Space & Time Efficient Algorithms for Lipschitz Problems

Barna Saha
University of Massachusetts Amherst

Collection of Some Basic Polynomial Time Problems

- Longest Increasing Subsequence [Schensted, 1961]
- String Edit Distan… …tation [Levenshtein, 19…
- Context Free Gr… parser, CYK, 196…-70]
- Language Edit Distance [Aho & Peterson, 1972]
- RNA Folding [Nussinov, Jacobson, 1980]

Dynamic Programming

1. Longest Increasing Subsequence [Schensted, 1961]

- *Given a sequence of integers s[1], s[2],..,s[n], find a subsequence* $1 \leq i_1 \leq i_2 \leq \ldots \leq i_k \leq n$ *such that* $s[i_1] < s[i_2] < \ldots < s[i_k]$ *and k is maximized.*

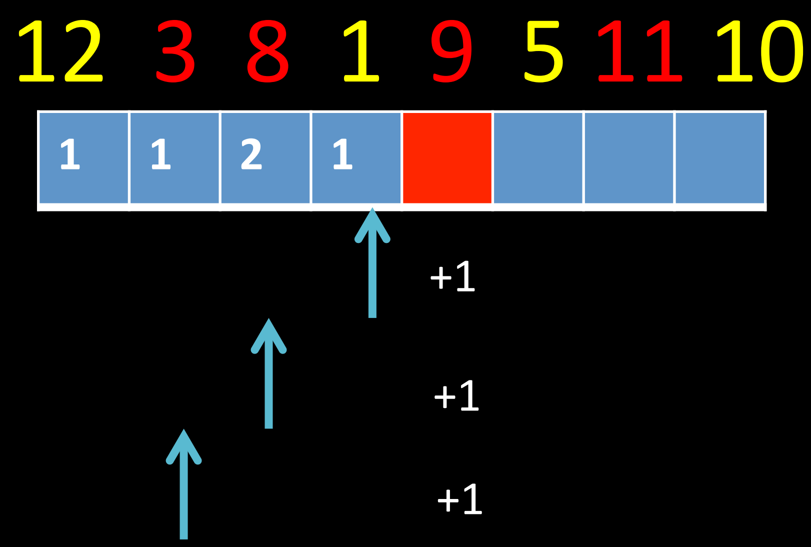
- Example

12 3 8 1 9 5 11 10

- The longest increasing subsequence has length 4

1. Longest Increasing Subsequence
[Schensted, 1961]

- Dynamic Programming
  - $LIS[1]=1$
  - $LIS[i]=\max_{j<l:s[j]<s[i]} LIS[j]+1$



1. Longest Increasing Subsequence
[Schensted, 1961]

- Dynamic Programming
  - LIS[1]=1
  - LIS[i]=$\max_{j<l:s[j]<s[i]}$ LIS[j]+1

12 3 8 1 9 5 11 10

1	1	2	1	3			

+1

+1

Time Complexity=O(n^2)
Space Complexity=O(n)

- More sophisticated dynamic programming with time complexity O(n log n) and O(n) space exists.

2. String Edit Distance [Levenshtein, 1965]

- *Given two strings s and t what is the minimum number of edits (insertion, deletion, substitution) needed to transform s to t?*
  - Example

s=ACCGGACGTT (reference string)

t=ATACGGACGT (input string)

C T

delete

substitute insert

Edit distance is 3

2. String Edit Distance [Levenshtein, 1965]

- Dynamic Programming
  - Edit[0,i]=Edit[i,0]=i
  - Edit[i,j]=min[1+Edit(i-1,j), 1+Edit(i,j-1), cost(s[i],t[j])+Edit(i-1,j-1)]

A T A C G G A C

A	C	C	G	G	A	C	T
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	2	3	4	5	6	6
2	2	2	3	4	4	5	6
3	2	2					

Time Complexity=O(n^2)
Space Complexity=O(n^2)

2. String Edit Distance [Levenshtein, 1965]

- Dynamic Programming
  - Edit[0,i]=Edit[i,0]=i
  - Edit[i,j]=min[1+Edit(i-1,j), 1+Edit(i,j-1), cost(s[i],t[j])+Edit(i-1,j-1)]

	A	C	C	G	G	A	C	T
A	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
T	1	1	2	3	4	5	6	6
A	2	2	2					
C	3	2						
G								
G								
A								
C								

Time Complexity=O(n^2)
Space Complexity=O(n)

- Assuming Strong Exponential Time Hypothesis no truly subquadratic algorithm exists for the exact computation [Backurs, Indyk, STOC'15]
- Even shaving arbitrary polylog factor is seemingly hard [Abboud, Dueholm, V Williams, Williams, STOC'16]

3. Context Free Grammar Parsing

[Earley's parser, CYK 1968-70]

- *Given a grammar G and a string s, can s be parsed according to rules of G?*

- G:

(Production rules)

A -> BC

B->XX

C-> AX

X->a

B->y

End index

Start Index	1	2	3	4	5	6	7
1							
2							
3			0	1	1		
4						1	
5						1	
6						0	
7							

3. Context Free Grammar Parsing

[Earley's parser, CYK 1968-70]

- *Given a grammar G and a string s, can s be parsed according to rules of G?*

- G:

(Production rules)

End index

1 2 3 4 5 6 7

Start Index

- **Without using fast matrix multiplication**, no truly subcubic exact algorithm
- **Using fast matrix multiplication** an $O(n^{\omega})$ exact algorithm [L Valiant, Ph.D. Thesis, 1978]
- Valiant's algorithm is the best possible [Abboud, Backurs, V. Williams, FOCS'15, Lee 2001]

B->y

6 0

7

Time Complexity=$O(n^3)$

Space Complexity=$O(n^2)$

4. Language Edit Distance [Aho & Peterson, 1972]

- *Given a grammar G and a string s, find the minimum number of edits required in s to be able to parse the edited string according to the rules of G.*

- G:

(Production rules)

A -> BC

End index

Start Index	1	2	3	4	5	6	7
1							
2							

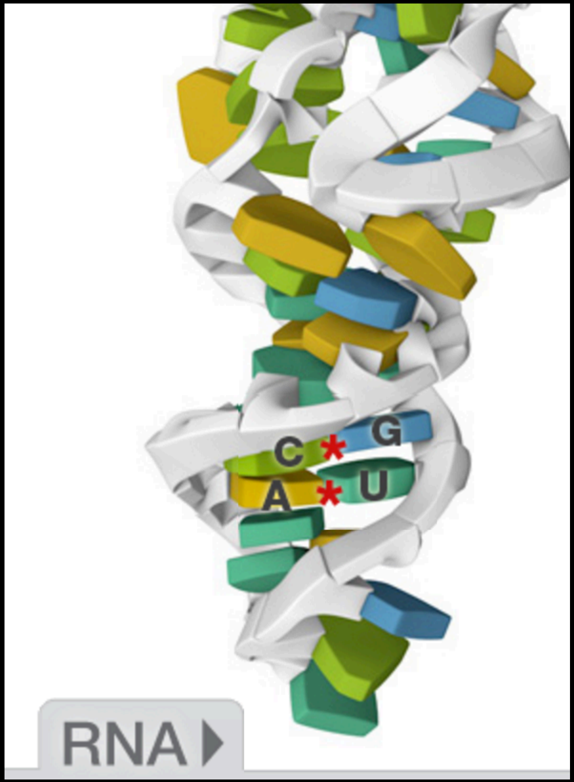
- **When only insertion is allowed**, LED is as hard as weighted All-Pairs-Shortest Paths [Saha, FOCS'15]
- For all possible edits, no conditional lower bound known that is stronger than parsing
- **Using fast matrix multiplication**, the **first** truly subcubic algorithm was developed last year [Bringmann, Grandoni, Saha, V. Williams, FOCS'16]

Tim

Space Complexity=O(n)

5. RNA Folding [Nussinov, Jacobson, 1980]

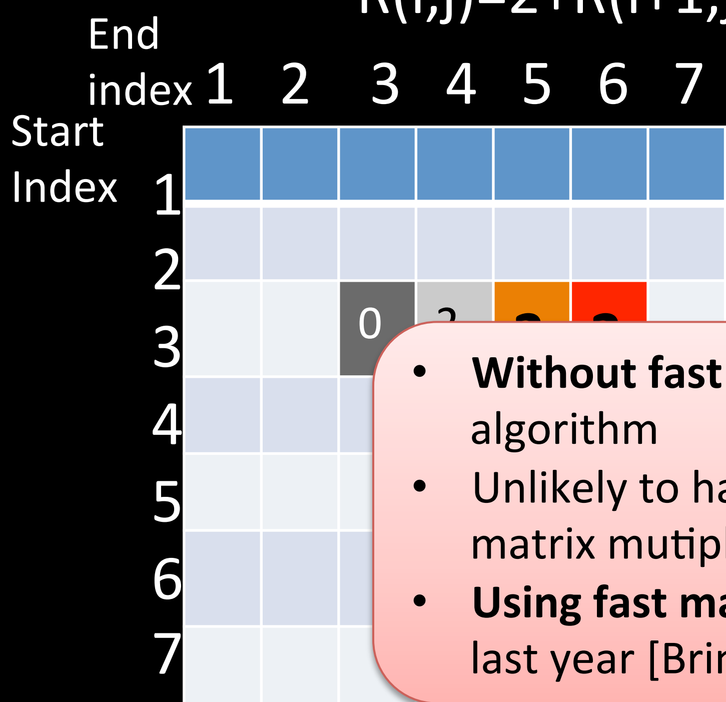
Nucleotides in RNA form complementary base pairs to form the RNA secondary structure: C pairs with G and A pairs with U.



GGCAGUACCGGUAAUAAGCUGCC

5. RNA Folding [Nussinov, Jackobson, 1980]

- Dynamic Programming
  - RNA[i,i]=0, RNA[i,j]=0 if j < i
  - RNA[i,j]=max(R[i,j], $\max_{i<=l<j}$ RNA[i,l]+RNA[l+1,j])
    - R(i,j)=0 if s[i] does not pair with s[j]
    - R(i,j)=2+R(i+1,j-1) if s[i] pairs with s[j]



Time Complexity=O(n^3)
Space Complexity=O(n^2)

- **Without fast matrix multiplication**, no truly subcubic exact algorithm
- Unlikely to have an algorithm with running time better than boolean matrix mutiplication [Abboud, Backurs, V. Williams, FOCS'15]
- **Using fast matrix multiplication**, the first truly subcubic algorithm last year [Bringmann, Grandoni, Saha, V. Williams, FOCS'16]

What is common among these Dynamic Programming Problems?

