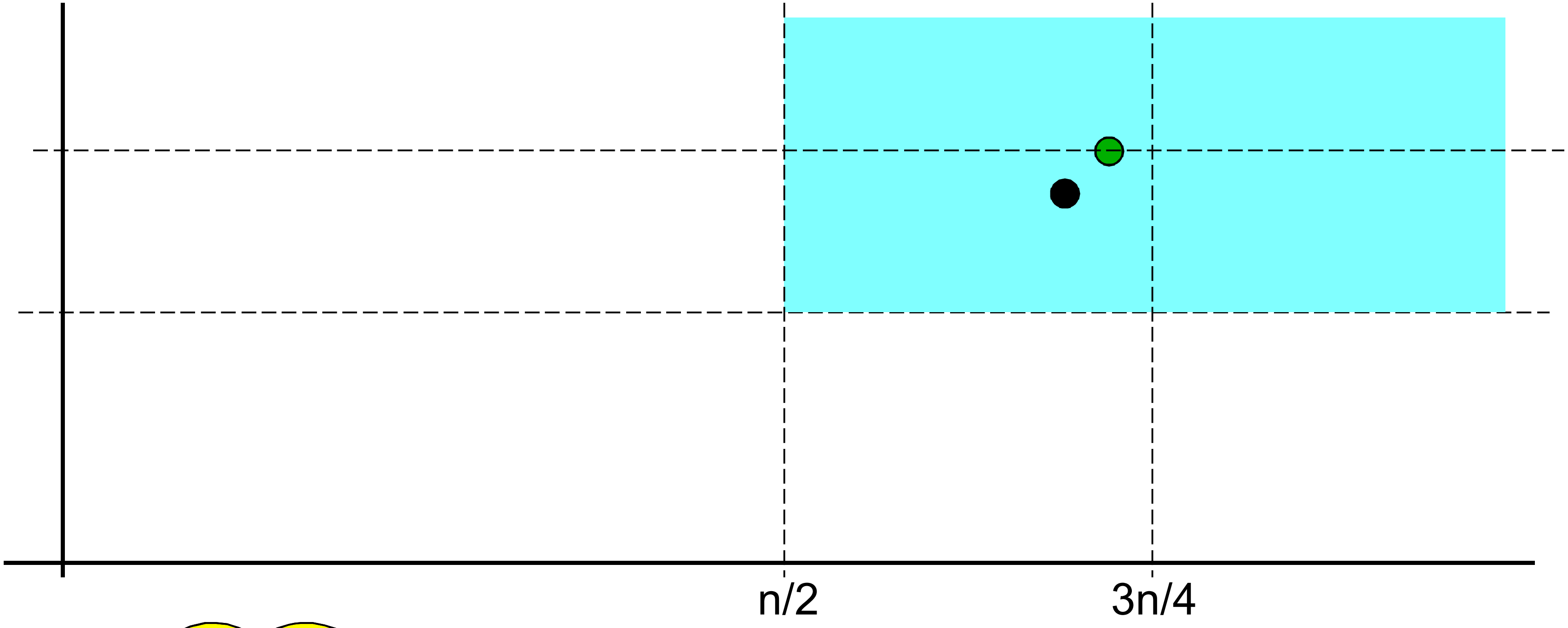
LIS is in blue region

n/2

Where is the splitter?

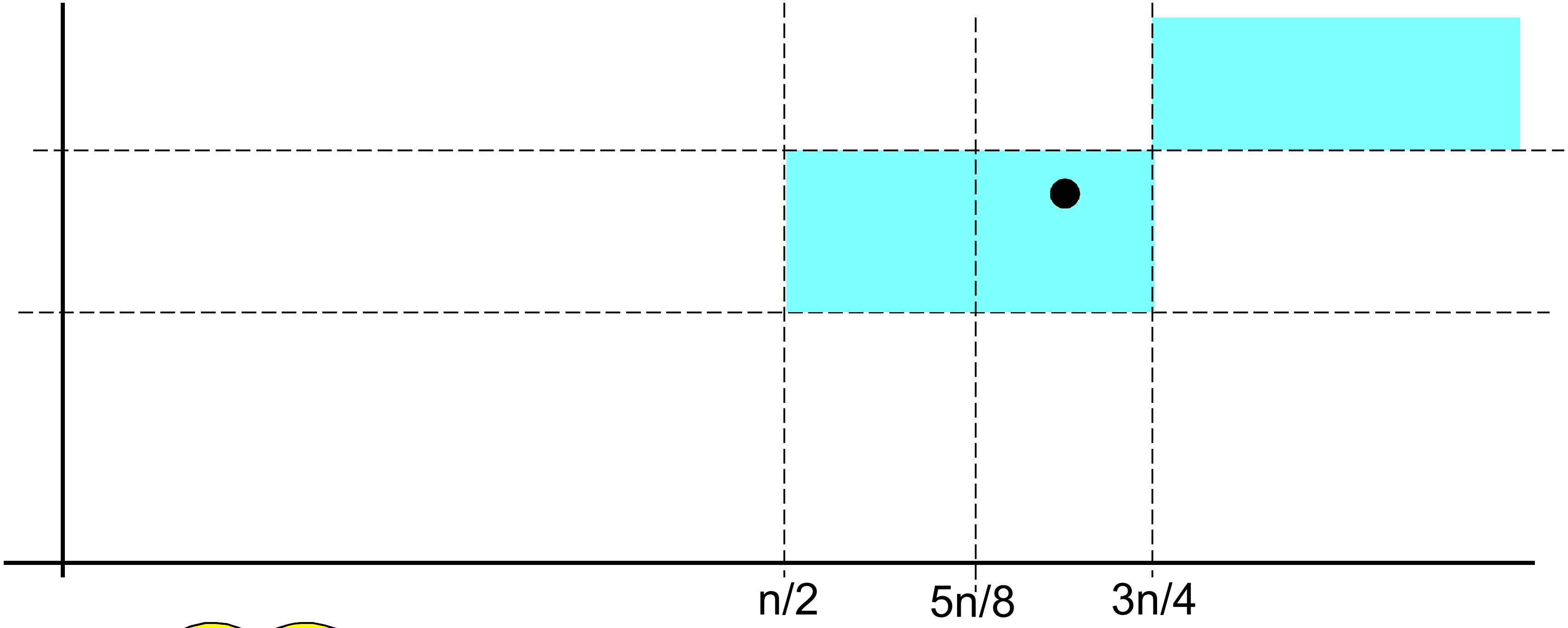
It is there.

# The IP



I wish we knew the splitter in that region

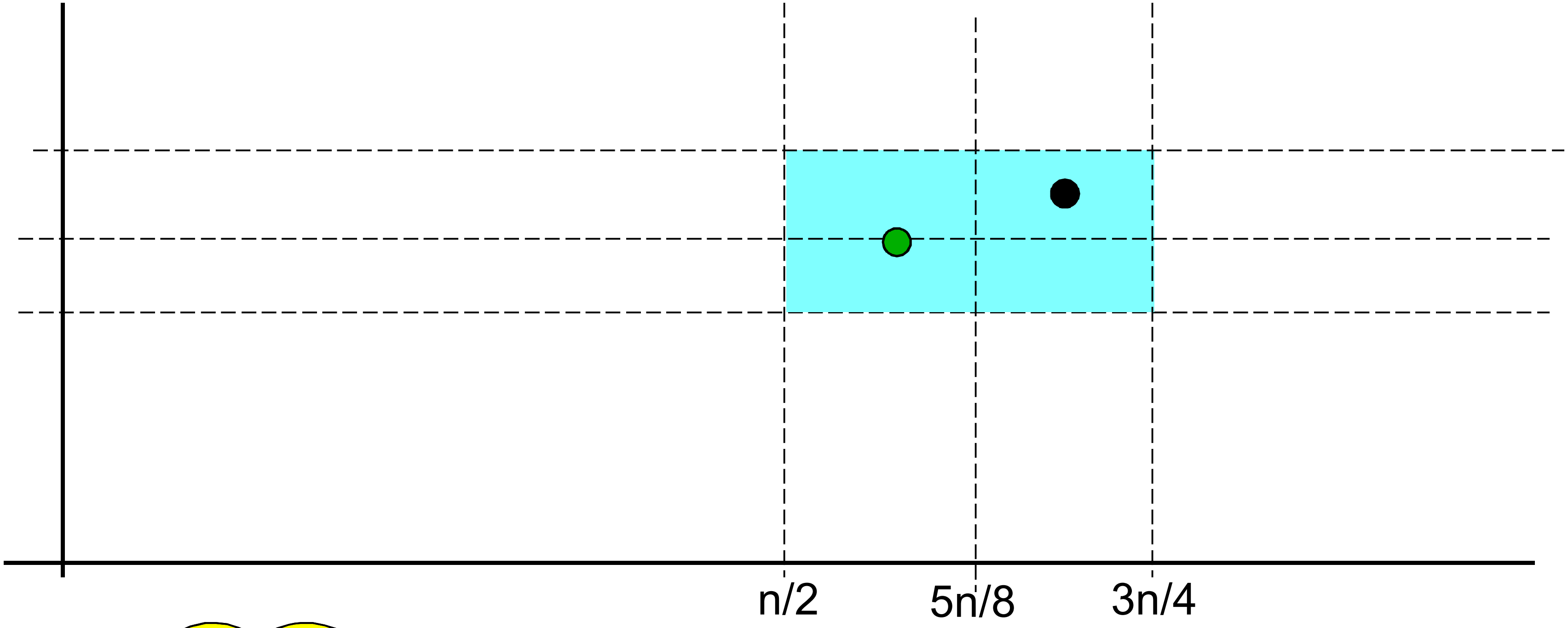
# The IP



I think I know what will happen next

You’re lucky I’m here

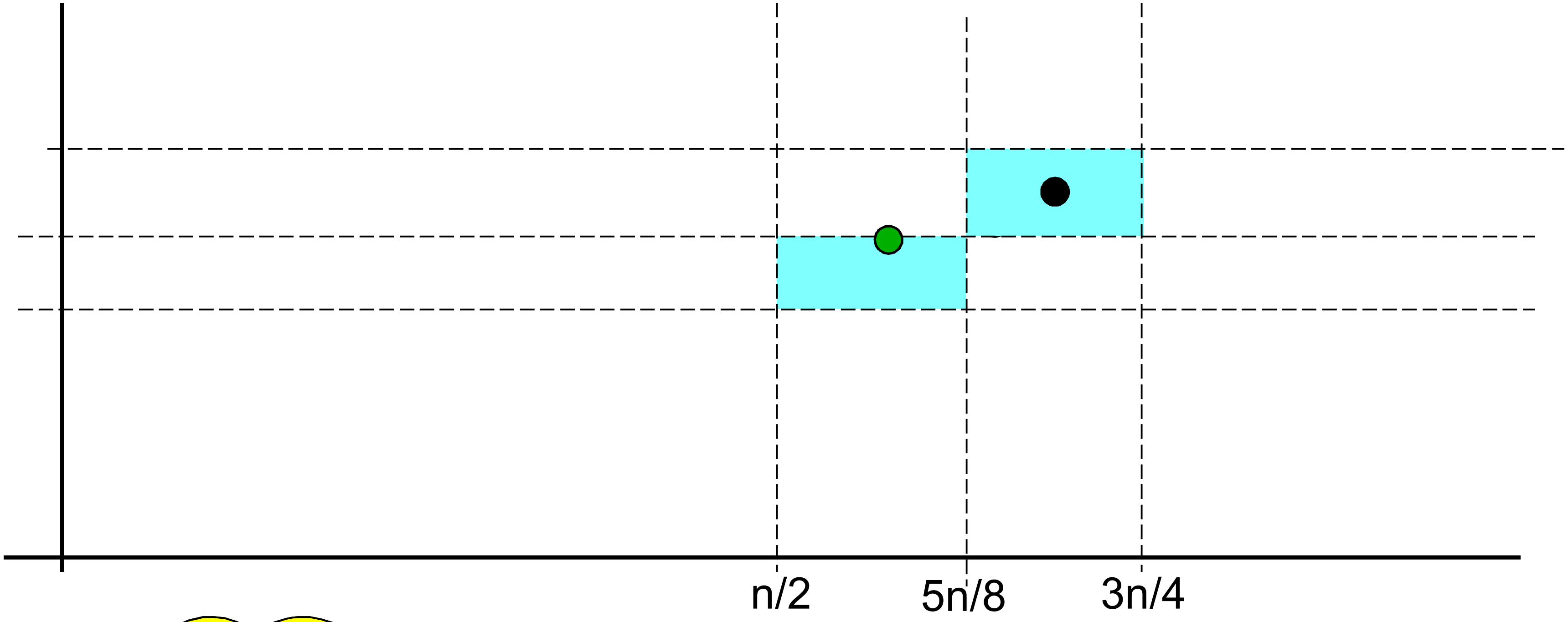
# The IP



I think I know what will happen next

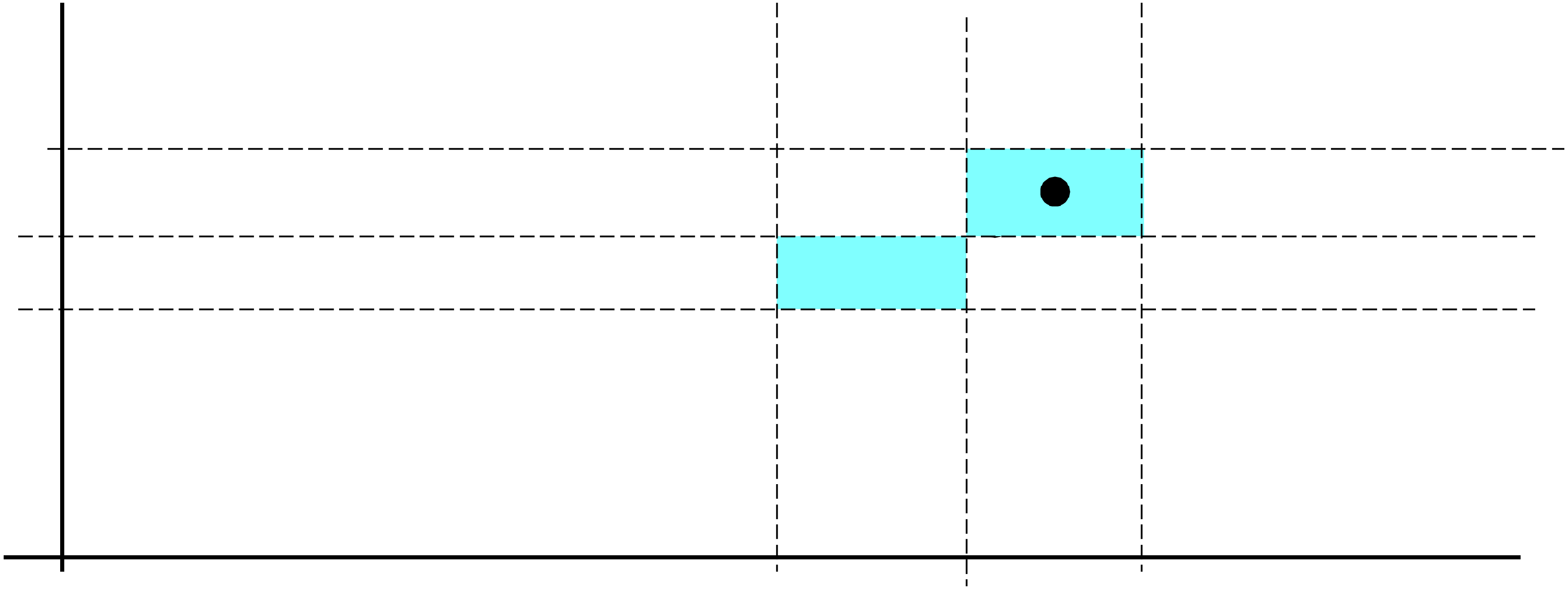
You’re lucky I’m here

# The IP



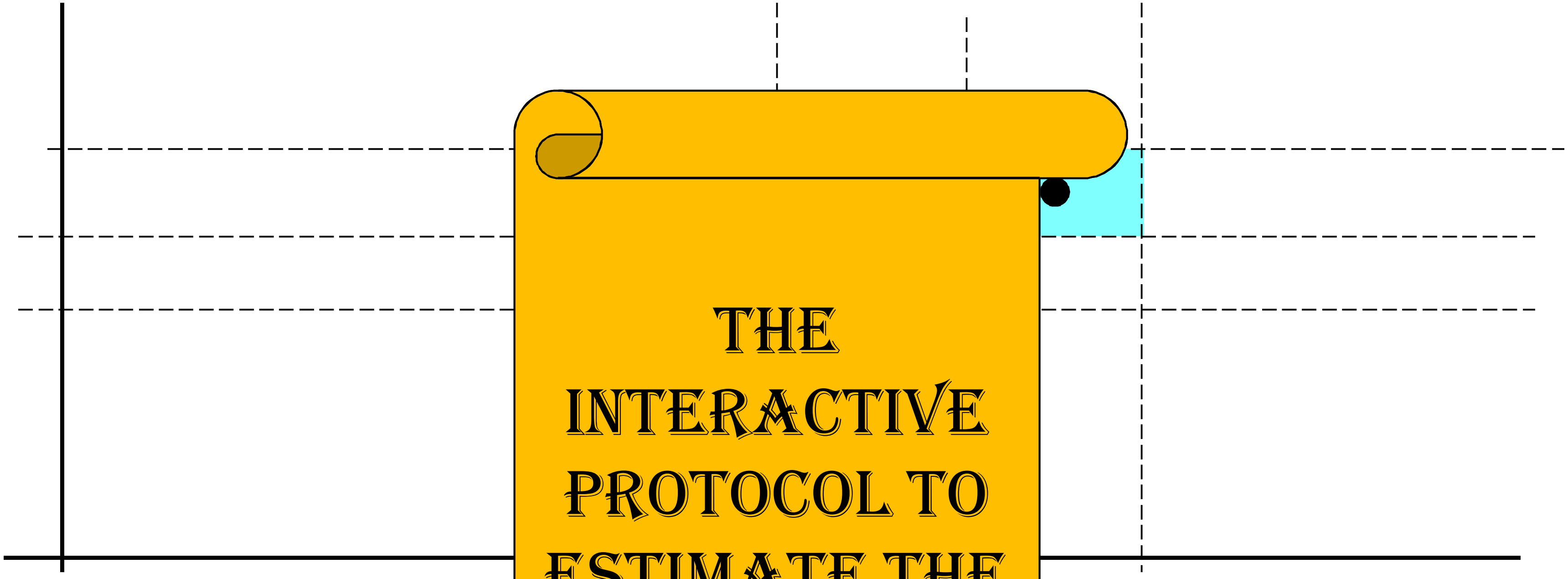
I think I know what will happen next

# The interactive protocol



- If point stays in blue region till very end, then it is good (on LIS). Otherwise, bad.
- This takes (log n) steps, with the help of the wizard

