Algorithms

Design by Induction

Ming-Hsien Tsai

Academia Sinica
National Taiwan University

Based on: U. Manber. Introduction to Algorithms - A Creative Approach. Addison-Wesley, 1989.
Also based on slides made by Professor Yih-Kuen Tsay.

Introduction

- It is not necessary to design the steps required to solve a problem from scratch

- It is sufficient to guarantee the following:

  - It is possible to solve one small instance or a few small instances of the problem (base case)

  - A solution to every problem/instance can be constructed from solutions to smaller problems/ instances (inductive step)

Evaluating Polynomials

Problem

Given a sequence of real numbers a_0, a_1, …, a_{n-1}, a_n, and a real number x, compute the value of the polynomial

$$P_n(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n$$

Motivation: different approaches to the inductive step may result in algorithms of very different time complexities

Evaluating Polynomials (cont'd)

- Let $P_{n-1}(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$

- Induction hypothesis (first attempt): We know how to evaluate a polynomial represented by the input $a_0, a_1, \cdots, a_{n-1}$, at the point x, i.e., we know how to compute $P_{n-1}(x)$

- $P_n(x) = P_{n-1}(x) + a_nx^n$

- Number of multiplications:

  - $n + (n - 1) + \cdots + 2 + 1 = n(n + 1) / 2$

$$P_n(x) = (((\cdots(((a_0) + a_1x) + a_2x^2) + \cdots) + a_{n-1}x^{n-1}) + a_nx^n)$$

Evaluating Polynomials (cont'd)

- Let $P_{n-1}(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$

- Induction hypothesis (second attempt): We know how to compute $P_{n-1}(x)$, and we know how to compute x^{n-1}

- $P_n(x) = P_{n-1}(x) + a_nx(x^{n-1})$

- Number of multiplications: 2n

x^{n-2} is cached when computing this monomial

$$P_n(x) = (((\cdots(((a_0) + a_1x) + a_2xx) + \cdots) + a_{n-1}xx^{n-2}) + a_nxx^{n-1})$$

Evaluating Polynomials (cont'd)

- Let $P'_{n-1}(x) = a_1 + a_2x + \cdots + a_{n-1}x^{n-2} + a_nx^{n-1}$

- Induction hypothesis (final attempt): We know how to evaluate a polynomial represented by the coefficients a_1, $a_2, \cdots, a_n$, at the point x, i.e., we know how to compute $P'_{n-1}(x)$

- $P_n(x) = a_0 + P'_{n-1}(x) \cdot x$

Evaluating Polynomials (cont'd)

- More generally
  - $P'_0(x) = a_n$
  - $P'_i(x) = a_{n-i} + P'_{i-1}(x) \cdot x$, for $1 \le i \le n$
- Number of multiplications: n

$$P_n(x) = a_0 + (a_1 + (a_2 + (\cdots + (a_{n-1} + (a_n)x)x)\cdots)x)x)x$$

Evaluating Polynomials (cont'd)

```
Algorithm Polynomial_Evaluation(ā, x):
begin
    P := a_n;
    for i := 1 to n do
        P := x * P + a_{n-1};
end
```

$(\bar{a} = a_0, a_1, \ldots, a_n)$

This algorithm is known as ***Horner's rule***

Maximal Induced Subgraph

Problem

Given an undirected graph G = (V, E) and an integer k, find an induced subgraph H = (U, F) of G of maximum size such that all vertices of H have degree ≥ k (in H), or conclude that no such induced subgraph exists

Direct approach: remove vertices whose degree is < k

What would be the order of removals?

- all vertices of degree < k first and then vertices whose degrees were reduced, or
- one vertex of degree < k and then continue with affected vertices

Maximal Induced Subgraph (cont'd)

Let's think of proving a theorem that the algorithm exists

Induction hypothesis

We know how to find maximum induced subgraphs all of whose vertices have degrees ≥ k, provided that the number of vertices is < n

Base case (n = k + 1)

Inductive step: G = (V, E) is a graph with n > k + 1 vertices

- Case 1: all vertices have degrees ≥ k
- Case 2: some vertex v has degree < k

Maximal Induced Subgraph (cont'd)

```
Algorithm Max_Ind_Subgraph(G, k):
begin
    if number of nodes in G ≤ k then
        Max_Ind_Subgraph := 0
    else if the degree of every vertex of G ≥ k then
        Max_Ind_Subgraph := G;
    else
        let v be a vertex of G with degree < k;
        Max_Ind_Subgraph := Max_Ind_Subgraph(G - v, k);
end
```

Recursive version

Maximal Induced Subgraph (cont'd)

```
Algorithm Max_Ind_Subgraph(G, k):
begin
    while the degree of some vertex v of G < k do
        G := G - v;
    Max_Ind_Subgraph := G;
end
```

Iterative version

One-to-One Mapping

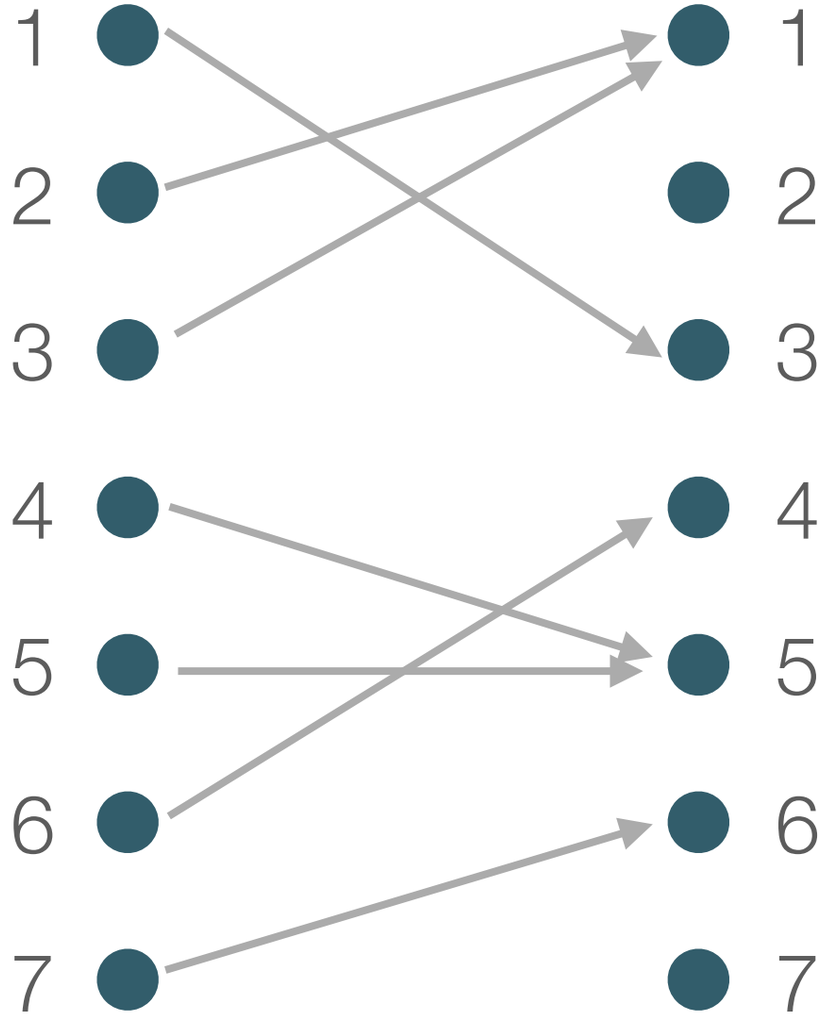
Problem

Given a finite set A and a mapping f from A to itself, find a subset $S \subseteq A$ with the maximum number of elements, such that

1. the function f maps every element of S to another element of S (i.e., f maps S into itself), and
2. no two elements of S are mapped to the same element (i.e., f is one-to-one with restricted to S)

Direct approach: When i and j are both mapped to k, remove one of them. What would be the order of the removal. Is the result maximal?

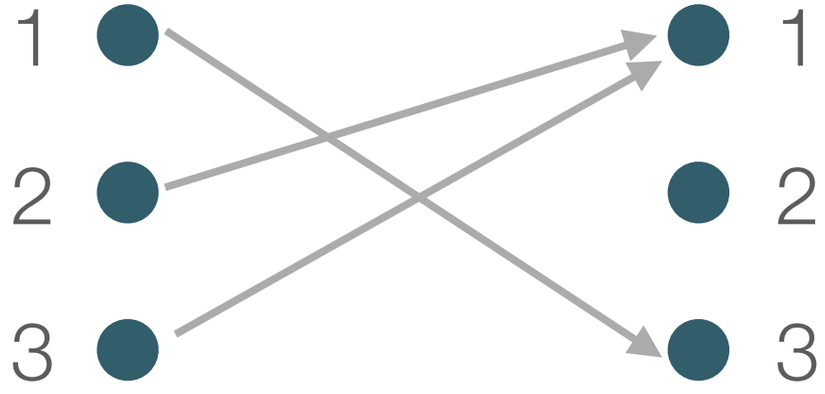
One-to-One Mapping (cont'd)



What would be the order of the removal?

