南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

02 Basics of Algorithm Analysis

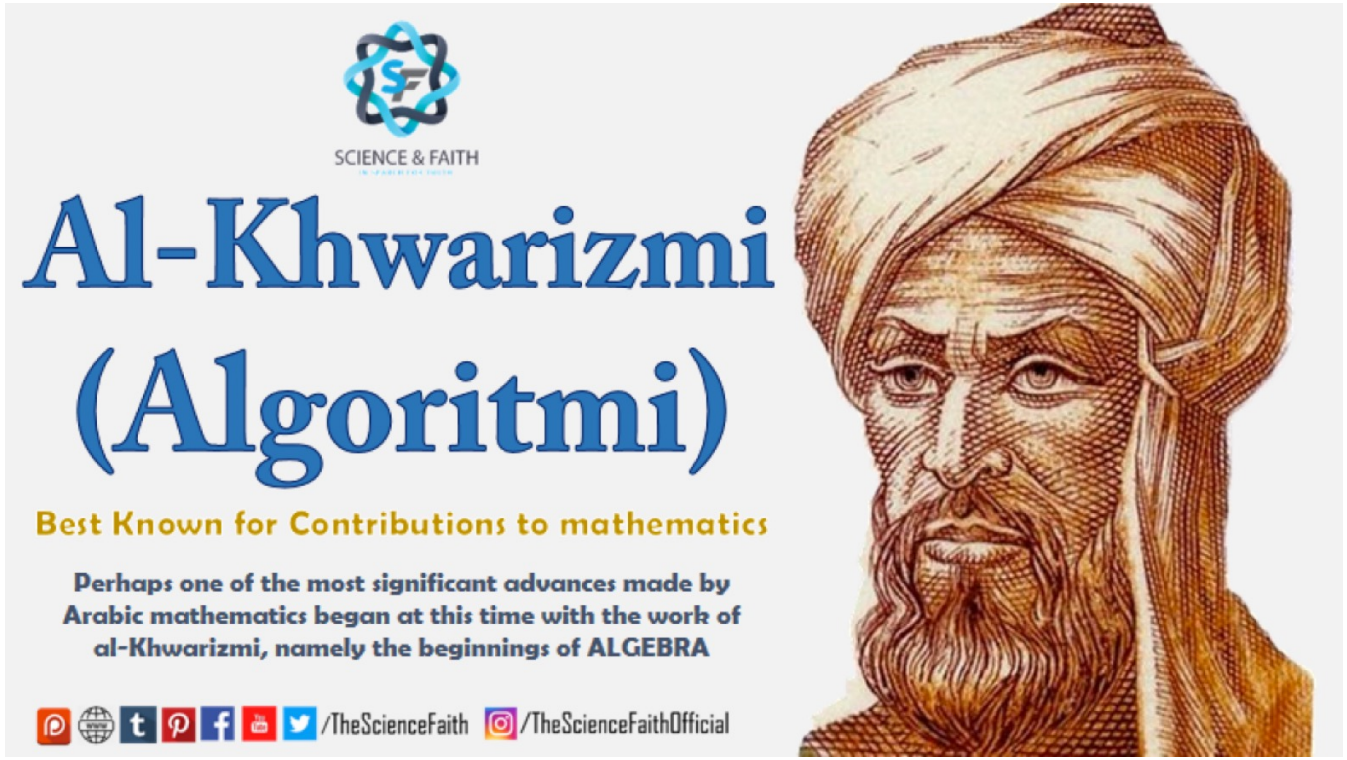
CS216 Algorithm Design and Analysis (H)

Instructor: Shan Chen

chens3@sustech.edu.cn

Algorithms

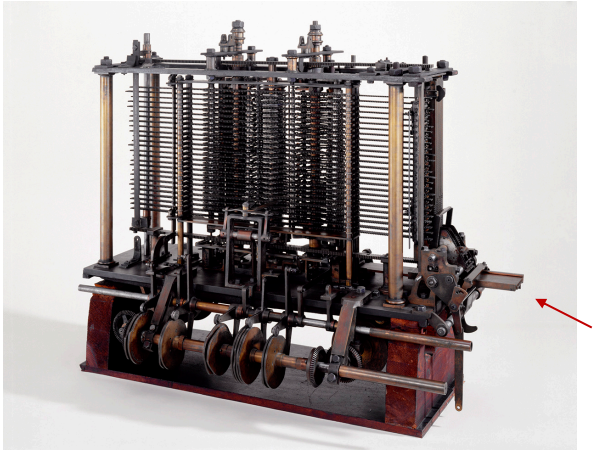
- An **algorithm** is a finite sequence of precise instructions for performing a computation or for solving a problem.