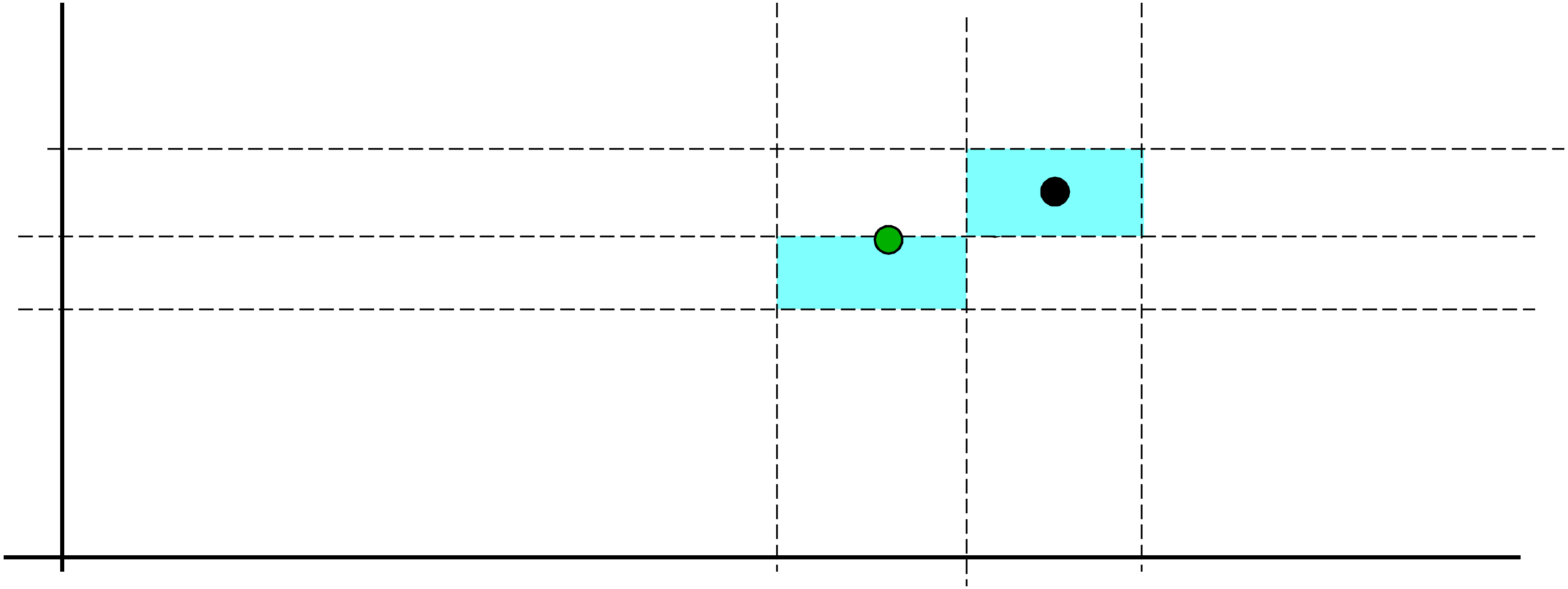
# The interactive protocol



THE INTERACTIVE PROTOCOL TO ESTIMATE THE LIS

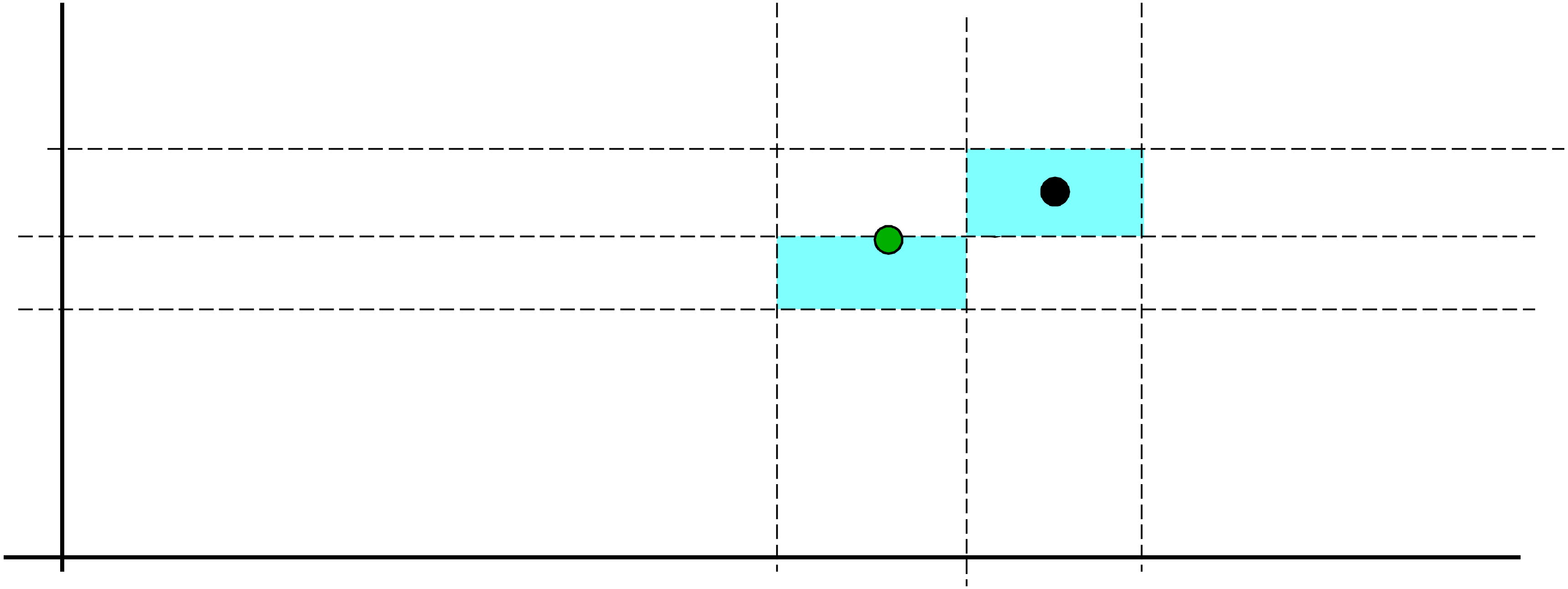
- If point stays in blu nd, then it is good (on LIS). Oth
- This takes (log p of the wizard

# The interactive protocol



- If point stays in blue region till very end, then it is good (on LIS). Otherwise, bad.
- This takes (log n) steps, with the help of the wizard
- If we could simulate the wizard…

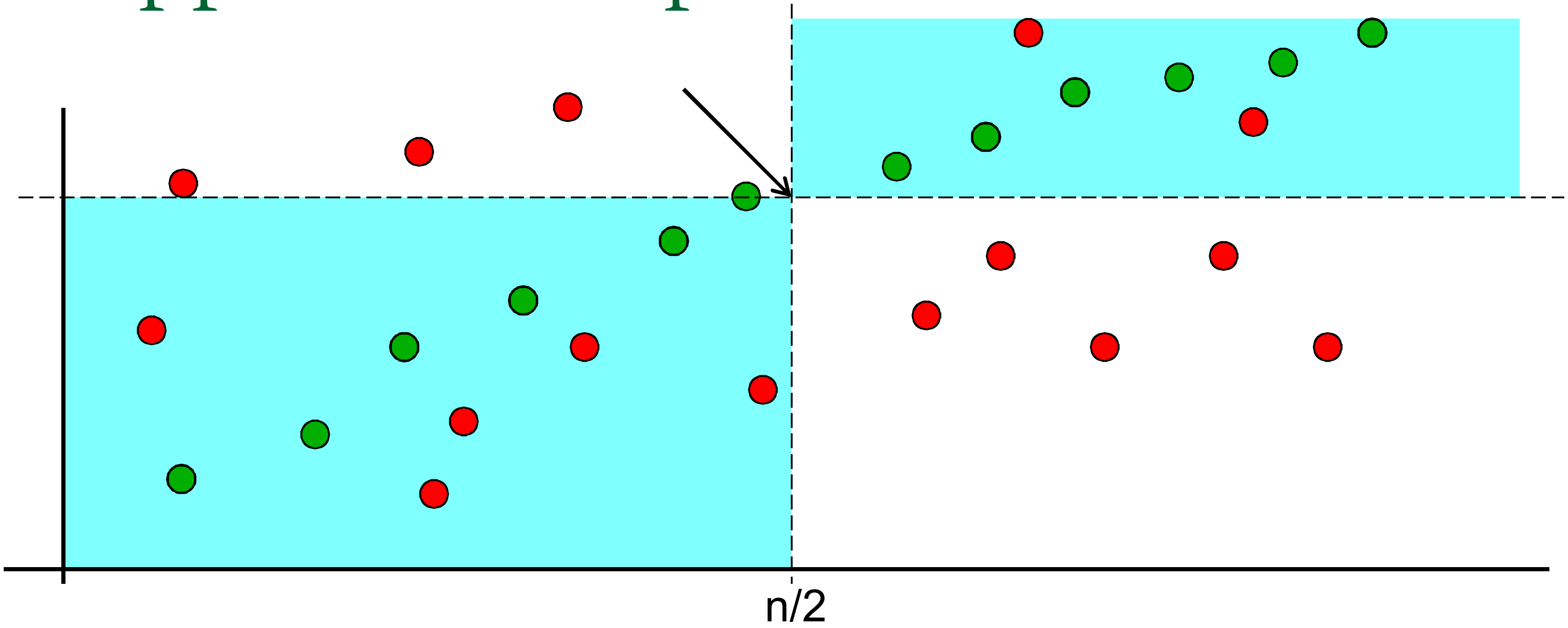
# The interactive protocol



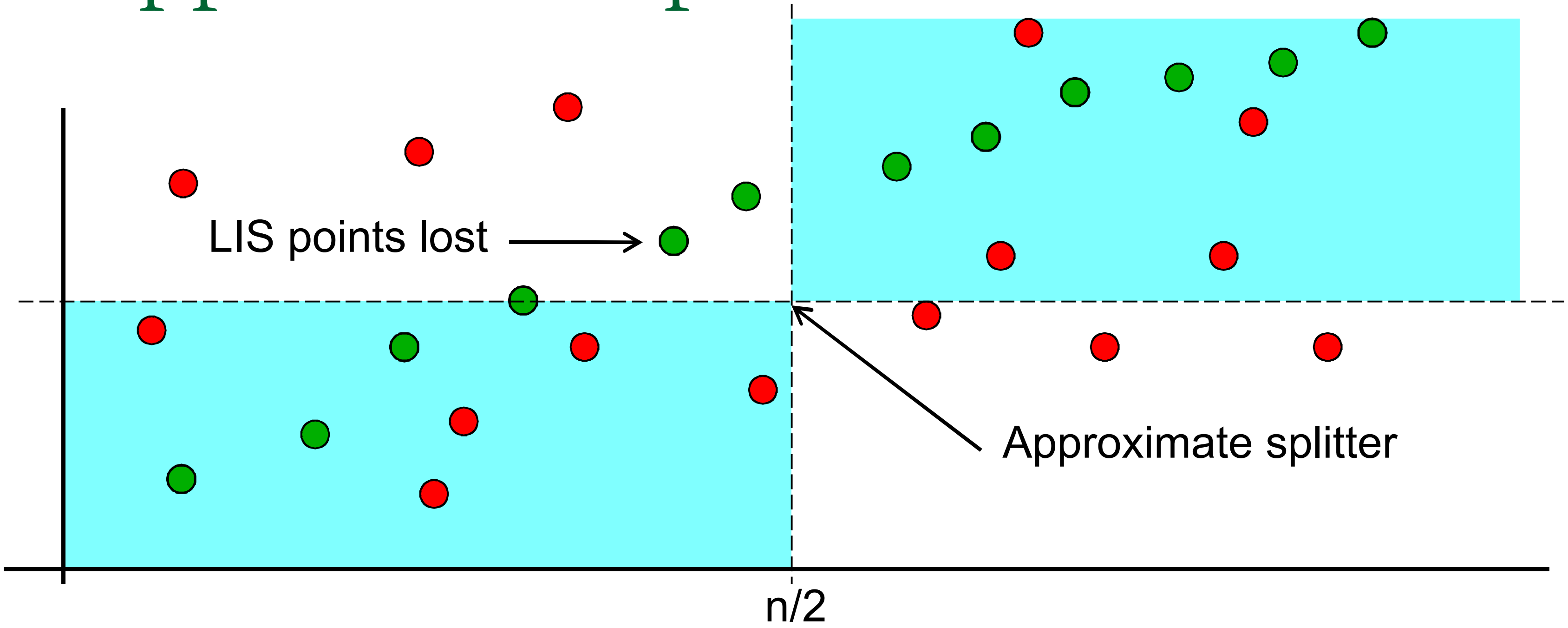
- If point stays in blue region till very end, then it is good (on LIS). Otherwise, ba
- This takes (log n) s izard
- If we could simu

What?? If you could simulate the wizard, you know the LIS!

# An approximate splitter

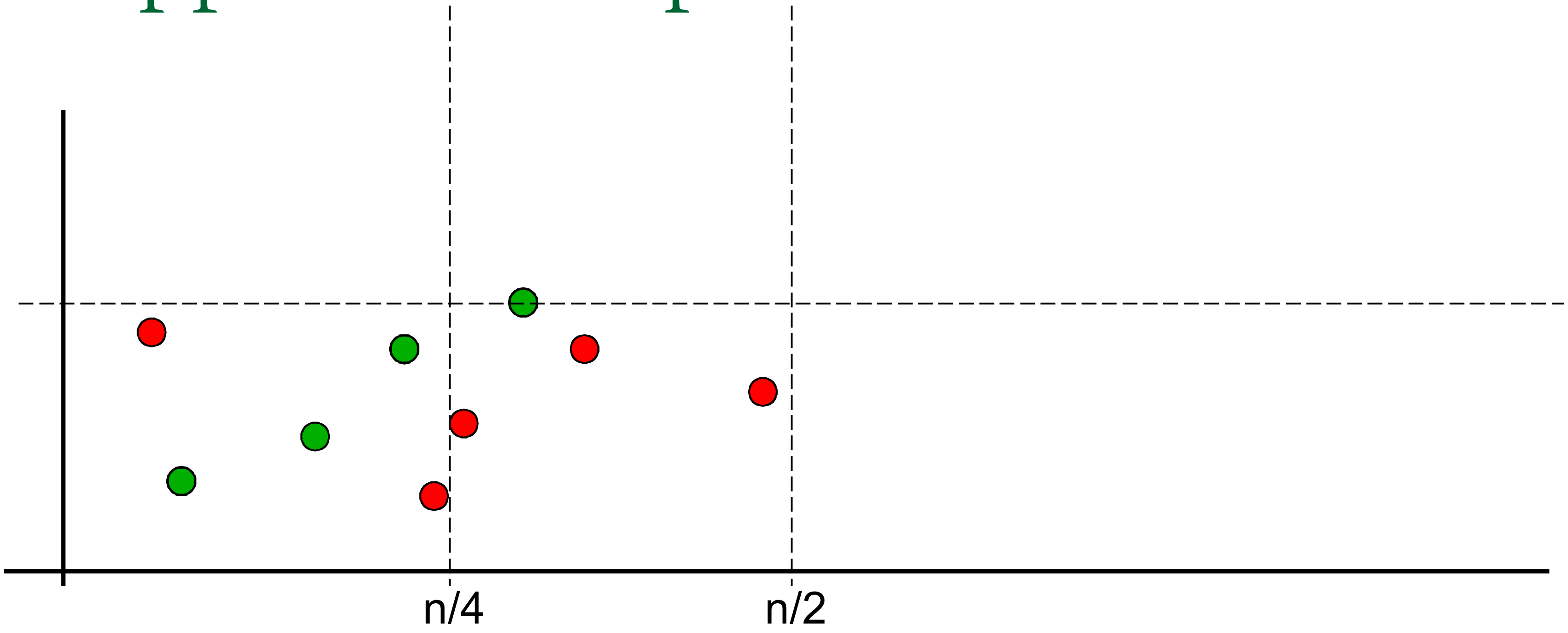


# An approximate splitter

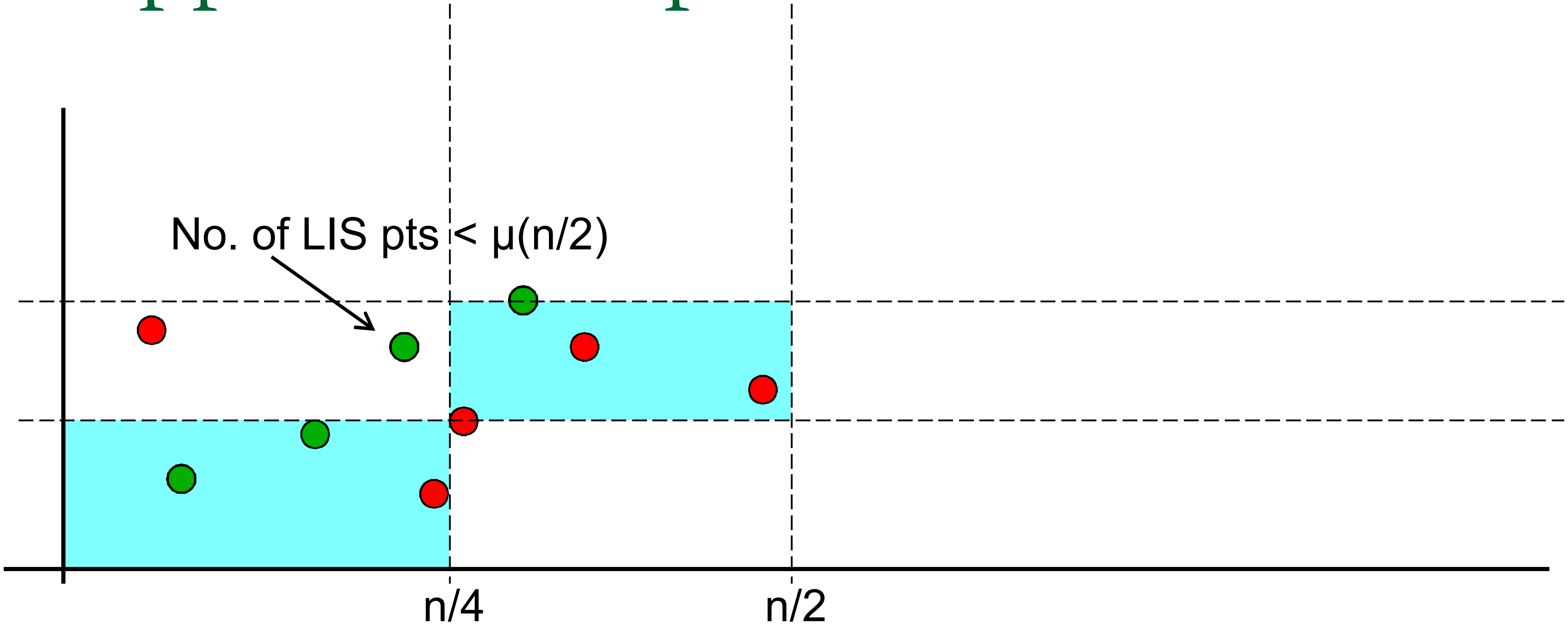


- No. of LIS points lost < μn (violations with splitter)

# An approximate splitter

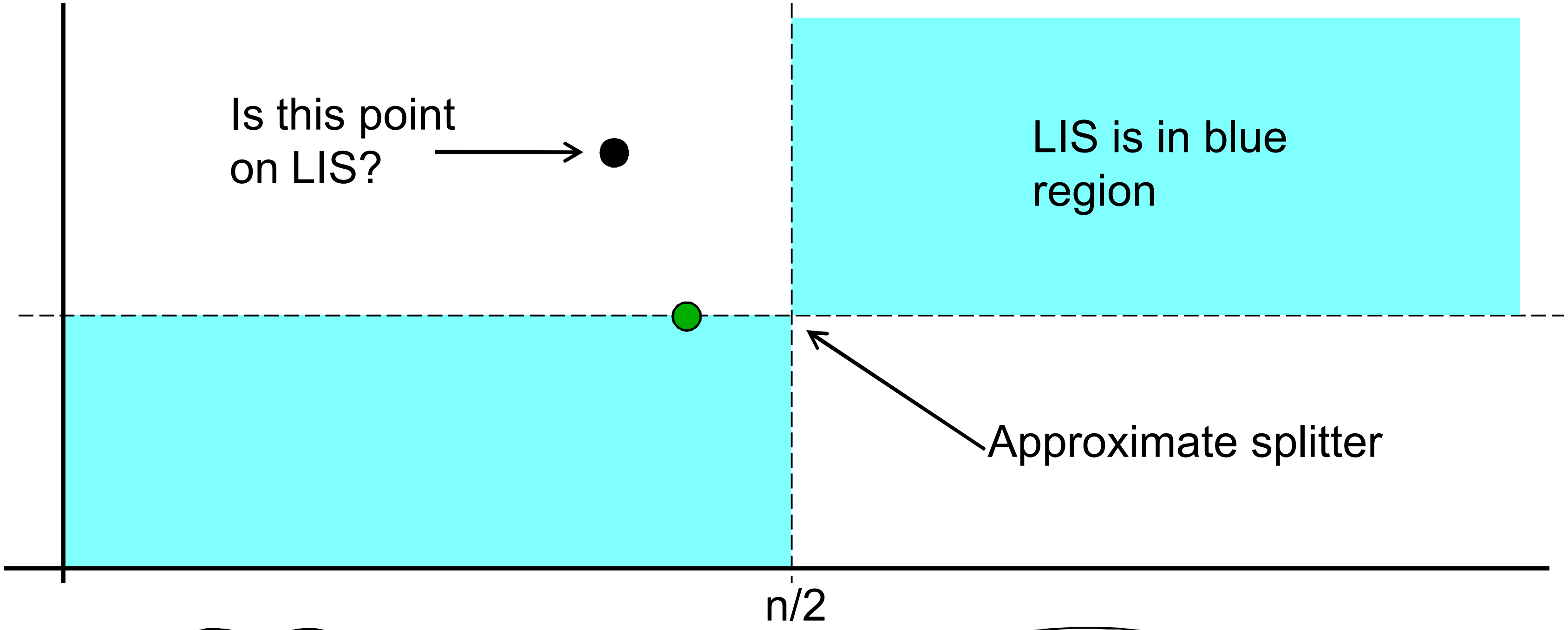


# An approximate splitter



- In interval of size k, we lose μk LIS points
- Total loss = μn log n
  - Set μ = 1/(100 log n), then total loss is n/100 (1% of points)

# The interactive protocol



Is this point on LIS?

