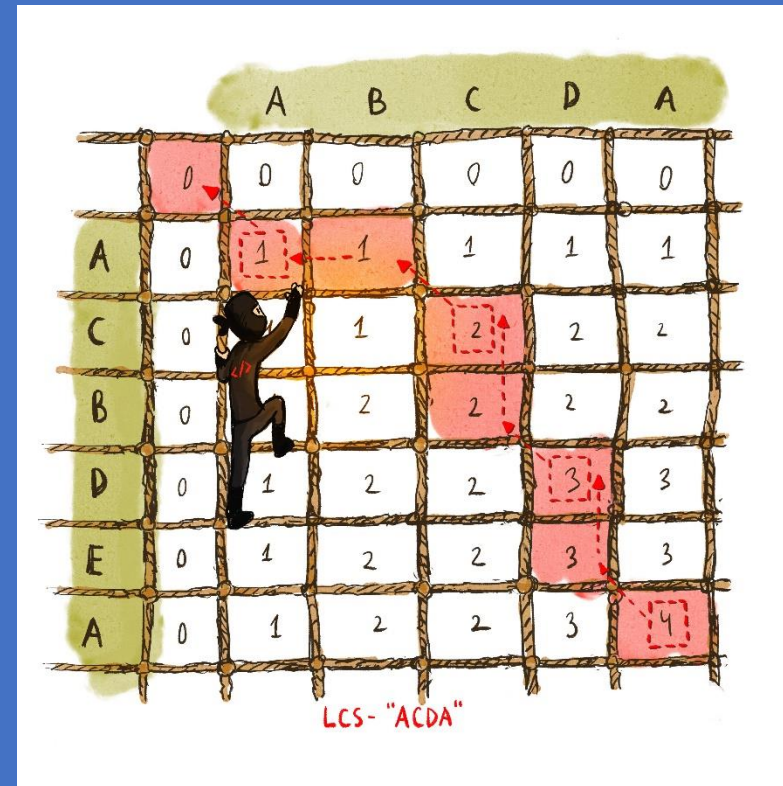


Dynamic Programming

Yan Gu



Minimum Edit Distance

- How to measure the similarity of words or strings?
- Auto corrections: “rationg” -> {“rating”, “ration”}
- Alignment of DNA sequences
- How many edits we need (at least) to transform a sequence X to Y?
 - Insertion
 - Deletion
 - Replace
- **rationg -> rating**
 - Delete o, edit distance 1
- **rationg -> action**
 - Delete r, add c, delete g
 - Edit distance 3

An Example of DNA sequence alignment

Human *LEP* gene
GTCACCAGGATCAATGACATTTACACACG - - TCAGTCTCCTCAAACAGAAAGTCACC
|||||
GTCACCAGGATCAATGACATTTACACACGCGAGTCGGTATCCGCCAAGCAGAGGGTCACT
Mouse *ob* gene
GGTTTGGACTTCA TTCTGGGCTCCACCCATCCTGACCTTATCCAAGATGGACCAGACA
||| |||||
GGCTTGGACTTCA TTCTGGGCTTCA CCCCATTCTGAGTTTGTCGAAGATGGACCAGACT

CTGGCAGTCTACCAACAGATCCTCACCAGTATGCCTTCCAGAAACGTGATCCA AATATCC
||||| |||||
CTGGCAGTCTATCAACAGGTCTCACCAGCTGCC TTCCCAAATGTGCTGCAGATA GCC



© 2010 Pearson Education, Inc.

Adapted from Klug p. 384

Determine the matching score.

Recurrence of Edit Distance

- Similar to LCS, consider the cost to transform $X[1..i]$ to $Y[1..j]$
- Look at the last character $X[i]$ and $Y[j]$
- What happens if $X[i] = Y[j]$?

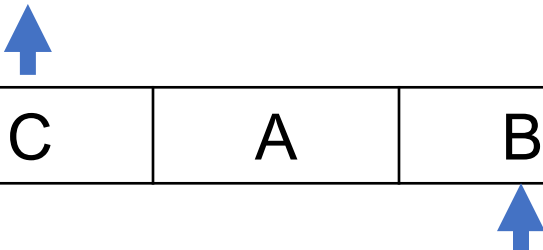
| | | | | | |
|---------|---|---|---|--|---|
| Index : | 1 | 2 | 3 | 4 | |
| X = | A | B | C | B | |
| | | | |  | |
| Y = | B | D | C | A | B |
| | | | |  | |

- Keep $X[i]$ and $Y[j]$ – no edit needed
- Need to transform ABC to BDCA
- $\rightarrow s[i-1, j-1]$

Recurrence of Edit Distance

- Similar to LCS, consider the cost to transform $X[1..i]$ to $Y[1..j]$
- Look at the last character $X[i]$ and $Y[j]$
- What happens if $X[i] \neq Y[j]$?

| | | | | | | |
|---------|---|---|---|---|---|--|
| Index : | 1 | 2 | 3 | | | |
| X = | A | B | C | | | |
| | | | | | | |
| Y = | B | D | C | A | B | |



- **Delete C.** Cost = (cost of transforming AB \Rightarrow BDCAB) + 1 $\rightarrow s[i-1, j] + 1$
- **Adding B.** Cost = (cost of transforming ABC \Rightarrow BDCA) + 1 $\rightarrow s[i, j-1] + 1$
- **Editing C to B.** Cost = (cost of transforming AB \Rightarrow BDCA) + 1 $\rightarrow s[i-1, j-1] + 1$
- Use the min of the above three!