Longest Increasing Subsequence

String Edit Distance

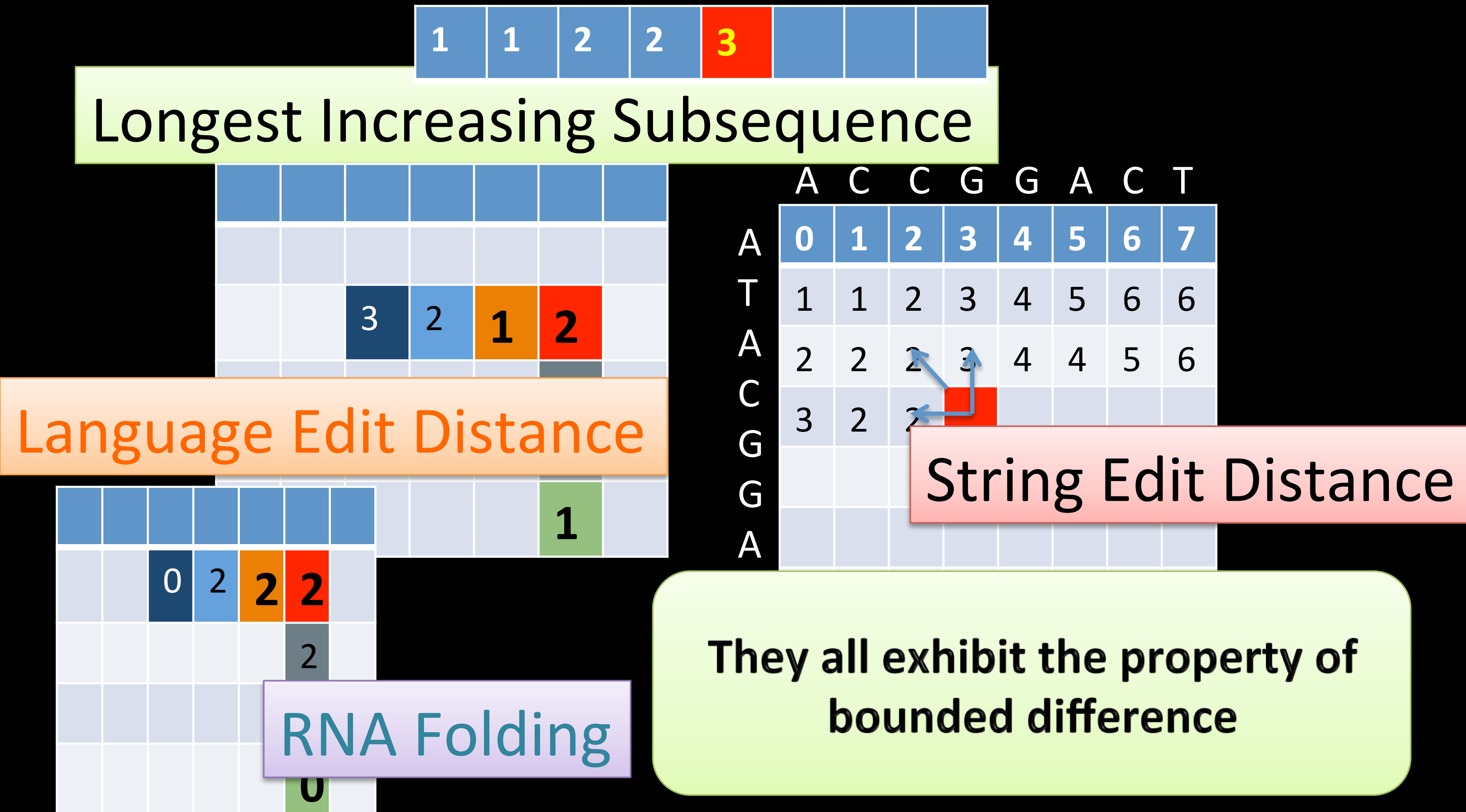
Context Free Grammar Parsing

Language Edit Distance

RNA Folding

They all exhibit the property of bounded difference

What is common among these Dynamic Programming Problems?



They all exhibit the property of bounded difference

What is the main difference?

Looks at many subproblems at a time

Longest Increasing Subsequence

Context Free Grammar Parsing

Language Edit Distance

RNA Folding

Looks at a constant number of subproblems at a time

String Edit Distance

What is the main difference?

Looks at many
at a t

Using Additive Approximation

onstant number
problems
a time

Longest Increasing Subsequence

Context Free Grammar Parsing

Language Edit Distance

RNA Folding

can improve both space and time complexity using amnesic dynamic programming

String Edit Distance

can improve space complexity using amnesic dynamic programming

Results: Language Edit Distance

– Previously Known
  - Conditional Lower Bound: No combinatorial subcubic algorithm exists even for any nontrivial multiplicative approximation. [Abboud, Backurs, V. Williams, FOCS'2015]
  - Upper Bound:
    – Combinatorial: $O(n^3)$ time complexity, $O(n^2)$ space [Aho & Peterson, 1972, Myers, 1985,..]
    – Using Fast Matrix Multiplication: $O(n^{2.8244})$ time complexity, $O(n^2)$ space [Bringmann, Grandoni, Saha, V. Williams, FOCS 2016]
    – Using Fast Matrix Multiplication: $O(n^\omega/\varepsilon^4)$ time complexity, $O(n^2)$ space randomized algorithm for multiplicative $(1+\varepsilon)$-approximation [Saha, FOCS 2015]

– What we show [Saha, FOCS'17]
  - Combinatorial & Deterministic algorithm with time complexity $O(n^2/\varepsilon)$, space $O(n/\varepsilon)$, εn-additive approximation
  - Sublinear space: $O(n^{2/3}/\varepsilon^{4/3})$ space for εn-additive approximation
  - Implies same bound for approximate membership checking for context free grammars

Results: RNA Folding

- Previously Known
  - Conditional Lower Bound: No combinatorial subcubic algorithm exists [Abboud, Backurs, V. Williams, FOCS'2015]
  - Upper Bound:
    - Combinatorial: $O(n^3)$ time complexity, $O(n^2)$ space [Nussinov, Jacokbson 1980]
    - Using Fast Matrix Multiplication: $O(n^{2.8244})$ time complexity, $O(n^2)$ space [Bringmann, Grandoni, Saha, V. Williams, FOCS'2016]
    - Using Fast Matrix Multiplication: $O(n^\omega/\varepsilon^4)$ time complexity, $O(n^2)$ space randomized algorithm for εn-approximation [Saha, FOCS 2015]
- What we show [Saha, FOCS'17]
  - Combinatorial & Deterministic algorithm with time complexity $O(n^2/\varepsilon)$, space $O(n/\varepsilon)$, εn-additive approximation
  - Sublinear space: $O(n^{2/3}/\varepsilon^{4/3})$ space for εn-additive approximation

Further Results: String Edit Distance, Linear Grammar, Map Reduce & More

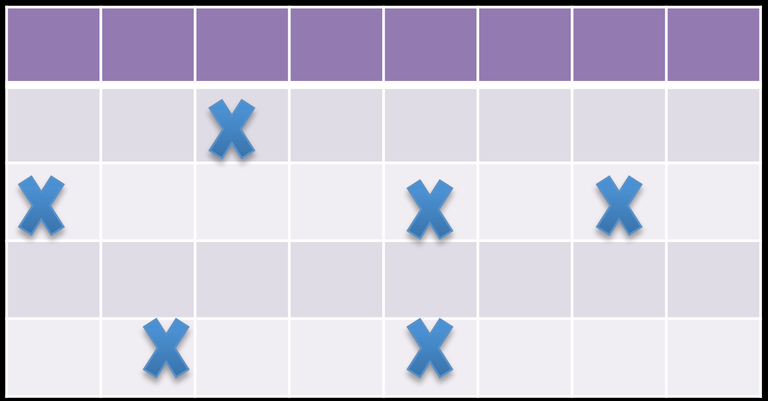
- New result
  - Linear grammar edit distance which generalizes string edit distance
    - Better space vs approximation trade offs: $O(n^{2/3}/ \varepsilon^{2/3})$ space for εn additive approximation
  - Map Reduce and multi-pass streaming algorithms for Language Edit Distance, RNA Folding, String Edit distance
  - Single pass streaming algorithm for string edit distance in asymmetric setting
    - Previously $O(n^{1/2l}/\varepsilon^{1/2})$ space [Saks and Seshadri, SODA'13]
    - This paper: $O(n^{1/2}/\varepsilon)$ space for εn-additive approximation

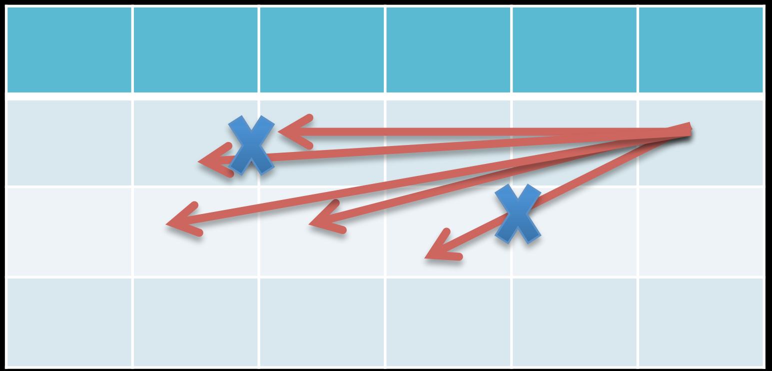
Dynamic Programming

↓

Amnesic Dynamic Programming

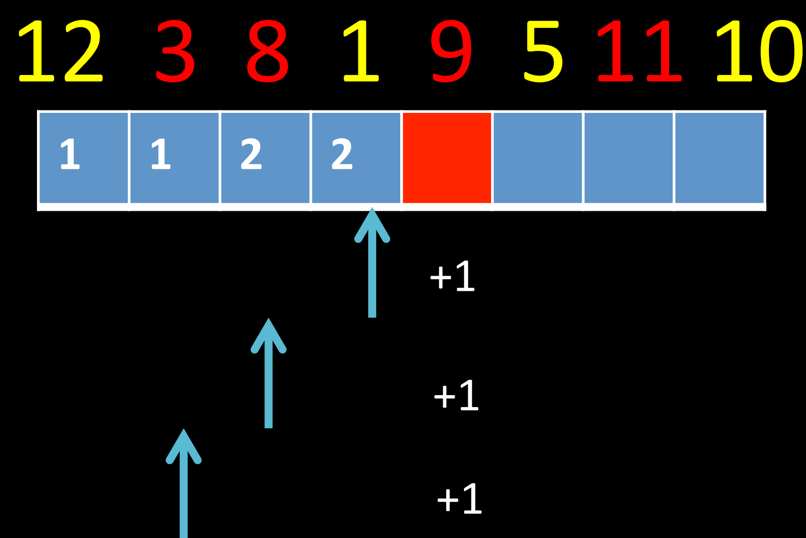
- A technique to forget DP states systematically to allow for fast running time
  - (1) Sample only part of the DP table for computation
  - (2) For computing DP(i,j) consider fewer subproblems





Amnesic DP for Longest Increasing Subsequence

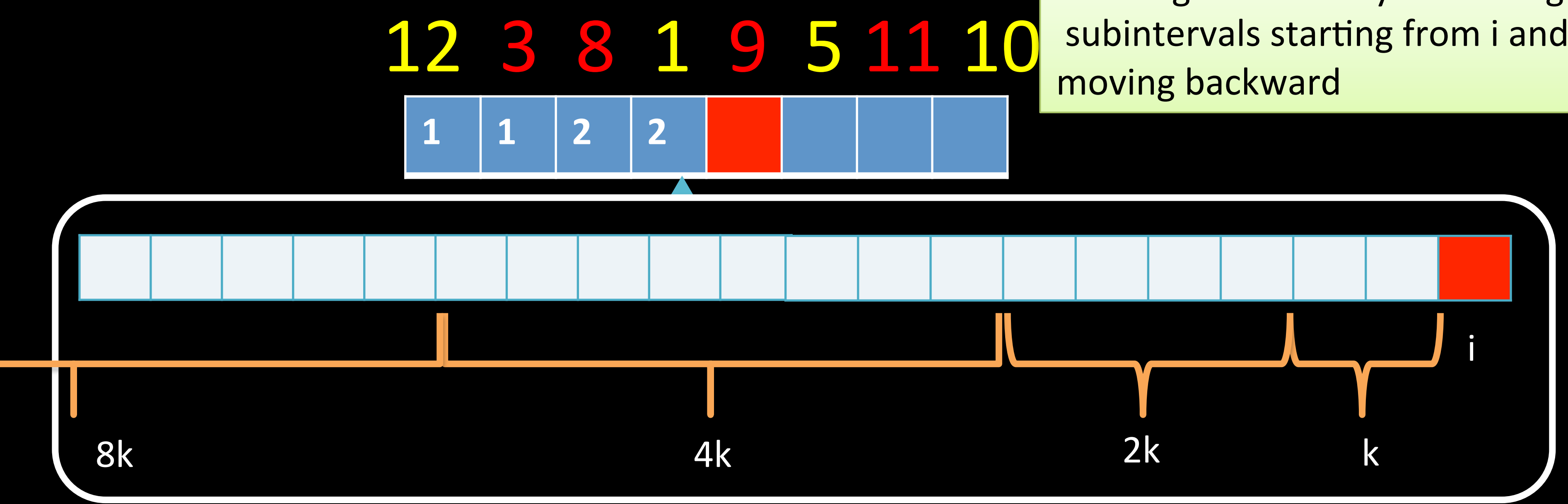
- Dynamic Programming
  - LIS[1]=1
  - $\text{LIS}[i]=\max_{j<l:s[j]<s[i]} \text{LIS}[j]+1$