LIS is in blue region

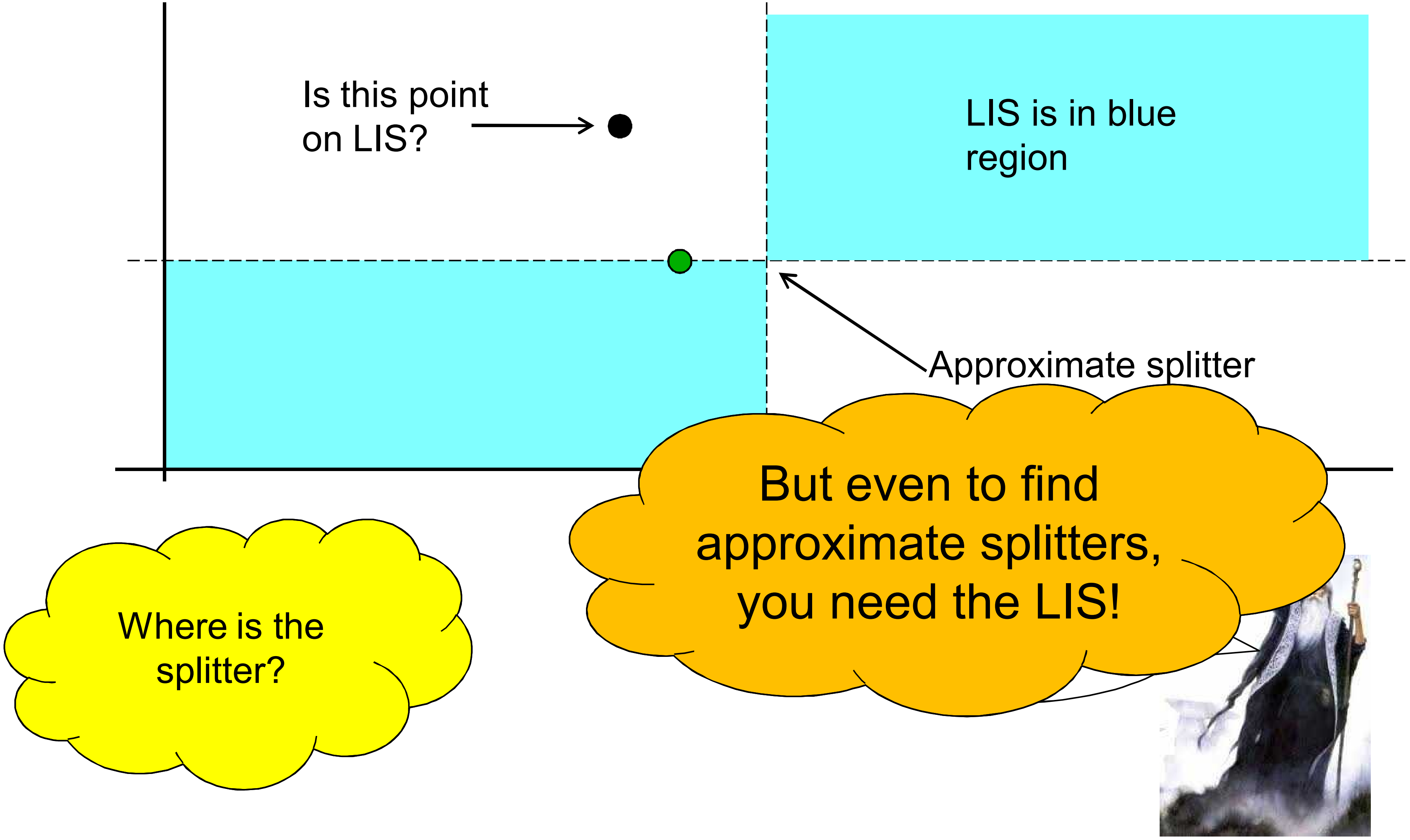
Approximate splitter

n/2

Where is the splitter?

Here is an approximate splitter

# The interactive protocol



Is this point on LIS?

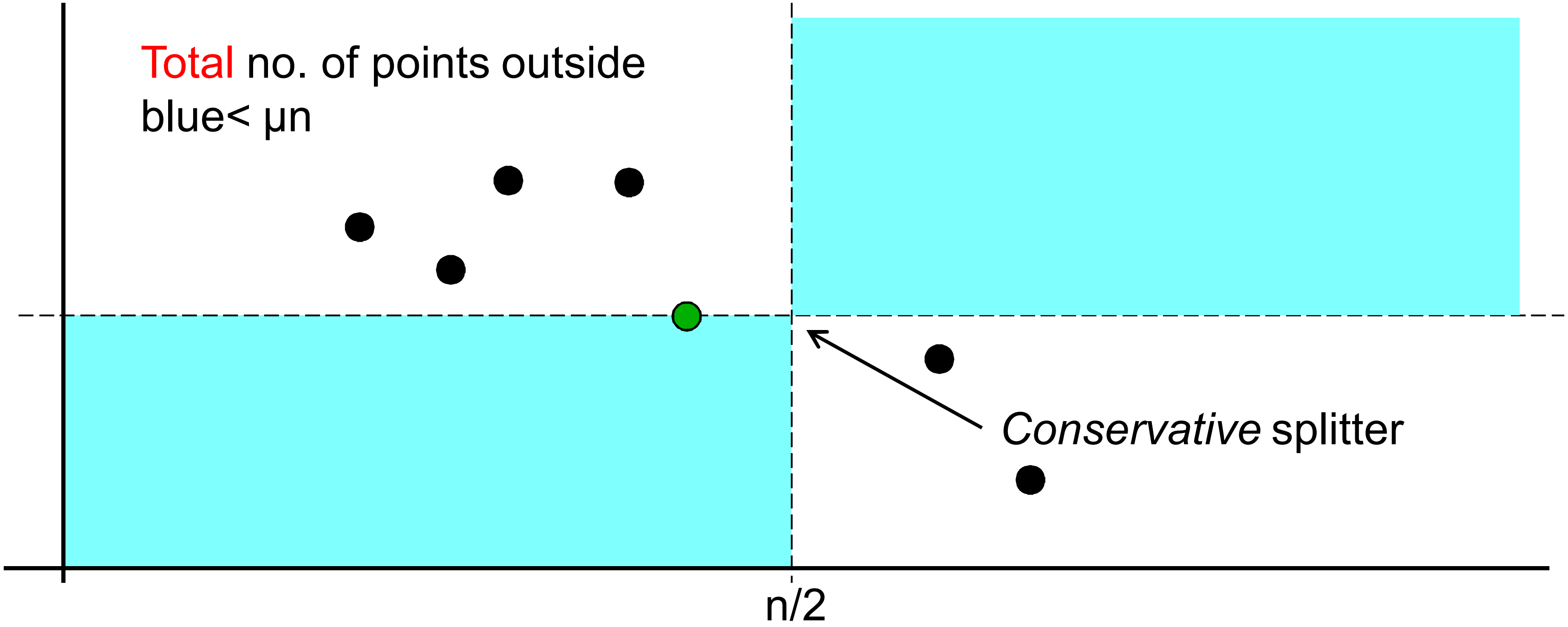
LIS is in blue region

Approximate splitter

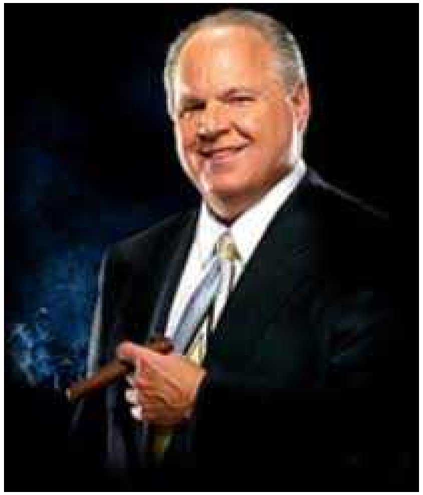
Where is the splitter?

But even to find approximate splitters, you need the LIS!

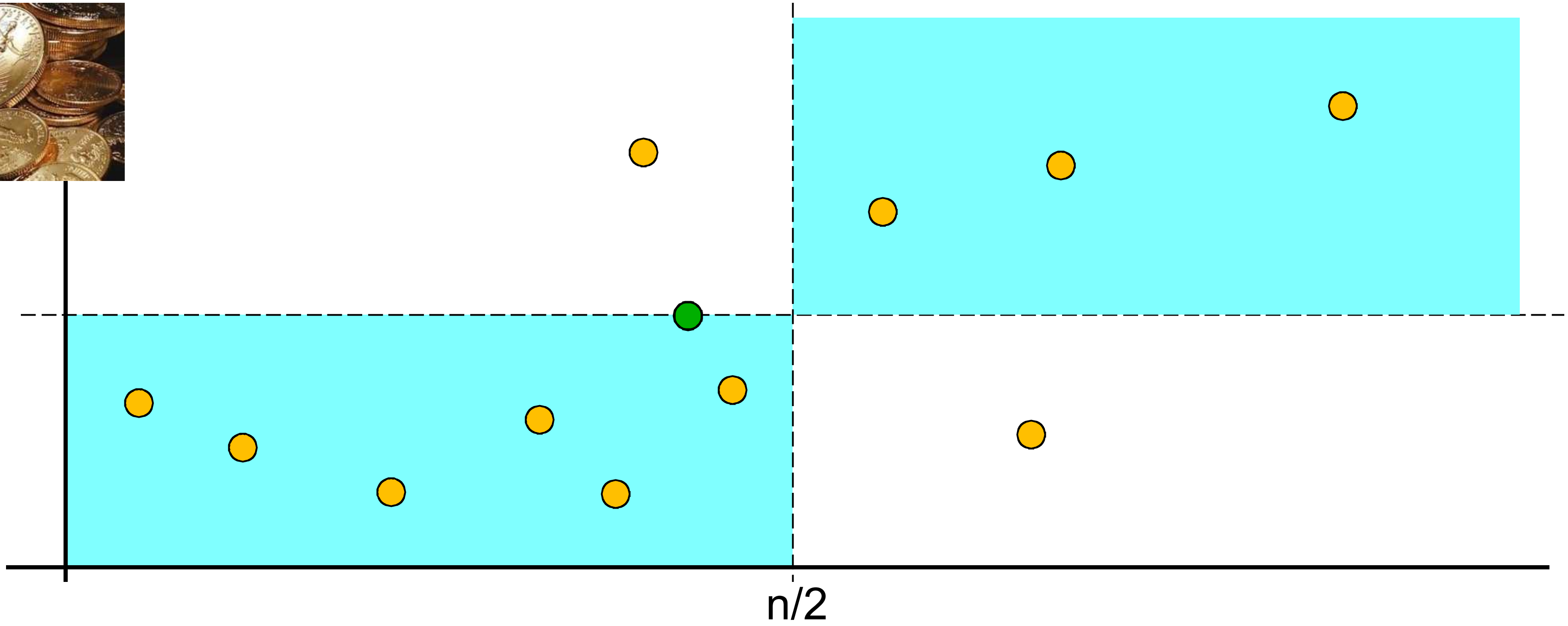
# Be conservative!



- Conservative splitter is definitely approximate splitter
- How to check whether point gives conservative splitter?

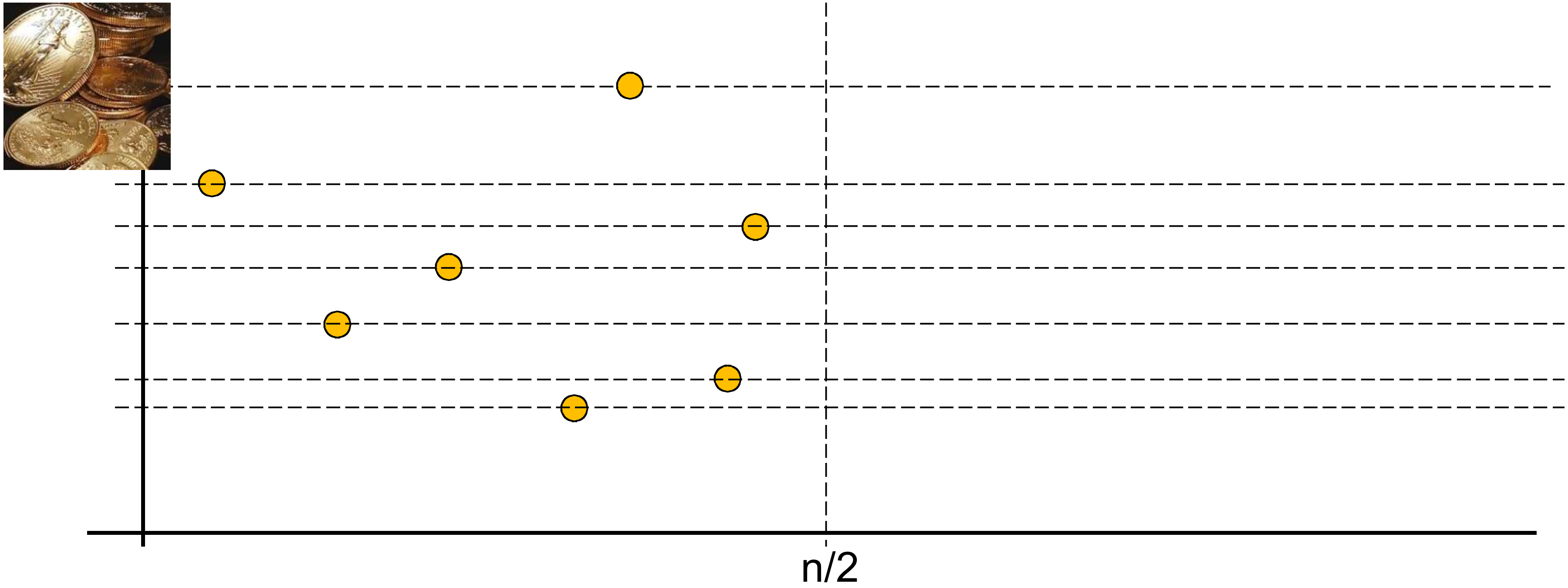


# Be conservative!



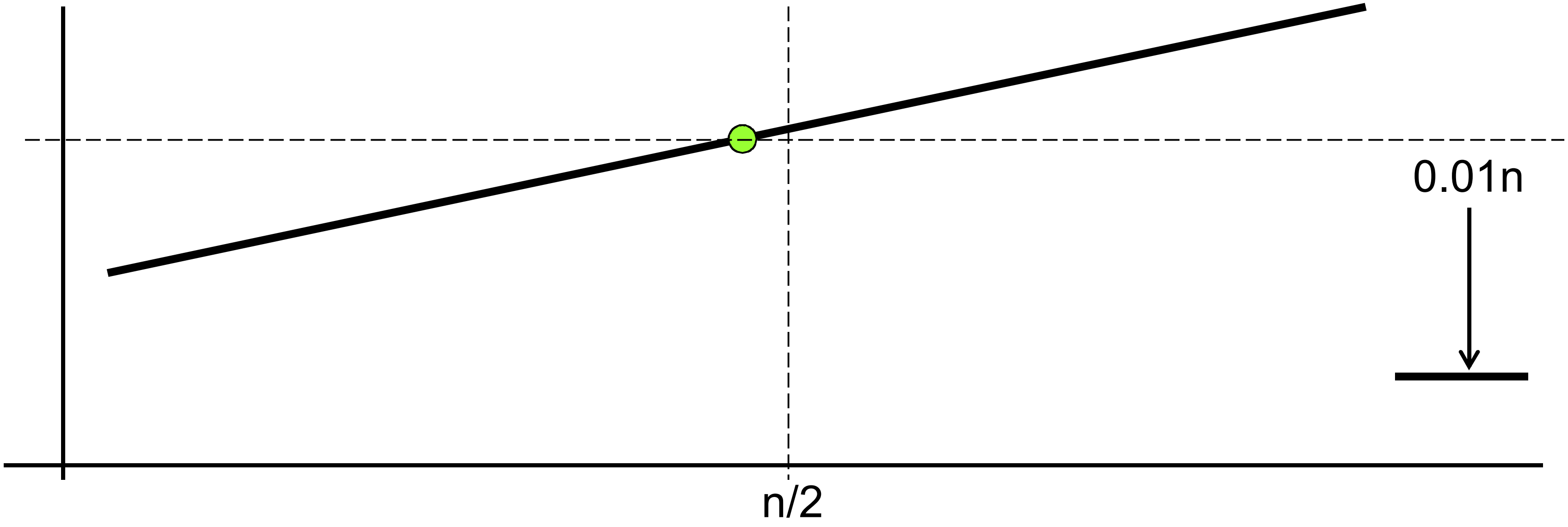
- Count fraction of sample outside blue
- Because μ = 1/(100 log n), poly(log n) samples checks this accurately
- With this, we can run interactive protocol

# Getting a conservative splitter



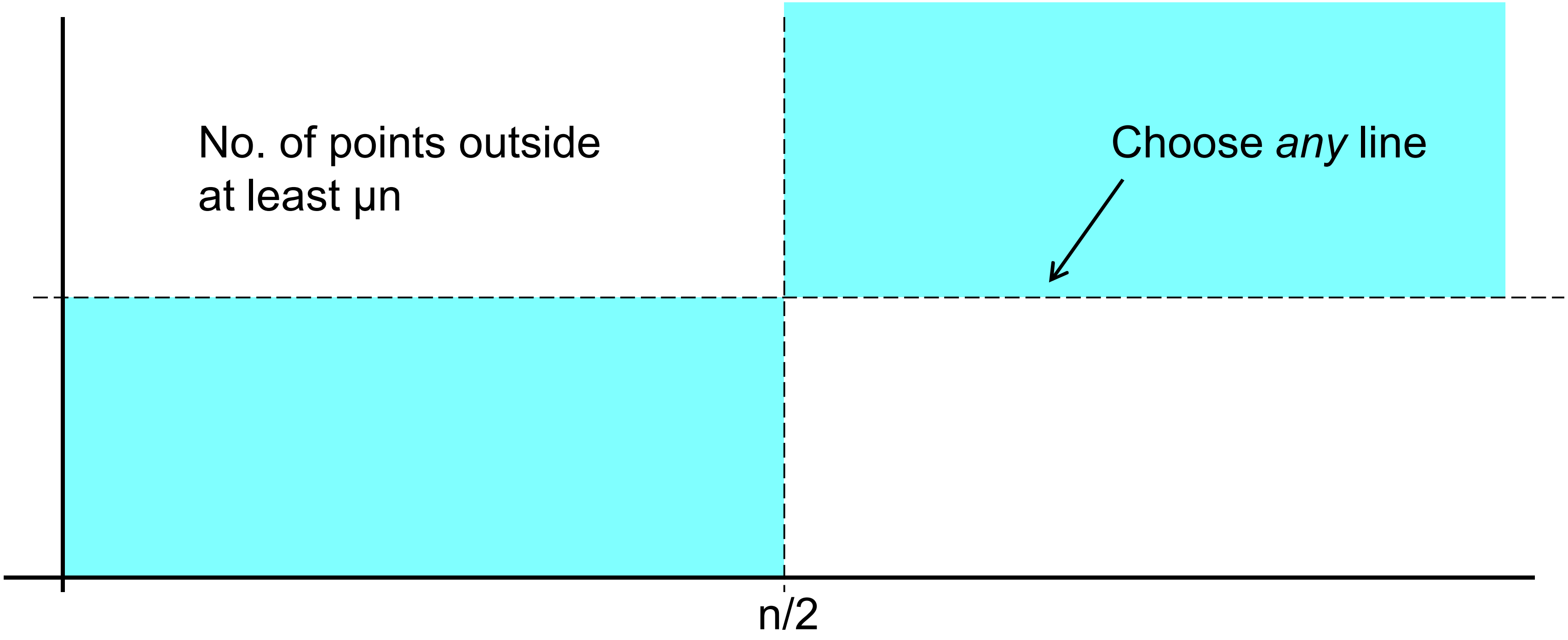
- We can sample (log n) different candidates and check all of them
- ~~You might miss a conservative splitter…~~
- What if no conservative splitter exists?

# No splitters



- Every point is violation with at least n/100 points
  - No conservative splitter

# A liberal paradise

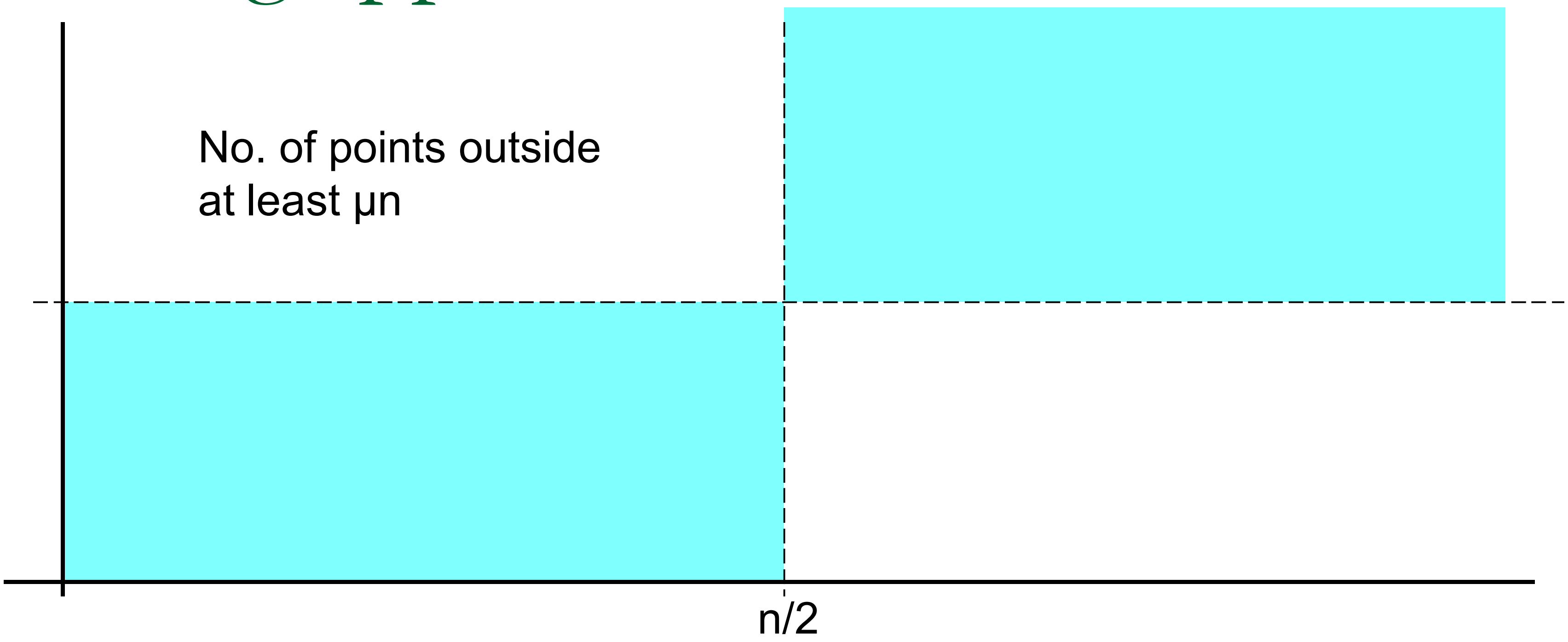


- So we know that |LIS| < (1-μ) n
- Leads to the next idea. Boosting approximations!