Recurrence Relation

- $s[i, j]$: The cost of transforming $X[1..i]$ to $Y[1..j]$

$$s[i, j] = \begin{cases} \max\{i, j\} & ; i = 0 \vee j = 0 \\ s[i - 1, j - 1] & ; i > 0 \wedge j > 0 \wedge x_i = y_j \\ \min \begin{cases} s[i, j - 1] + 1 \\ s[i - 1, j] + 1 \\ s[i - 1, j - 1] + 1 \end{cases} & ; i > 0 \wedge j > 0 \wedge x_i \neq y_j \end{cases}$$

Recursive Algorithm

- ED(i, j)
 - If computed or base case then return the value
 - if $X[i] == Y[j]$
 - return ED(i-1, j-1)
 - if $X[i] != Y[j]$
 - return min(ED(i, j-1)+1,
ED(i-1, j)+1,
ED(i-1, j-1)+1)

$$s[i, j] = \begin{cases} \max\{i, j\} & ; i = 0 \vee j = 0 \\ s[i-1, j-1] & ; i > 0 \wedge j > 0 \wedge x_i = y_j \\ \min \begin{cases} s[i, j-1] + 1 \\ s[i-1, j] + 1 \\ s[i-1, j-1] + 1 \end{cases} & ; i > 0 \wedge j > 0 \wedge x_i \neq y_j \end{cases}$$

Designing a DP algorithm / recurrence

- **Step 1: find the correct (appropriate) subproblems (a polynomial number of the states/subproblems)**
- **Step 2: find the relationships between the subproblems**
- **Particular goals for CS 141 (Intermediate Data Structures and Algorithms)**
 - Understand the high-level ideas for dynamic programming
 - Understand the DP algorithms for three specific problems (knapsack, LCS, LIS)
 - Use the variance of these algorithms to solve some related questions
- **More related practice will be given in CS 218 (Spring 2021)**
 - More algorithm design practice
 - Optimizing dynamic programming algorithms
 - DP on trees, graphs, games, etc.

Longest Increasing Subsequence (LIS) and Other Similar Problems

What is an increasing subsequence?

4 **2** **7** **0** **1** **6** **3** **8** **5** **9**

- Increasing subsequence:

2 **6** **8** **9**

- Longest increasing subsequence (LIS):

0 **1** **3** **8** **9**

What is an increasing subsequence?

4 **2** **7** **0** **1** **6** **3** **8** **5** **9**

Why studying LIS?

- **The length of LIS reflect some intrinsic properties of the sequence**
 - Consider the length of LIS as the “eigenvalue” of a sequence (LIS as the “eigenvector”)
 - Applications in many algorithms and [quantum computing](#)
- **Many similar DP algorithms are similar to the DP algorithm for LIS**
 - More examples are given later in this lecture



What are the states for LIS?

- Let l_i be the longest LIS that ends at the i -th element
- What is the recurrence of LIS?

| | | | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 1 | 1 | 2 | 1 | 2 | 3 | 3 | 4 | 4 | 5 |
| 4 | 2 | 7 | 0 | 1 | 6 | 3 | 8 | 5 | 9 |

What are the states for LIS?

- Let l_i be the longest LIS that ends at the i -th element
- What is the recurrence of LIS?

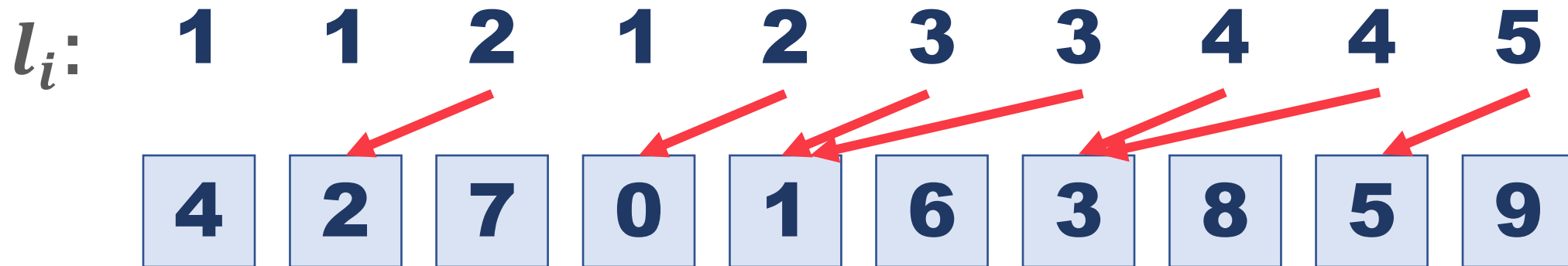
$$l_i = \max \begin{cases} 1 \\ \max_{0 < j < i, a_j < a_i} \{l_j + 1\} \end{cases}$$

- Why is it an optimal substructure?



Running the example input

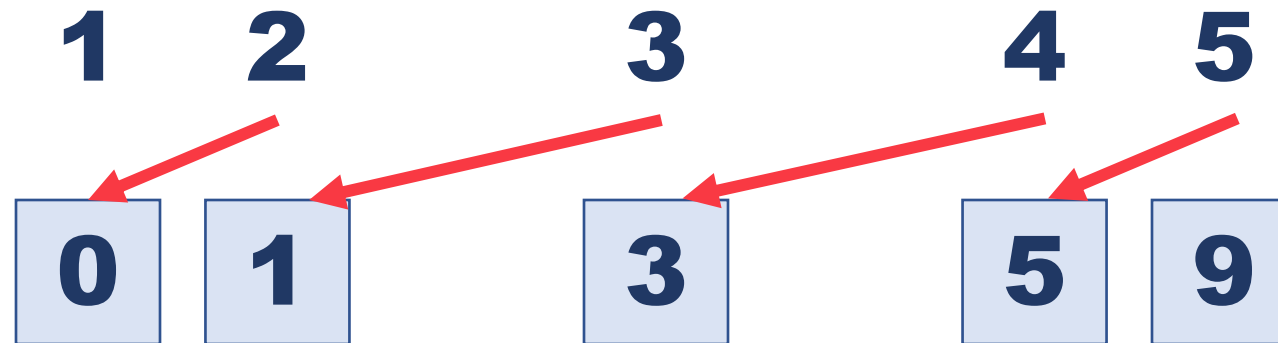
$$l_i = \max \begin{cases} 1 \\ \max_{0 < j < i, a_j < a_i} \{l_j + 1\} \end{cases}$$