Amnesic DP for Longest Increasing Subsequence

- Dynamic Programming
  - $LIS[1]=1$
  - $LIS[i]=\max_{j<l:s[j]<s[i]} LIS[j]+1$

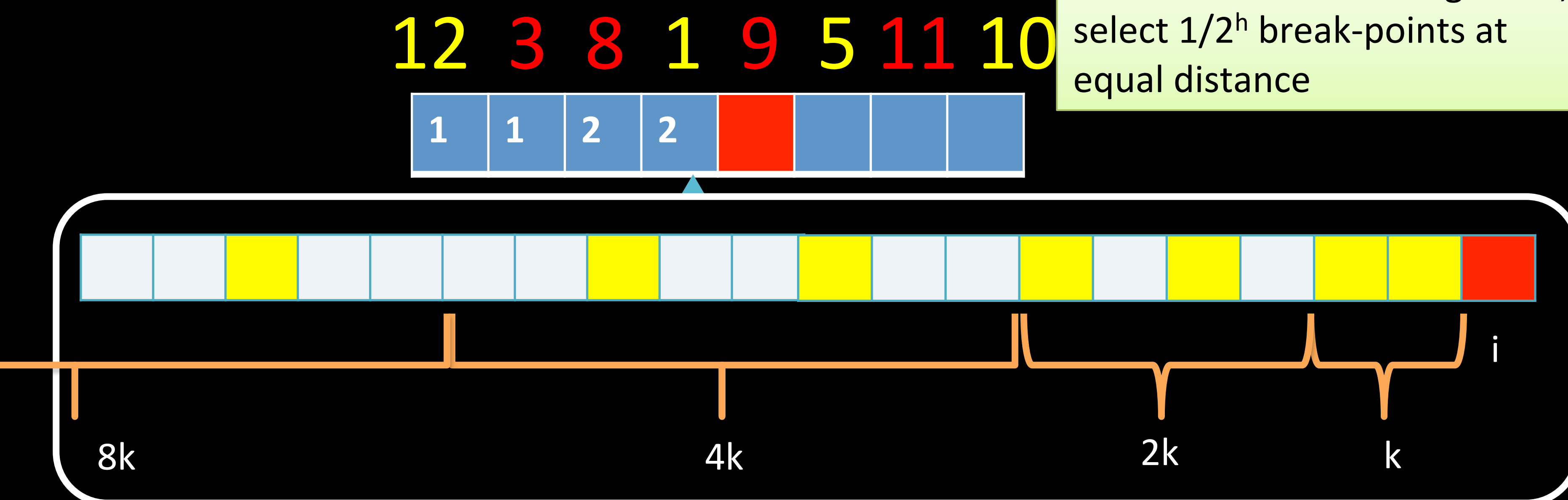
Create geometrically increasing subintervals starting from i and moving backward



Amnesic DP for Longest Increasing Subsequence

- Dynamic Programming
  - $LIS[1]=1$
  - $LIS[i]=\max_{j<l:s[j]<s[i]} LIS[j]+1$

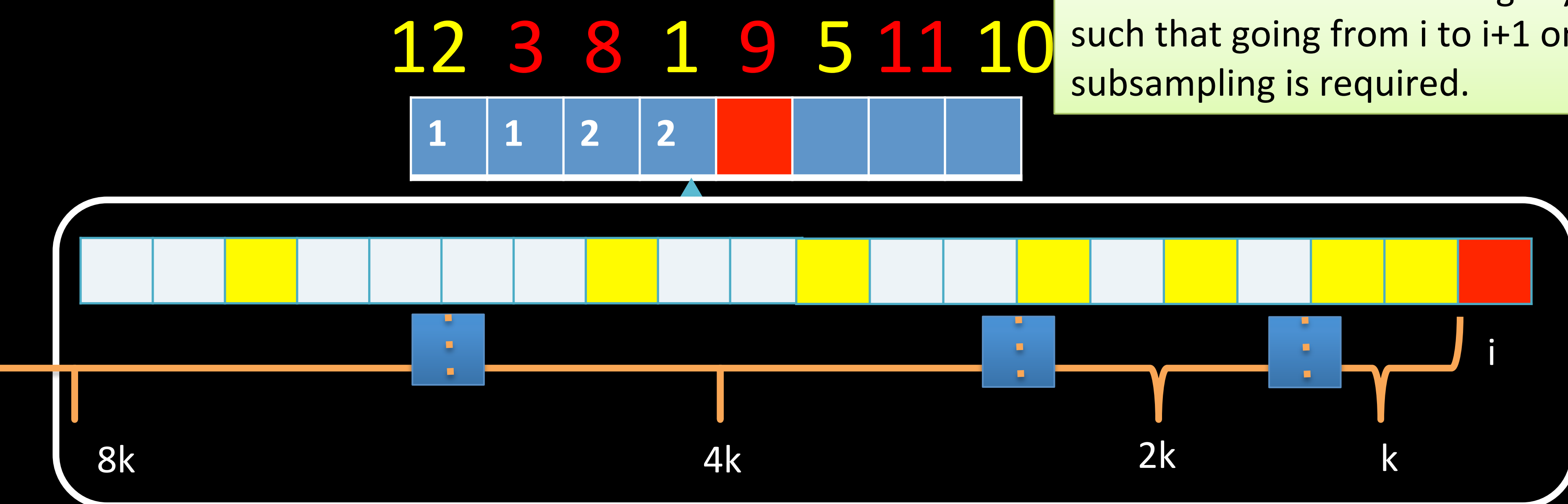
On a subinterval of length $2^h k$, select $1/2^h$ break-points at equal distance



Amnesic DP for Longest Increasing Subsequence

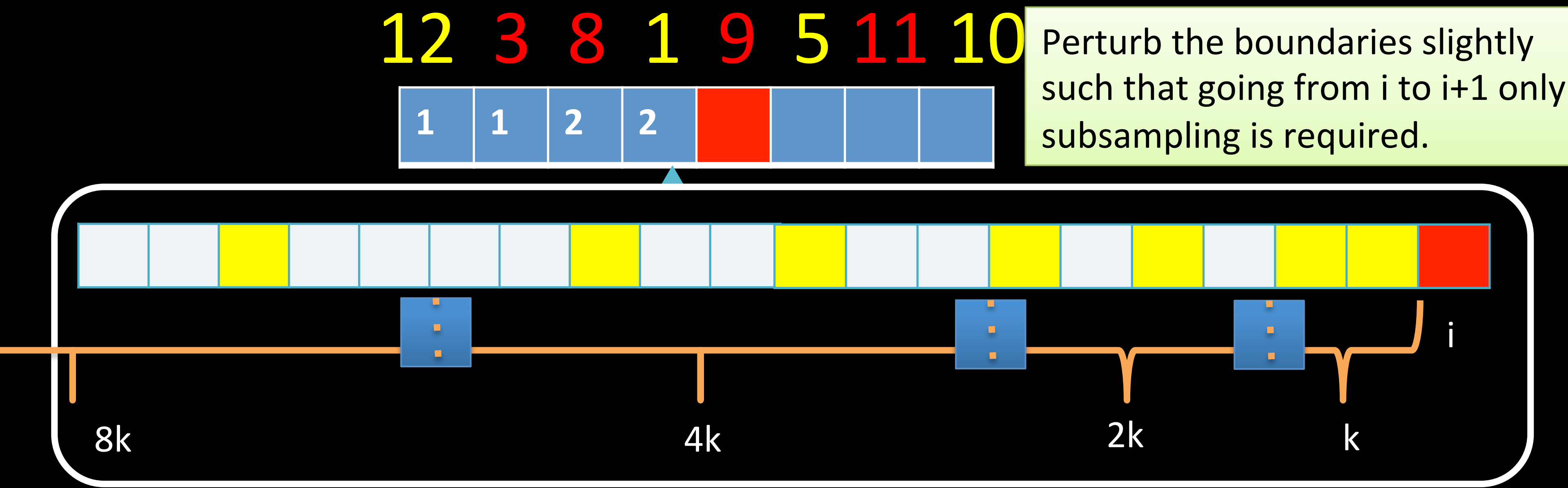
- Dynamic Programming
  - $LIS[1]=1$
  - $LIS[i]=\max_{j<l:s[j]<s[i]} LIS[j]+1$

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.



Amnesic DP for Longest Increasing Subsequence

- To show: $LIS^{approx}[i] \geq LIS[i] - \lfloor \frac{i}{k} \rfloor$
  - LIS[i]: Optimal LIS sequence for s[1,..i]
  - LIS^{approx}[i]: Approximate LIS sequence for s[1,…,i]
  - Cost[i]=i-LIS[i], Cost^{approx}[i]=i-LIS^{approx}[i]



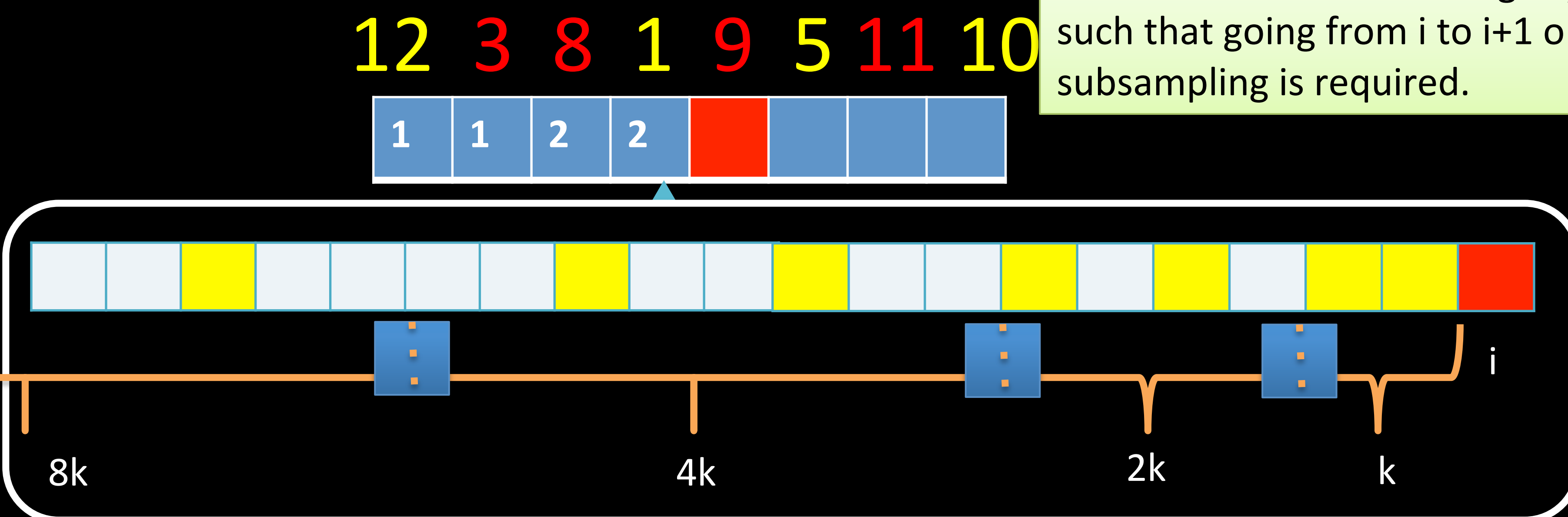
Amnesic DP for LIS

$$LIS^{approx}[i] \geq LIS[i] - \lfloor \frac{i}{k} \rfloor$$

$$\boxed{cost^{approx}[i] \leq cost[i] + \lfloor \frac{i}{k} \rfloor}$$

- Proof:
  - $i \leq k : LIS^{approx}[i] = LIS[i], cost^{approx}[i] = cost[i]$
  - Suppose true for $i \leq kT$

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.



Amnesic DP for LIS

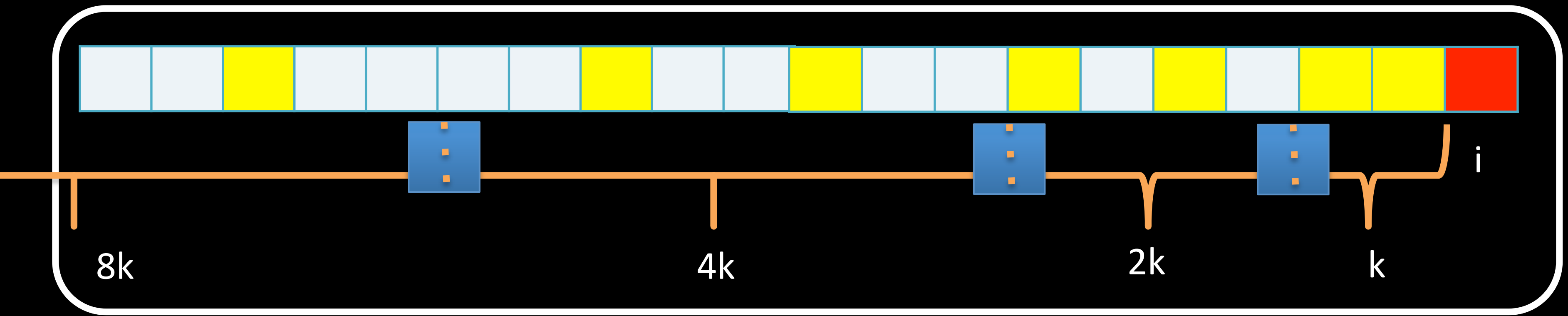
$$LIS^{approx}[i] \geq LIS[i] - \lfloor \frac{i}{k} \rfloor \quad \boxed{cost^{approx}[i] \leq cost[i] + \lfloor \frac{i}{k} \rfloor}$$

$$i \leq k : LIS^{approx}[i] = LIS[i], cost^{approx}[i] = cost[i$$

- Proof: Suppose true for $i \leq kT$
  - Take $i \in [kT+1, k(T+1))]$
  -
  -

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.

