COMP90038 Algorithms and Complexity

Brute Force Methods

Michael Kirley

Lecture 5

Semester 1, 2021

Brute Force Algorithms

Straightforward problem solving approach, usually based directly on the problem's statement.

Exhaustive search for solutions is a prime example.

- Selection sort
- String matching
- Closest pair
- Exhaustive search for combinatorial solutions
- Graph traversal

Example: Selection Sort

We already saw this algorithm:

```
function SelSort(A[0..n − 1])
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[min] then
                min ← j
        swap A[i] and A[min]
```

The complexity is $\Theta(n^2)$.

We shall soon meet better sorting algorithms.

Properties of Sorting Algorithms

A sorting algorithm is

- in-place if it does not require additional memory except, perhaps, for a few units of memory.
- stable if it preserves the relative order of elements that have identical keys.
- input-insensitive if its running time is fairly independent of input properties other than size.

Properties of Selection Sort

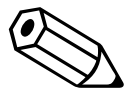
While running time is quadratic, selection sort makes only about n exchanges.

So: A good algorithm for sorting small collections of large records.

In-place?

Stable?

Input-insensitive?



Brute Force String Matching

Pattern p: A string of m characters to search for.

Text t: A long string of n characters to search in.

```
for i ← 0 to n − m do
    j ← 0
    while j < m and p[j] = t[i + j] do
        j ← j + 1
    if j = m then
        return i
return −1
```

Analysing Brute Force String Matching

For each of $n - m + 1$ positions in t, we make up to m comparisons.

Assuming n is much larger than m, this means $O(mn)$ comparisons.

However, for random text over a reasonably large alphabet (as in English), the average running time is linear in n.

There are better algorithms, in particular for smaller alphabets such as binary strings or strings of DNA nucleobases.

But for many purposes, the brute-force algorithm is acceptable.

Later we shall see more sophisticated string search.

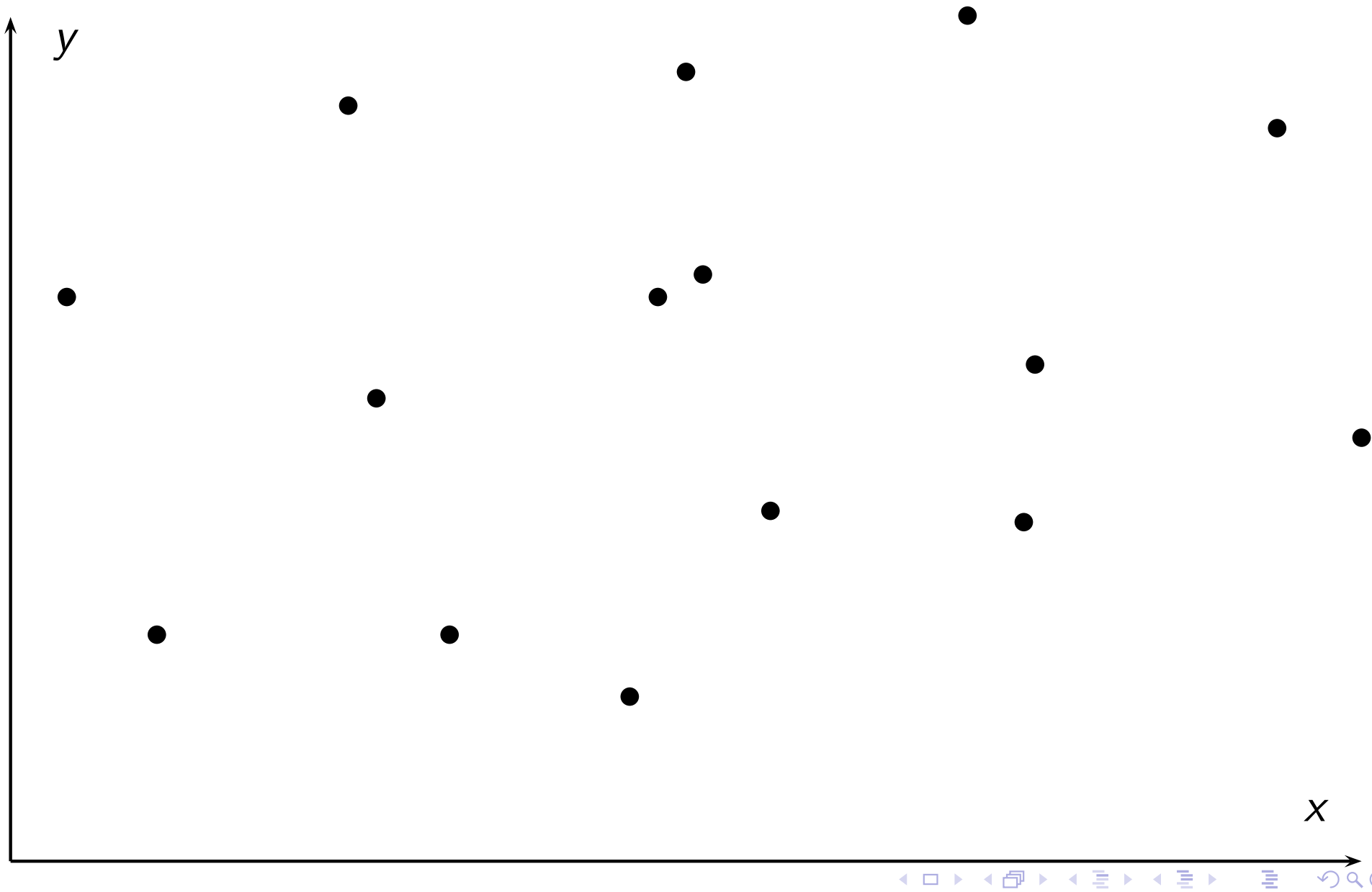
Brute Force Geometric Algorithms: Closest Pair