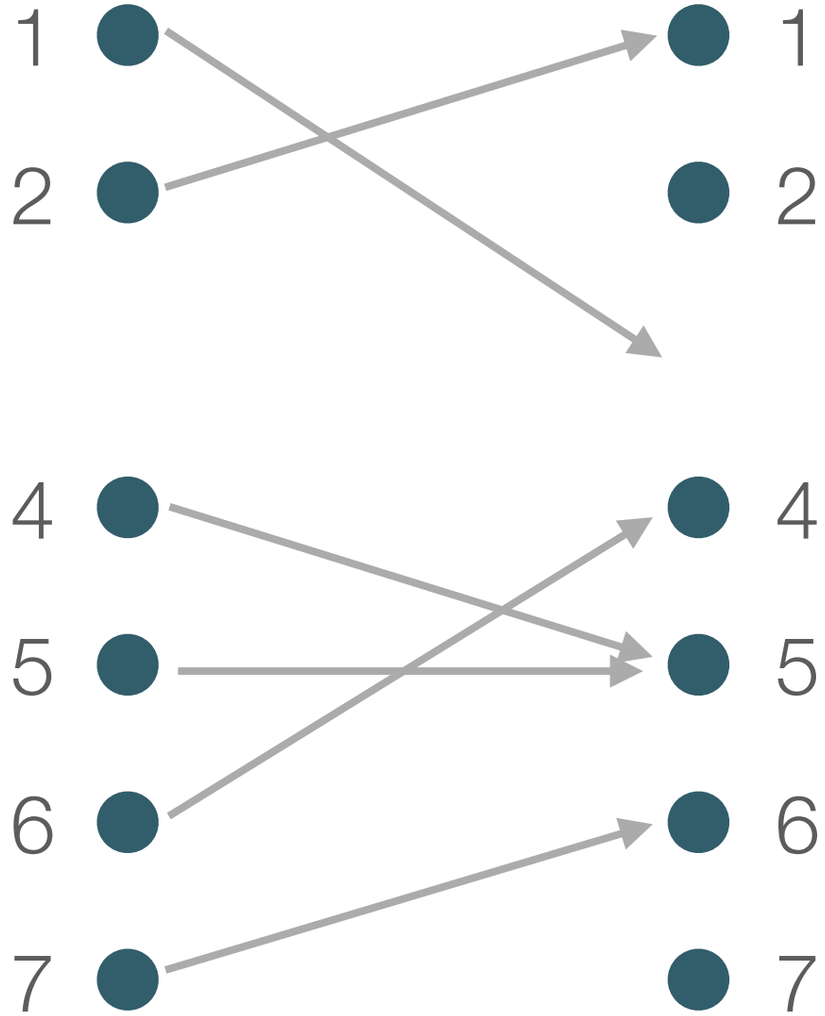
Is the result maximal?

One-to-One Mapping (cont'd)



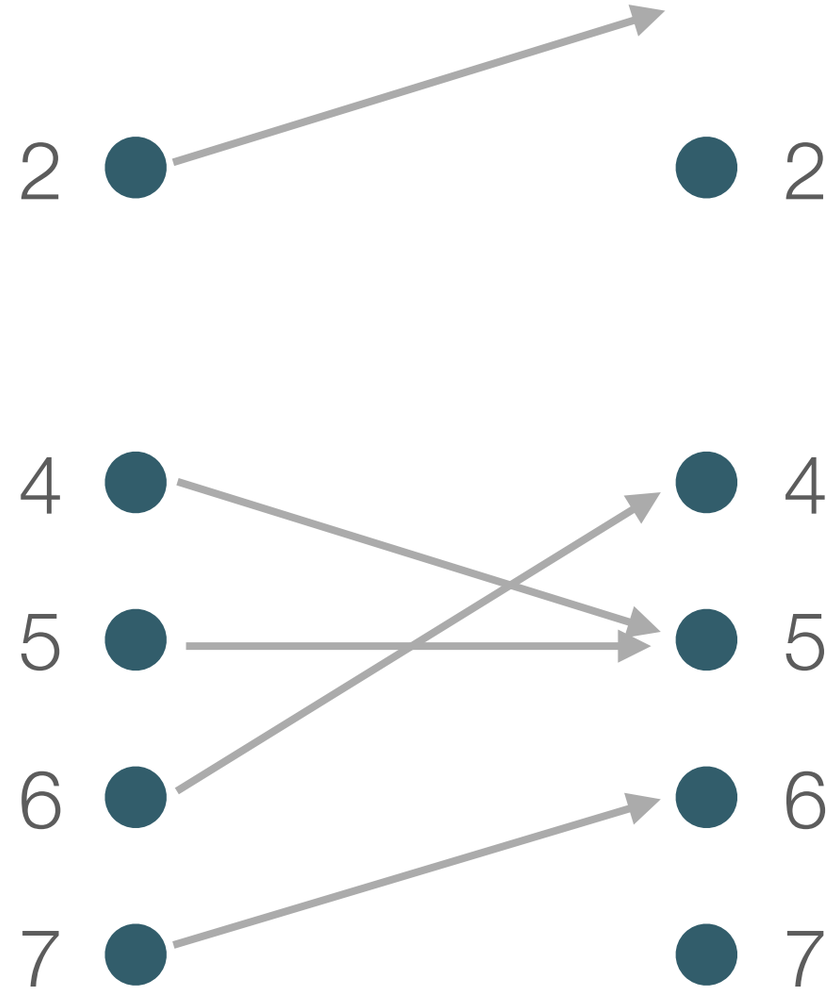
Case 1: remove 3 first

One-to-One Mapping (cont'd)



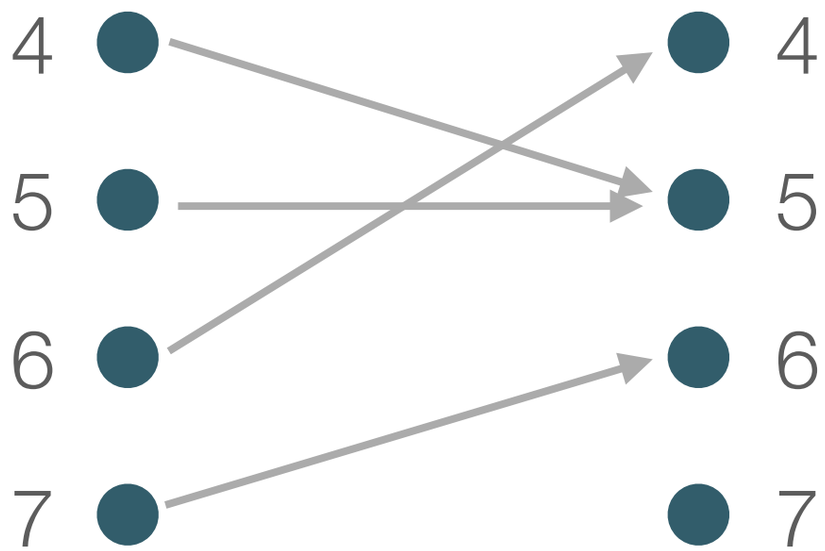
Case 1: remove 3 first

One-to-One Mapping (cont'd)



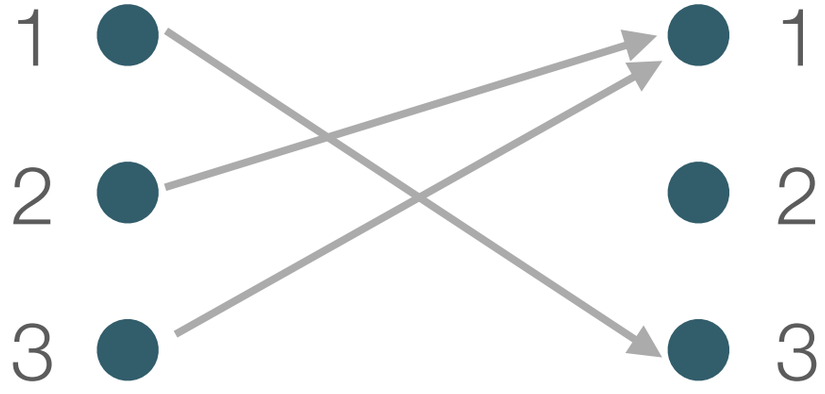
Case 1: remove 3 first

One-to-One Mapping (cont'd)



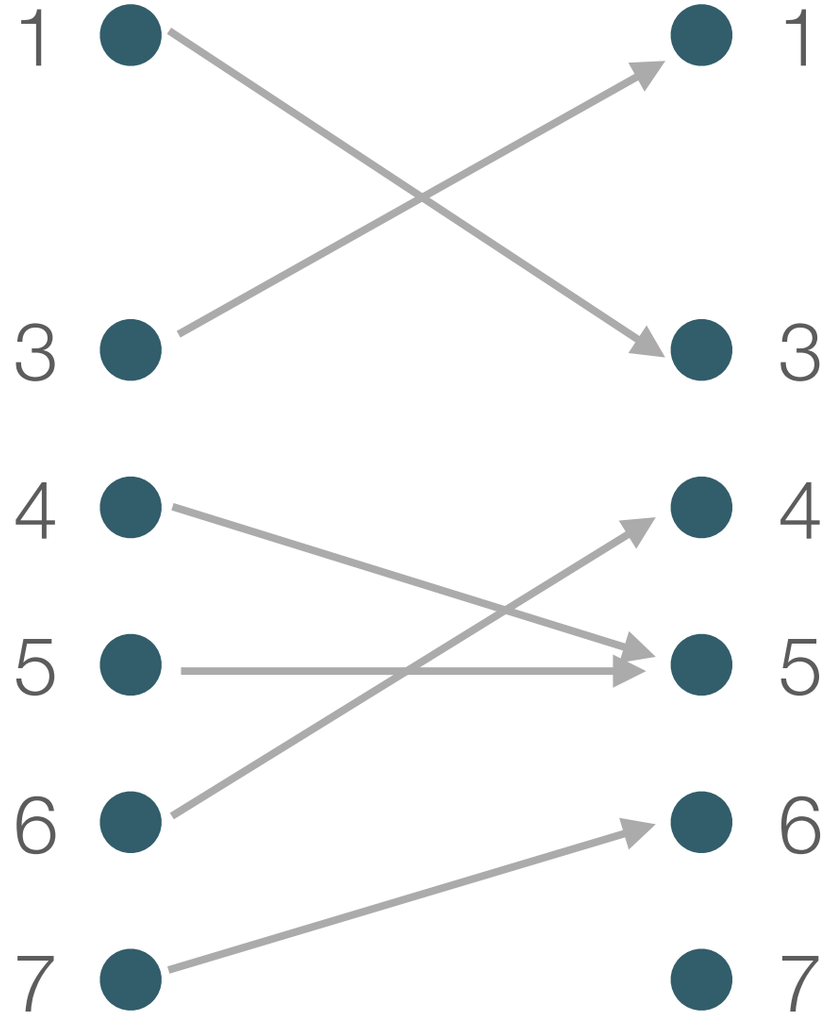
Case 1: remove 3 first

One-to-One Mapping (cont'd)



Case 2: remove 2 first

One-to-One Mapping (cont'd)



Case 2: remove 2 first

One-to-One Mapping (cont'd)

We can reduce the size of the problem by finding an element that does not belong to S