Let l be the optimum break-point considered.



Amnesic DP for LIS

$$\boxed{cost^{approx}[i] \leq cost[i] + \lfloor \frac{i}{k} \rfloor}$$

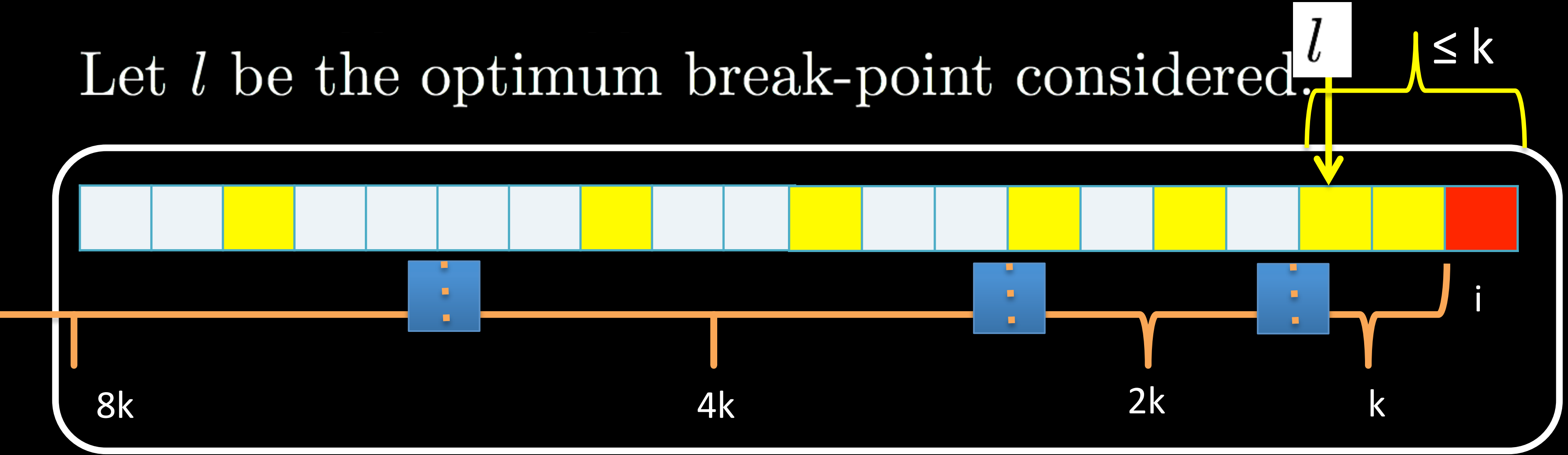
- Proof:
  - Take $i \in [kT+1, k(T+1)]$

If $(i - l + 1) \leq k$:

Move to $LIS[l]$ and $LIS^{approx}[l]$

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.

Let l be the optimum break-point considered.



Amnesic DP for LIS

$$\boxed{cost^{approx}[i] \leq cost[i] + \lfloor \frac{i}{k} \rfloor}$$

- Proof:

Take $i \in [kT+1, k(T+1)]$

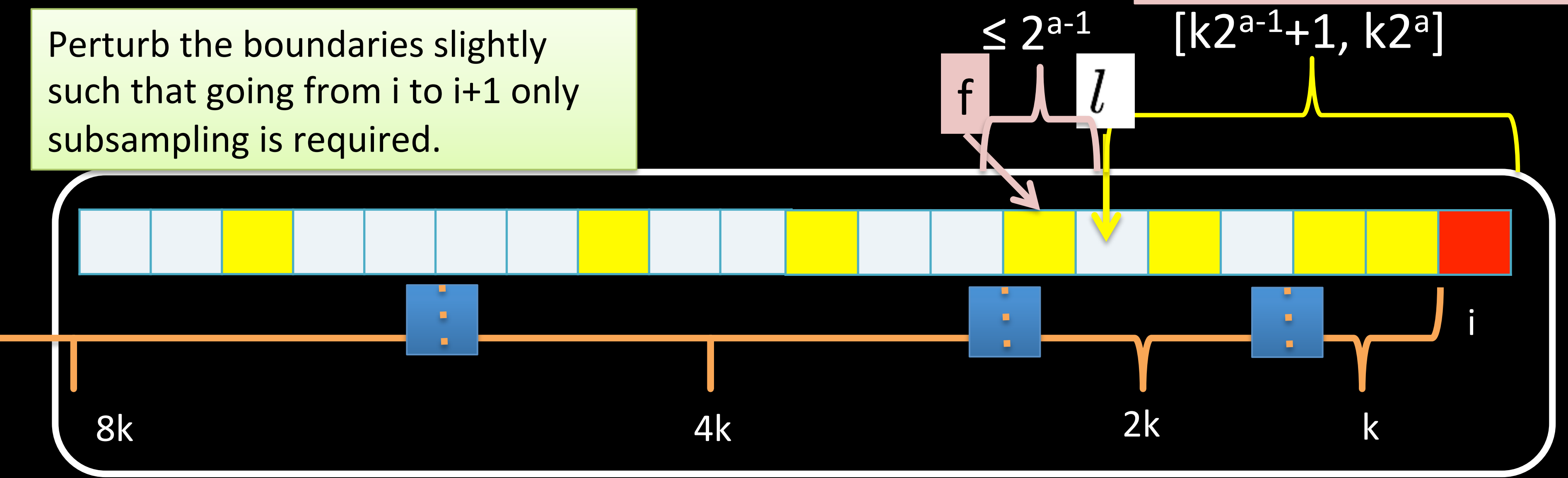
Let l be the optimum break-point considered.

f is the sampled break-point nearest to the left

Let $k2^{a-1} + 1 \leq (i - l + 1) \leq k2^a$

Sample every 2^{a-1}th point

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.



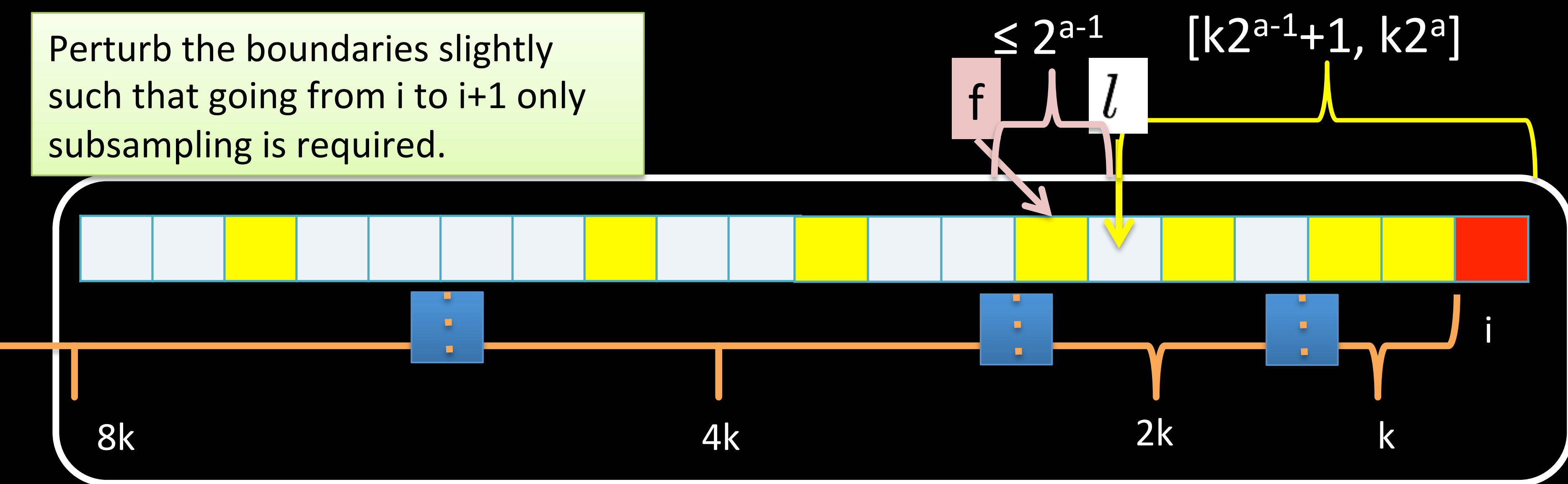
Amnesic DP for LIS

$$\boxed{cost^{approx}[i] \leq cost[i] + \lfloor \frac{i}{k} \rfloor}$$

- Proof:
  - Take $i \in [kT+1, k(T+1)]$
  - By induction hypothesis

$$cost^{approx}[l] \leq cost[l] + \lfloor \frac{l}{k} \rfloor$$

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.



Amnesic DP for LIS

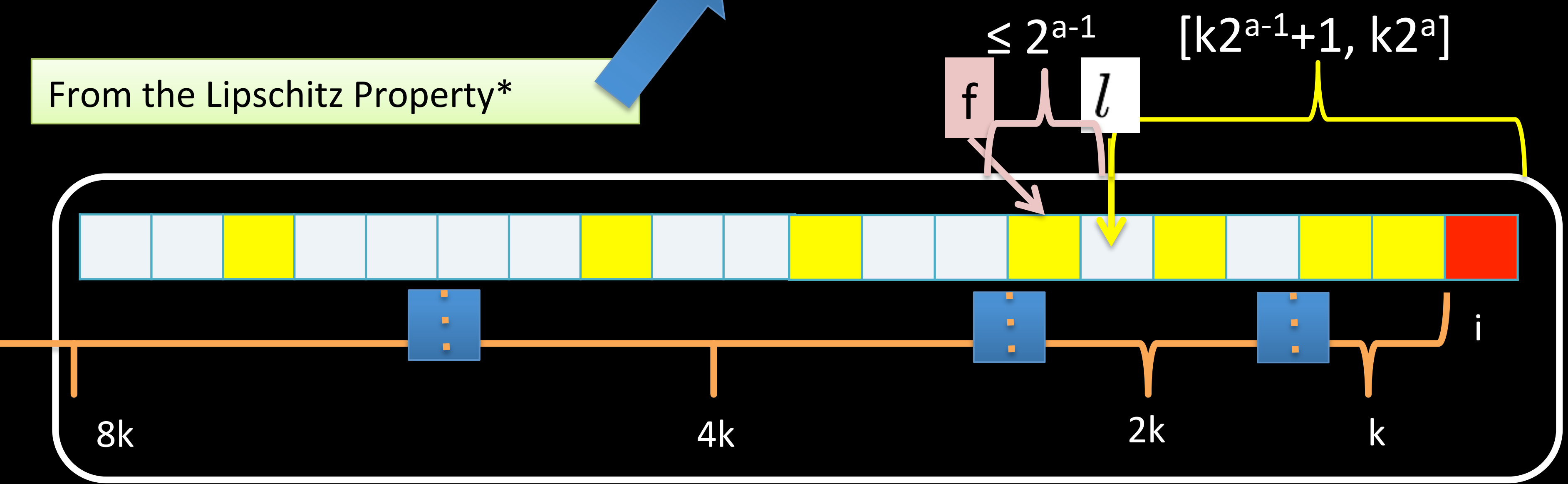
$$\boxed{cost^{approx}[i] \leq cost[i] + \lfloor \frac{i}{k} \rfloor}$$

- Proof:
  - Take $i \in [kT+1, k(T+1)]$
  - $cost^{approx}[l] \leq cost[l] + \lfloor \frac{l}{k} \rfloor$
  - $LIS^{approx}[l] \leq LIS^{approx}[f] + (l - f)$

*Does not hold quite because of the subtleties of the LIS DP.

But would hold if we randomly sample the points with a rate higher by a log n factor.