Problem: Given n points is k-dimensional space, find a pair of points with minimal separating Euclidean distance.

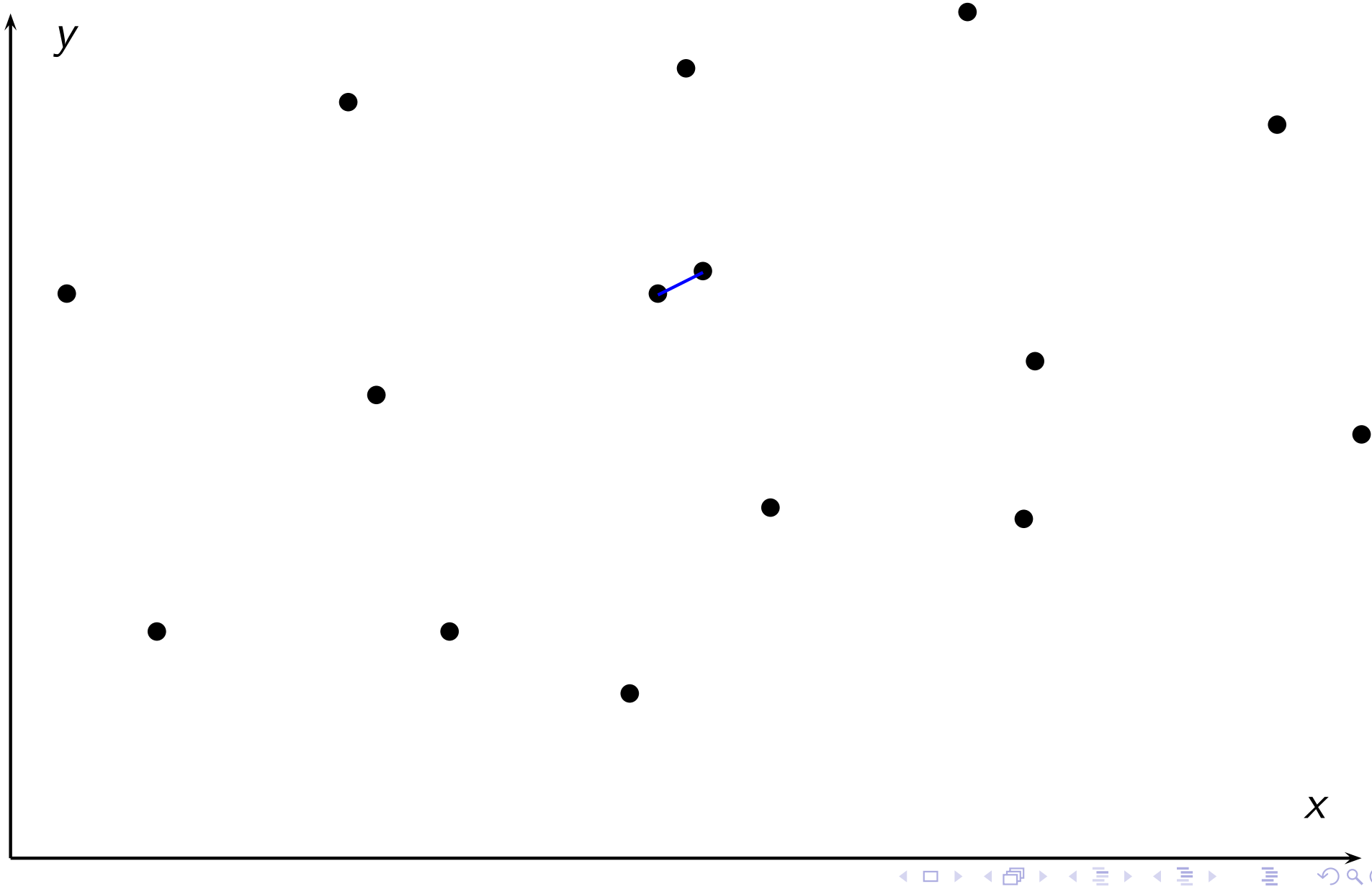
The brute force approach considers each pair in turn (except that once it has found the distance from x to y, it does not need to consider the distance from y to x).

For simplicity, we look at the 2-dimensional case, the points being $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$.

The Closest Pair Problem (Two-Dimensional Case)



The Closest Pair Problem (Two-Dimensional Case)



Brute Force Geometric Algorithms: Closest Pair

$min \leftarrow \infty$
for $i \leftarrow 0$ to $n-2$ **do**
    for $j \leftarrow i+1$ to $n-1$ **do**
        $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$
        if $d < min$ **then**
            $min \leftarrow d$
            $p_1 \leftarrow i$
            $p_2 \leftarrow j$
return p_1, p_2

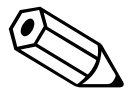
Analysing the Closest Pair Algorithm

It is not hard to see that the algorithm is $\Theta(n^2)$.

Note, however, that we can speed up the algorithm considerably, by utilising the monotonicity of the square root function.

How?

Does this contradict the $\Theta(n^2)$ claim?



Later we shall see how a clever divide-and-conquer approach leads to a $\Theta(n \log n)$ algorithm for this problem.

Brute Force Summary

Simple, easy to program, widely applicable.

Standard approach for small tasks.

Reasonable algorithms for some problems.

But: Generally inefficient—does not scale well.

Use brute force for prototyping, or when it is known that input remains small.