Running the example input

$$l_i = \max \begin{cases} 1 \\ \max_{0 < j < i, a_j < a_i} \{l_j + 1\} \end{cases}$$

l_i :



What is the time complexity of LIS?

$$l_i = \max \left\{ \begin{array}{c} 1 \\ \max_{0 < j < i, a_j < a_i} \{l_j + 1\} \end{array} \right.$$

- n element, each takes $O(n)$ time to compute, so $O(n^2)$ cost in total
- Answer: $\max_{0 < j \leq n} \{l_j\}$
- LIS can be computed in $O(n \log n)$ time ([link](#))
 - Will be covered in CS 218: design and analysis of algorithms

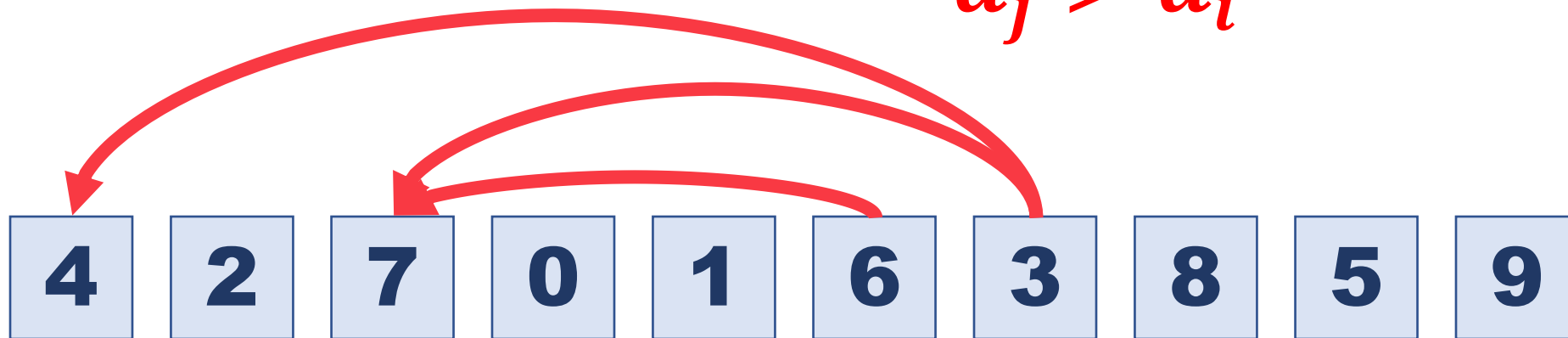
LIS and similar problems

$$l_i = \max \left\{ \begin{array}{c} 1 \\ \max_{0 < j < i, a_j < a_i} \{l_j + 1\} \end{array} \right.$$

Longest decreasing subsequence?

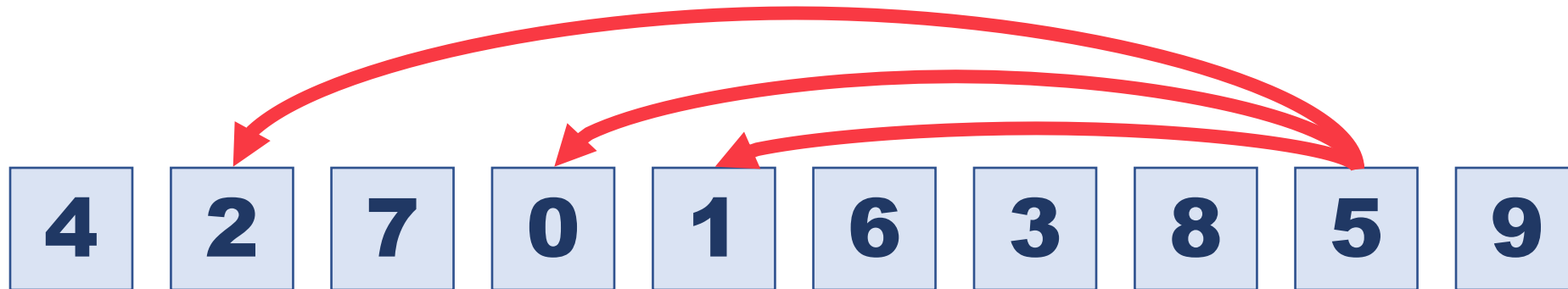
$$l_i = \max \left\{ \begin{array}{l} 1 \\ \max_{0 < j < i, \cancel{a_j < a_i}} \{l_j + 1\} \end{array} \right.$$

$a_j > a_i$



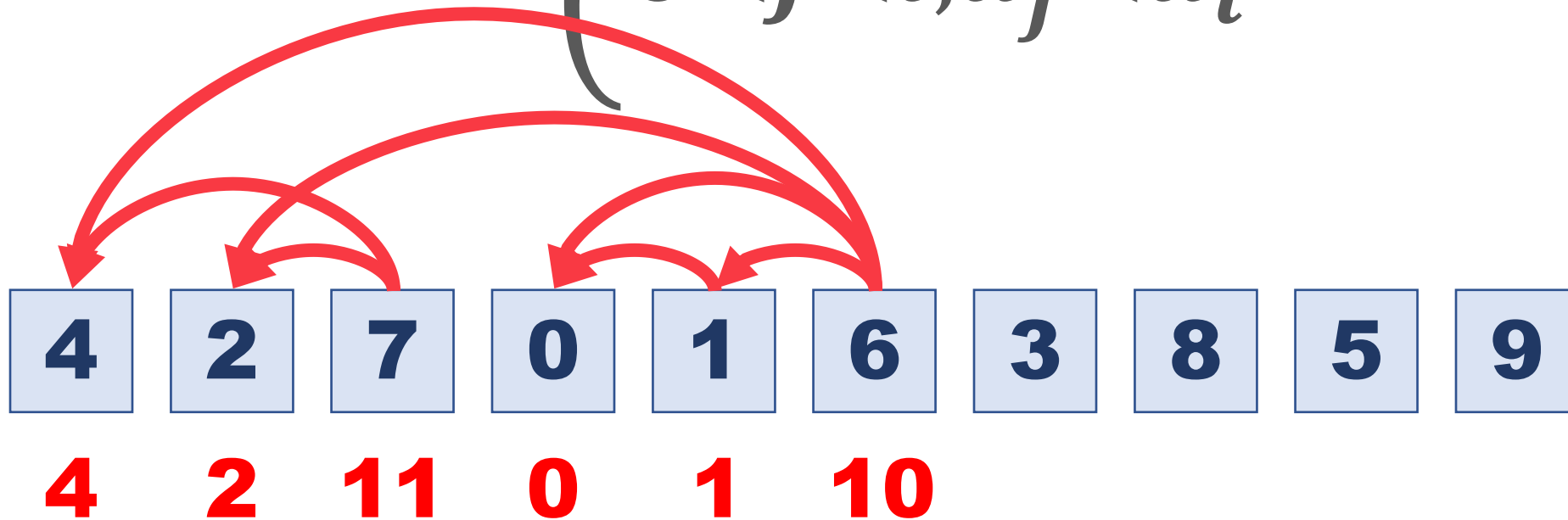
Longest increasing subsequence with gap ≥ 3 ?