From the Lipschitz Property*

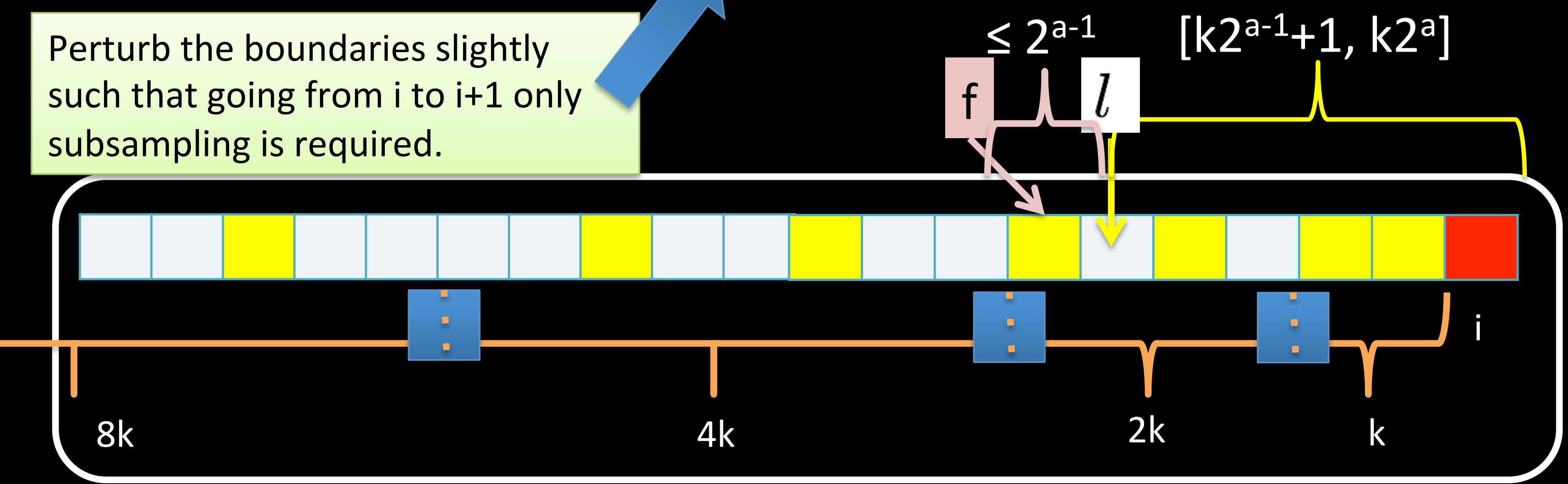


Amnesic DP for LIS

$$\boxed{cost^{approx}[i] \leq cost[i] + \lfloor \frac{i}{k} \rfloor}$$

- Proof:
  - Take $i \in [kT+1, k(T+1)]$
  - $cost^{approx}[l] \leq cost[l] + \lfloor \frac{l}{k} \rfloor$
  - $cost^{approx}[f] \leq cost^{approx}[l]$

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.

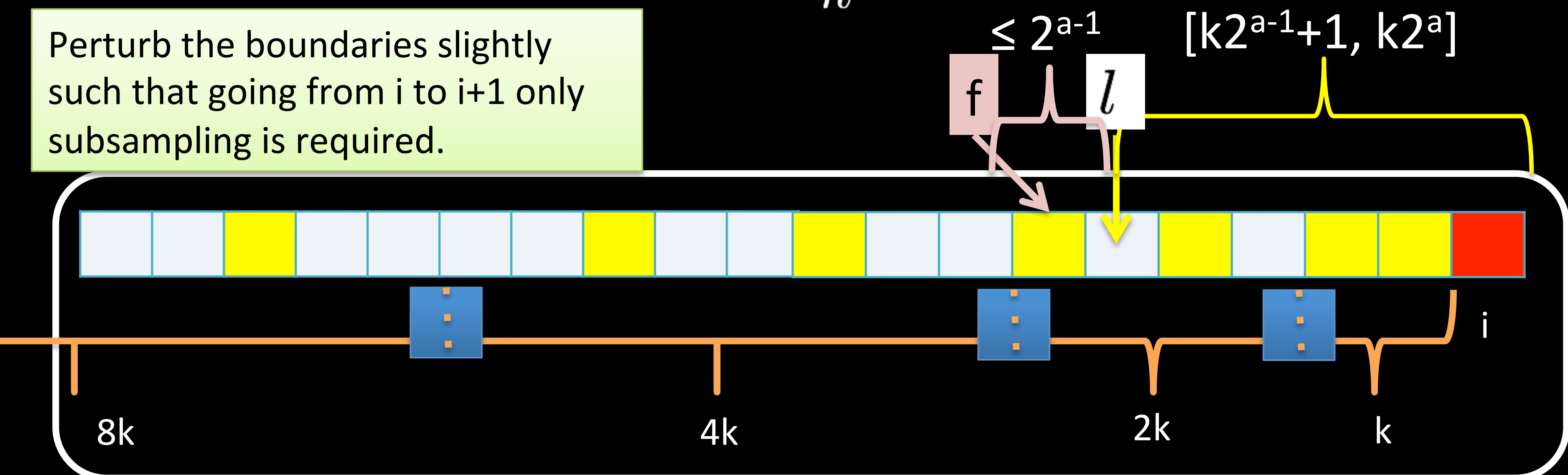


Amnesic DP for LIS

$$\boxed{cost^{approx}[i] \leq cost[i] + \lfloor \frac{i}{k} \rfloor}$$

$$\begin{aligned} cost^{approx}[i] &\leq (i-f) + cost^{approx}[f] \\ &\leq (i-f) + cost^{approx}[l] \\ &\leq (i-l) + (l-f) + cost[l] + \lfloor \frac{l}{k} \rfloor \\ &\leq cost[i] + \lfloor \frac{i}{k} \rfloor \end{aligned}$$

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.



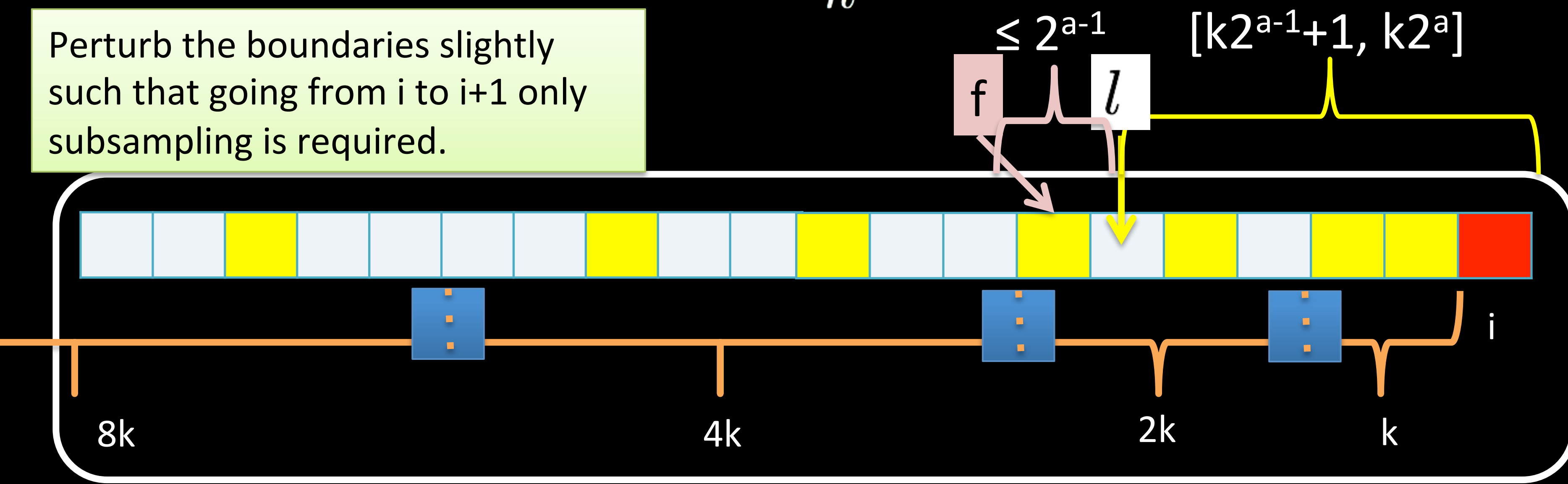
Amnesic DP for LIS

$$\boxed{cost^{approx}[i] \le cost[i] + \lfloor \frac{i}{k} \rfloor}$$

$$\begin{aligned} cost^{approx}[i] &\le (i-f) + cost^{approx}[f] \\ &\le (i-f) + cost^{approx}[l] \\ &\le (i-l) + (l-f) + cost[l] + \lfloor \frac{l}{k} \rfloor \\ &\le cost[i] + \lfloor \frac{i}{k} \rfloor \end{aligned}$$

$$\lfloor \frac{i-l}{k} \rfloor$$

Perturb the boundaries slightly such that going from i to i+1 only subsampling is required.



Amnesic DP for LIS

$$\boxed{cost^{approx}[i] \leq cost[i] + 2\lfloor \frac{i}{k} \rfloor}$$

$$cost^{approx}[i] \leq (i - f) + cost^{approx}[f]$$

$$\lfloor \frac{i-l}{k} \rfloor$$

$$\text{Space usage} = O(k \log \frac{n}{k})$$

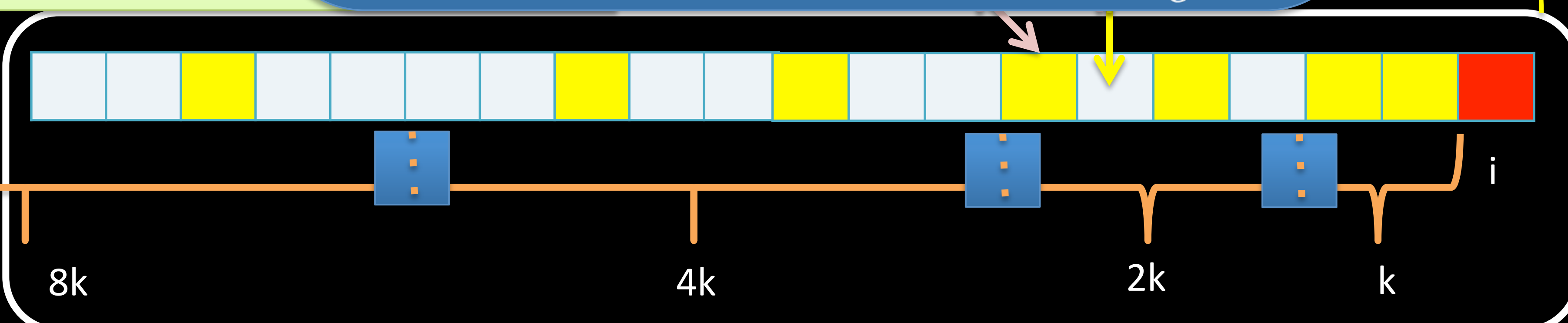
$$\text{Time usage} = O(nk \log \frac{n}{k})$$