1. Computational Tractability

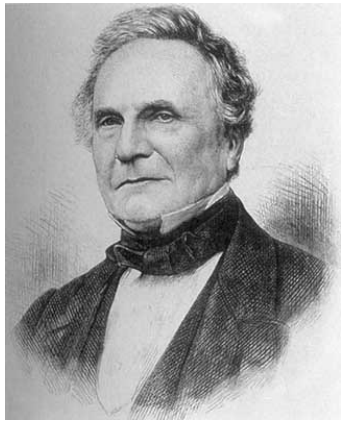
A strikingly modern thought…

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - Charles Babbage (1864)



how many times do you have to turn the crank?

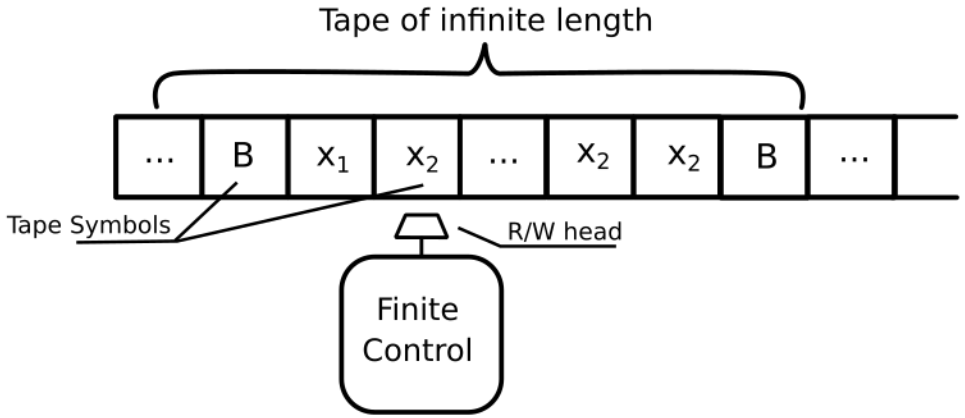
Analytical Engine



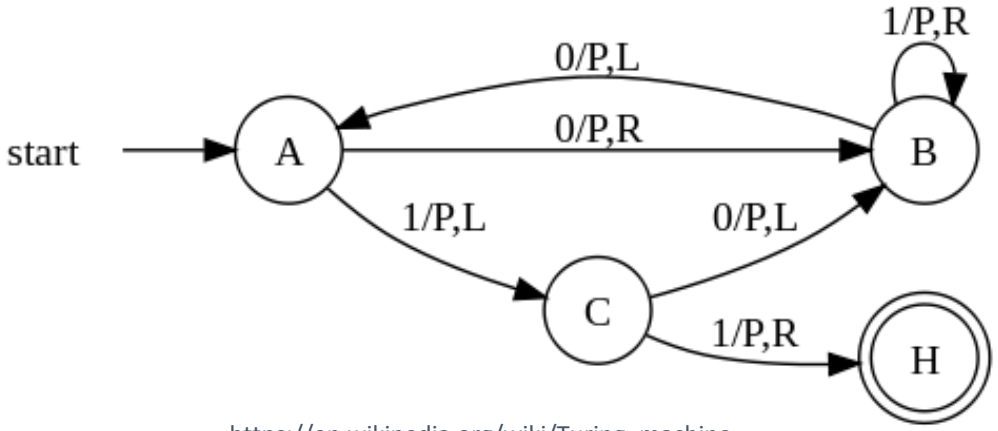
Charles Babbage
(1791 - 1871)

Models of Computation: Turing Machines

- **Deterministic Turing machine.** Simple and idealistic model.