Exhaustive Search

Problem type:

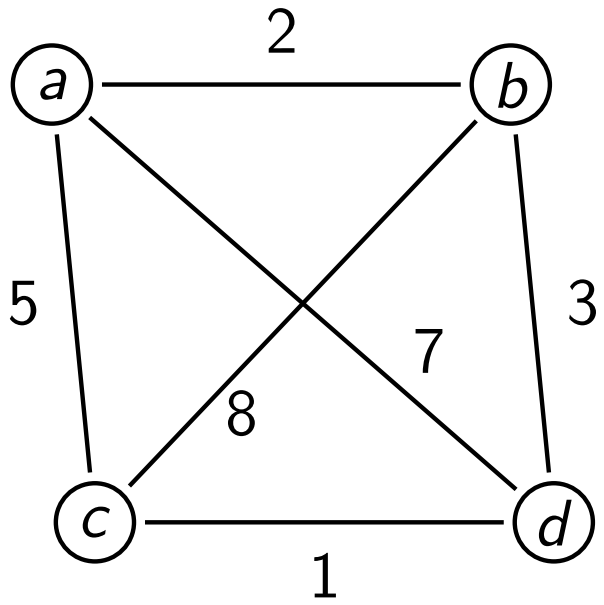
- Combinatorial decision or optimization problems
- Search for an element with a particular property
- Domain grows exponentially, for example all permutations

The brute-force approach—generate and test:

- Systematically construct all possible solutions
- Evaluate each, keeping track of the best so far
- When all potential solutions have been examined, return the best found

Example 1: Travelling Salesperson (TSP)

Find the shortest tour (visiting each node exactly once before returning to the start) in a weighted undirected graph.



$$
\begin{array}{lcl}
a-b-c-d-a & : & 18 \\
a-b-d-c-a & : & 11 \\
a-c-b-d-a & : & 23 \\
a-c-d-b-a & : & 11 \\
a-d-b-c-a & : & 23 \\
a-d-c-b-a & : & 18
\end{array}
$$