$$l_i = \max \left\{ \begin{array}{l} 1 \\ \max_{0 < j < i, \cancel{a_j < a_i}} \{l_j + 1\} \\ a_j \leq a_i - 3 \end{array} \right.$$

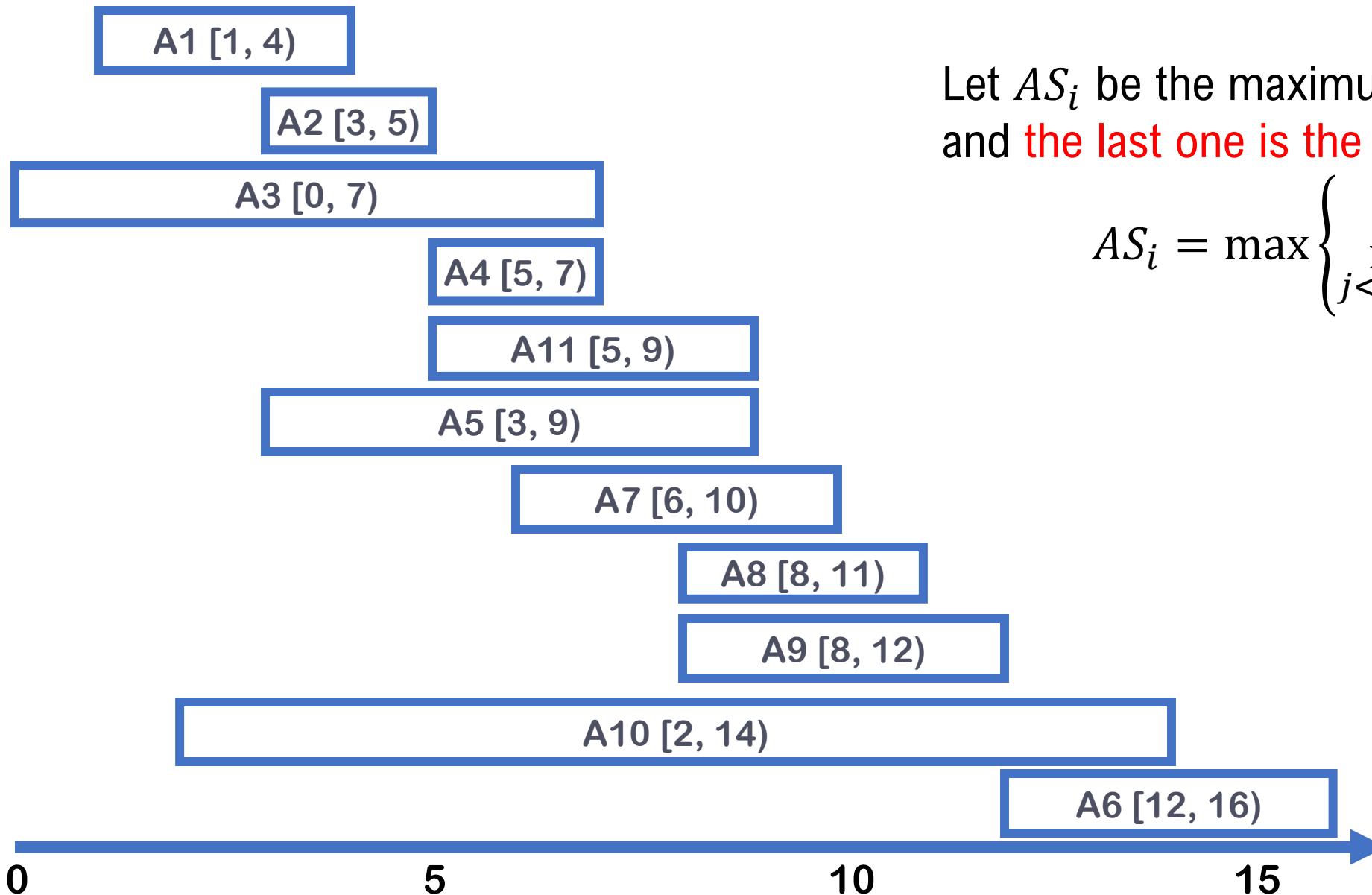


Increasing subsequence with MAX SUM?

$$l_i = \max \left\{ \max_{0 < j < i, a_j < a_i} \left\{ l_j + \cancel{1} \right\} \right. \quad \left. \cancel{1} + a_i \right. \\ \left. a_i \right\}$$