$$\text{Additive Approximation} = \frac{n}{k}$$

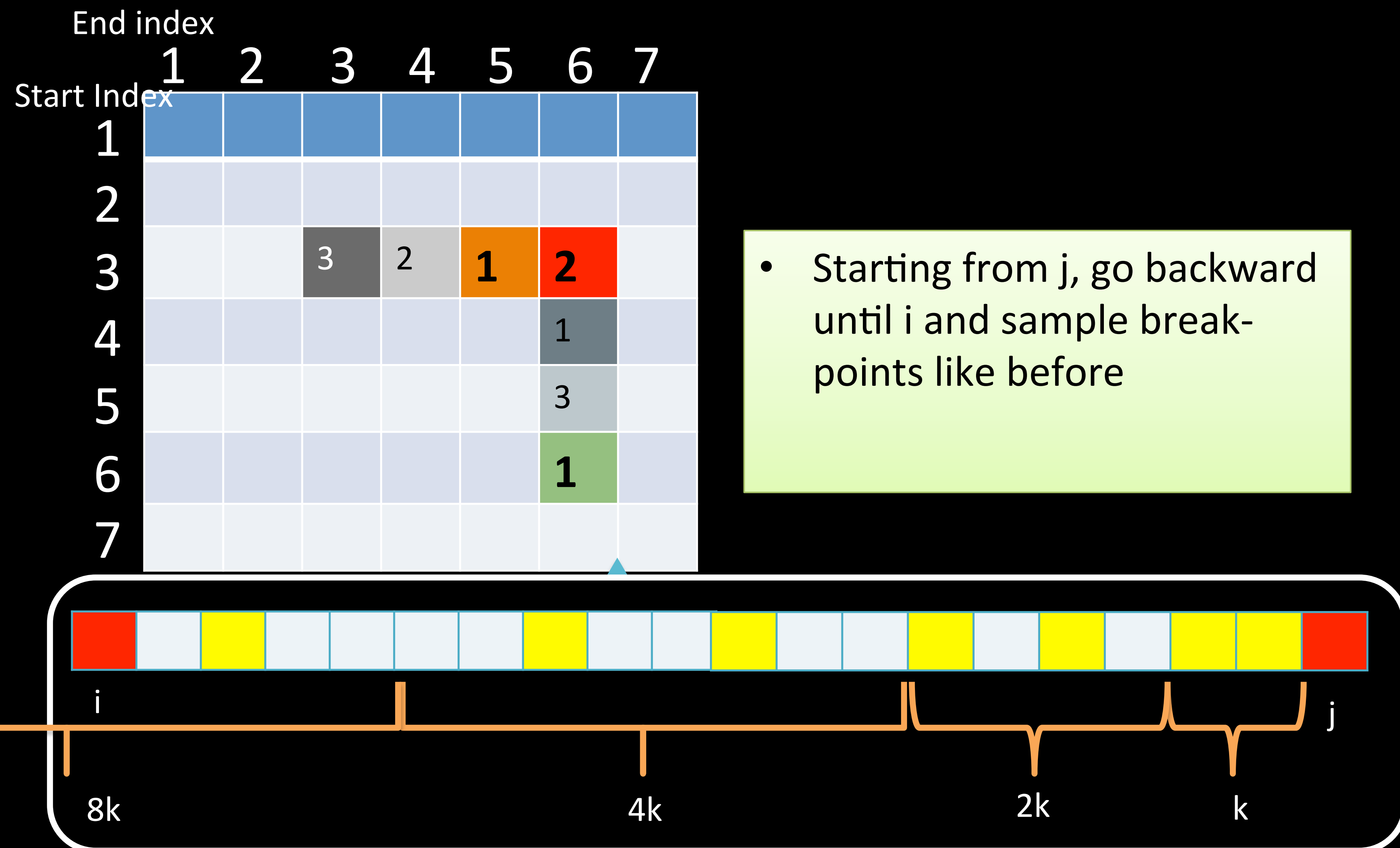
$$\text{Set } k = \frac{1}{\epsilon}$$

Perturb the bou
such that going
subsampling is re

k2a]

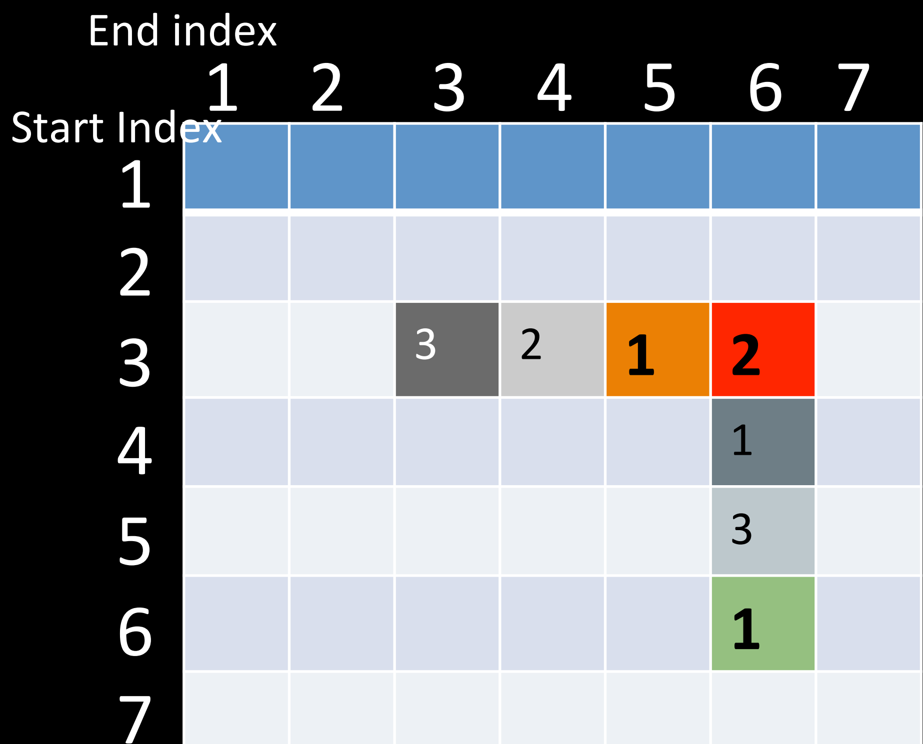


Amnesic DP for Language Edit Distance

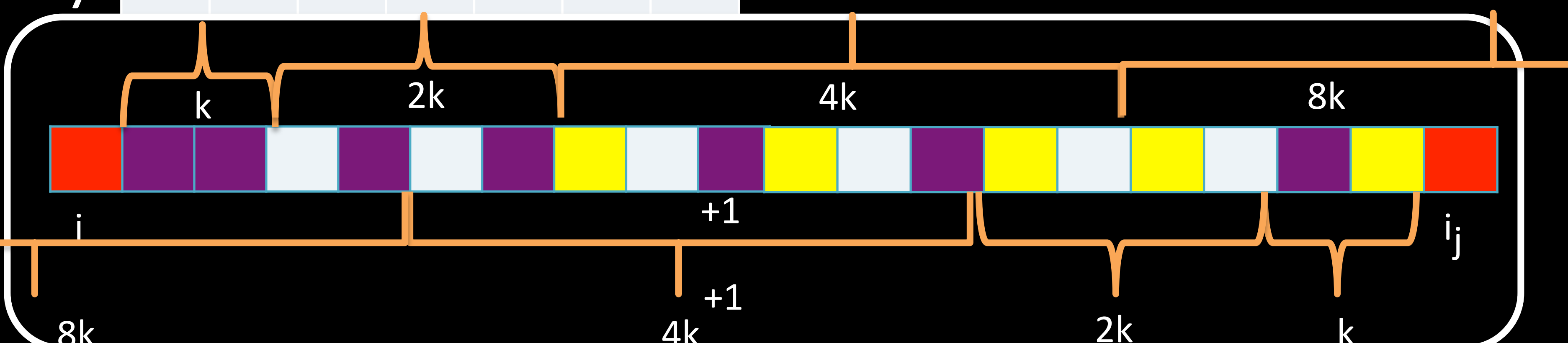


- Starting from j, go backward until i and sample break-points like before

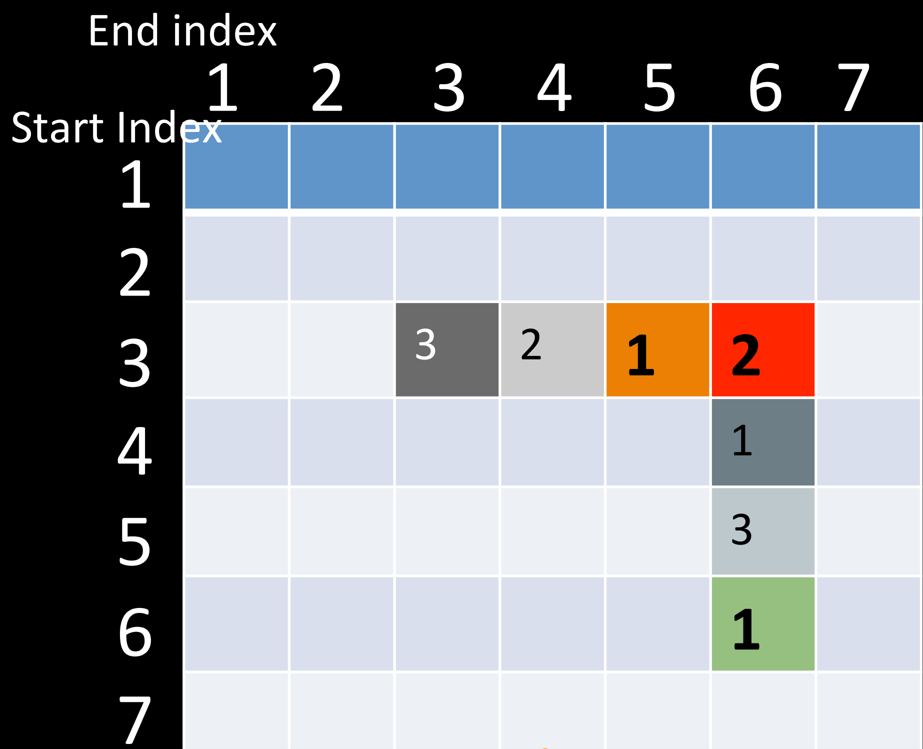
Amnesic DP for Language Edit Distance



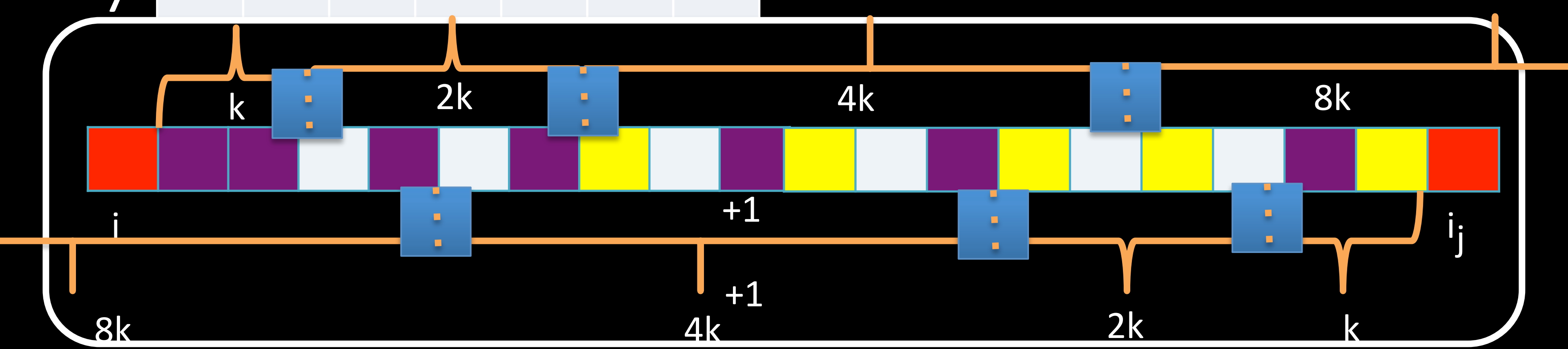
- Perturb the boundaries slightly such that going from (i’,j’) to (i,j) where (i’,j’) is a subinterval of (i,j), the algorithm only subsamples break-points within (i’,j’)



Amnesic DP for Language Edit Distance



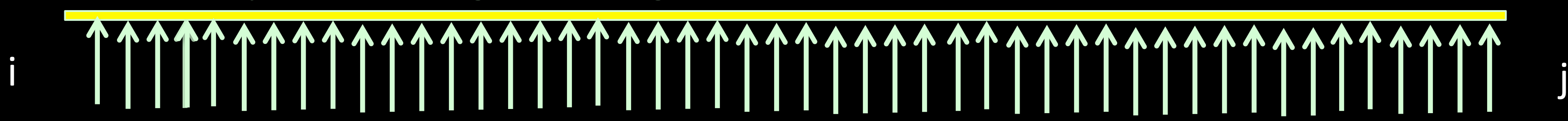
- Perturb the boundaries slightly such that going from (i',j') to (i,j) where (i',j') is a subinterval of (i,j), the algorithm only subsamples break-points within (i',j')



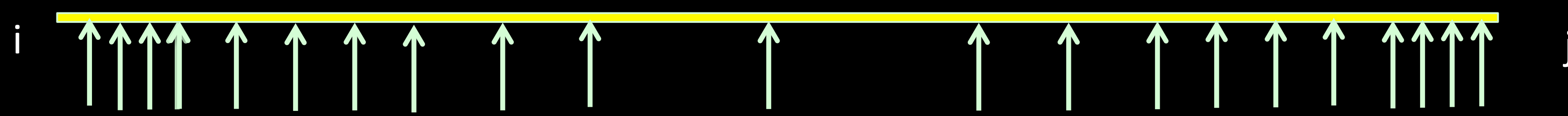
Amnesic DP for Language Edit Distance

- Language Edit Distance
  - Amnesic DP: from O(n) subproblems to O(klog(n))

Exact Dynamic Programming



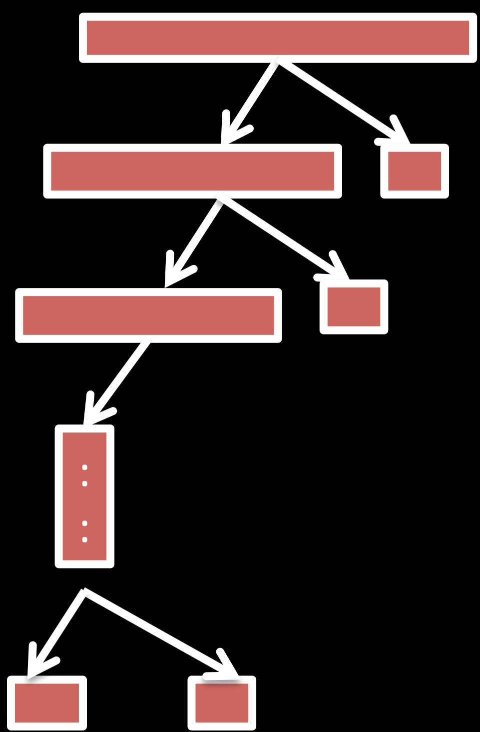
Amnesic Dynamic Programming



Sparsified in the middle

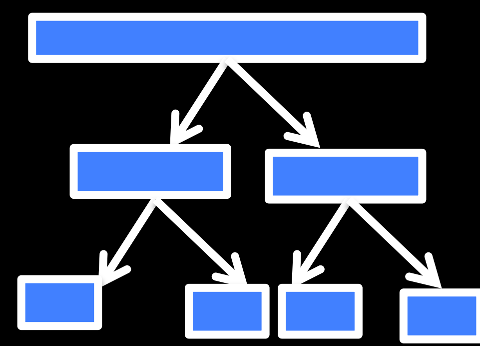
Understanding the Intuition Behind Break Point Selection

- Long recursion



Cannot make mistake either
 in the short or in the long substrings

Short Recursion



Can make mistake on the long substrings
 but not on the short substrings—too many
of them

Understanding the Intuition Behind Break Point Selection

- Long recursion

Short Recursion

Let P(i,j) denote the optimum recursion tree for s(i,i +1,…,j) and $P^{approx}(i,j)$ denote the approximate recursion tree computed by us.