Inductive hypothesis

We know how to solve the problem for sets of n - 1 elements

Base case (one element)

Inductive step (n elements): any element i that has no other element mapped to it cannot belong to S

Case 1: there is such element

Case 2: there is no such element

One-to-One Mapping (cont'd)

```
Algorithm Mapping(f, n):
begin
    S := A;
    for j := 1 to n do c[j] := 0;
    for j := 1 to n do increment c[f[j]];
    for j := 1 to n do
        if c[j] = 0 then put j in Queue;
    while Queue is not empty do
        remove i from the top of Queue;
        S := S - { i };
        decrement c[f[i]];
        if c[f[i]] = 0 then put f[i] in Queue;
end
```

Celebrity

Problem

Given an n × n adjacency matrix, determine whether there exists an i (the “celebrity”) such that all the entries in the i-th column (except for the ii-th entry) are 1, and all the entries in the i-th row (except for the ii-th entry) are 0

- Celebrity: someone who is known by everyone but does not know anyone
- ij-th entry is one: i knows j, i.e., j is known by i
- A celebrity corresponds to a sink (with in-degree n - 1 and out-degree 0) of the directed graph
- Every directed graph has at most one sink

Celebrity (cont'd)

- Assume that we can find the celebrity among the n - 1 persons
- There are three possibilities
  - The celebrity is among the first n - 1 (ask two more questions)
  - The celebrity is the n-th person (ask 2(n - 1) questions in the n-th step and $O(n^2)$ in total)
  - There is no celebrity

Celebrity (cont'd)

- Instead of identifying a celebrity, it is easier to identify someone as a noncelebrity
- Ask if i knows j:
  - Yes: i is noncelebrity
  - No: j is noncelebrity
- Always remove one person with one question
- Check whether the last one is a celebrity or not

Celebrity (cont'd)

```
Algorithm Celebrity(Know):
begin
    i := 1;
    j := 2;
    next := 3;
while next ≤ n + 1 do
    if Know[i, j] then i := next
    else j := next;
    next := next + 1;
if i = n + 1 then candidate := j
else candidate := i;
…
```

Celebrity (cont'd)

Algorithm Celebrity(Know):

```
    …
    wrong := false;
    k := 1;
    Know[candidate, candidate] := false;
    while not wrong and k ≤ n do
        if Know[candidate, k] then wrong := true;
        if not Know[k, candidate] then
            if candidate ≠ k then wrong := true;
        k := k + 1
    if not wrong then celebrity := candidate
    else celebrity := 0
end
```

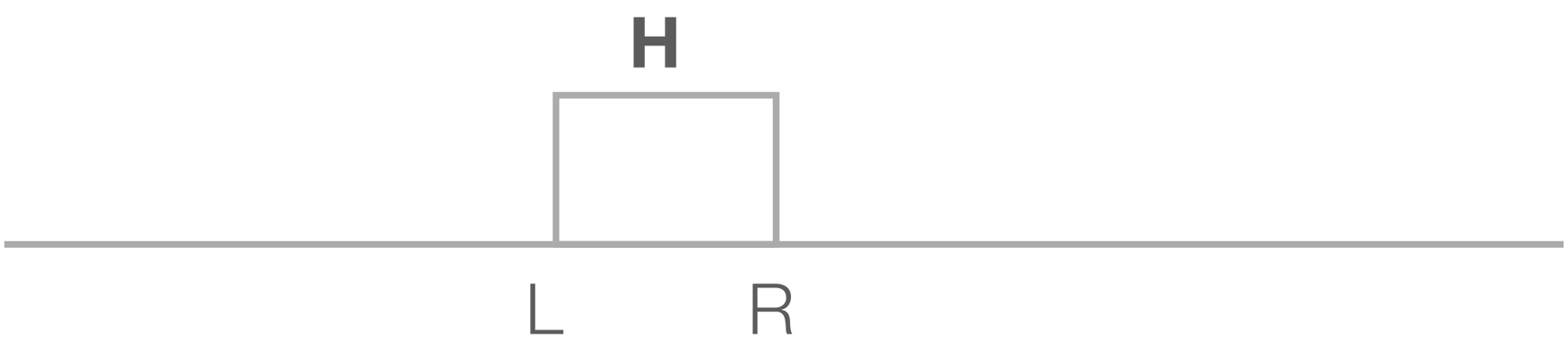
The Skyline Problem

Problem

Given the exact locations and shapes of several rectangular buildings in a city, draw the skyline (in two dimensions) of these buildings, eliminating hidden lines

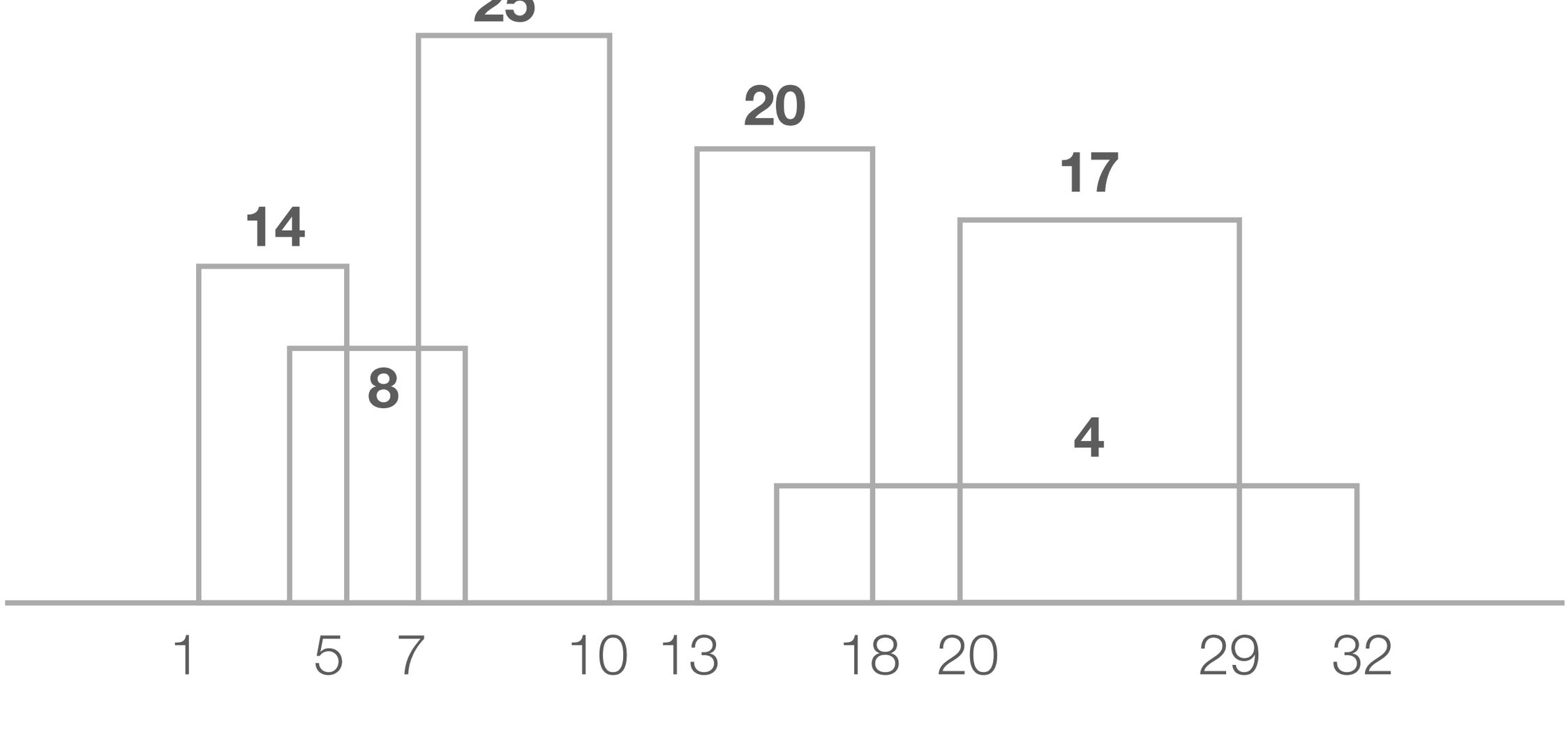
Motivation: different approaches to the inductive step may result in algorithms of very different time complexities

A building is represented by a triple (L, **H**, R)



The Skyline Problem (cont'd)

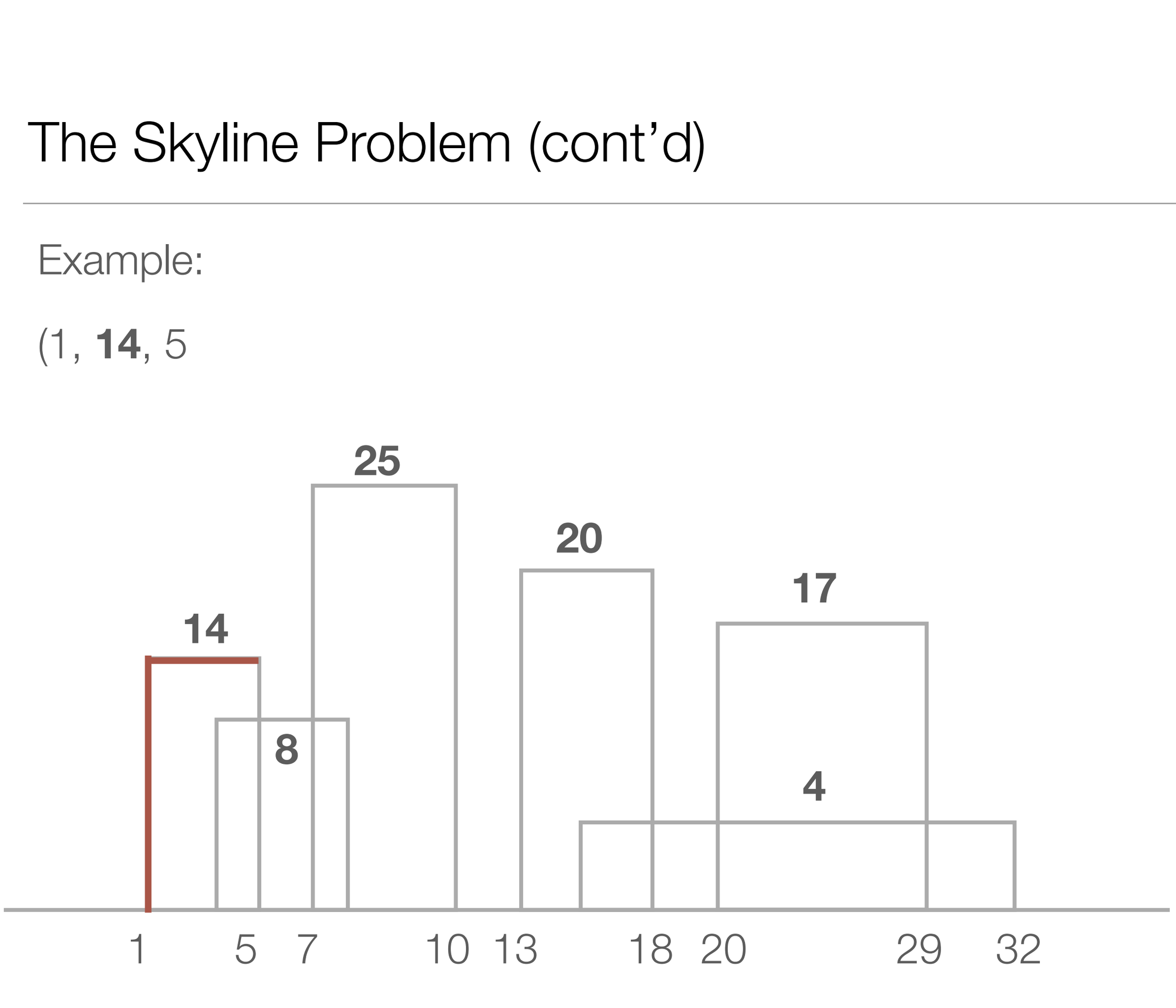
Example:



The Skyline Problem (cont'd)

Example:

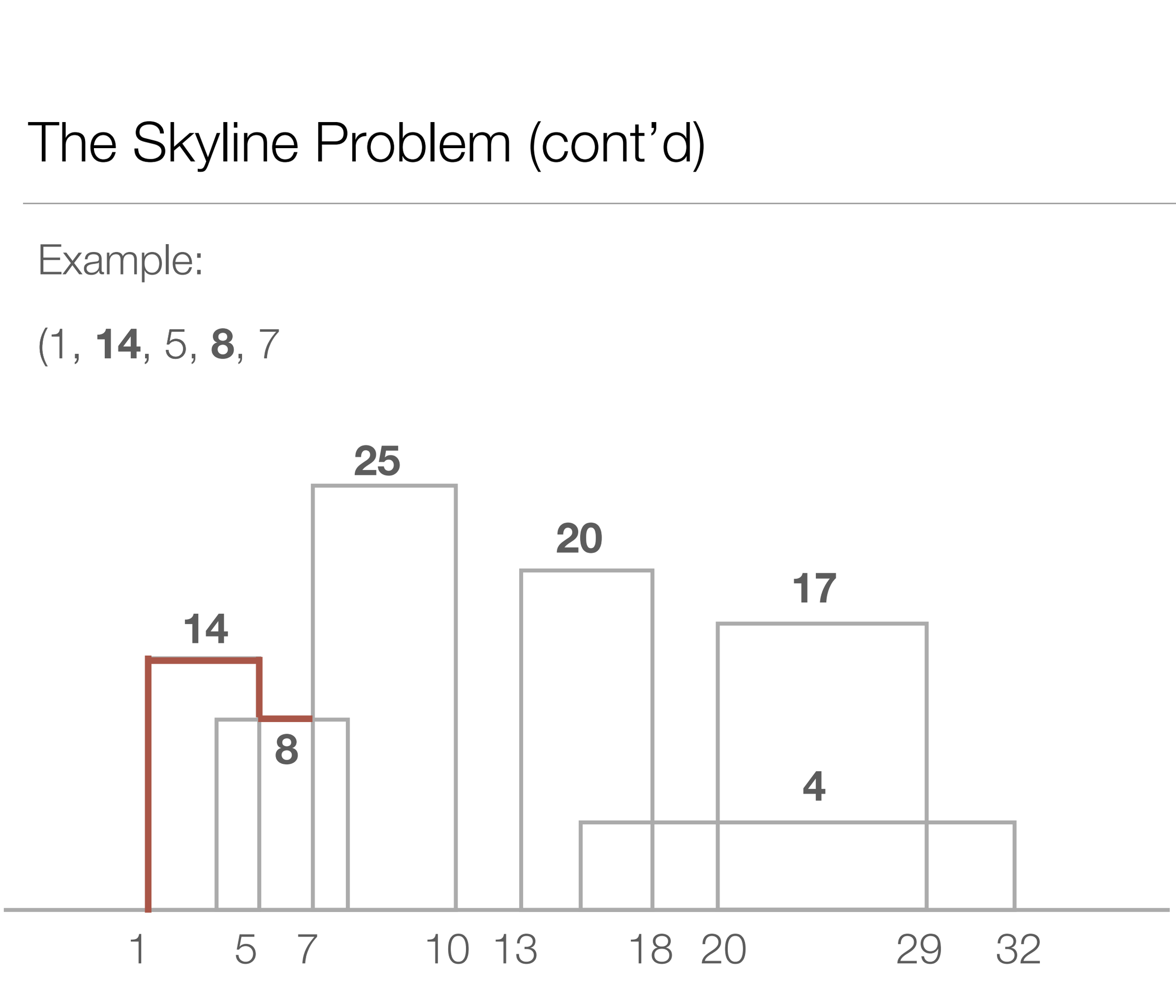
(1, **14**, 5



The Skyline Problem (cont'd)

Example:

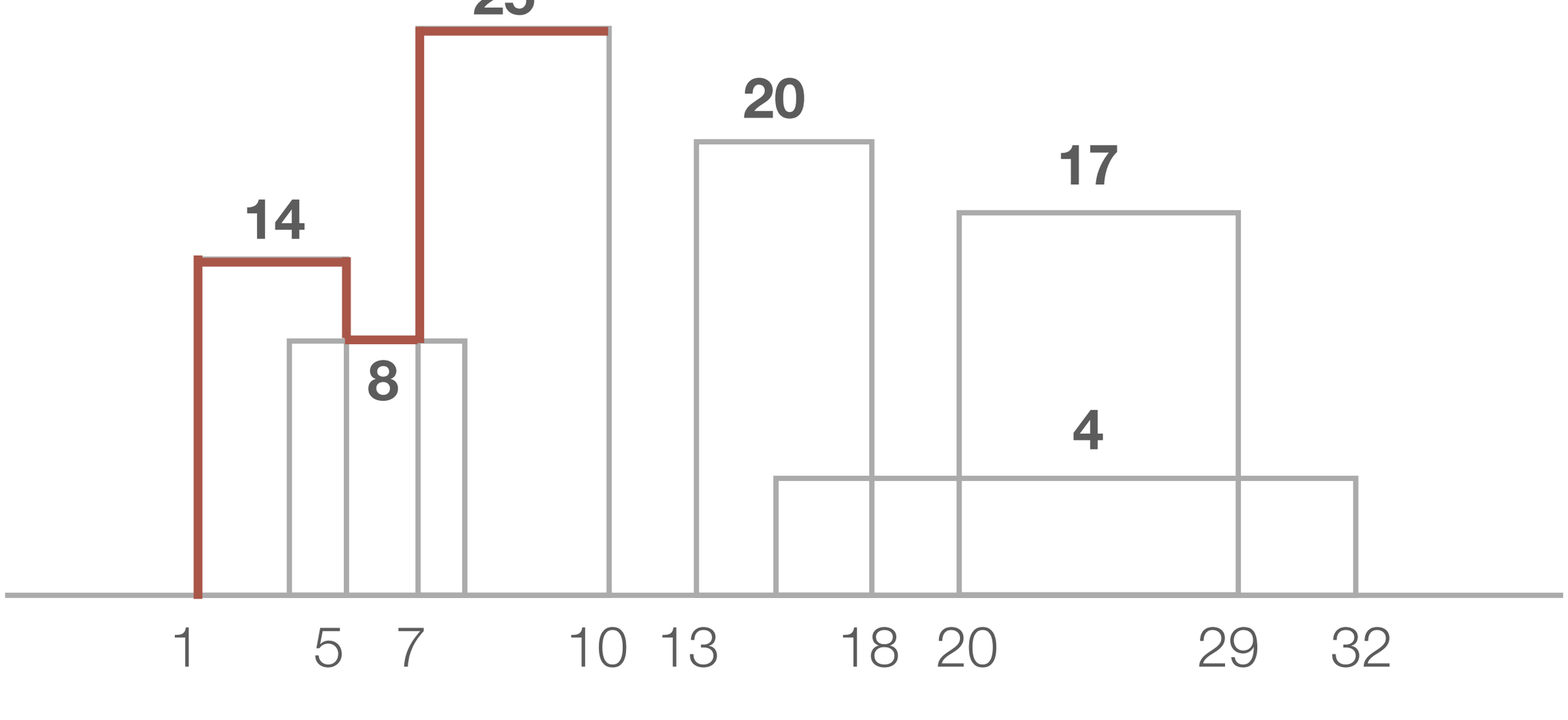
(1, **14**, 5, **8**, 7



The Skyline Problem (cont'd)

Example:

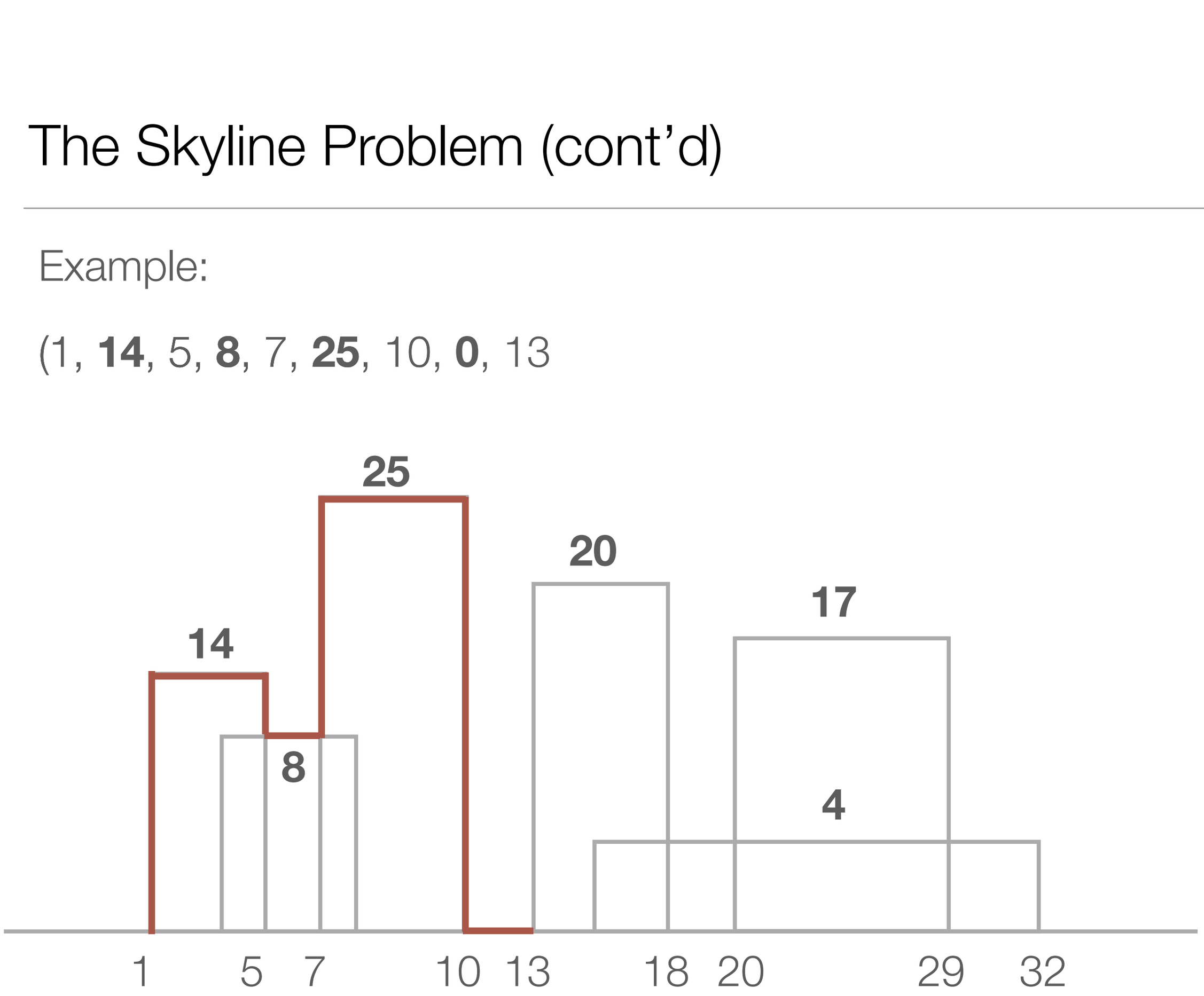
(1, **14**, 5, **8**, 7, **25**, 10



The Skyline Problem (cont'd)

Example:

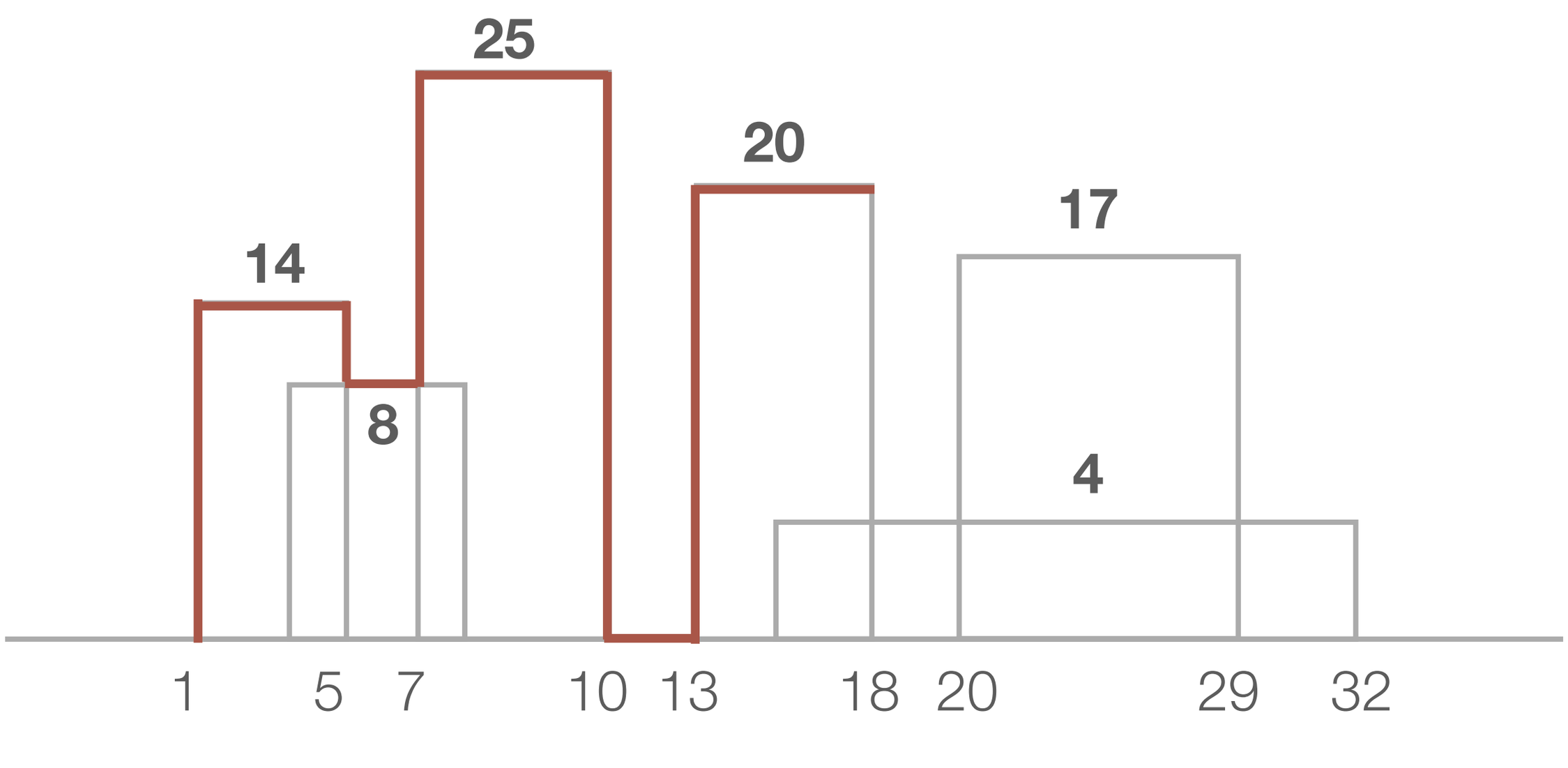
(1, **14**, 5, **8**, 7, **25**, 10, **0**, 13



The Skyline Problem (cont'd)

Example:

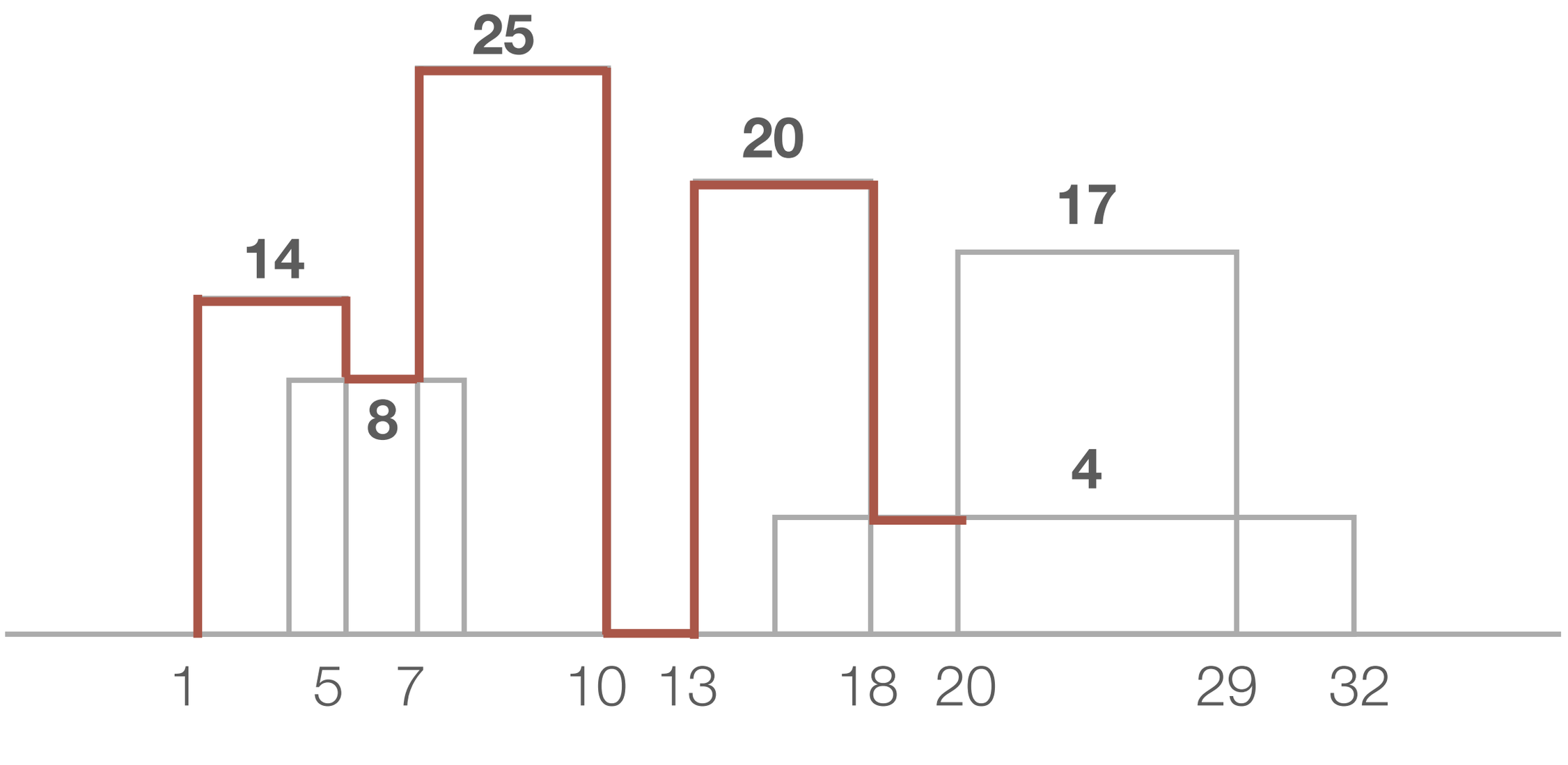
(1, **14**, 5, **8**, 7, **25**, 10, **0**, 13, **20**, 18