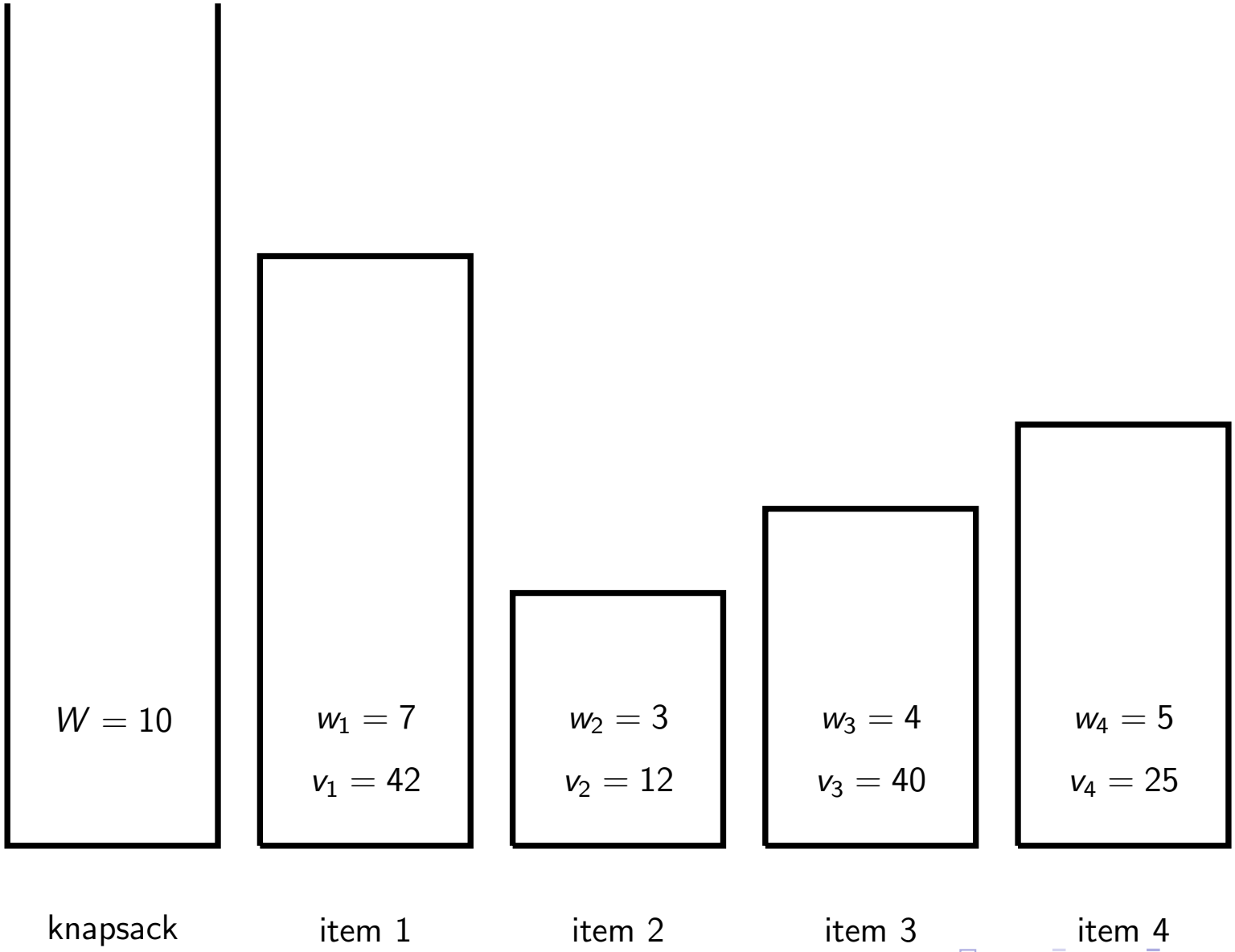
Example 2: Knapsack

Given n items with

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of capacity W

find the most valuable selection of items that will fit in the knapsack.

Example 2: Knapsack



Example 2: Knapsack

Set	Weight	Value
$\emptyset$	0	0
$\{1\}$	7	42
$\{2\}$	3	12
$\{3\}$	4	40
$\{4\}$	5	25
$\{1,2\}$	10	54
$\{1,3\}$	11	NF
$\{1,4\}$	12	NF

Set	Weight	Value
$\{2,3\}$	7	52
$\{2,4\}$	8	37
$\{3,4\}$	9	65
$\{1,2,3\}$	14	NF
$\{1,2,4\}$	15	NF
$\{1,3,4\}$	16	NF
$\{2,3,4\}$	12	NF
$\{1,2,3,4\}$	19	NF

NF means "not feasible": exhausts the capacity of the knapsack.