https://iq.opengenus.org/general-introduction-to-turing-machine/

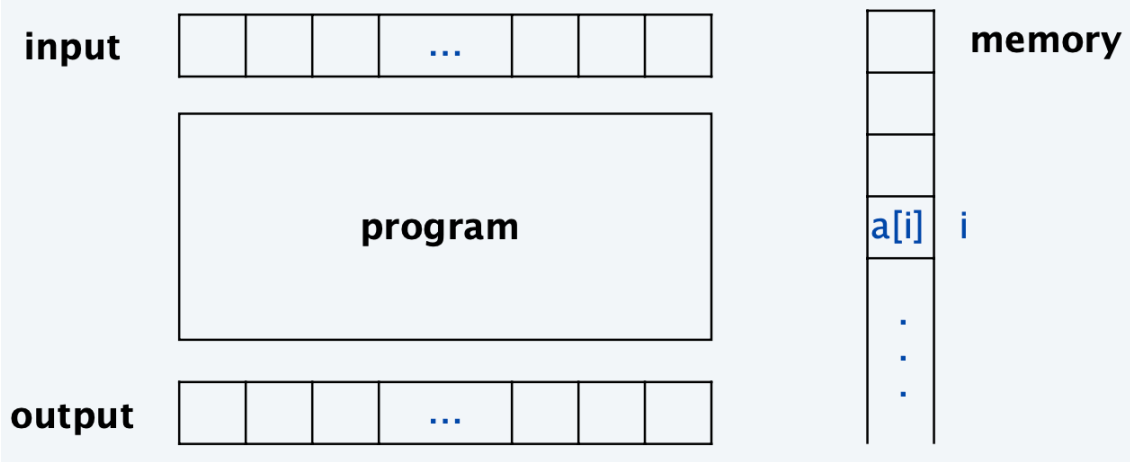


https://en.wikipedia.org/wiki/Turing_machine

- **Running time.** Number of steps. **Memory.** Number of tape cells utilized.
- **Caveat.** No random access of memory.
  - Single-tape TM requires $\geq n^2$ steps to detect n-bit palindromes.
  - Easy to detect palindromes in $\leq cn$ steps on a real computer.

Models of Computation: Word RAM

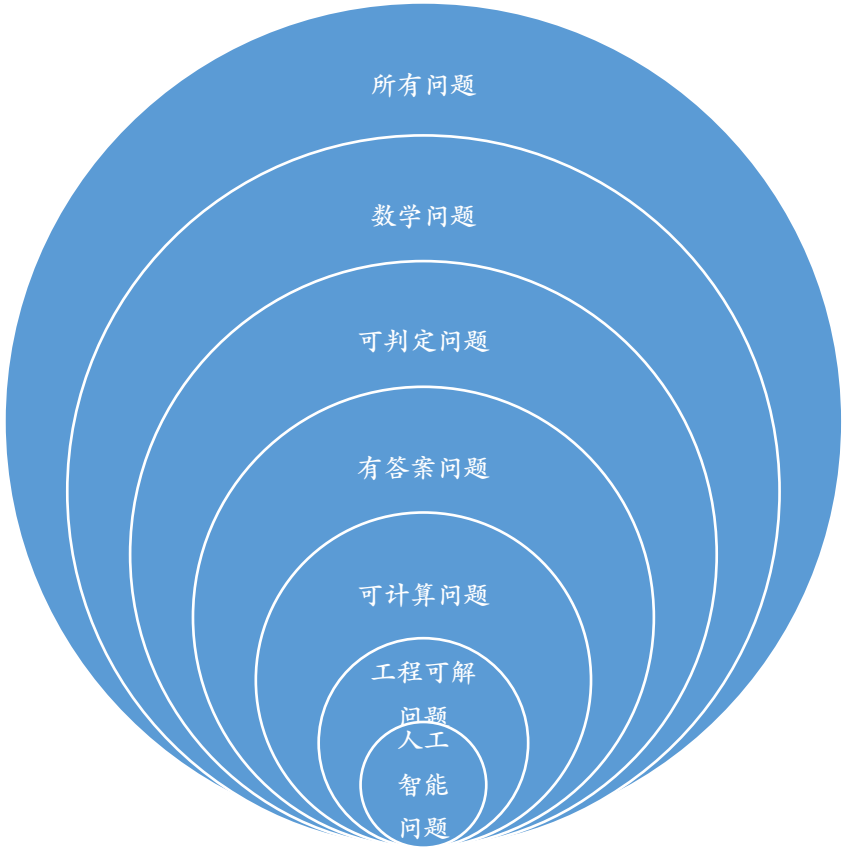
- **Word random-access machine (RAM).** $w \geq \log n$
  - Each memory location and input/output cell stores a w-bit integer/word.
  - Primitive operations:
    - ✓ arithmetic/logic operations
    - ✓ read/write memory
    - ✓ array indexing
    - ✓ following a pointer
    - ✓ conditional branch
    - ✓ ...



- **Running time.** Number of primitive operations.
- **Memory.** Number of memory cells utilized.
- **Caveat.** At times, need more refined model, e.g., multiply n-bit integers.

Limits of Artificial Inteligence

- All problems
- Math problems
  - halting problem is undecidable
- Decidable problems
  - some solutions exist but hard to find
- Solvable problems
  - by any algorithms (e.g., quantum ones)
- Computable problems
  - by Turing machines in finite steps
- Engineering solvabe problems
  - efficiently solvable in practice
- Artificial inteligence problems



吴军，计算之魂，人民邮电出版社, 2021

Computational Complexity

- **Q.** How do we measure the efficiency of an algorithm?
  - We human beings are not good at perception of numbers.