The Skyline Problem (cont'd)

Example:

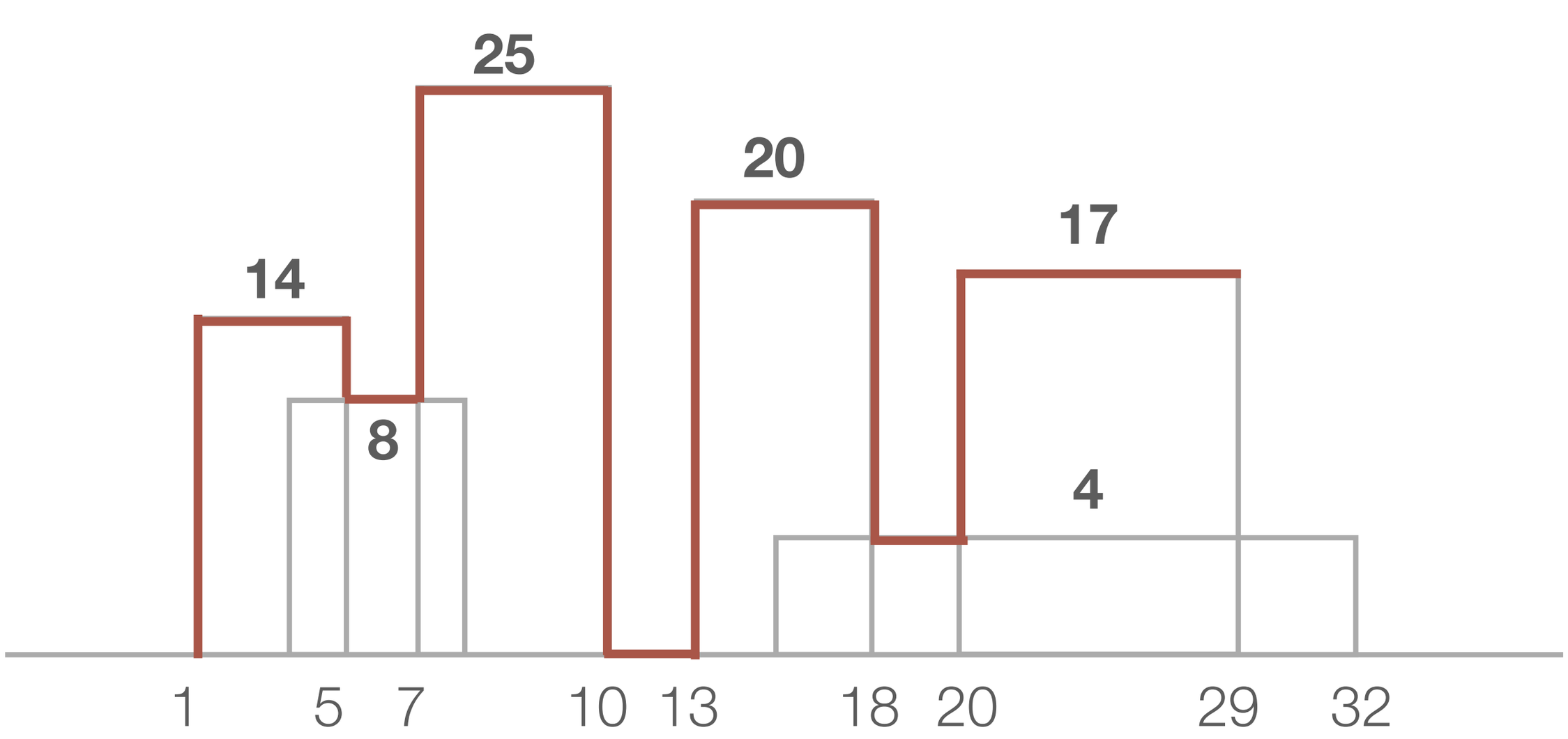
(1, **14**, 5, **8**, 7, **25**, 10, **0**, 13, **20**, 18, **4**, 20



The Skyline Problem (cont'd)

Example:

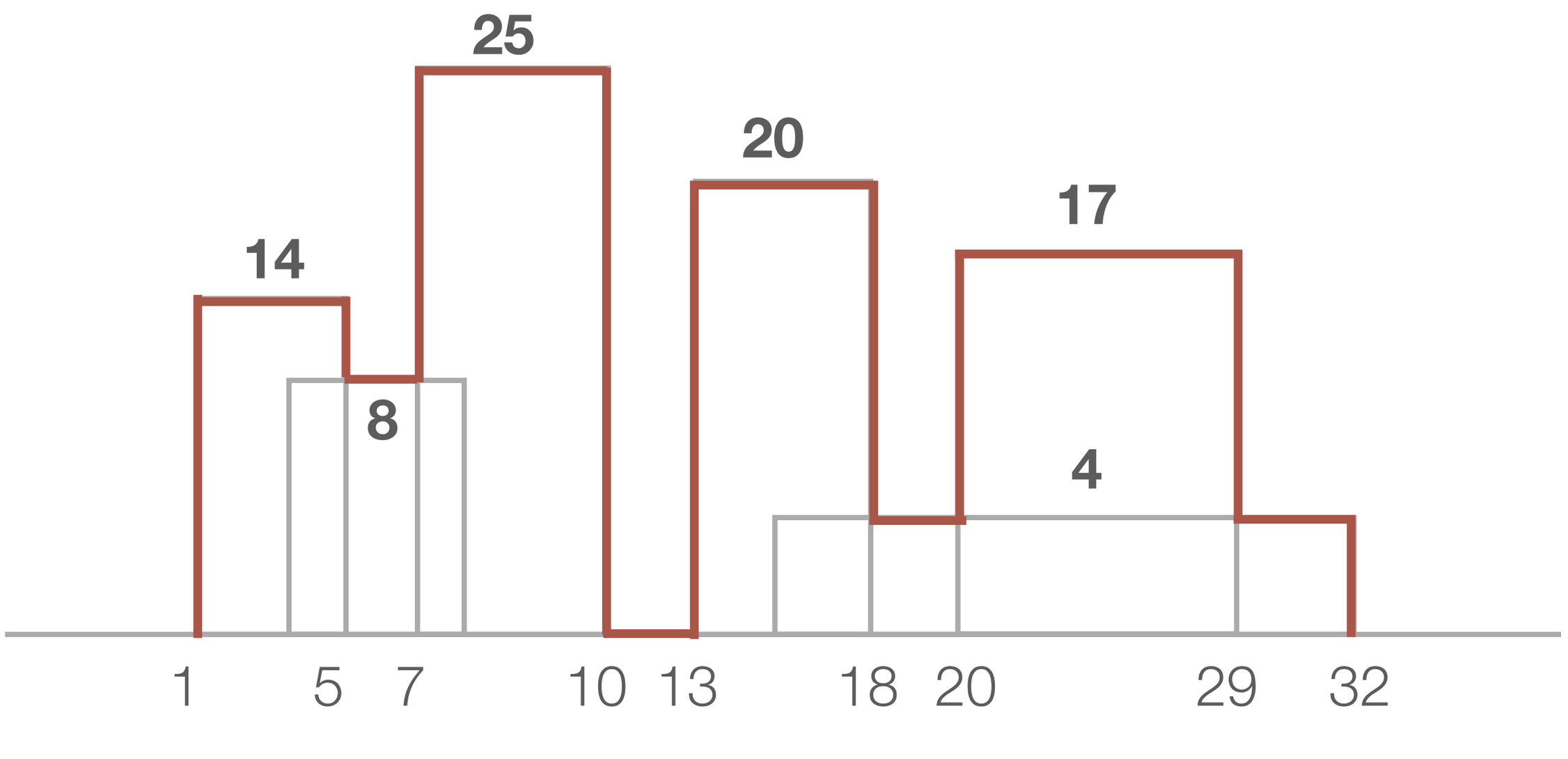
(1, **14**, 5, **8**, 7, **25**, 10, **0**, 13, **20**, 18, **4**, 20, **17**, 29



The Skyline Problem (cont'd)

Example:

(1, **14**, 5, **8**, 7, **25**, 10, **0**, 13, **20**, 18, **4**, 20, **17**, 29, **4**, 32)