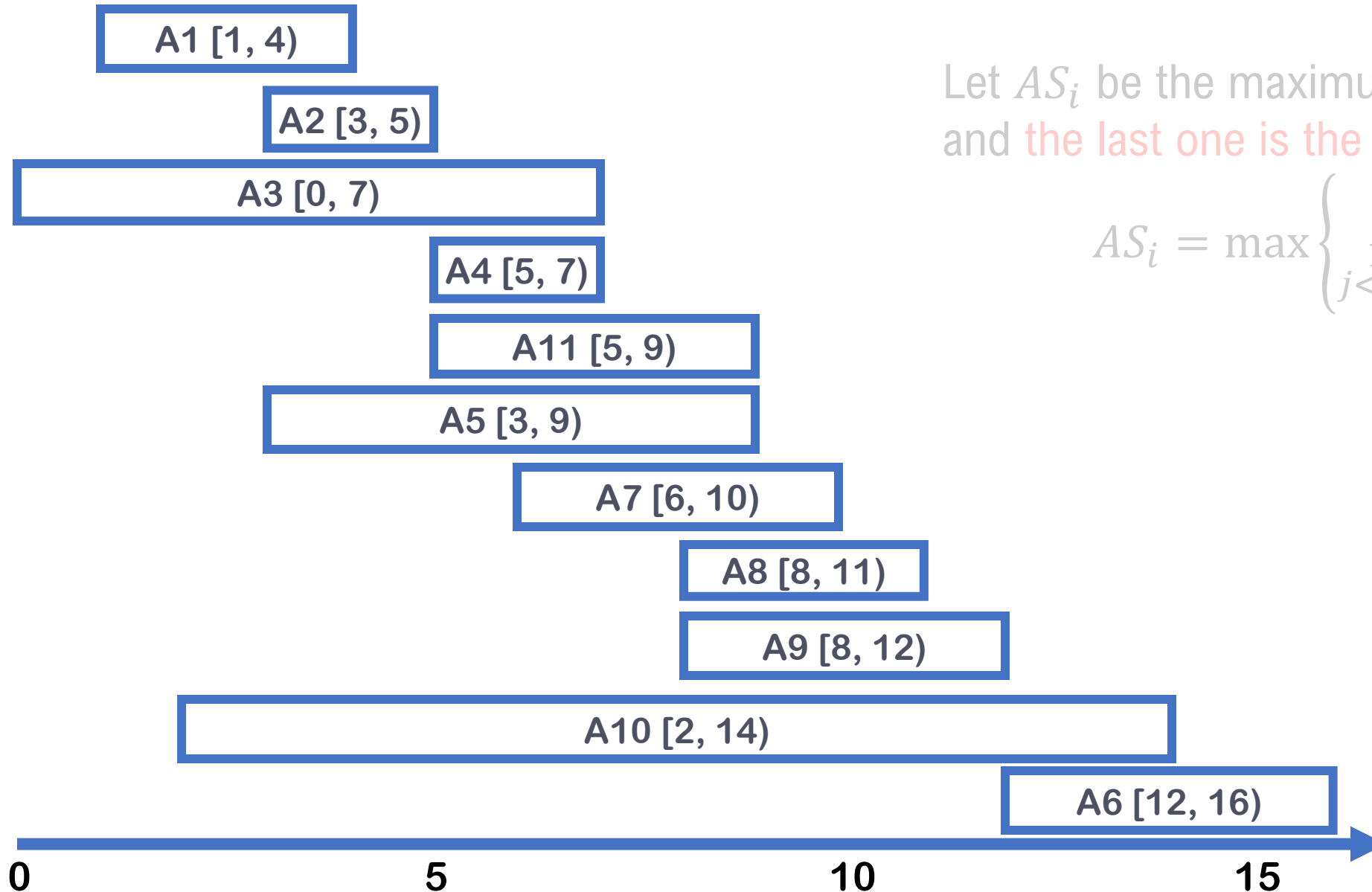
Revisit: activity selection



Let AS_i be the maximum number of activities and **the last one is the i -th activity**, then

$$AS_i = \max \left\{ \begin{array}{l} 1 \\ \max_{j < i, e_j \leq s_i} \{AS_j + 1\} \end{array} \right.$$

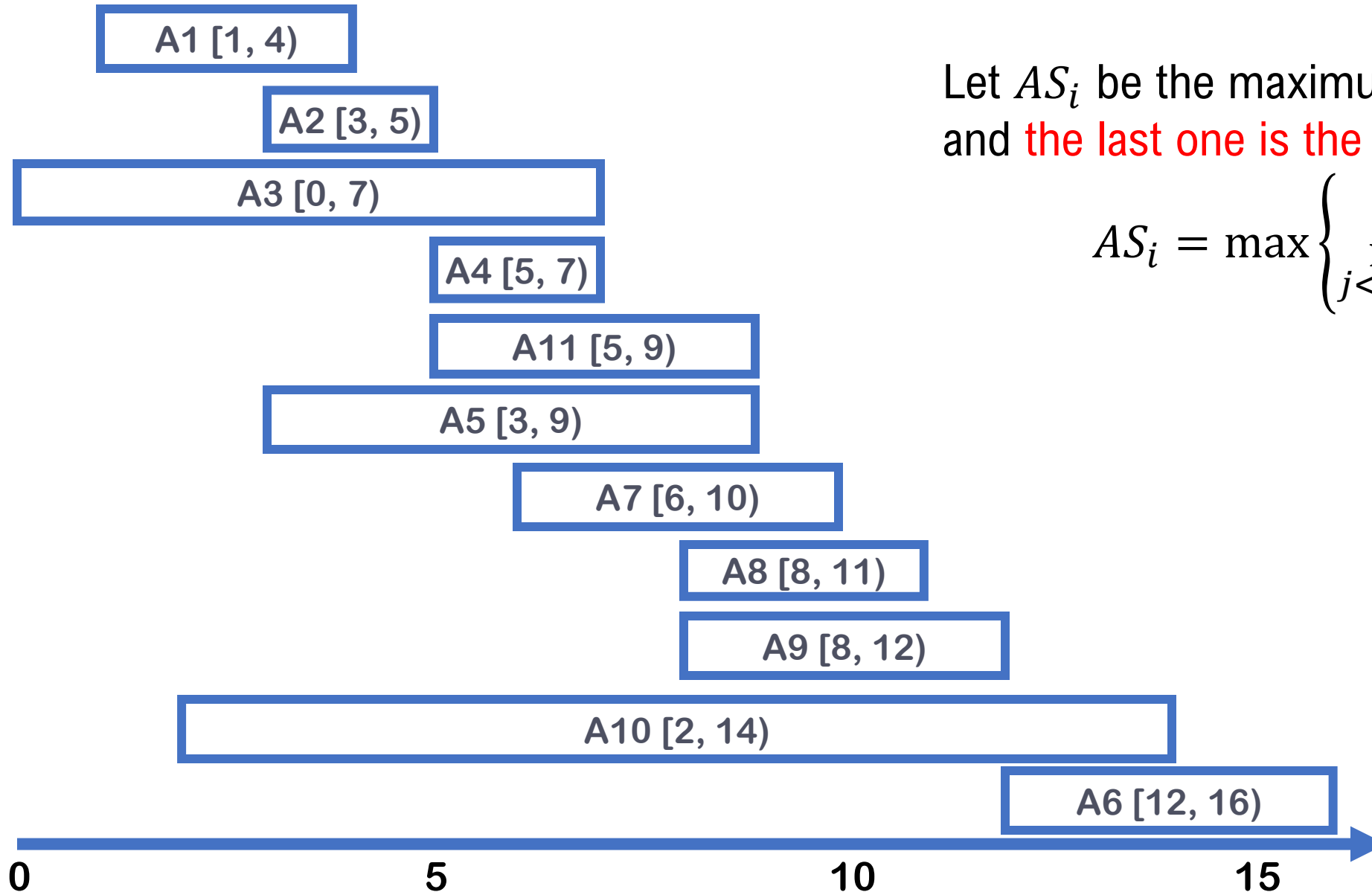
Revisit: activity selection: maximize **total time**?