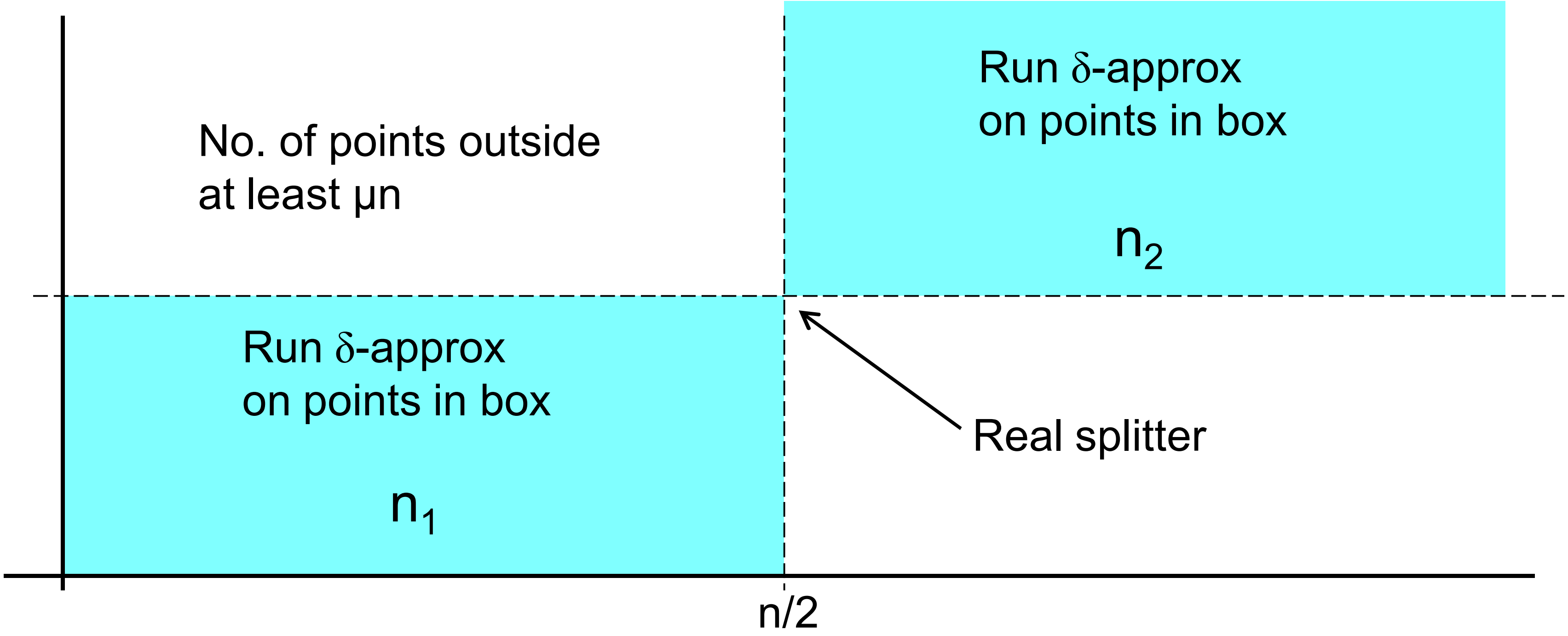
# Can we beat the n/2-approx??

- We have an n/2-approx for |LIS|
- Can we get *anything* better than that?
- Maybe we can somehow convert a δn-approx to δ□n-approx (δ > δ□)
    - Call this a δ-approx algorithm

# Boosting approximations

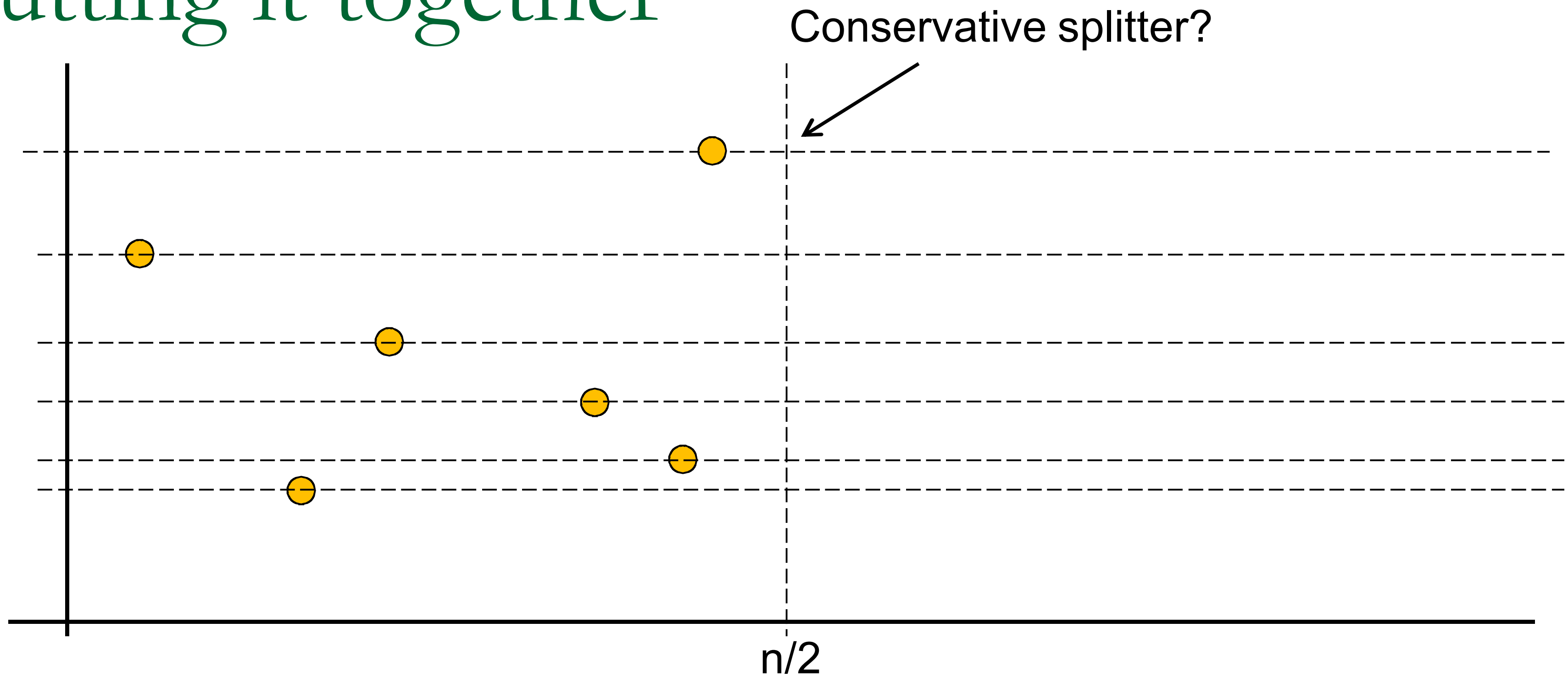


# Boosting approximations



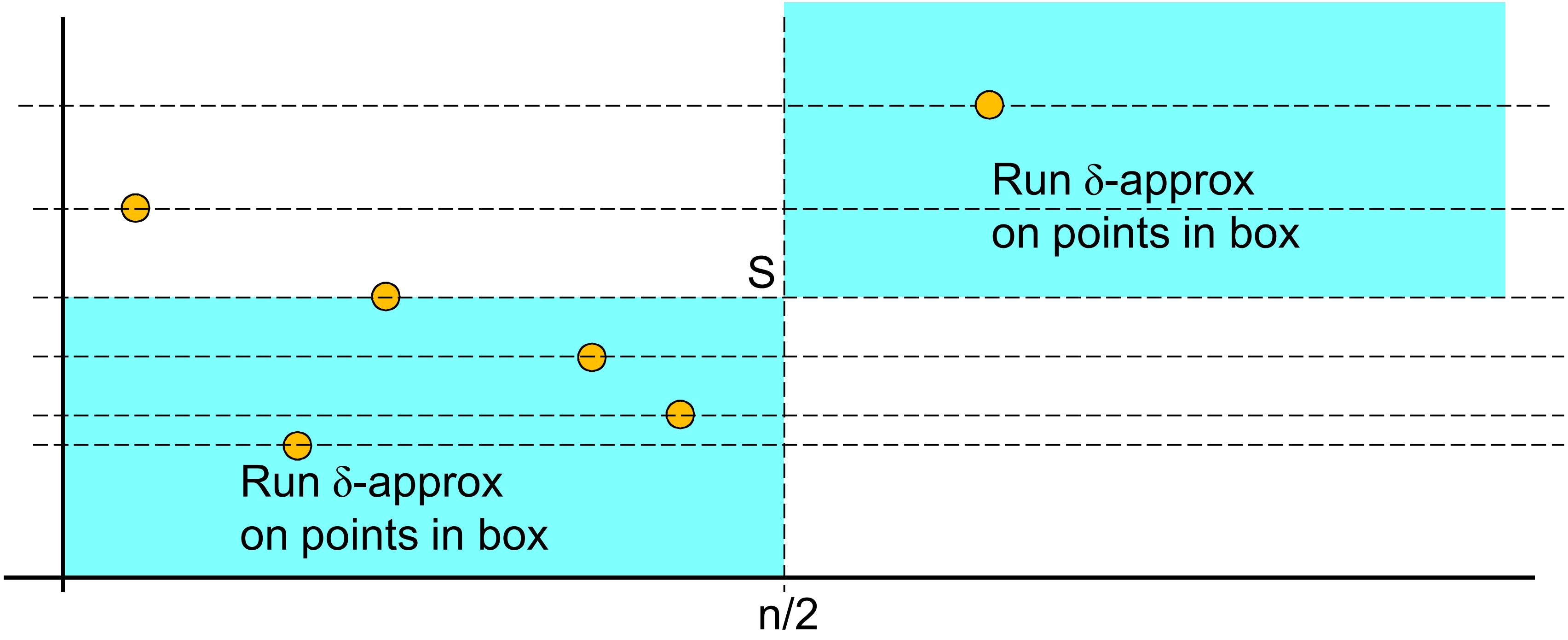
- Take sum of outputs as total LIS estimate
- $|LIS| = |LIS_1| + |LIS_2|$, $Est = Est_1 + Est_2$
- $|Est_1 – LIS_1| < \delta n_1$ $|Est_2 – LIS_2| < \delta n_2$
- So $|Est – LIS| < \delta(n_1 + n_2)$
- $n_1+n_2 < (1-\mu)n$, so $|Est – LIS| < \delta(1-\mu)n$ !

# Putting it together



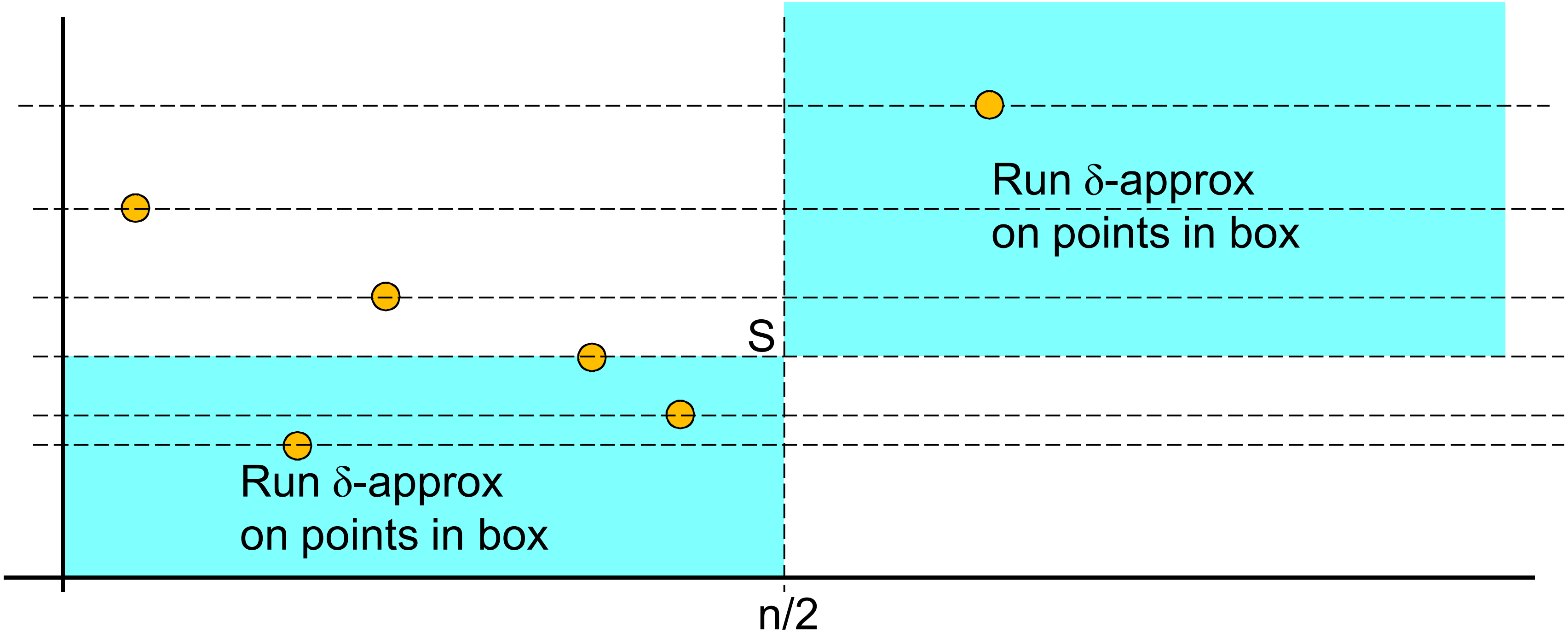
- Check if each is conservative splitter
  - If it is, we're all set to run IP
- Otherwise…

# Putting it together



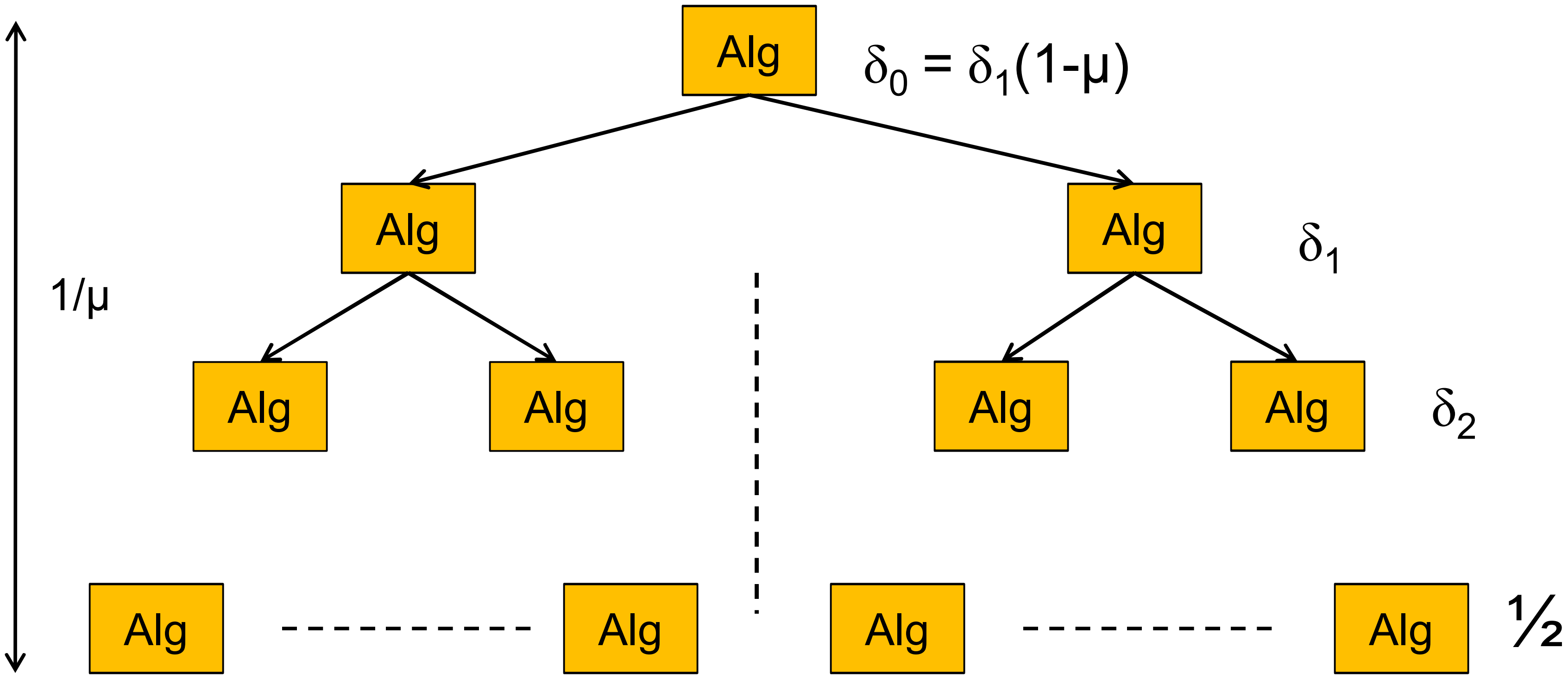
- One of these is “close enough” to real splitter
- Est(S) = Left-Est(S) + Right-Est(S)

# Putting it together



- One of these is “close enough” to real splitter
- Est(S) = Left-Est(S) + Right-Est(S)
- Final Estimate = $\max_S$ Est(S)
- Looks like a great idea!
  - We go from δ to δ(1- μ).  Recurse to keep improving approximation

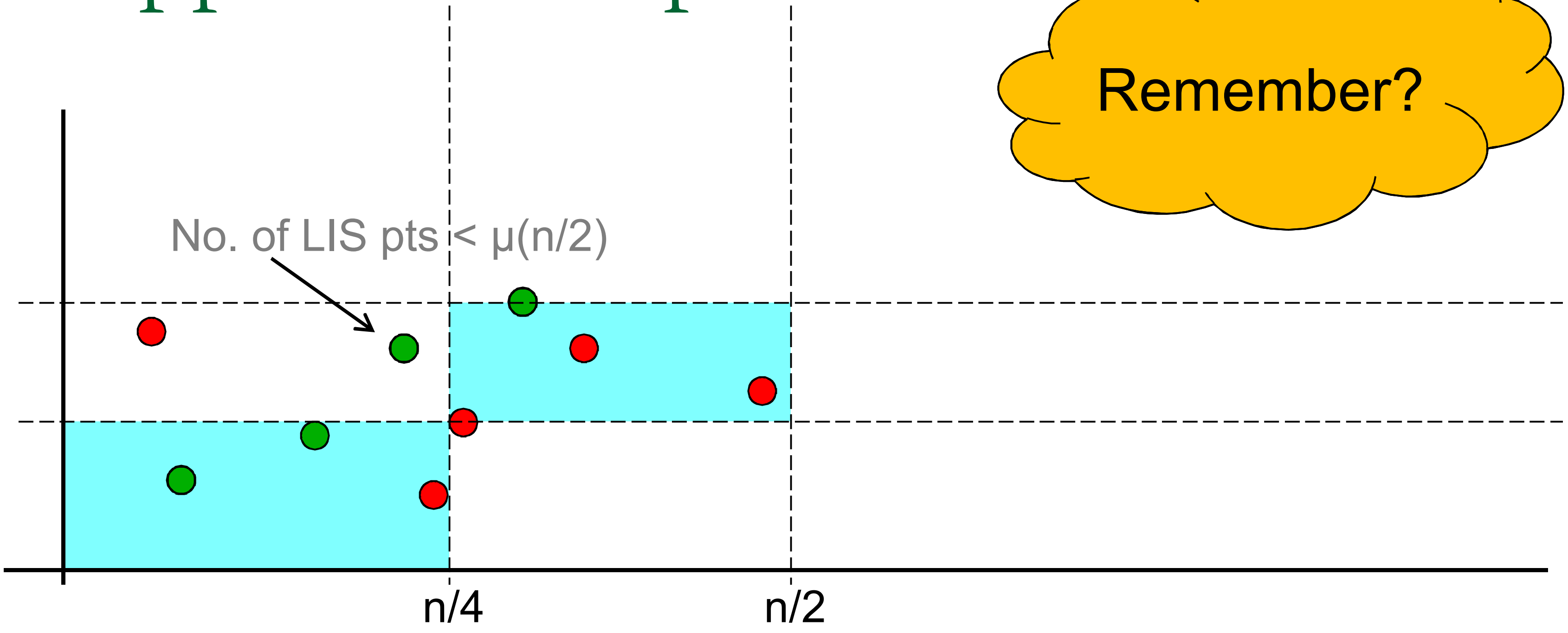
# It fails, miserably



- As we go up each level, approx gets better by (1-μ).
- So to get $\delta_0$ = ¼, how many levels needed?
  - $\frac{1}{4} = \frac{1}{2}(1-\mu)^t$ So $t = 1/\mu$
- We have running time at least $2^{1/\mu}$.