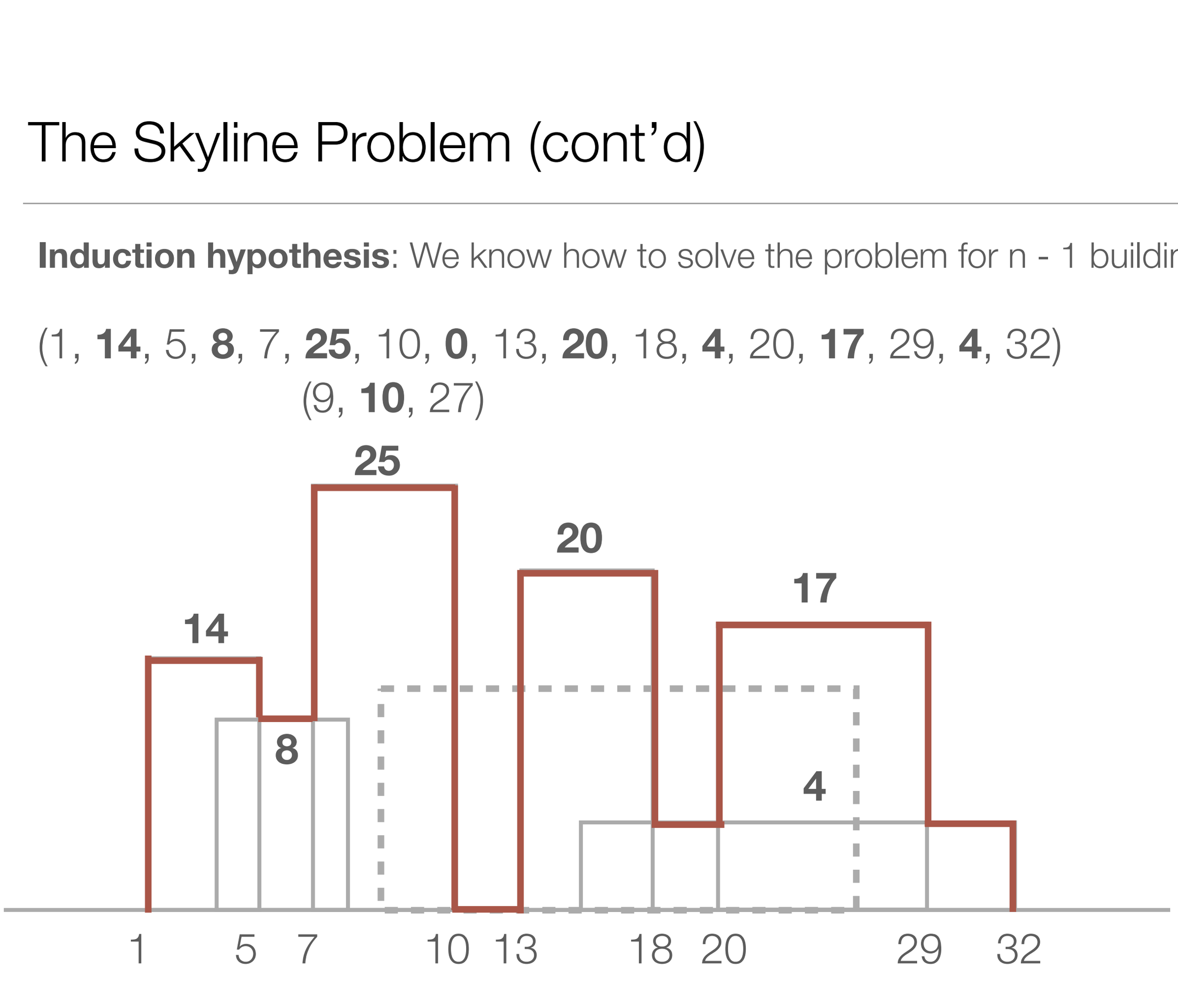


The Skyline Problem (cont'd)

Induction hypothesis: We know how to solve the problem for n - 1 buildings

(1, **14**, 5, **8**, 7, **25**, 10, **0**, 13, **20**, 18, **4**, 20, **17**, 29, **4**, 32)

(9, **10**, 27)



The Skyline Problem (cont'd)

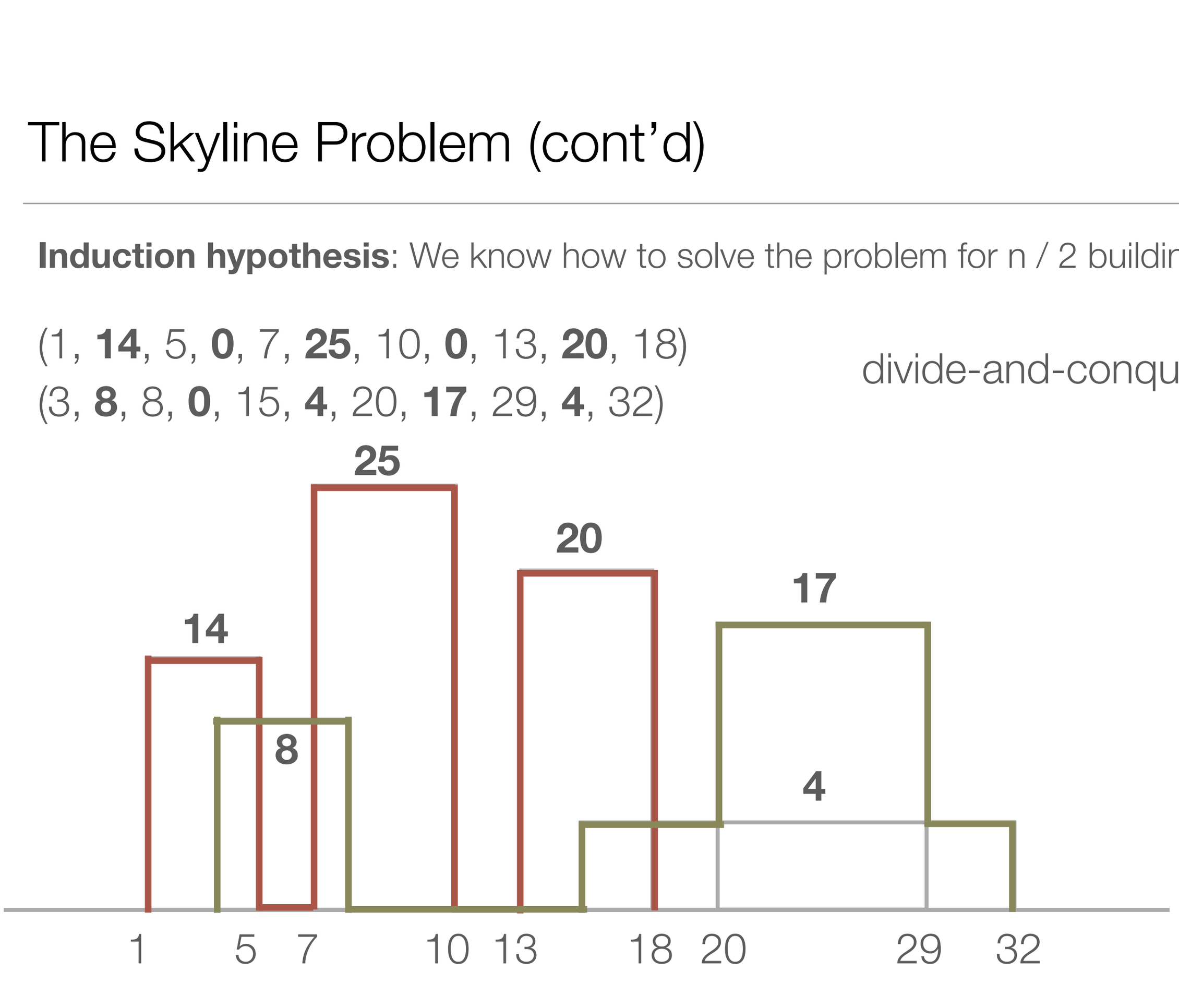
- Adding one building at a time:
  - $T(1) = O(1)$
  - $T(n) = T(n - 1) + O(n), n \geq 2$
- Time complexity: $O(n^2)$

The Skyline Problem (cont'd)

Induction hypothesis: We know how to solve the problem for n / 2 buildings

(1, **14**, 5, **0**, 7, **25**, 10, **0**, 13, **20**, 18)
(3, **8**, 8, **0**, 15, **4**, 20, **17**, 29, **4**, 32)

divide-and-conquer



The Skyline Problem (cont'd)

- Merge two skylines every round:
  - $T(1) = O(1)$
  - $T(n) = 2T(n / 2) + O(n), n \geq 2$
- Time complexity: $O(n \log n)$

Balance Factors in Binary Trees

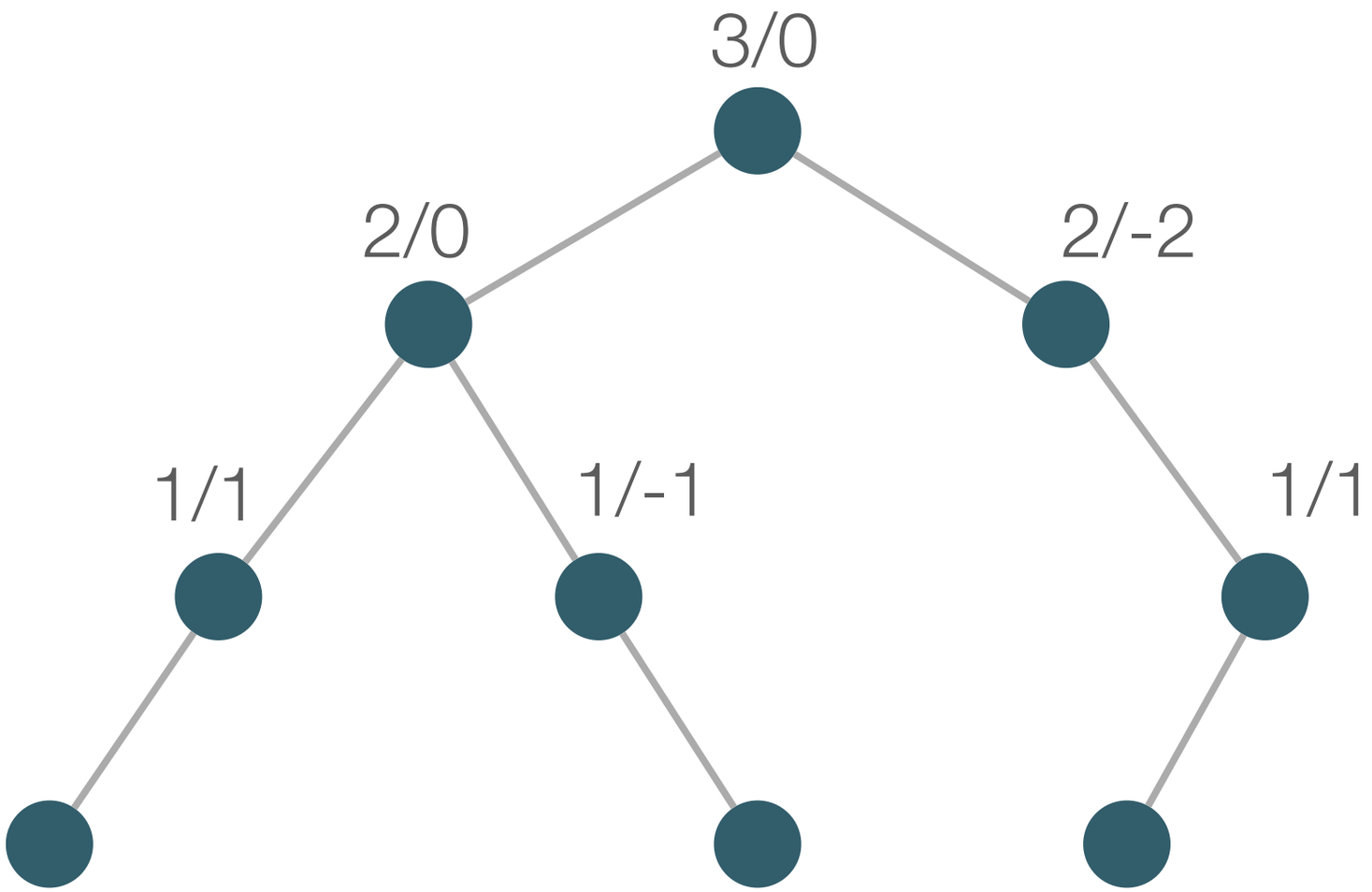
Problem

Given a binary tree T with n nodes, compute the balance factors of all nodes

The ***balance factor*** of a node is defined as the difference between the height of the node's left subtree and the height of the node's right subtree