Let AS_i be the maximum number of activities and **the last one is the i -th activity**, then

$$AS_i = \max \left\{ \begin{array}{l} 1 \\ \max_{j < i, e_j \leq s_i} \{AS_j + 1\} \end{array} \right.$$

Revisit: activity selection: maximize **total time**?

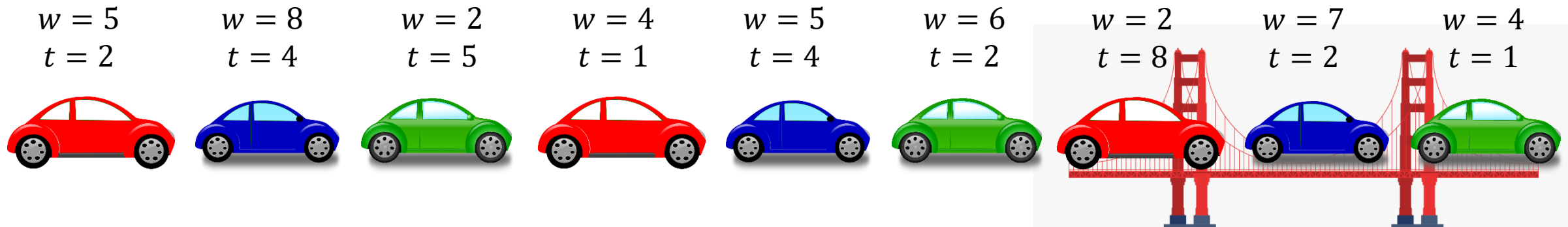


Let AS_i be the maximum number of activities and **the last one is the i -th activity**, then

$$AS_i = \max \left\{ \max_{j < i, e_j \leq s_i} \{AS_j + \cancel{1}\} \right. \\ \left. e_i - s_i \right. \\ \left. \text{(duration of activity } i) \right\}$$

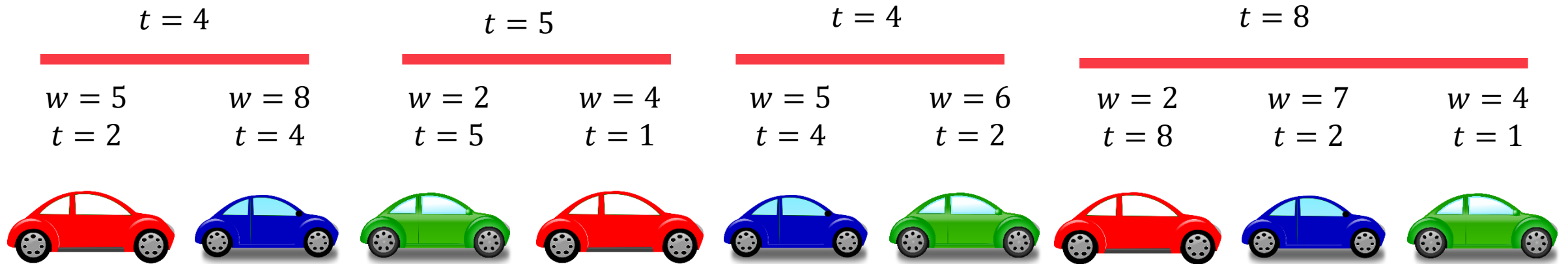
Motorcade

- A list of cars each with different weight and speed
 - Weight w_i , time needed to cross bridge t_i
- Cross a bridge with weight limit k
- They have to go in the original order
- Multiple cars can cross the bridge together, but sum of weight must be within weight limit k
- The time needed is the longest time among them
- What is the shortest time needed?