Later we shall consider a better algorithm based on dynamic programming.

Comments on Exhaustive Search

Exhaustive search algorithms have acceptable running times only for very small instances.

In many cases there are better alternatives, for example, Eulerian tours, shortest paths, minimum spanning trees, ...

For some problems, it is known that there is essentially no better alternative.

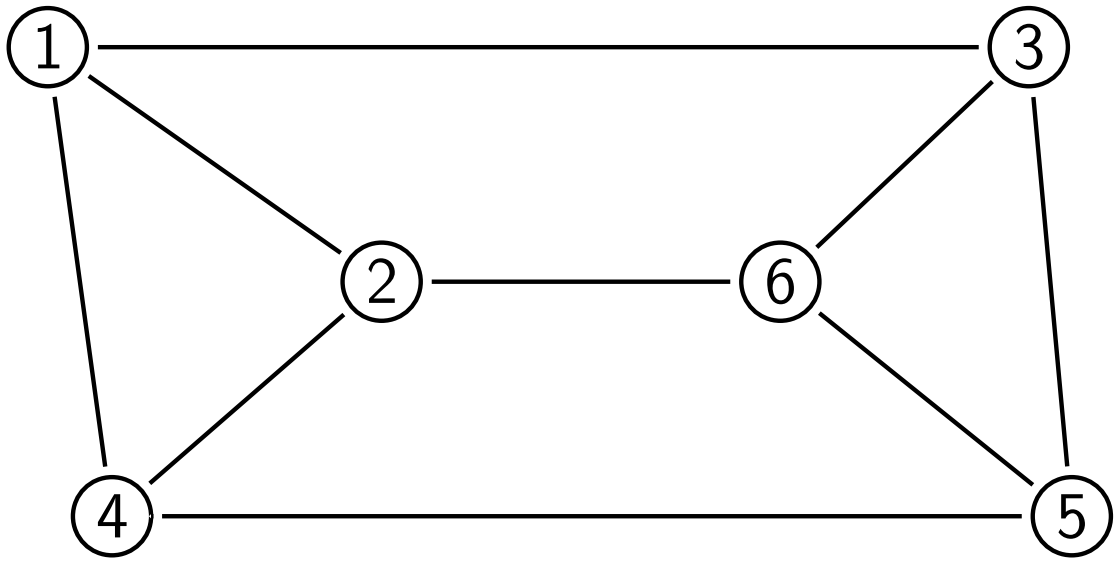
For a large class of important problems, it appears that there is no better alternative, but we have no proof either way.