Motivation: an example of why we must ***strengthen the hypothesis*** (and hence the problem to be solved)

Balance Factors in Binary Trees (cont'd)



h/b: h = height, b = balance factor

Balance Factors in Binary Trees (cont'd)

- **Induction hypothesis**: We know how to compute balance factors of all nodes in trees that have < n nodes

  - We still need to compute the heights of the children of a node when computing the balance factor of the node

- **Induction hypothesis**: We know how to compute balance factors ***and heights*** of all nodes in trees that have < n nodes

Note: in the inductive step for a tree of n nodes, both the balance factor and the height of the root have to be computed

Maximum Consecutive Subsequence

Problem

Given a sequence $x_1, x_2, \ldots, x_n$ of real numbers (not necessarily positive) find a subsequence $x_i, x_{i+1}, \ldots, x_j$ (of consecutive elements) such that the sum of the numbers in it is maximum over all subsequences of consecutive elements

Example:

Sequence: (2, -3, 1.5, -1, 3, -2, -3, 3)

Maximum subsequence: (1.5, -1, 3)

Maximum Consecutive Subsequence (cont'd)

Induction hypothesis

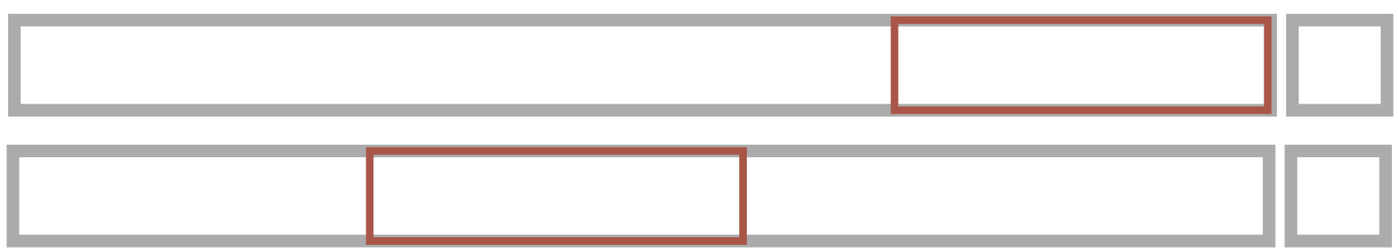
We know how to find the maximum subsequence in sequences of size < n

Consider a sequence $S = (x_1, x_2, \ldots, x_n)$ of size $n > 1$

The maximum subsequence of $S' = (x_1, x_2, \ldots, x_{n-1})$ is $S_M' = (x_i, x_{i+1}, \ldots, x_j)$ for certain i and j such that $1 \leq i \leq j \leq n - 1$ (by induction)

Case 1: $j = n - 1$

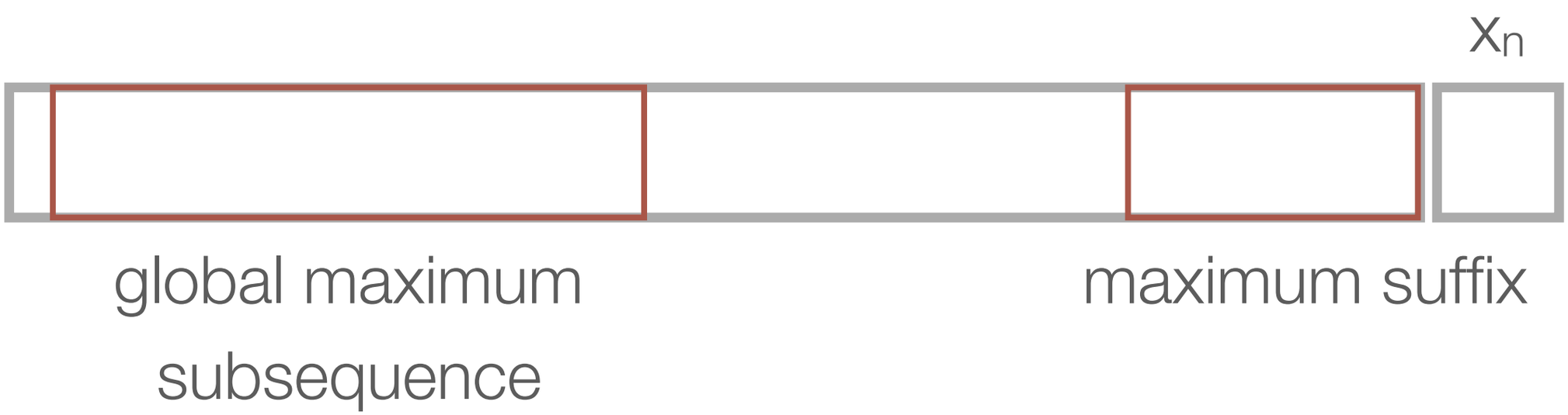
Case 2: $j < n - 1$



Maximum Consecutive Subsequence (cont'd)

Stronger induction hypothesis

We know how to find, in sequences of size < n, the maximum subsequence overall ***and the maximum subsequence that is a suffix***



Maximum Consecutive Subsequence (cont'd)

```
Algorithm Max_Consec_Subseq(X, n):
begin
    Global_Max := 0;
    Suffix_Max := 0;
    for i := 1 to n do
        if X[i] + Suffix_Max > Global_Max then
            Suffix_Max := Suffix_Max + X[i];
            Global_Max := Suffix_Max
        else if X[i] + Suffix_Max > 0 then
            Suffix_Max := Suffix_Max + X[i]
        else
            Suffix_Max := 0
end
```

Strengthening the Induction Hypothesis

- Denote a theorem by P
- The induction hypothesis can be denoted by P(< n)
- The proof must conclude that P(< n) ⇒ P(n)

- We can add another assumption Q, under which the proof becomes easier
  - [P and Q](< n) ⇒ P(n) v.s. P(< n) ⇒ P(n)
- We need to prove [P and Q](< n) ⇒ [P and Q](n)

The Knapsack Problem

Problem

Given an integer K and n items of difference sizes such that the i-th item has an integer size k_i, find a subset of the items whose sizes sum to exactly K, or determine that no such subset exists

Denote the problem by P(n, K) where n is the number of items and K is the size of the knapsack

P(i, k) denotes the problem with the first i items and a knapsack of size k

The Knapsack Problem (cont'd)

Induction hypothesis

We know how to solve P(n - 1, K)

Base case (P(1, K)): trivial

Inductive step (P(n, K)):

- Case 1: There is a solution for P(n - 1, K)
- Case 2: There is no solution for P(n - 1, K)
  - n-th item must be included
  - Need to solve P(n - 1, K - k_n)

The Knapsack Problem (cont'd)

Induction hypothesis

We know how to solve P(n - 1, K) for all $0 \leq k \leq K$

K in this induction hypothesis can be any number

Base case (P(1, k)): trivial

Inductive step: P(n, k) has a solution if one of the following subproblem has a solution

- P(n - 1, k)
- P(n - 1, k - k_n)

The time complexity is exponential!

The Knapsack Problem (cont'd)

P(n, k) has a solution if one of the following subproblem has a solution

- P(n - 1, k)
- P(n - 1, k - k_n)

The time complexity is exponential.

There are only nK different possible problems.

Why the complexity is exponential?

How to reduce the time complexity?

The Knapsack Problem (cont'd)

- We can reduce the time complexity by solving all the nK problems with a table construction
- The table is constructed iteratively
- Each entry is computed from a combination of other entries above is or to the left of it in the table
- This technique is called ***dynamic programming***

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2																	
k_2=3																	
k_3=5																	
k_4=6																	

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3																	
k_3=5																	
k_4=6																	

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	?											
k_3=5																	
k_4=6																	

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	?											
k_4=6																	

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8
k_1=2	O	-	I	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I
k_4=6	O	-	O	O	-	O	I	O	O

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8
k_1=2	O	-	I	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I
k_4=6	O	-	O	O	-	O	I	O	O

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8
k_1=2	O	-	I	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I
k_4=6	O	-	O	O	-	O	I	O	O

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8
k_1=2	O	-	I	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I
k_4=6	O	-	O	O	-	O	I	O	O

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

	0	1	2	3	4	5	6	7	8
k_1=2	O	-	I	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I
k_4=6	O	-	O	O	-	O	I	O	O

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Trace back to see what items are included

The Knapsack Problem (cont'd)

```
Algorithm Knapsack(S, K):
begin
    P[0, 0].exist := true;
    for k := 1 to K do
        P[0, k].exist := false;
    for i := 1 to n do
        for k := 0 to K do
            P[i, k].exist := false;
            if P[i - 1, k].exist then
                P[i, k].exist := true; P[i, k].belong := false
            else if k - S[i] ≥ 0 then
                if P[i - 1, k - S[i]].exist then
                    P[i, k].exist := true; P[i, k].belong := true
end
```