Cost(P(i,j)): Total edit cost paid by P(i,j)
Cost($P^{approx}(i,j)$): Total edit cost paid by $P^{approx}(i,j)$

gs
many

Cannot make mistake either
in the short or in the long substrings

Understanding the Intuition Behind Break Point Selection

- Long recursion

Short Recursion

$$cost(P^{approx}[i,j]) \leq cost(P[i,j]) + 4 \sum_{v \in \text{ internal nodes of } P(i,j)} \min\left(\lfloor \frac{w(v_L)}{k} \rfloor, \lfloor \frac{w(v_R)}{k} \rfloor\right)$$

ings

many

$$\sum_{v \in \text{ internal nodes of } P(i,j)} \min\left(\lfloor \frac{w(v_L)}{k} \rfloor, \lfloor \frac{w(v_R)}{k} \rfloor\right) \leq \frac{(j-i+1)}{k} \log(j-i+1)$$

Cannot make mistake either
in the short or in the long substrings

Understanding the Intuition Behind Break Point Selection

- Long recursion

Short Recursion

$$cost(P^{approx}[i,j]) \leq cost(P[i,j]) + 4 \sum \min\left(\lfloor \frac{w(v_L)}{k} \rfloor, \lfloor \frac{w(v_R)}{k} \rfloor\right)$$

Theorem 6. *Given a parameter $k \geq 1$, there exists an algorithm which for any grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$, and $\sigma \in \Sigma^*$ of $|\sigma| = n$, computes an $O(\frac{n}{k}\log n)$-additive approximation for LED in $O(n^2 k \log n)$ time and $O(n^2)$ space.*

ings

many

$$\sum_{v \in \text{ internal nodes of } P(i,j)} \min\left(\lfloor \frac{w(v_L)}{k} \rfloor, \lfloor \frac{w(v_R)}{k} \rfloor\right) \leq \frac{(j-i+1)}{k} \log(j-i+1)$$

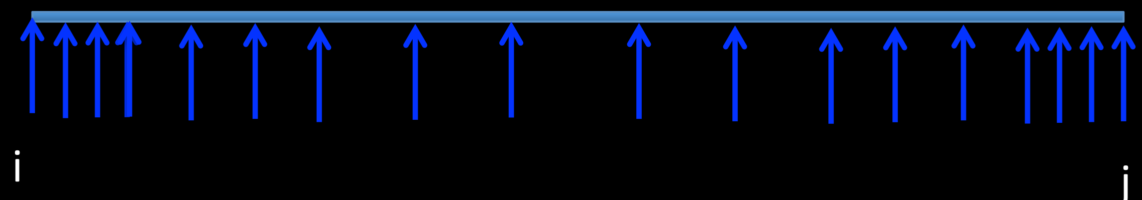
Cannot make mistake either
in the short or in the long substrings

Amnesic DP: Improving Space Complexity

- Improved time complexity → Improved space complexity
- Few subproblems to look at implies less space complexity
- Locality among subproblems:
  - Store the solution for all subproblems (i,j) of length 2k
  - For **(j-i+1)=r>2k**: among the subproblems that are accessed to solve for **(i,j)**, keep only those which will be required by **(i,j+1)** and **(i-1,j)**. Also store all the solutions for **length r substrings**.
  - Space complexity: O(nk log n)

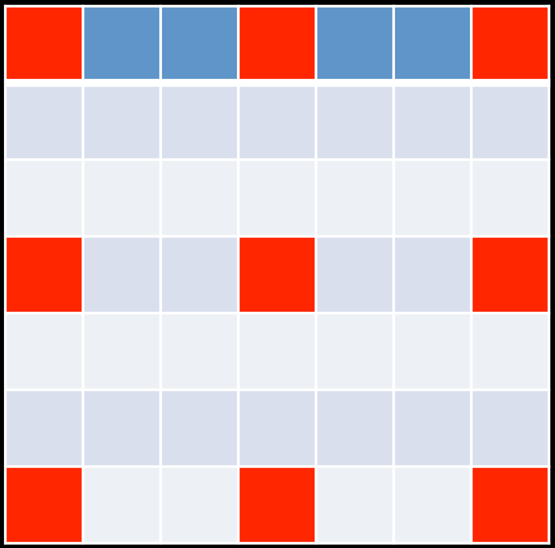
Amnesic DP for Sublinear Space Complexity

- Exact dynamic programming computes/stores solutions for every subproblem
  - Compute/Store solutions for only a subset of the entries



Break points considered by time and space efficient DP

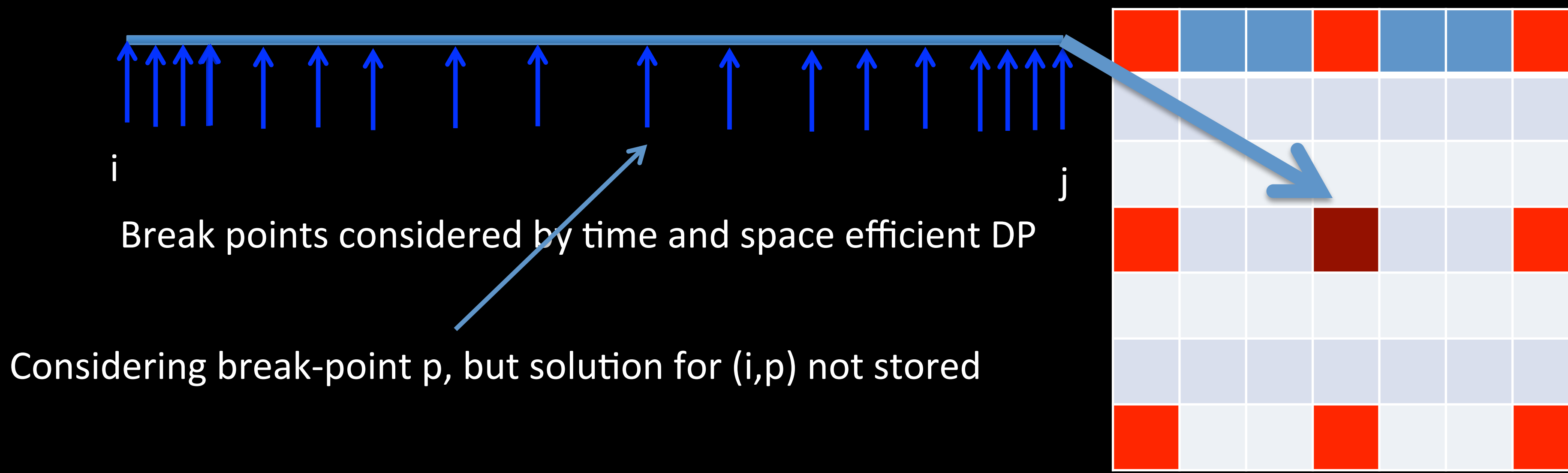
For small substrings: compute using auxiliary space with the time and space efficient DP



For long substrings: use the nearest DP value that is computed

Amnesic DP for Sublinear Space Complexity

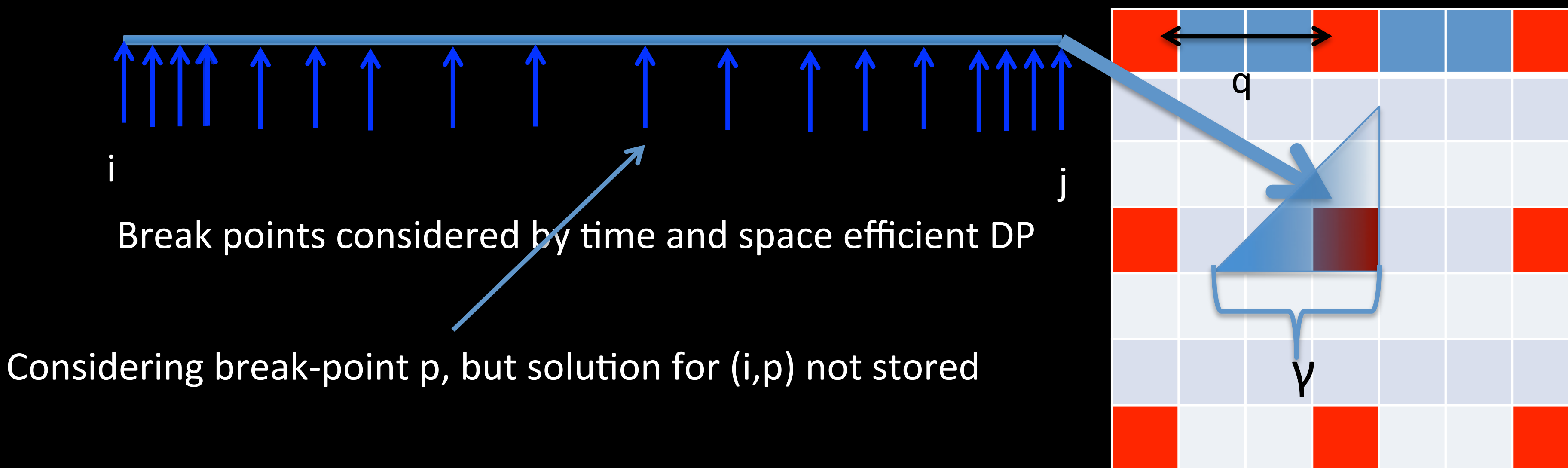
- Exact dynamic programming computes/stores solutions for every subproblem
  - Compute/Store solutions for only a subset of the entries



Length of (i,p) and (j,p+1) are large: select the l_1 nearest neighbor for which a solution has been computed and use that.

Amnesic DP for Sublinear Space Complexity

- Exact dynamic programming computes/stores solutions for every subproblem
  - Compute/Store solutions for only a subset of the entries



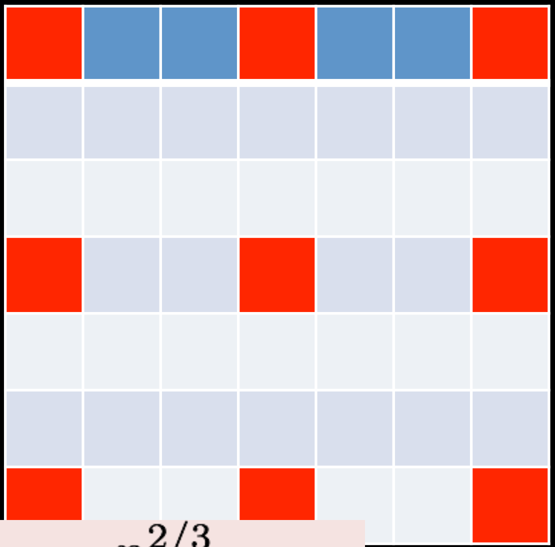
Length of (i,p) or (j,p+1) is small: Initialize the Δ with the nearest computed solutions and then recompute….