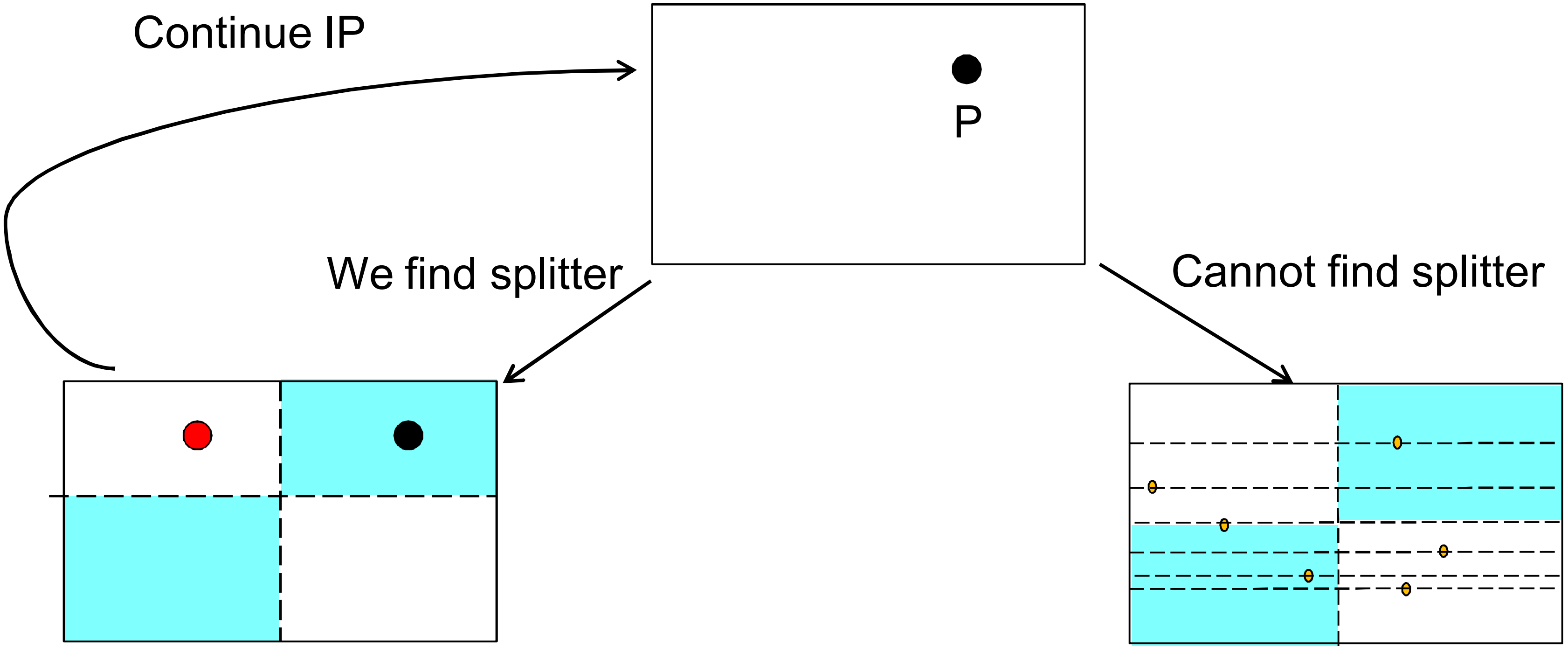
So, μ needs to be > 1/log log n.

# An approximate splitter



- No. of LIS points lost < μn (violations with splitter)
- In interval of size k, we lose μk LIS points
- Total loss = μn log n
    - **Set μ = 1/(100 log n)**, then total loss is n/100 (1% of points)

# The basic dichotomy



Continue IP

P

We find splitter

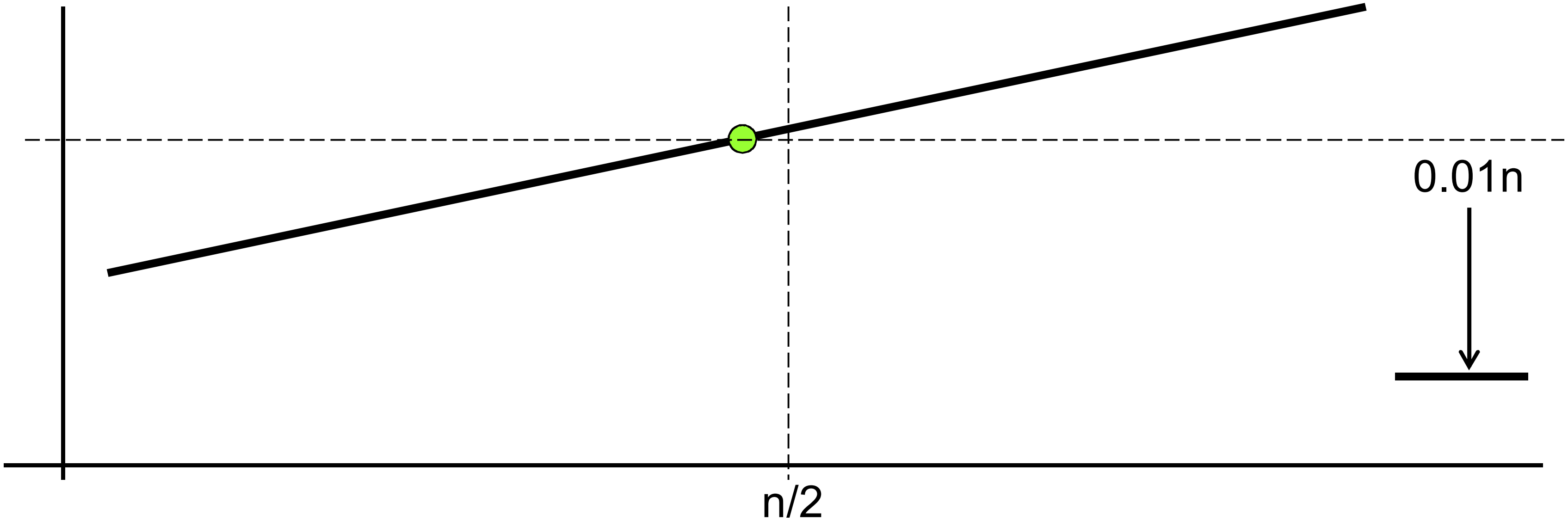
Cannot find splitter

The “Interactive Protocol” phase

The “Dynamic Programming” phase

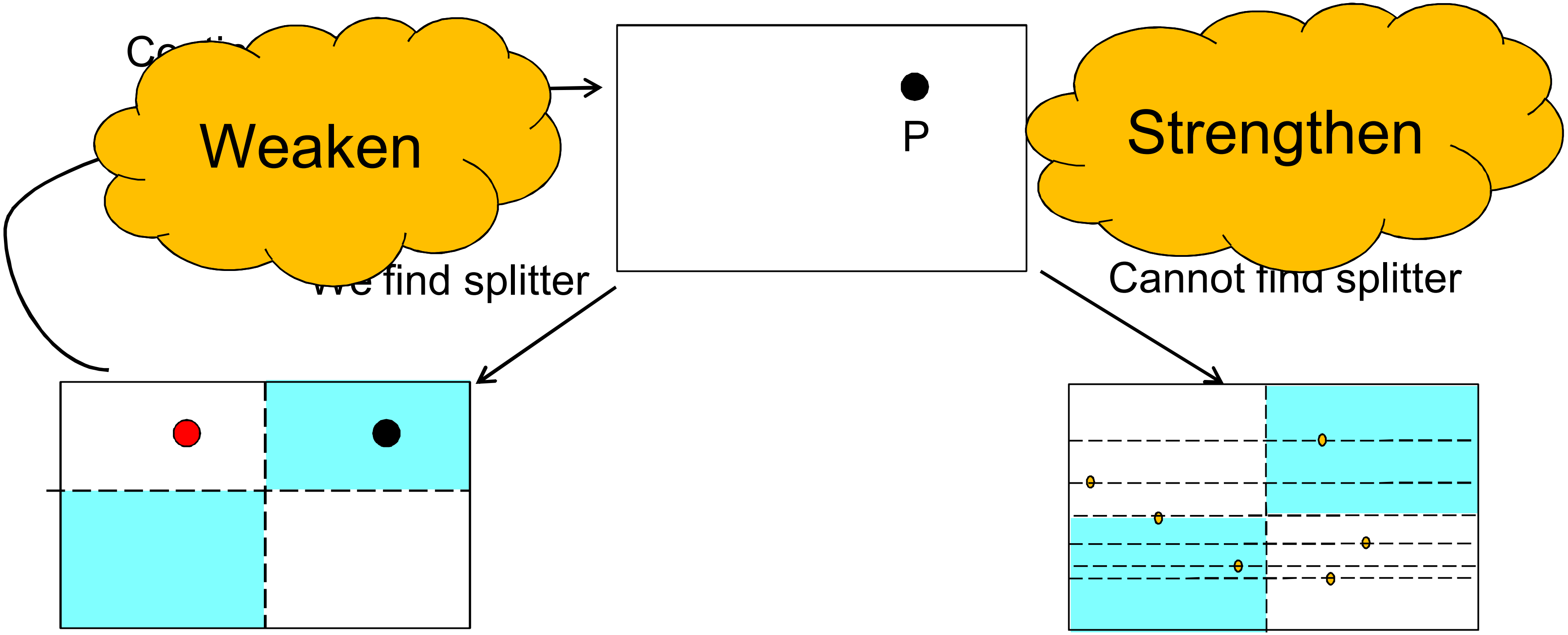
- For IP, we need μ < 1/log n
  - μn is error in each “level” of IP
- For DP, we need μ > 1/log log n
  - (1-μ) is decrease in error

# A weaker splitter



- Every point is violation with at least n/100 points
  - No conservative splitter
- But surely, splitters are easy to find
  - We’re being too conservative

# The basic dichotomy

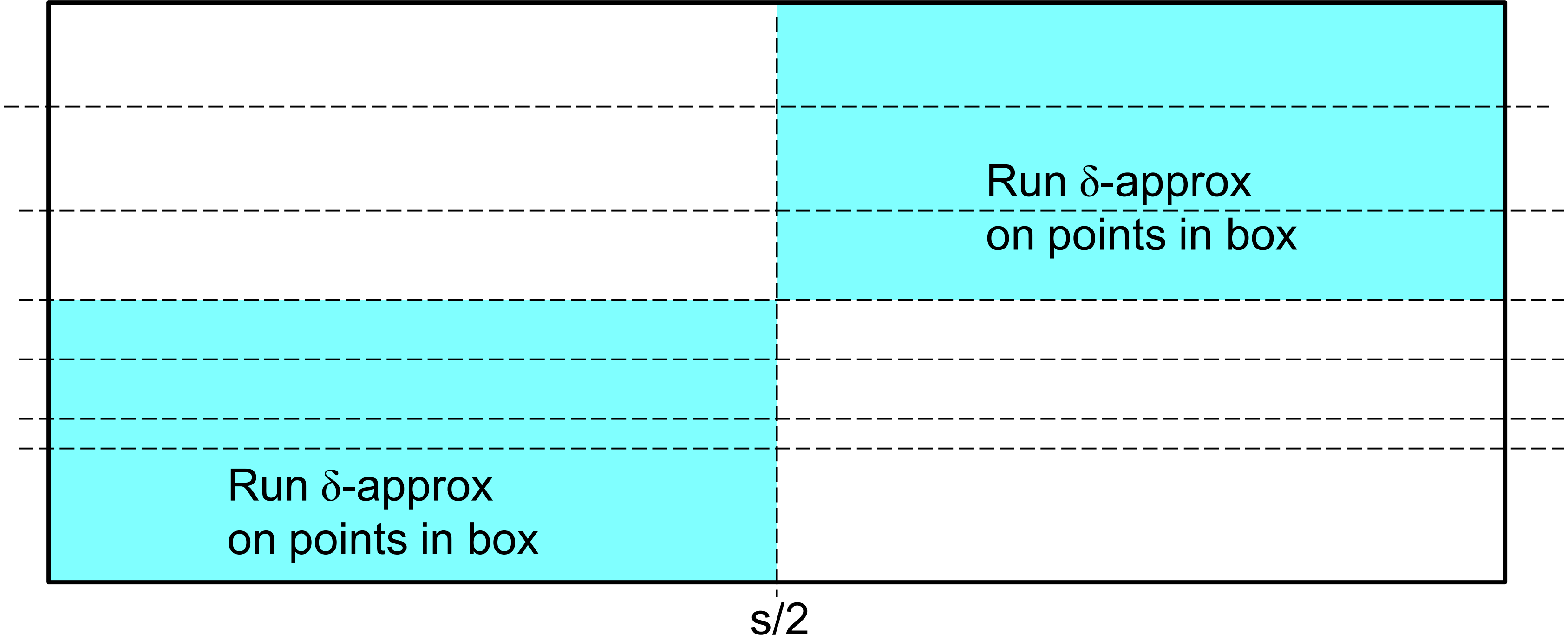


The “Interactive Protocol” phase

The “Dynamic Programming” phase

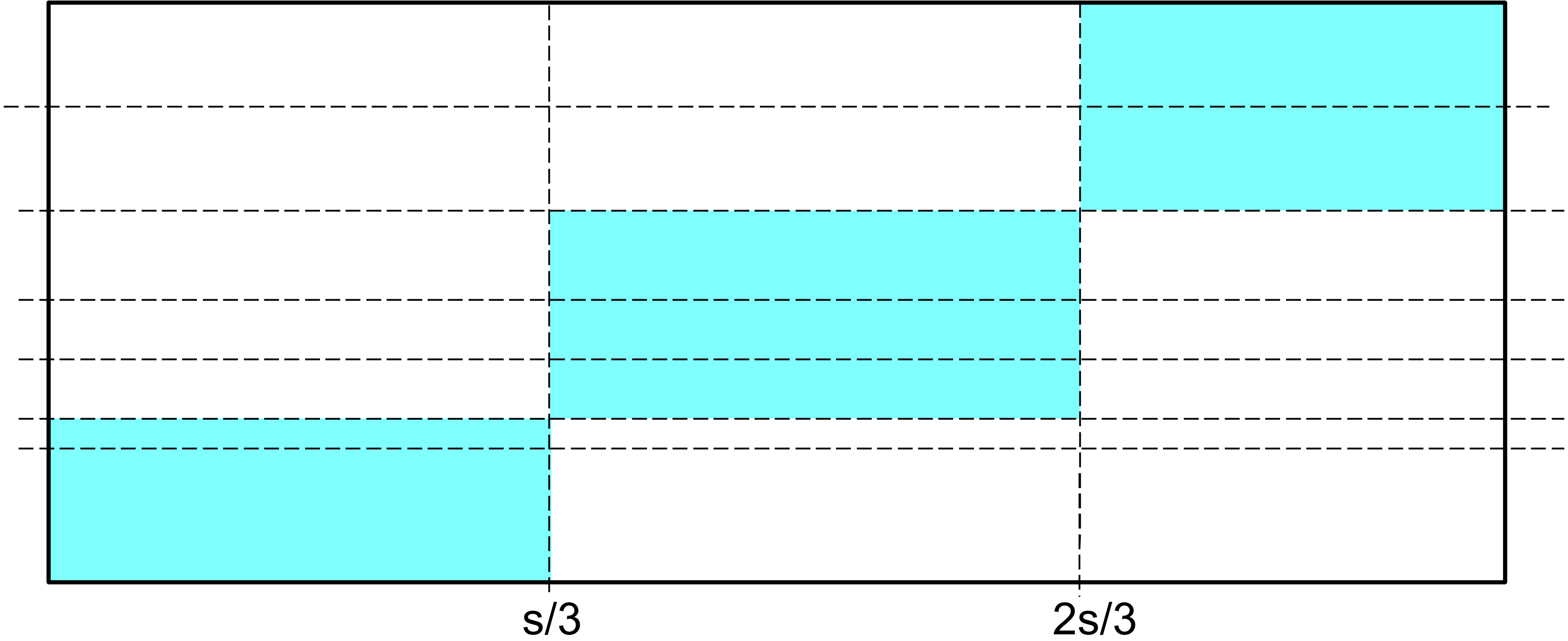
- For IP, we need μ < 1/log n
  - μn is error in each “level” of IP
- For boosting, we need μ > 1/log log n
  - (1-μ) is decrease in error

# Have a look again…



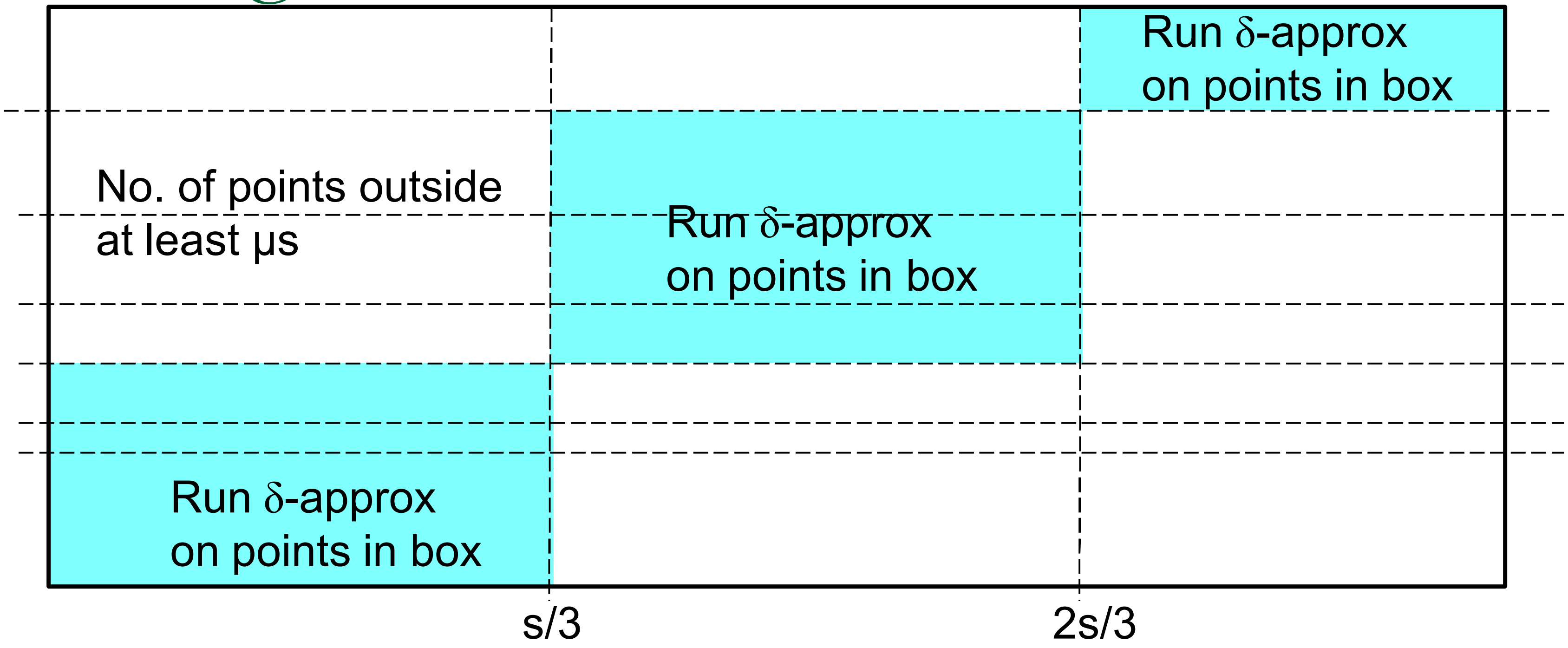
- If no splitter, then only (1-μ)s points in any division
- We take max of all possible sums
  - We get δ(1-μ)-approx

# A small generalization



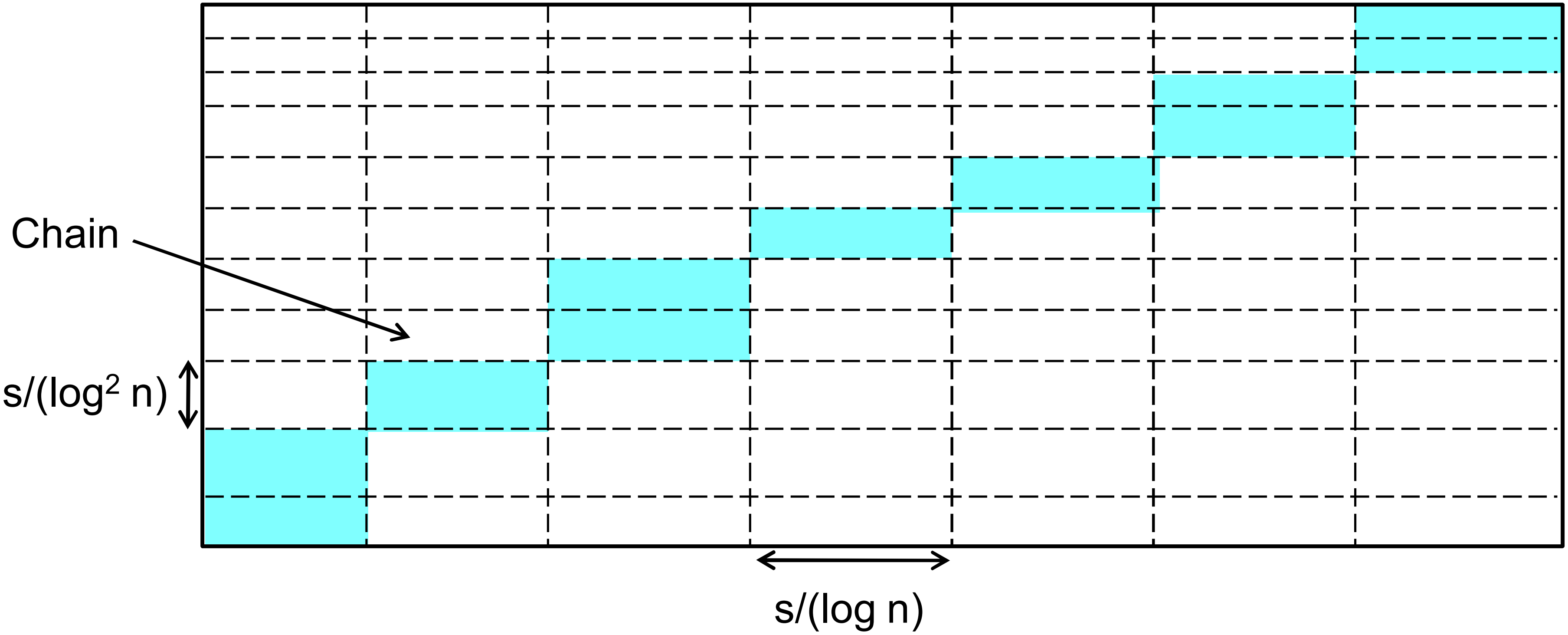
- Suppose every “3-way” division has at most (1-μ)s points