Motorcade

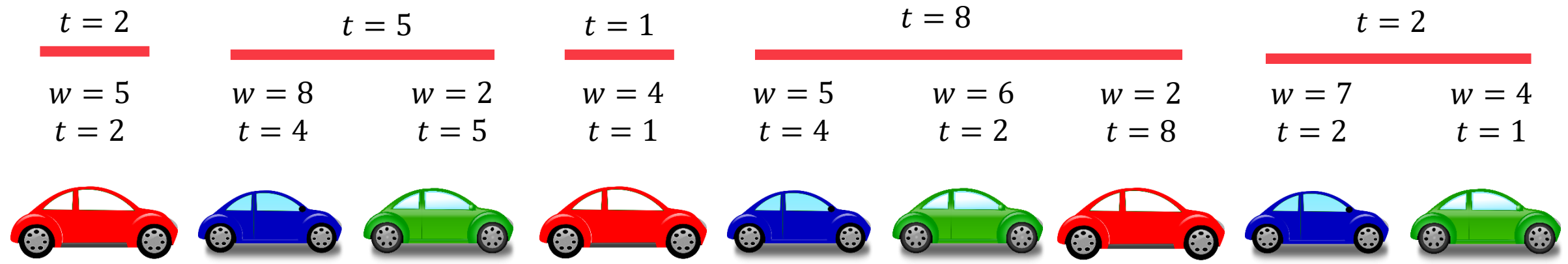
Total = 21min



$$k = 13$$

Motorcade

Total = 18min



$k = 13$

Motorcade

- Consider the first i cars (a prefix of the entire problem)
- What is the “last move”?
 - Which cars are in the last batch?
- What is the subproblem?
 - Other than the last batch, what is the best solution?

| | | | | | | | | |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| $w = 5$ | $w = 8$ | $w = 2$ | $w = 4$ | $w = 5$ | $w = 6$ | $w = 2$ | $w = 7$ | $w = 4$ |
| $t = 2$ | $t = 4$ | $t = 5$ | $t = 1$ | $t = 4$ | $t = 2$ | $t = 8$ | $t = 2$ | $t = 1$ |



$k = 13$

$$s[i] = \min_{\substack{0 < j < i \\ \text{sum}(w[j..i]) \leq k}} (s[j-1] + \max(t[j..i]))$$

Boundary: $s[0] = 0$

Summary for Dynamic Programming

Dynamic Programming (DP)

- DP is not an algorithm, but an algorithm design idea (methodology)
- DP works on problems with optimal substructure
- A DP **recurrence** of the **states**, with **boundary cases**
- We can convert a DP recurrence to a DP algorithm
 - Recursive implementation: straightforward
 - Non-recursive implementation: faster, and easy to be optimized

A high-level approach to design DP algorithms

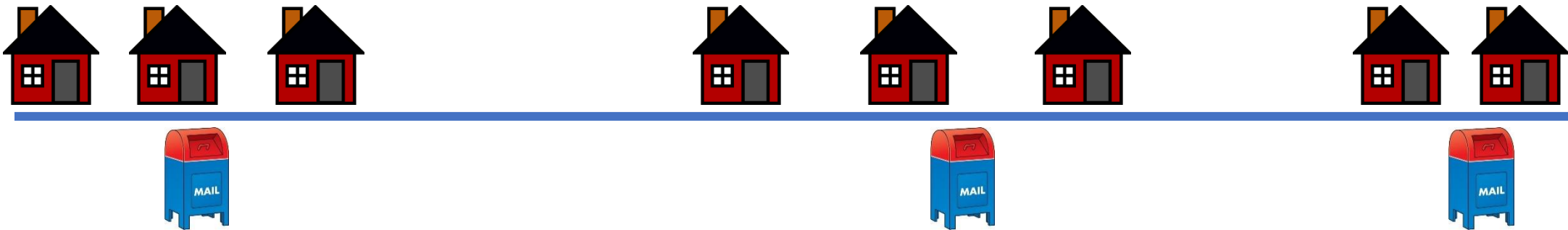
- DP is not an algorithm, but an algorithm design idea (methodology)
- Ideas in this lecture
- Subproblems: a prefix of the problem
- Decisions: what is the possible “last move” (second last element)?
- Boundary: what is the end of the recursion?

How to prepare midterm exam

- **First of all, the midterm exam is a review, not to check who is better than others (of course, it's a good chance for you to check if you fully understand the course materials based on a sample set of problems)**
 - Problems are similar to homework and quiz problems
- **Prepare the cheatsheet well**
 - Summarize the knowledge that you think are important for you in the midterm exam
- **Do the homework problems**
 - Prog HW 3 and 4, Written HW 3 are all available (for the next 3 weeks)
 - Dynamic programming is hard, so we will give you as much exercise as possible
 - Try to solve as many problems as possible before the exam, which maximizes your chances to design the DP algorithms in the exam
 - Also alleviate the workload in the second half of this quarter

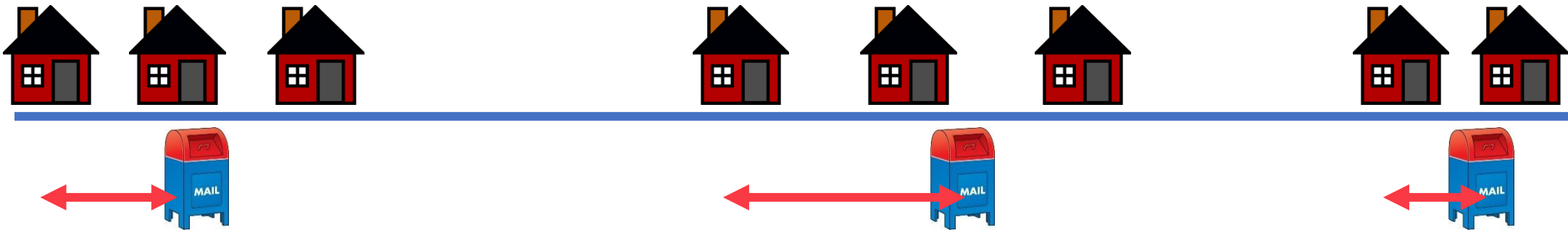
Other similar problem: the famous “post-office problem”

- First proposed by Donald Knuth in vol. 3 of TAOCP (1973)
- Let's consider the 1d case
- Installing each mailbox has certain cost (installation and maintenance)
- But we also want to minimize the residents' walking distances