Amnesic DP for Sublinear Space Complexity

- Exact dynamic programming computes/stores solutions for every subproblem
  - Compute/Store solutions for only a subset of the entries

Theorem 2. *Given two parameters γ and q such that $\gamma > \sqrt{q} \geq 1$, there exist efficient algorithms for LED, RNA folding, and approximate CFG recognizer that use space $O(\max(\frac{n^2}{q}, \frac{\gamma^2 \log n}{\sqrt{q}}))$ and achieve an additive approximation of $O(\frac{n\sqrt{q}\log n}{\gamma})$.*

j



with the time and space efficient DP

Setting $\gamma = \dfrac{\sqrt{q}\log n}{\epsilon}$ we get ϵn-additive approximation in $\tilde{O}(\dfrac{n^{2/3}}{\epsilon^{4/3}})$space

Sublinear space algorithm even for additive approximation $n^{3/4+\delta}, \delta > 0$

From Sublinear Space to Parallel and Streaming Algorithms

- First Map-Reduce algorithm for Language Edit Distance and RNA Folding
  - Each machine stores only the computed entries of the dynamic programming table
  - Part of input
- Multi-pass streaming algorithm for Language Edit Distance and RNA Folding
- Better space vs approximation trade offs for linear grammar edit distance that generalizes string edit distance
  - Single pass algorithm for edit distance in asymmetric setting

What is the main difference?

Using Additive Approximation

Looks at many
at a t

onstant number
problems
a time

Longest Increasing Subsequence

Context Free Grammar Parsing

Language Edit Distance

RNA Folding

can improve both space and time complexity using amnesic dynamic programming

String Edit Distance

can improve space complexity using amnesic dynamic programming

What is the main difference?

Looks at many
at a t

Exact Solution

ooks at a
nstant number
problems
a time

Context Free Grammar Parsing

Language Edit Distance

RNA Folding

String Edit Distance

Using fast matrix multiplication can beat the dynamic programming

Conditional hardness rules out better running time

What is the main difference?

Looks at many subproblems at a time

can improve both space and time complexity using amnesic dynamic programming via additive approximation

Parsing

✓ Language Edit Distance

✓ RNA Folding

Using fast matrix multiplication can beat the dynamic programming

Looks at a constant number of subproblems

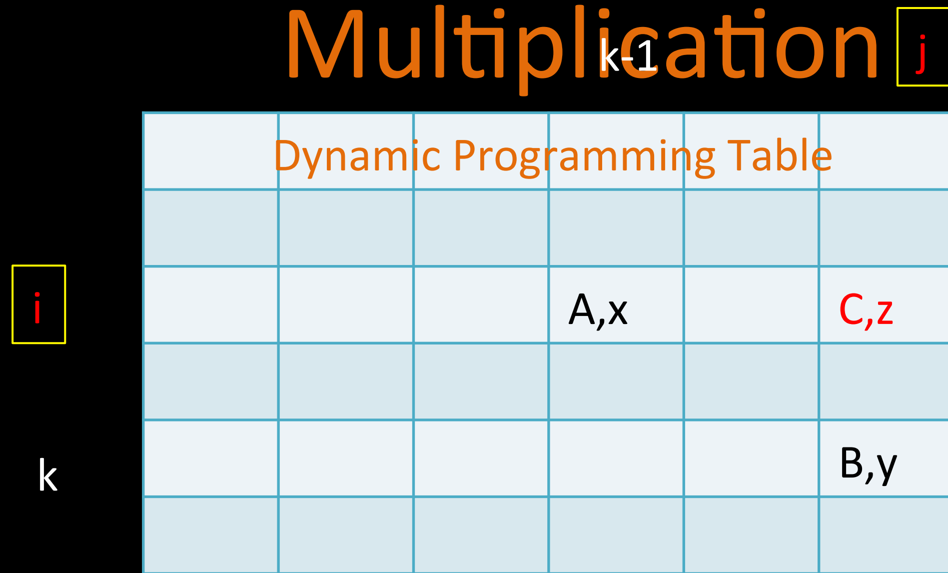
cannot improve time complexity using amnesic dynamic programming via additive approximation

String Edit Distance

Conditional hardness rules out better running time

From Dynamic Programming to Matrix Multiplication

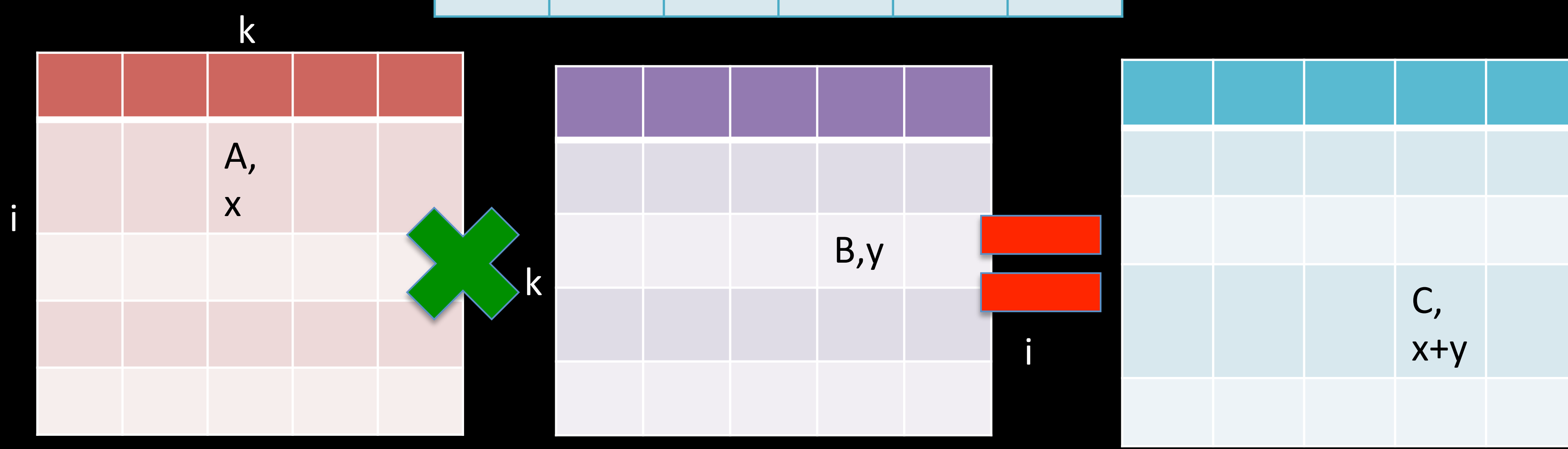
S→ AC
C→ AB
B→AR
A→AL
R→a
L→ b
R→b score(1)
L→a score(1)



C→AB and z=x+y

A derives s(i,k-1) with a score of x

B derives s(k,j) with a score of y



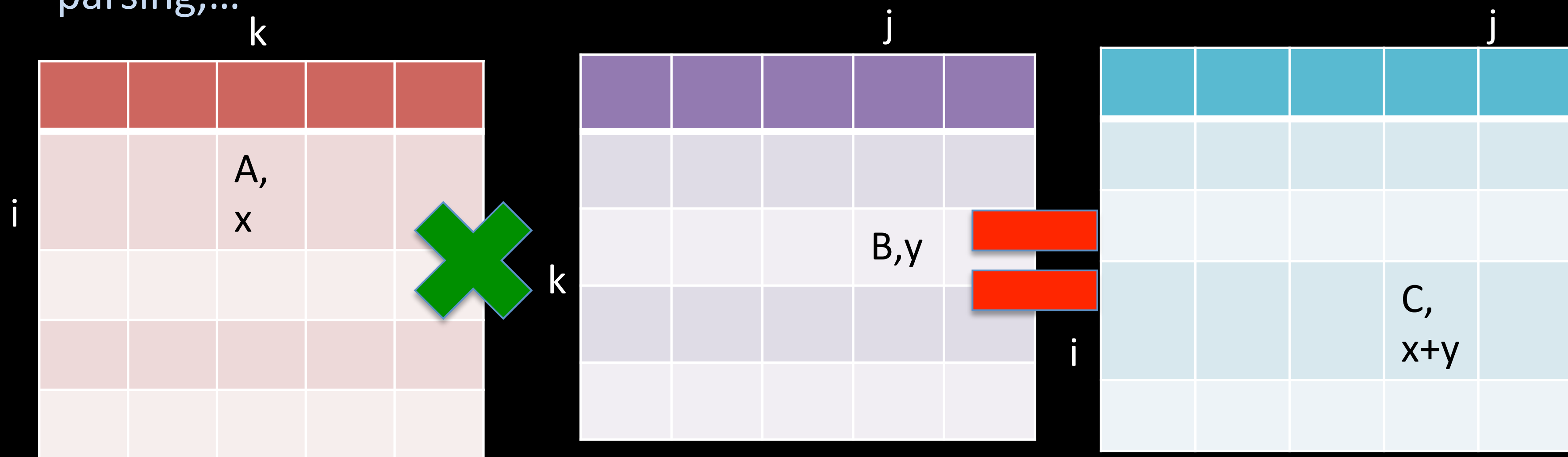
Use production C→AB

Compute Transitive Closure

To Matrix Multiplication [Saha, FOCS'15]

- Compute Transitive Closure Computation under a Special Matrix Product→(min,+)-product
- The product is nonassociative→computing transitive closure is difficult→keeping approximation error low is difficult

From cubic to $O(n^{2.327})$ time for $(1+\varepsilon)$-approximation LED, APSP, stochastic CFG parsing,…



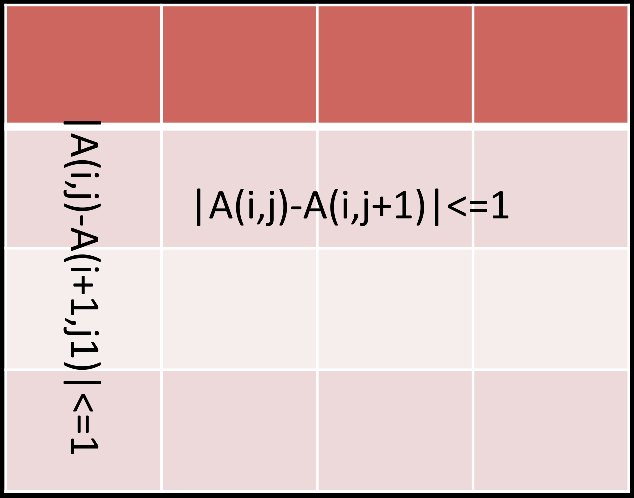
Use production C→AB

Can we develop an exact fast algorithm for LED?

- Conditional Lower Bound: A subcubic exact algorithm for LED is weighted APSP hard if we allow only “insertion” as edit operation. [Saha FOCS’15]
- The lower bound breaks down when all three types of edits are allowed!!!

Can we develop an exact fast algorithm for LED?

– When all three types of edits are allowed, the corresponding (min,+)-product matrices have bounded difference property
  - No of edits required for substring s(i,j) and s(i,j+1) can not differ much—similarly for s(i,j) and s(i+1,j)



Can we develop an exact fast algorithm for LED?

- When all three types of edits are allowed, the corresponding (min,+)-product matrices have bounded difference property
  - No of edits required for substring s(i,j) and s(i,j+1) can not differ much—similarly for s(i,j) and s(i+1,j)

(min,+)-matrix product with bounded difference can be computed in truly subcubic time.

[Bringmann, Grandoni, Saha, V. Williams, FOCS'16]

Open Problems

- Improve the upper bounds
  - Amnesic Dynamic Programming---Running time below $O(n^2)$ would give new approximation results even for string edit distance
  - Improve the bound for Bounded-difference (min,+)-product towards a truly subcubic algorithm for integer APSP
  - Is Real APSP >> Integer APSP?
  - Higher Dimensional Amnesic DP for Lipschitz problems
- Lower bound for space and time complexity trade-off
- Understand the true complexity of LED

Towards an Exact fast algorithm for (min,+) product

- Will be a major breakthrough resulting in a huge number of graph and string problems to have faster running time.
- Technique for bounded difference matrices------ matrices with integer entries in [1,n] ?

How close are we?

- As long as the absolute difference in any one dimension for any one matrix is at most $n^{3-\omega-\varepsilon}$
- To start with there are n^3 triplets (i,k,j) that can contribute to (min,+)-product computation.
- A triplet (i,k,j) is relevant if $A(i,k)+B(k,j) \approx C(i,j)$ [an approximate value of the $(i,j)^{th}$ entry of the product matrix C]
- Even for two arbitrary matrices after subcubic amount of processing, we are only left with subcubic number of relevant triplets which we have not looked at.
- How do we find these few relevant triplets which are yet to be considered?