# A small generalization



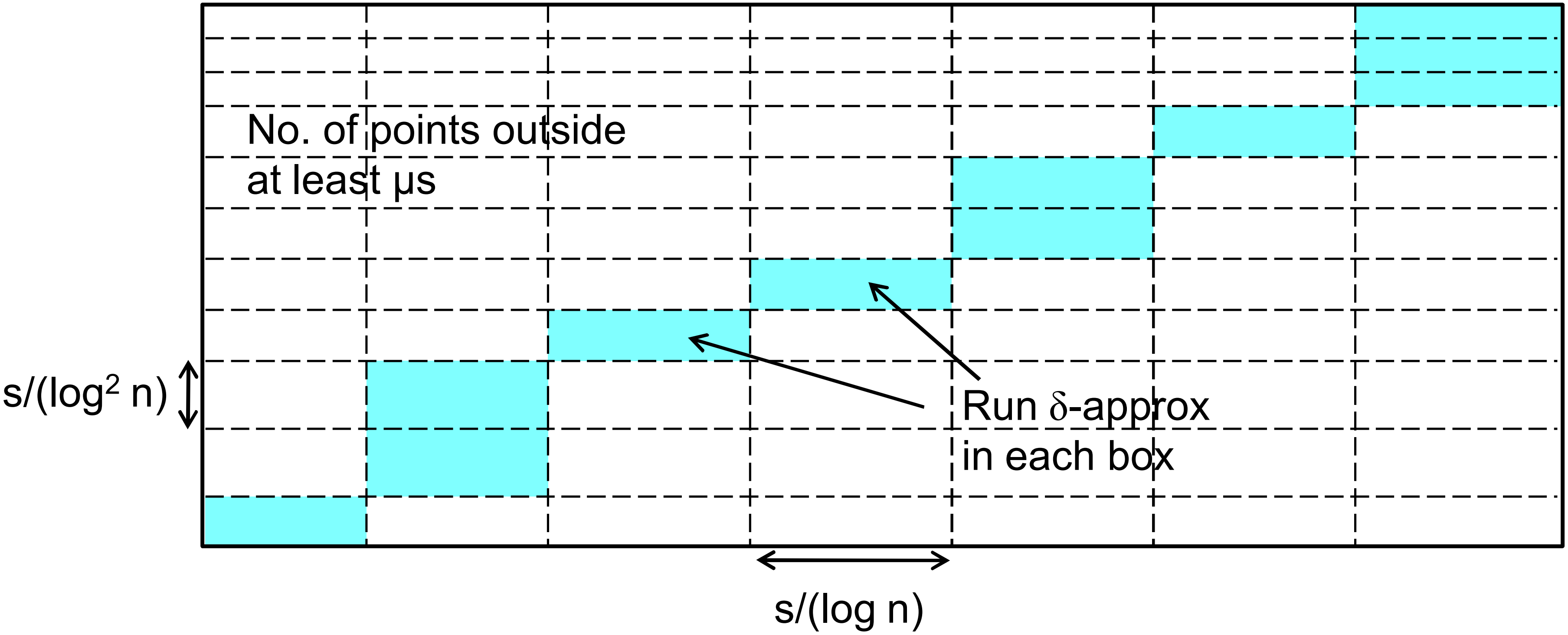
- Suppose every "3-way" division has at most (1-μ)s points
- Do this for all 3-way splits (only poly(log n) many)
- Take max over all sums – we get δ(1-μ)-approx

# A big generalization



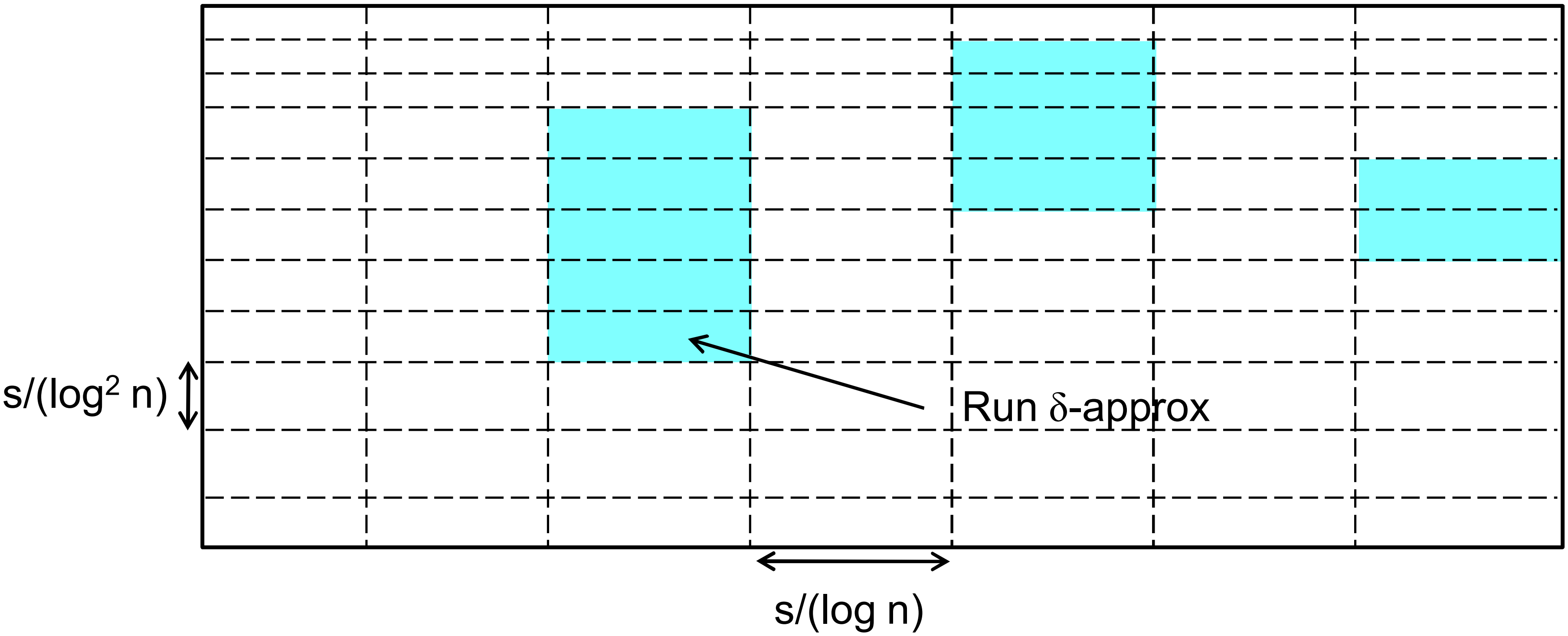
- Suppose every "chain" has at most $(1-\mu)s$ points

# A big generalization



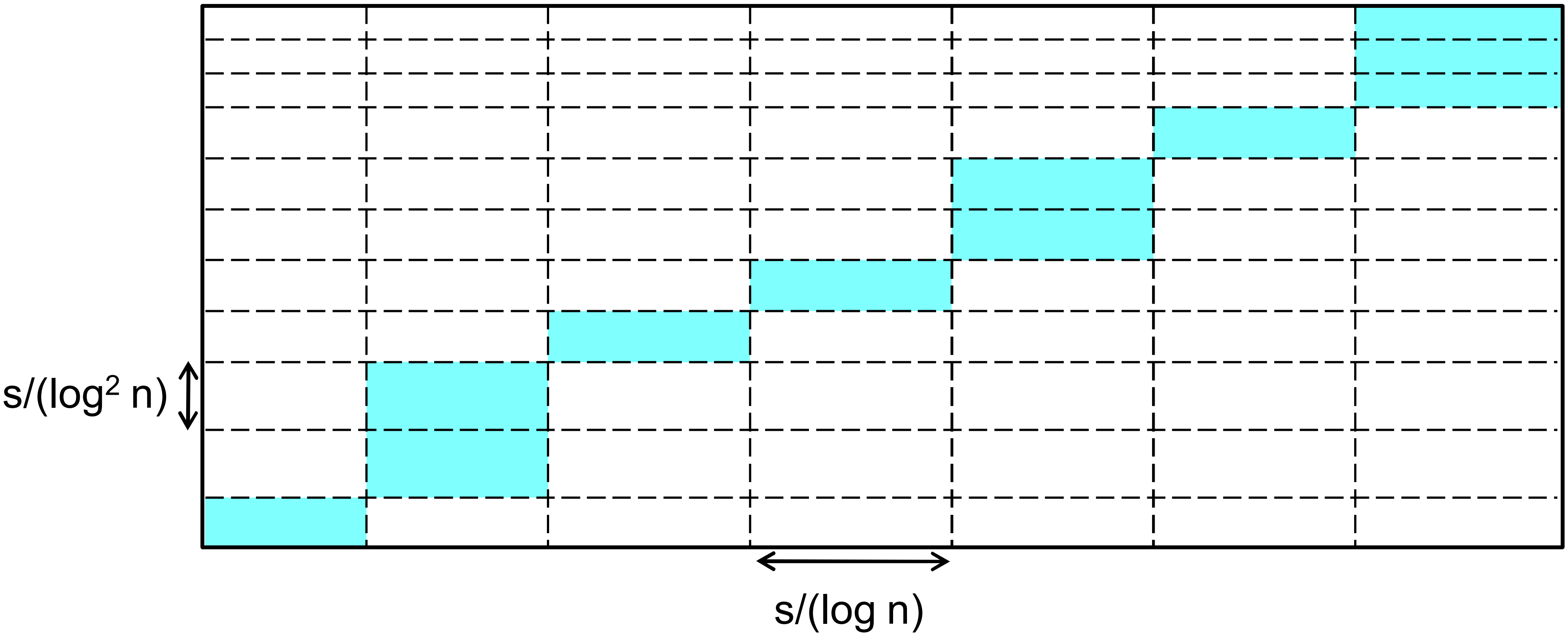
- Suppose every “chain” has at most (1-μ)s points
- Find chain with largest sum of estimates
- We get δ(1-μ)-approx
- But there are more than poly(n) chains!
  - (Not a problem. Why?)

# It's…a DP!



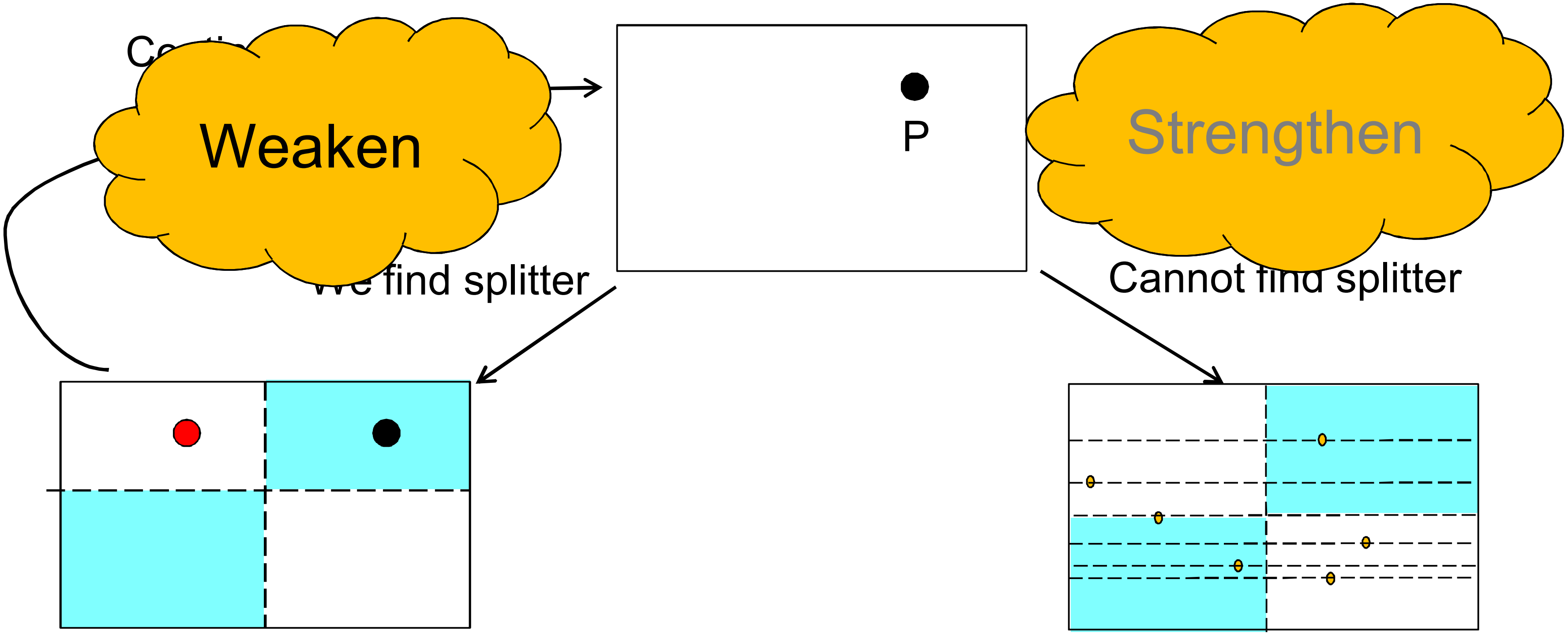
- Run δ-approx on all poly(log n) such boxes
- Use Dynamic Program to find chain with largest sum of estimates
  - Longest path in DAG
  - Can solve in poly(log n) time

# A big generalization



- Suppose every “chain” has at most (1-μ)s points
- In poly(log n) time, with poly(log n) calls to δ-approx, we get δ(1-μ)-approx

# The basic dichotomy

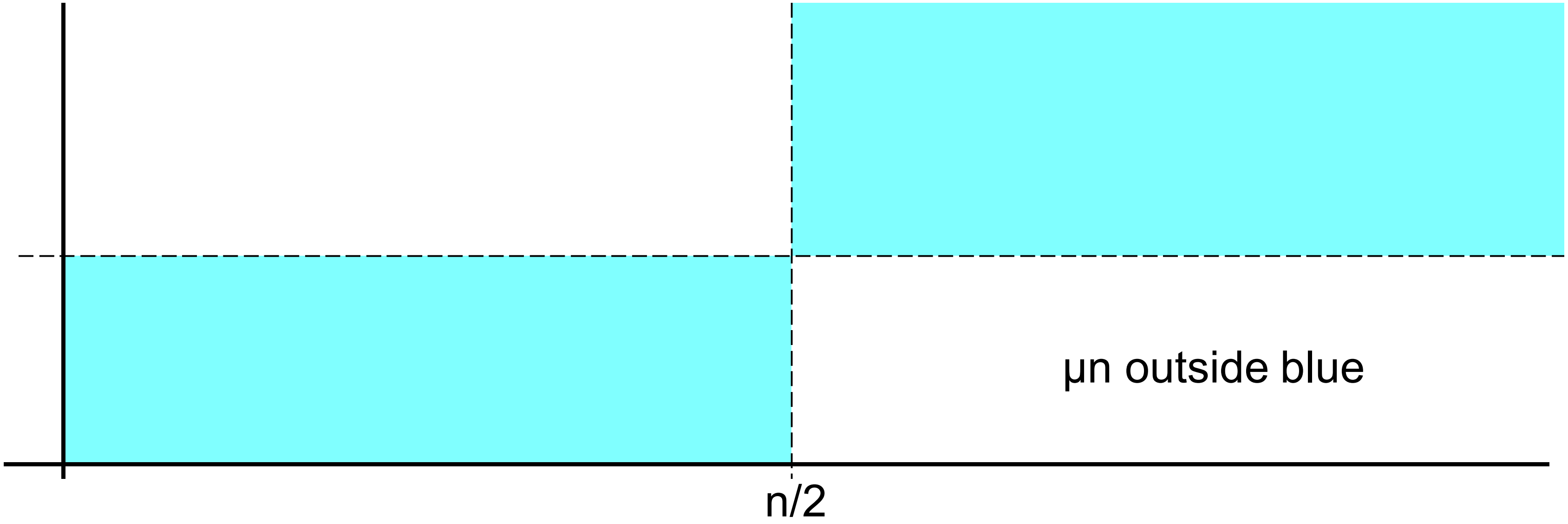


The “Interactive Protocol” phase

The “Dynamic Programming” phase

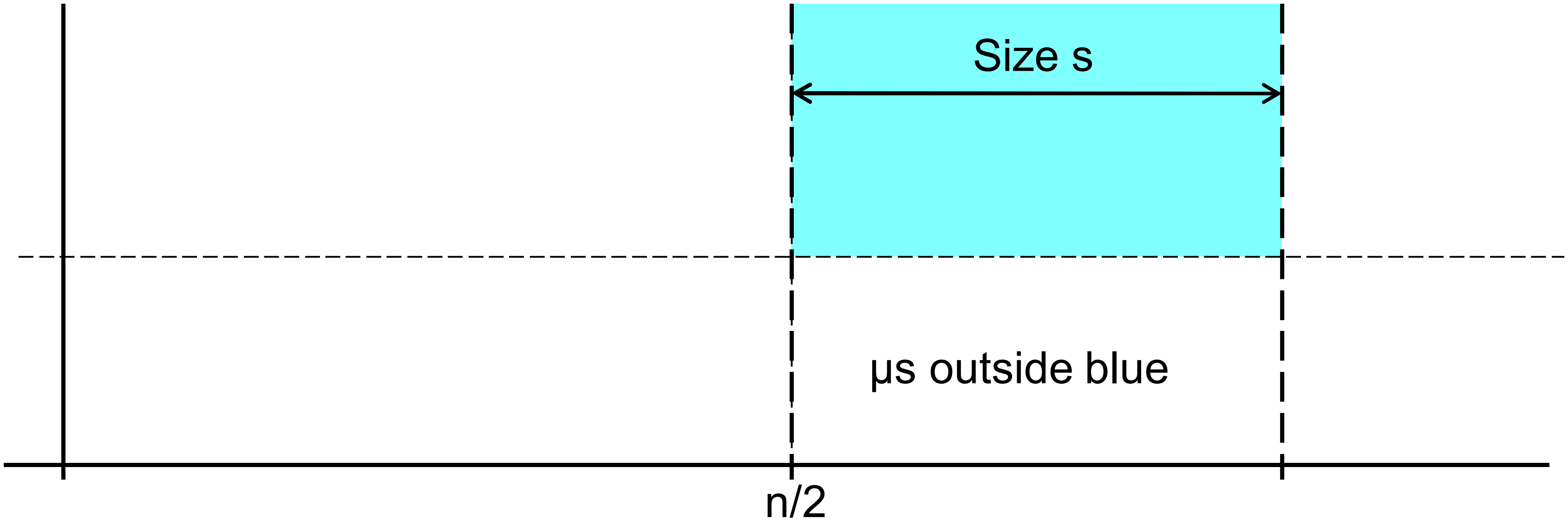
- For IP, we need μ < 1/log n
    - μn is error in each “level” of IP
- For boosting, we need μ > 1/log log n
    - (1-μ) is decrease in error

# The new and improved…



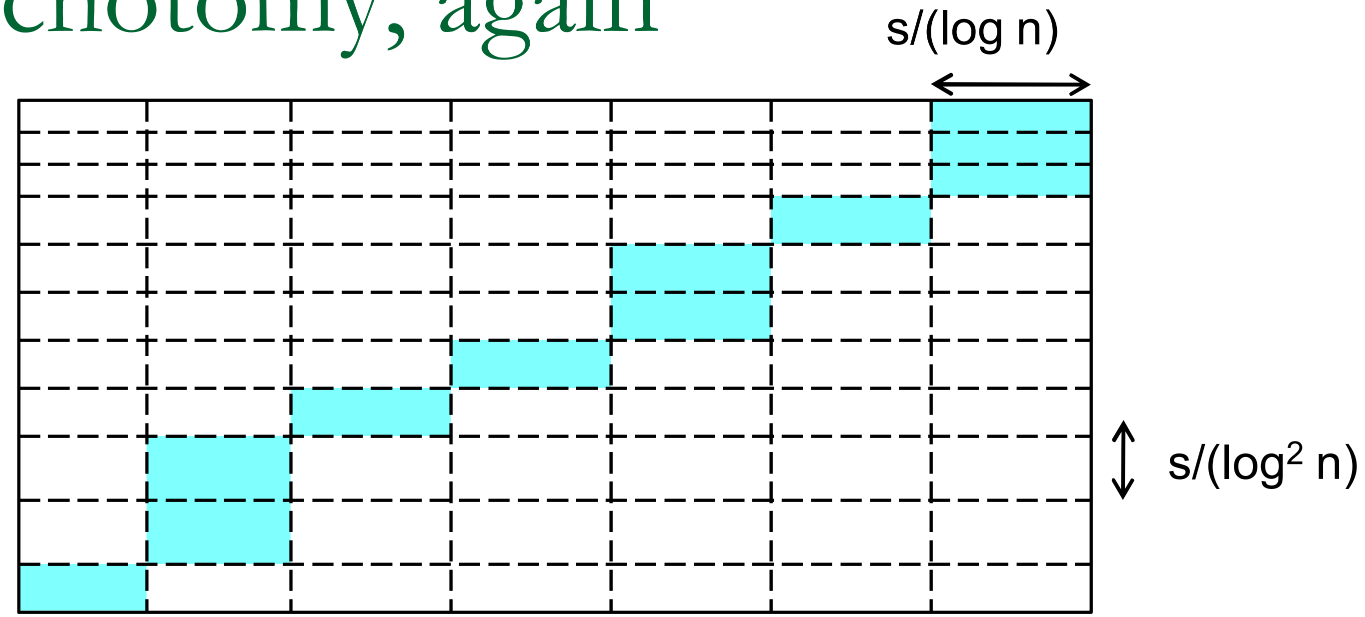
- We set μ to be constant (say μ = 1/50)