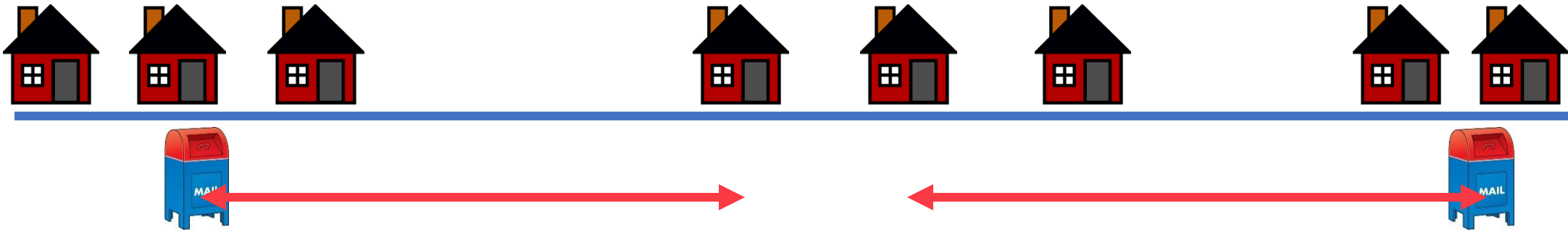
Formalize the problem

- Installing each mailbox has certain cost m
- The residents' unhappiness is the sum of the longest walking distances for each mailbox



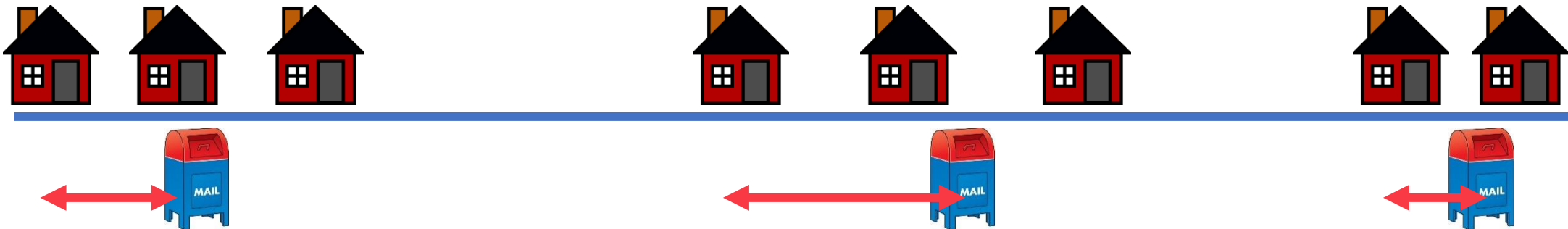
Formalize the problem

- Installing each mailbox has certain cost m
- The residents' unhappiness is the sum of the longest walking distances for each mailbox



Solving the problem

- Installing each mailbox has certain cost m
- The residents' unhappiness is the sum of the longest walking distances for each mailbox
- Let p_i be the optimal solution of the first i residents:
$$p_i = \min_{j < i} \{p_j + m + (c_i - c_{j+1})/2\}$$
- Boundary: $p_0 = 0$
- Answer: p_n



The line-breaking problem in LaTeX

Randomized Incremental Convex Hull is Highly Parallel

Guy E. Blelloch

Carnegie Mellon University

guyb@cs.cmu.edu

Yan Gu

UC Riverside

ygu@cs.ucr.edu

Julian Shun

MIT CSAIL

jshun@mit.edu

Yihan Sun

UC Riverside

yihans@cs.ucr.edu

ABSTRACT

The randomized incremental convex hull algorithm is one of the most practical and important geometric algorithms in the literature. Due to its simplicity, and the fact that many points or facets can be added independently, it is also widely used in parallel convex hull implementations. However, to date there have been no non-trivial theoretical bounds on the parallelism available in these implementations. In this paper, we provide a strong theoretical analysis showing that the standard incremental algorithm is inherently parallel. In particular, we show that for n points in any constant dimension, the algorithm has $O(\log n)$ dependence depth with high probability. This leads to a simple work-optimal parallel algorithm with polylogarithmic span with high probability.

Our key technical contribution is a new definition and analysis of the configuration dependence graph extending the traditional configuration space, which allows for asynchrony in adding configurations. To capture the “true” dependence between configurations,

the convex hull. A newly-added point either falls into the current convex hull and thus no further action is needed, or it removes existing faces (henceforth facets) that it is *visible* from, while adding new facets. For example, in Figure 1, adding c to the existing hull $u-v-w-x-y-z-t$ would replace edges $v-w$, $w-x$, $x-y$, and $y-z$ with $v-c$ and $c-z$. Clarkson and Shor, in their seminal work over 30 years ago [28], showed that the incremental convex hull algorithm on n points in d -dimensions has optimal $O(n^{\lfloor d/2 \rfloor} + n \log n)$ expected runtime when points are added in random order. Their results are based on a more general setting, which they also used to solve several other geometry problems, and the work led to over a hundred papers and survey articles on the topic of random incremental algorithms. Their proof has been significantly simplified over the years, and is now described in several textbooks [21, 32, 35, 50]. Analysis techniques, such as backwards analysis [54], were developed in this context and are now studied in many intermediate algorithms courses.

The line-breaking problem in LaTeX

- You have n words in a paragraph with lengths l_1, \dots, l_n
- You want to break them into lines so each line should contain 50 characters
- The penalty for each line is the $|x - 50|$ when x is the number of characters in that line
- You want to find an optimal line-breaking result

The line-breaking problem in LaTeX

- Let b_i be the optimal penalty for the first i words

$$b_i = \min_{j < i} \left\{ b_j + \left| i - j - 1 + \sum_{k=j+1}^i l_k - 50 \right| \right\}$$

- **Boundary:** $b_0 = 0$
- **Answer:** $\min_{i < n} \{b_i + w(i)\}$
 - $w(i)$ is the penalty for the last line, which is 0 if the last line has no more than 50 letters, or $n - i - 1 + \sum_{k=i+1}^n l_k - 50$ otherwise
- **Can add additional penalty to break the words (states changed to letters)**
- **How to implement it in $O(n^2)$ time?**