Hamiltonian Tours

The Hamiltonian tour problem is this:

In a given undirected graph, is there a simple tour (a path that visits each node exactly once, except it returns to the starting node)?

Is there a Hamiltonian tour of this graph?

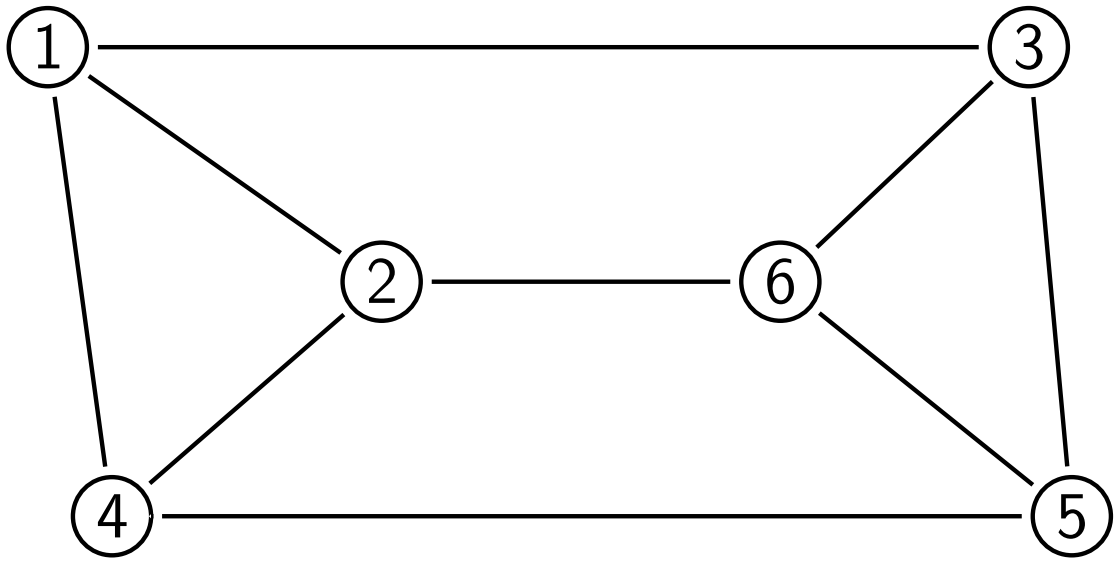


Eulerian Tours

The Eulerian tour problem is this:

In a given undirected graph, is there a path which visits each edge exactly once?

Is there a Eulerian tour of this graph?



Hard and Easy Problems

Recall that by a problem we usually mean a parametric problem: an infinite family of problem "instances".

Sometimes our intuition about the difficulty of problems is not very reliable. The Hamiltonian Tour problem and the Eulerian Tour problem look very similar, but one is hard and the other is easy. We'll see more examples of this phenomenon later.

For many important optimization problems we do not know of solutions that are essentially better than exhaustive search (a whole raft of NP-complete problems, including TSP and knapsack).

In those cases we may look for approximation algorithms that are fast and still find solutions that are reasonably close to the optimal.

We return to this idea in Week 12.

Next Up

Recursion as a problem-solving technique.