# The new and improved…



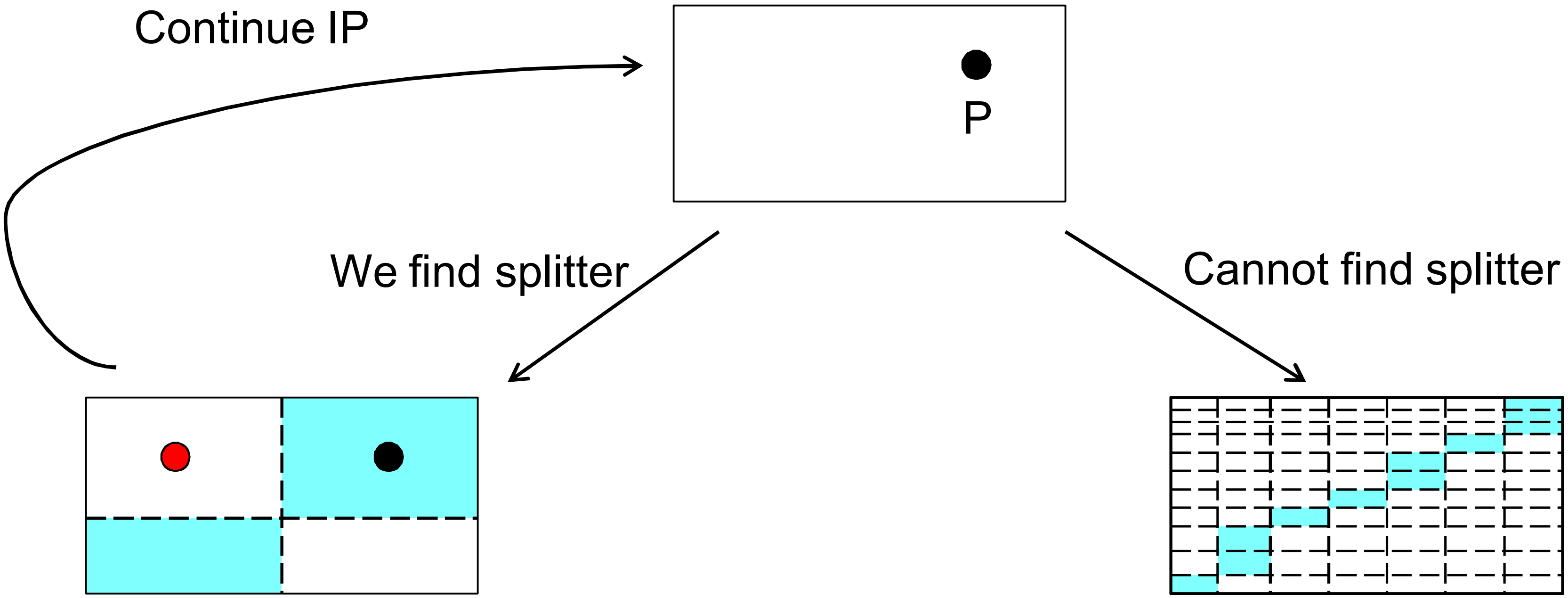
- We set μ to be constant (say μ = 1/50)
- But we now ask for < μ-fraction of violations in every interval
  - As long as interval size s > n/(100log n)
- This is “improved splitter”

# The dichotomy, again



- We are unable to find improved splitter in this box
- Build grid in the box
- Lemma: Since we cannot find splitter, no chain has more than (1-μ)s points
  - We can find δ(1-μ)-approx for LIS in box

# The algorithm, in one slide



Continue IP

P

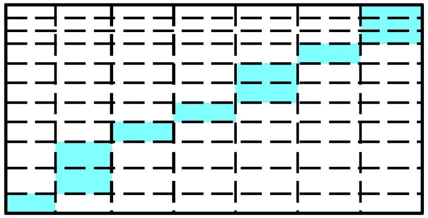
We find splitter

Cannot find splitter

Make poly(log n) calls to δ-approx. Solve DP of poly(log n) size.

- We get δ(1-μ)-approx
- Overall running time becomes $(\log n)^{1/\delta}$
  - *&^#$% miracle that the math works out

# The even better version



- Don’t exactly solve this dynamic program!
- Use our sublinear algo to approximately solve in (loglog n) time. Then do it recursively…
  - I know, my head is spinning too

- Greek list: α β γ δ ε ζ λ μ ξ ρ θ τ ψ
  - We had ν, but got rid of it

# What next?

- We get $2^{1/\delta}$ $(\log n)^c$ time. Can we get $(\log n)/\delta$ time?
  - Would be extremely cool. Completely optimal

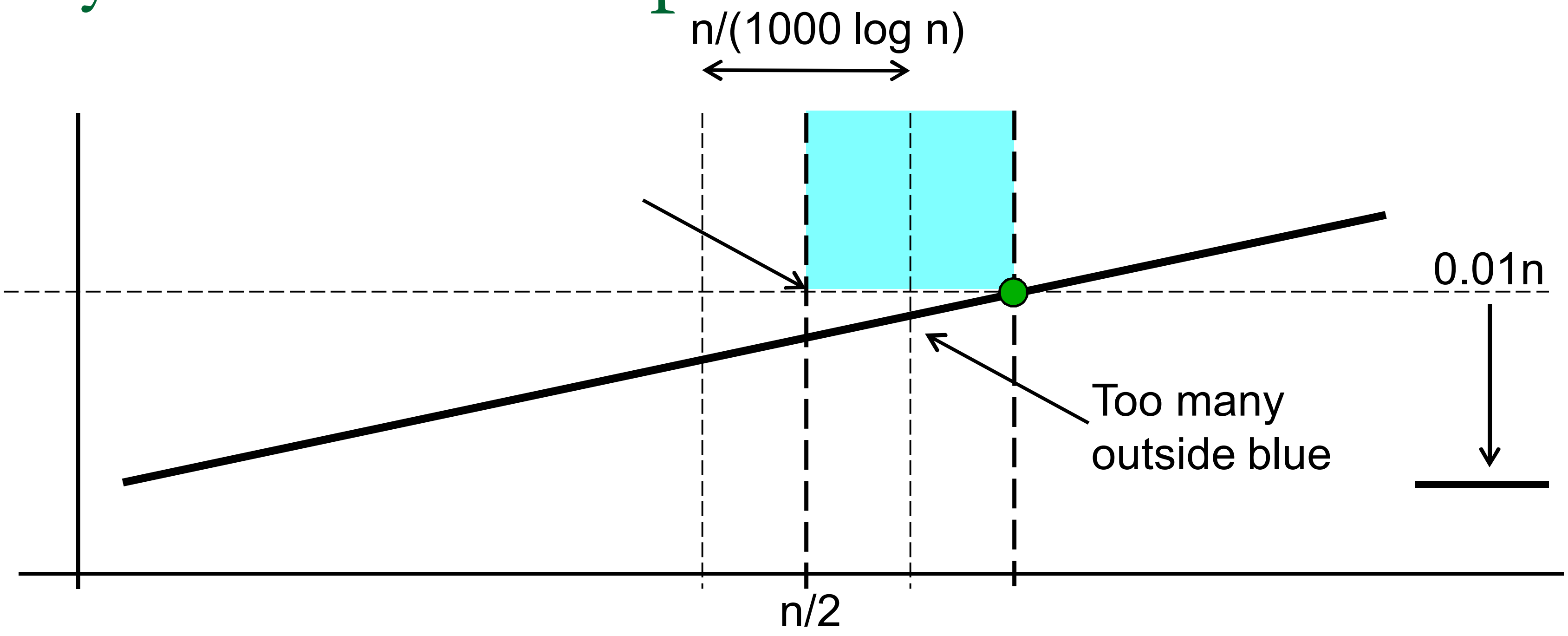
- There's a lot going on here. We don't completely get it.
- Applications for other dynamic programs?
  - Longest common subsequence
  - Approximating edit distance
  - …?

# Questions?

Surely, the talk was not *that* clear.

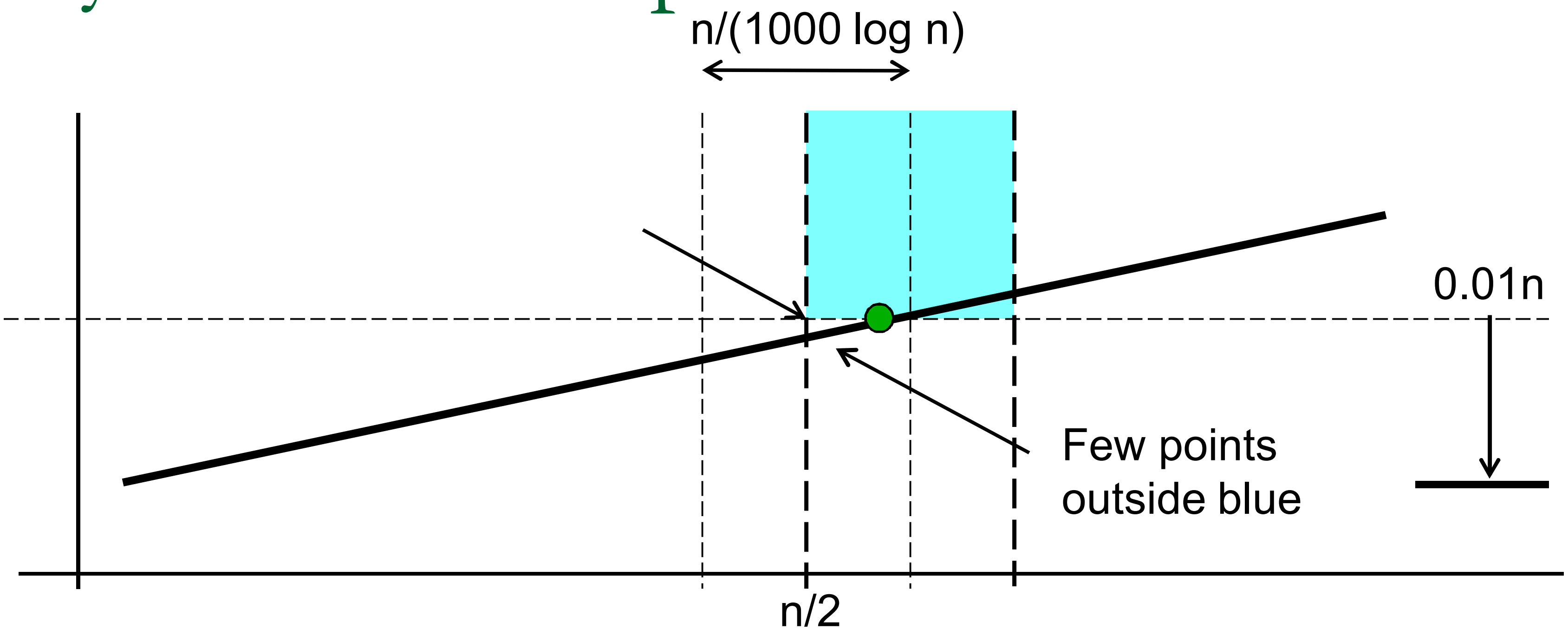
Or that confusing…

# Why does this help?



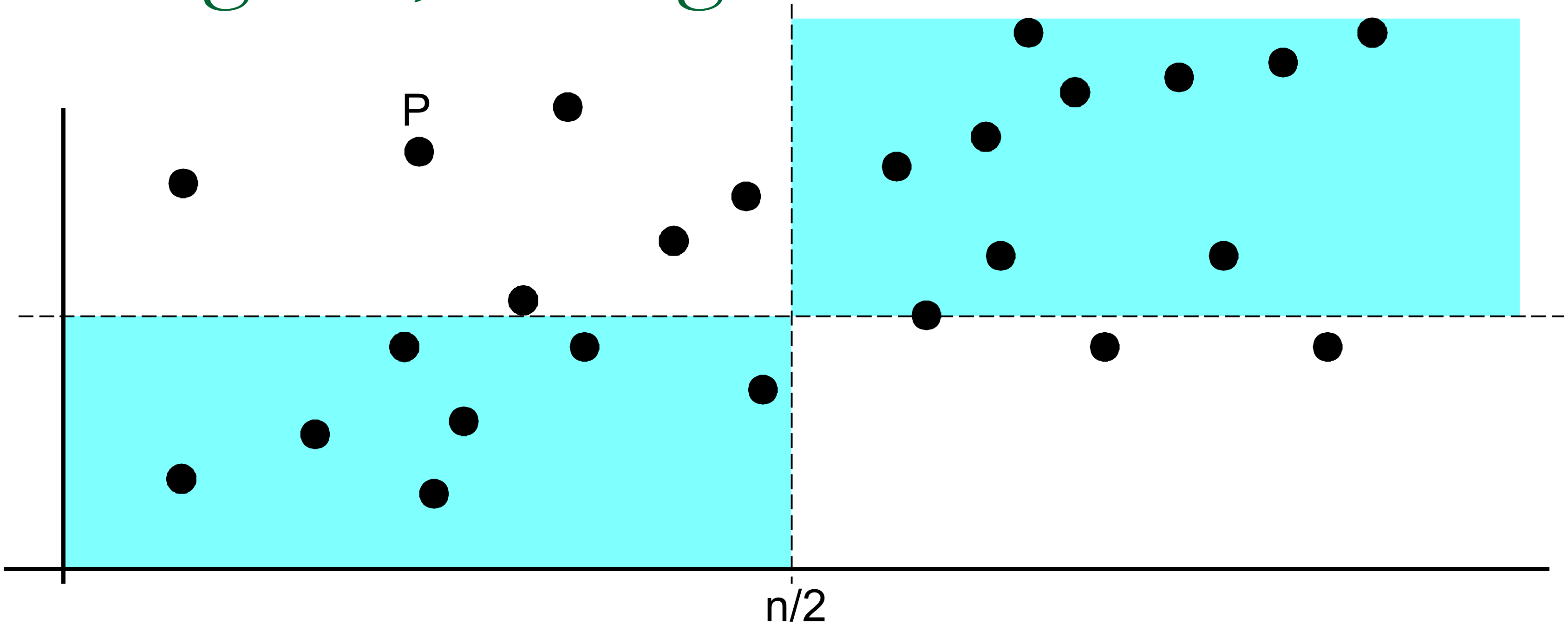
- μ = 1/50, so the small set of violations doesn't hurt
- Point "too far" from real splitter not allowed

# Why does this help?



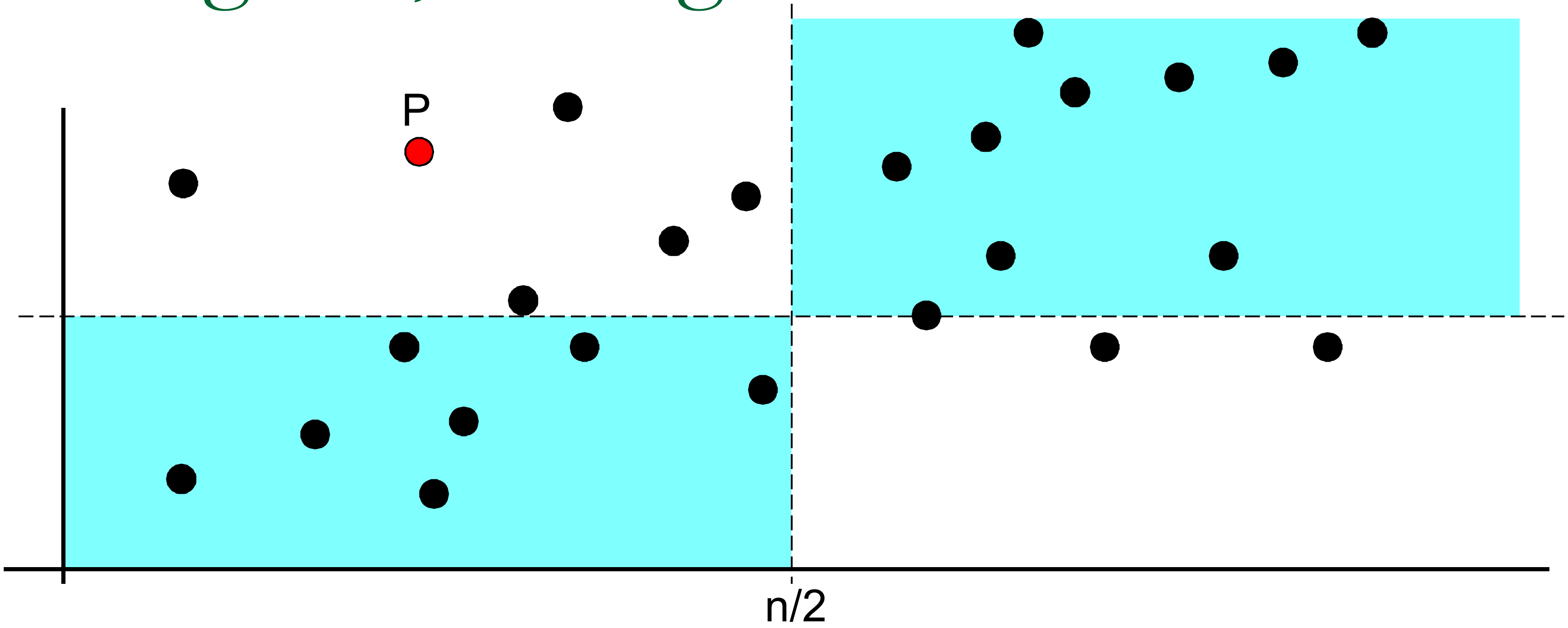
- μ = 1/50, so the small set of violations doesn't hurt
- Point "too far" from real splitter not allowed
- At least n/(100log n) points are improved splitters
    - So we can find one by sampling

# At long last, the algorithm



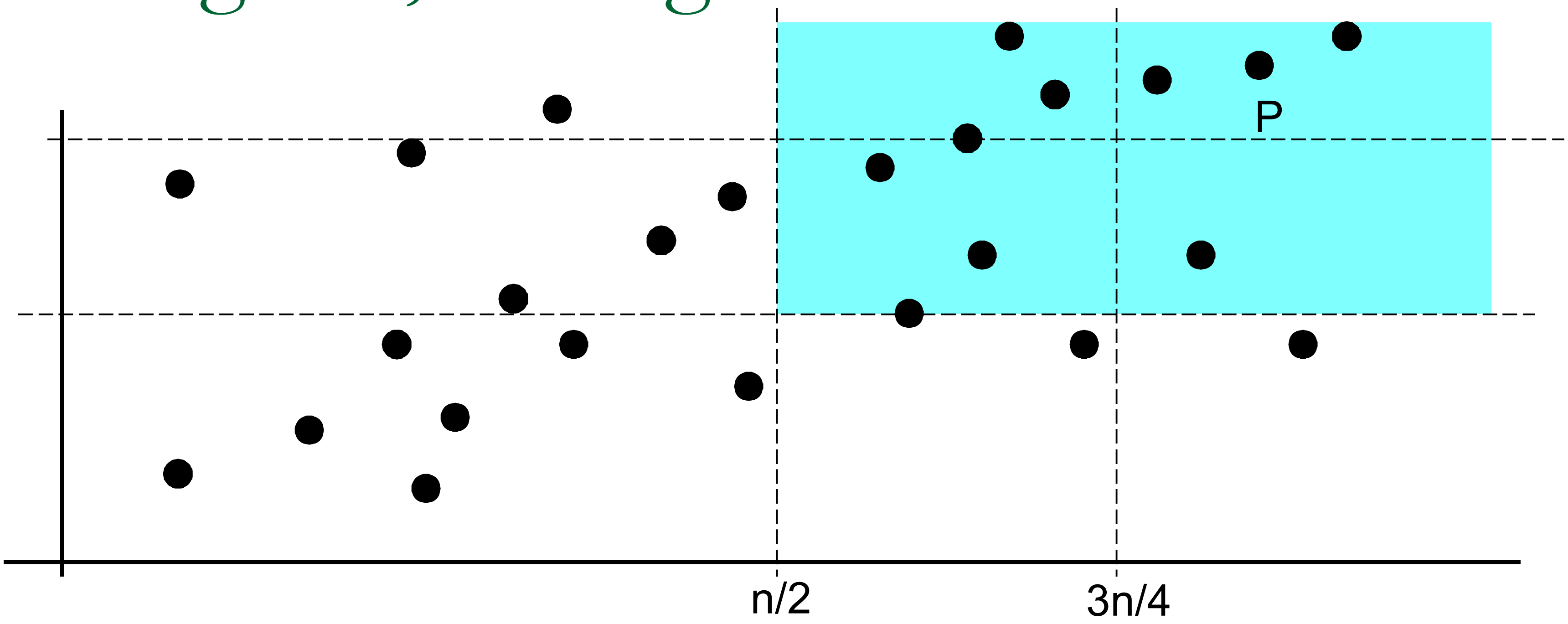
- Is P good or bad?
- Find improved splitter

# At long last, the algorithm



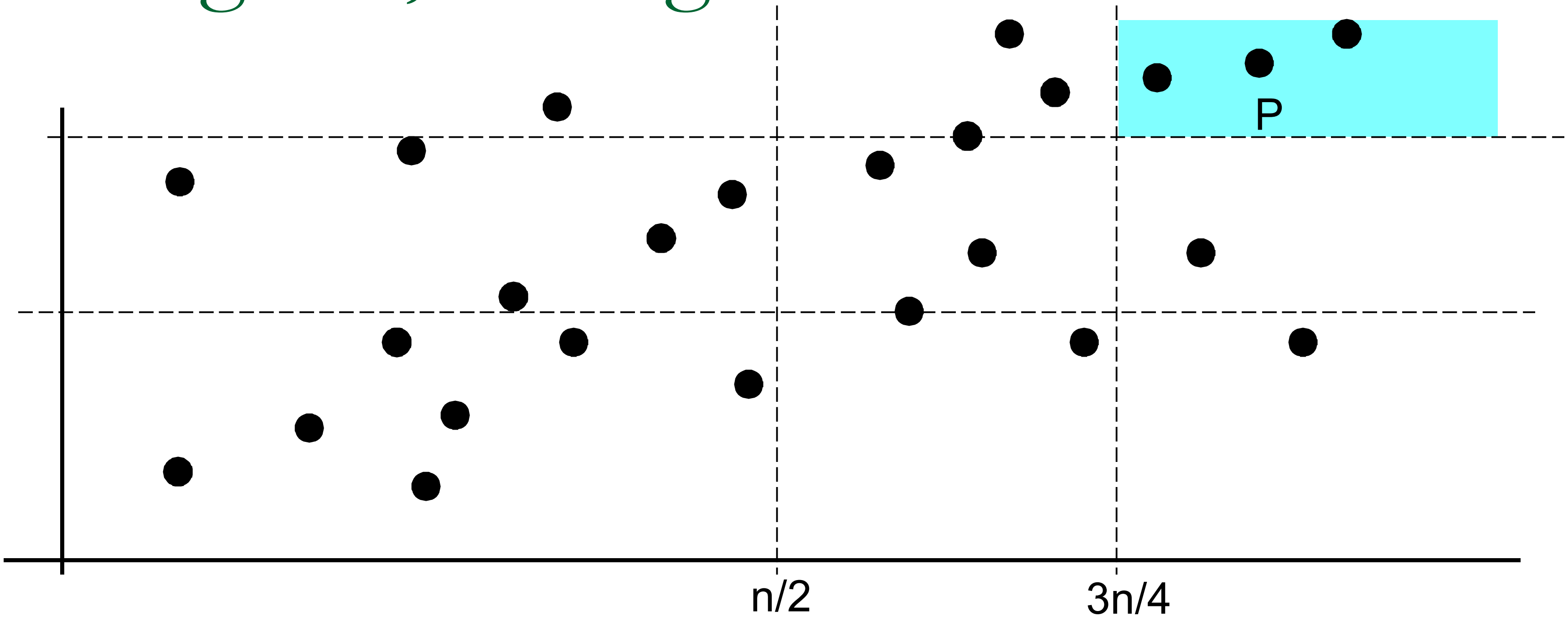
- Is P good or bad?
- Find improved splitter
- So P is bad. We’re done

# At long last, the algorithm



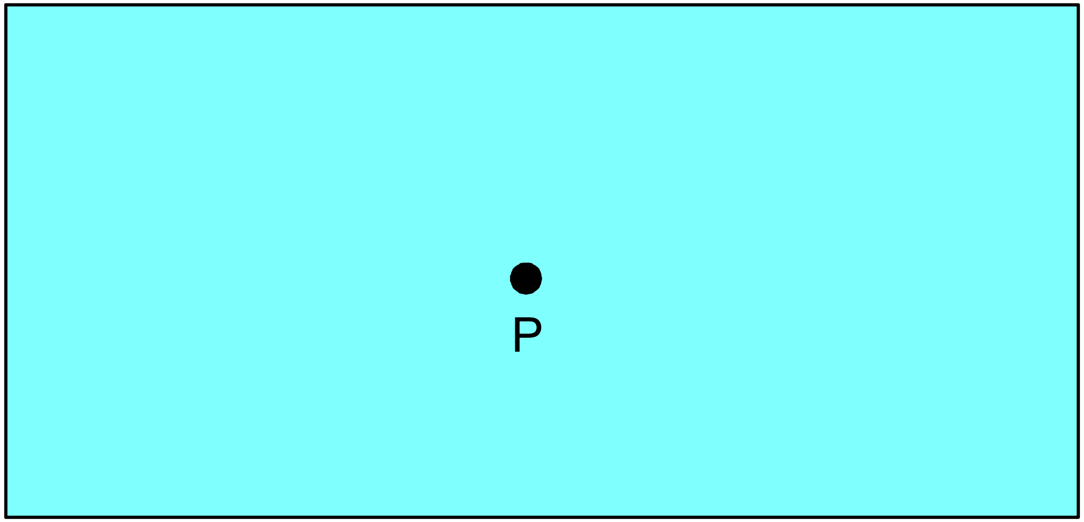
- Is P good or bad?
- Find improved splitter
- Continue with the IP
    - Find next improved splitter, etc, etc,

# At long last, the algorithm



- Is P good or bad?
- Find improved splitter
- Continue with the IP
  - Find next improved splitter, etc, etc,

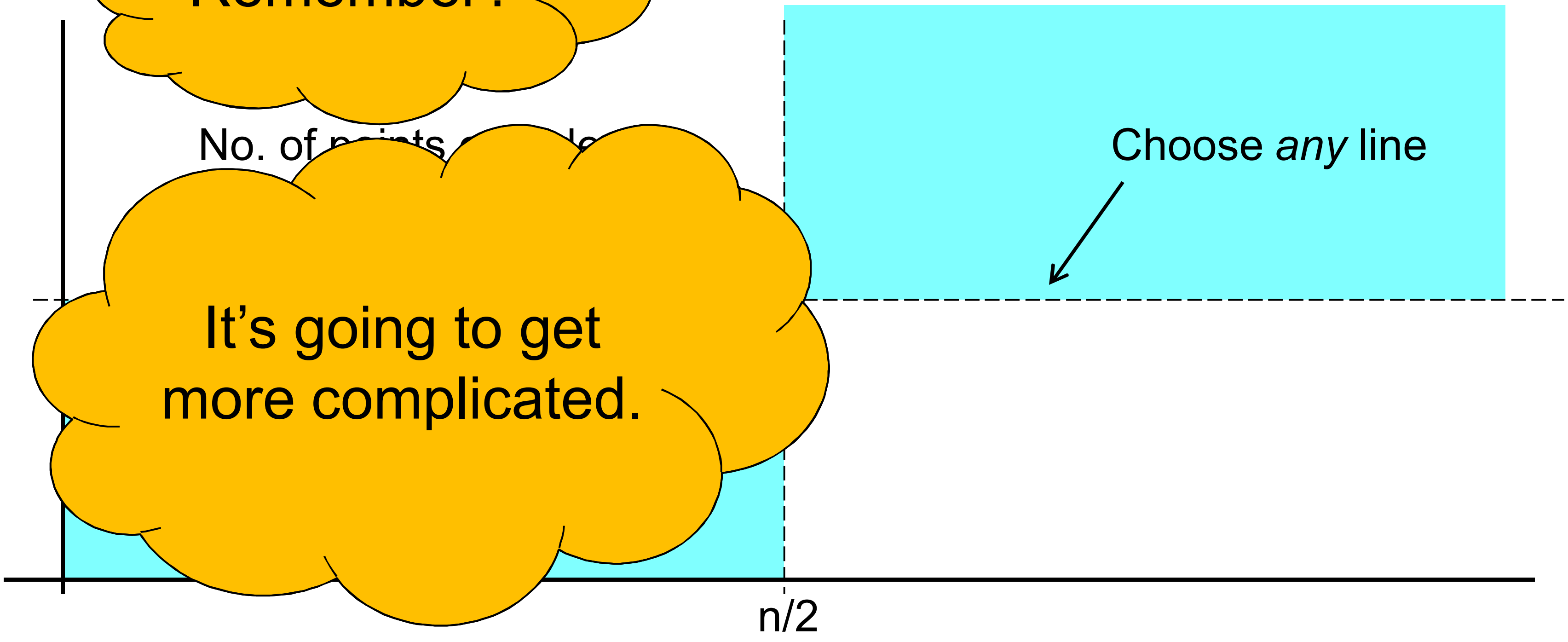
# What if…?



- We are unable to find splitter in this box

# A lib

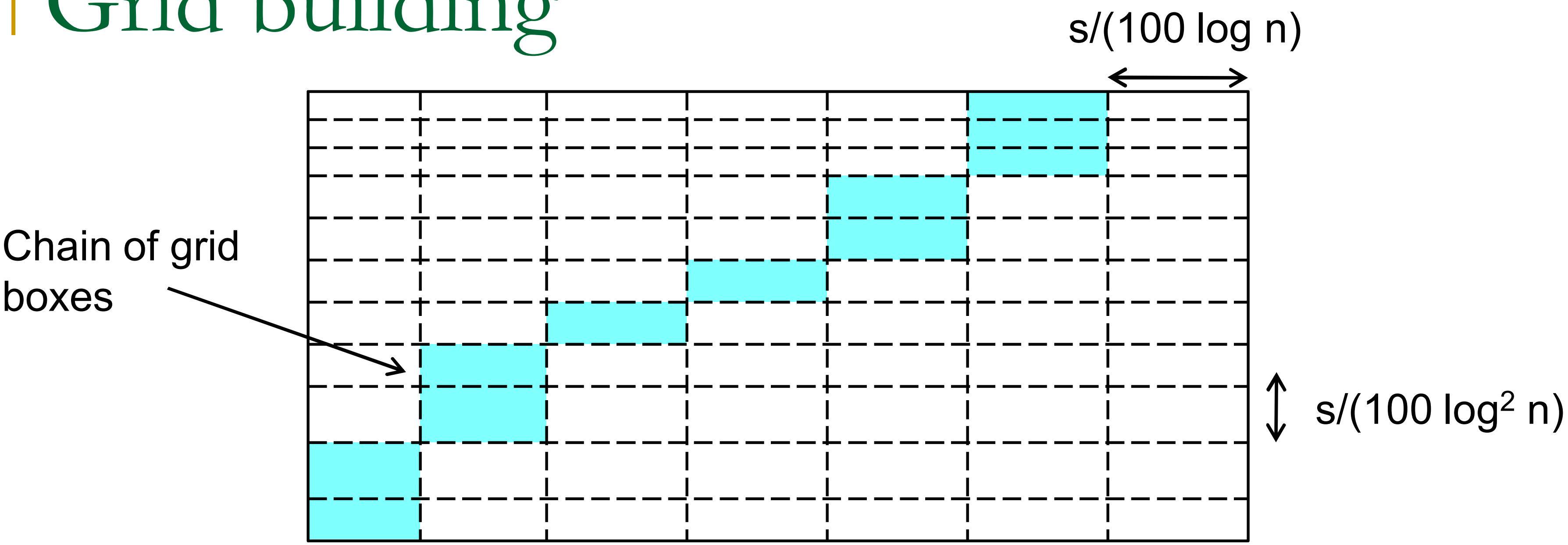
Remember?



It's going to get more complicated.

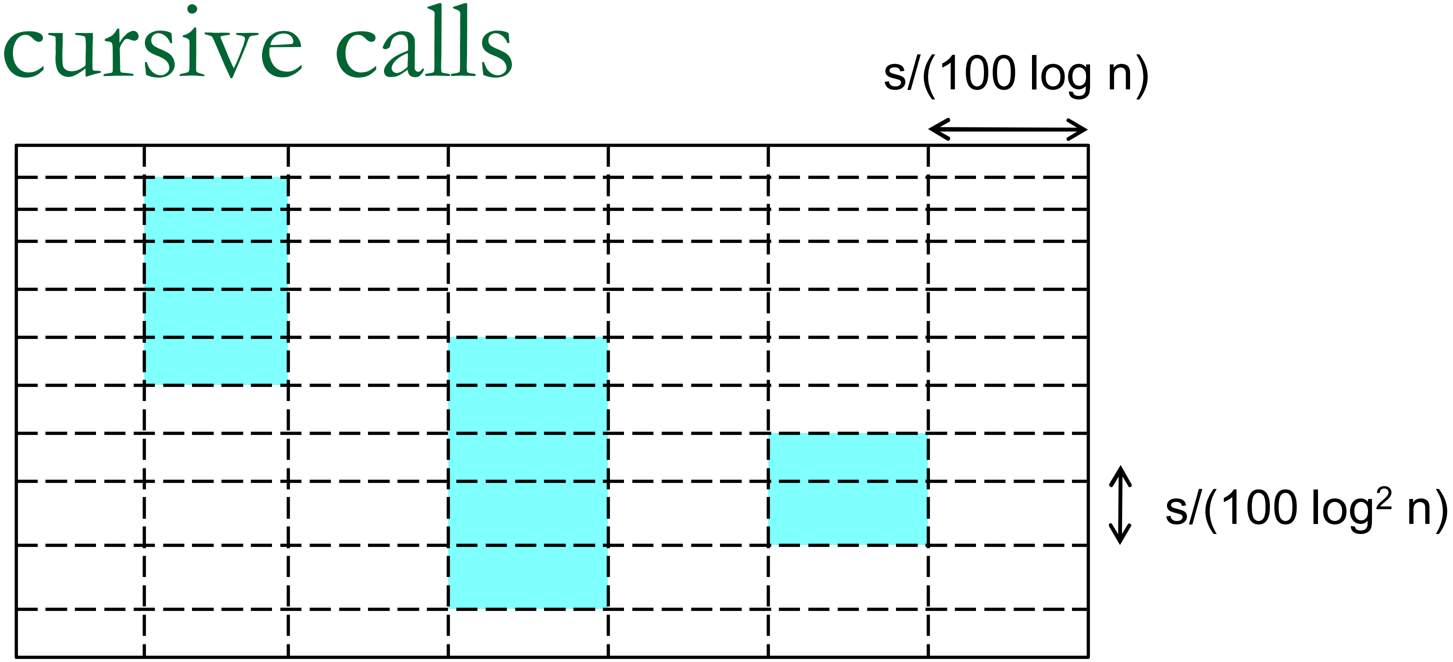
- So we know that |LIS| < (1-μ) n
- Leads to the next idea. Boosting approximations!

# Grid building



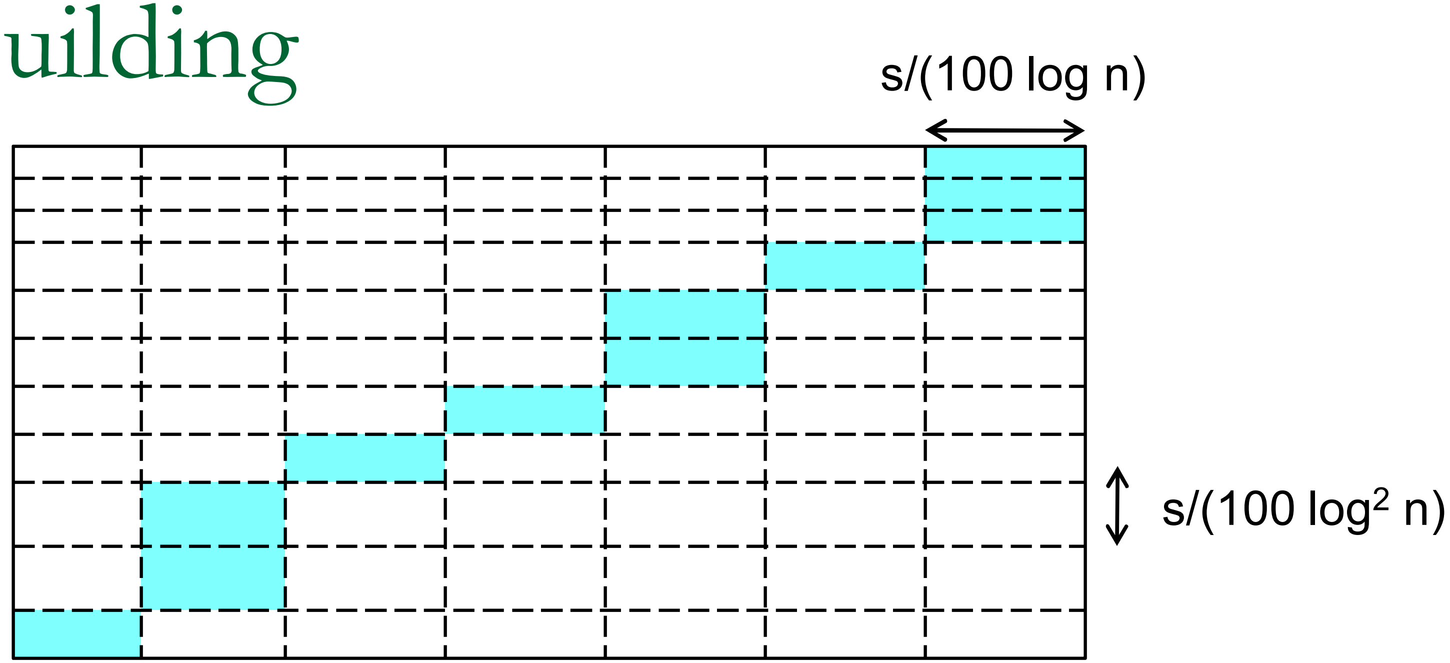
- We are unable to find splitter in this box
- Build grid in the box

# The recursive calls



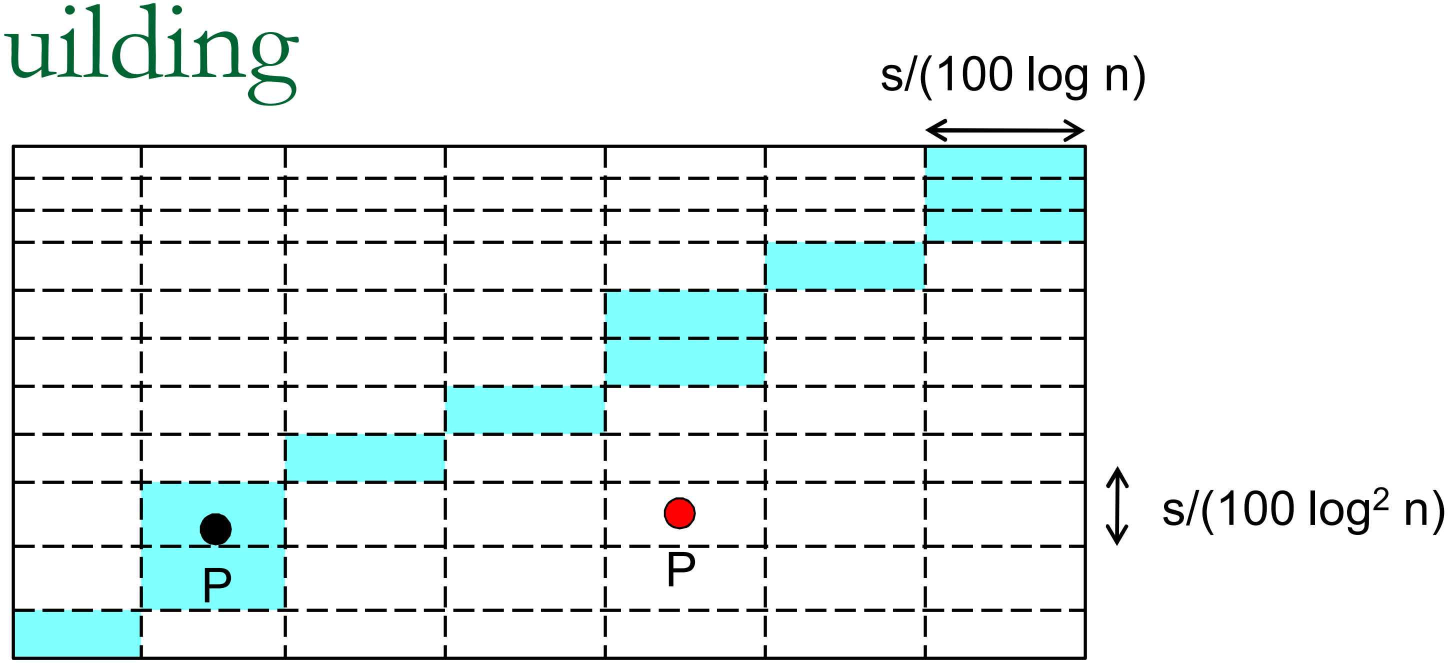
- For grid box B, length(B) = δ-approx LIS estimate in B
- Only poly(log n) calls to δ-approx algorithm

# Grid building



- Length of chain is just sum of length of boxes
  - This is sort of estimate of LIS inside chain
- The longest chain is our estimate of where is LIS of box
  - Longest path in DAG of size (log n)$^c$: Solve by dynamic program

# Grid building



- The longest chain is our estimate of where is LIS of box
  - Longest path in DAG of size (log n)$^c$: Solve by dynamic program
- If P in longest chain: in box B, use δ-approx to check if P is good.