- **A.** Computational complexity: measure efficiency as a function of the algorithm's input size.
  - time complexity: number of primitive computation steps
  - space complexity: number of memory units

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Brute Force vs Polynomial Running Time

- **Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
  - Typically takes 2^n steps (or worse) for inputs of size n.
  - Unacceptable in practice.

- **Desirable scaling property.** If the input size is increased by a constant factor C_1, the algorithm should also slow down by a constant factor C_2.
- **Def.** An algorithm is polynomial-time if there exist constants $a > 0$, $b > 0$ such that the running time of the algorithm is bounded by an^b.
  - Note that the above scaling property holds, e.g., choose $C_2 = C_1^b$.

Polynomial Running Time

- We say that an algorithm is efficient if has a polynomial running time.
- **Theory.** Definition is (relatively) insensitive to models of computation.
- **Practice.** It really works!
  - The polynomial-time algorithms that people develop almost always have both small constants and small exponents.
  - Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
- **Exceptions:**
  - Some polynomial-time algorithms in the wild have galactic constants and/or huge exponents, then useless in practice, e.g., $6.02 \times 10^{23} \times n^{20}$.
  - Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare, e.g., the simplex method.

Worst-Case/Average-Case Analysis

- **Worst-case analysis.** Worst possible running time on any input of size *n*.
  - Generally captures efficiency in practice.
  - Draconian view, but hard to find effective alternative.

- **Average-case analysis.** Average running time on random input of size *n*.
  - Hard (or impossible) to accurately model real instances by random distributions.
  - Algorithms tuned for a certain distribution may perform poorly on other distributions.

Other Types of Analyses

- **Probabilistic.** Expected running time of a randomized algorithm.
  - Note that here the algorithm itself is randomized.
  - E.g., the expected number of compares to quicksort *n* elements is ~ *2n ln n*.

shown in later sections

- **Amortized analysis.** Worst-case running time running time for any sequence of *n* operations.
  - E.g., pushing *n* elements to a JAVA ArrayList (dynamic array) takes ~ *4n* steps.

will be discussed in our first lab lecture

2. Asymptotic Order of Growth

Notations

- Let function $T(n)$ denote the worse-case running time of an algorithm on any input of size n.
- **Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $0 \leq T(n) \leq c \cdot f(n)$.
- **Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n) \geq 0$.
- **Tight bounds.** $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.
- Example: $T(n) = 32n^2 + 17n + 32$
  - $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
  - $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Remark on the Use of Notations

- **Slight abuse of notation.** $T(n) = O(f(n))$
  - $=$ is not transitive:
    - ✓ $f(n) = 5n^3$; $g(n) = 3n^2$
    - ✓ $f(n) = O(n^3) = g(n)$
    - ✓ $f(n) \neq g(n)$.
  - better notation: $T(n) \in O(f(n))$

- **Vacuous statement.** Any compare-based sorting algorithm requires at least $O(n \log n)$ compares in the worst case.
  - The statement does not "type-check".
  - Should use Ω for lower bounds.

Properties

- **Reflexivity and constants:**
  - $f = O(f)$, $f = \Omega(f)$, $f = \Theta(f)$
  - If $c > 0$ is a constant, then $cf = O(f)$, $cf = \Omega(f)$, $cf = \Theta(f)$.
- **Transitivity:**
  - If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
  - If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
  - If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.
- **Sums and Products:**
  - If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$ and $f \cdot g = O(h^2)$.
  - If $f = \Omega(h)$ and $g = \Omega(h)$, then $f + g = \Omega(h)$ and $f \cdot g = \Omega(h^2)$.
  - If $f = \Theta(h)$ and $g = \Theta(h)$, then $f + g = \Theta(h)$ and $f \cdot g = \Theta(h^2)$.

Asymptotic Bounds for Some Common Functions

- **Polynomials.** If $a_d > 0$, then $a_0 + a_1 n + \ldots + a_d n^d = \Theta(n^d)$.
- **Polynomial time.** The running time is $O(n^d)$ for some constant $d > 0$ independent of the input size n.
- **Logarithms.** $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 1$.

  no need to specify log base

- **Logarithms < polynomials.** For every $a > 1, c > 0$, $\log_a n = O(n^c)$.
- **Exponentials > polynomials.** For every $r > 1, d > 0$, $n^d = O(r^n)$.
- **Factorials.** $n! = 2^{\Theta(n \log n)}$, i.e., $\log_2 n! = \Theta(n \log n)$.
  - Can be proved by Stirling's formula: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
- **Factorials > exponentials.** For every $r > 1$, $r^n = O(n!)$.

3. A Survey of Common Running Times

Sublinear Time: *O(log n)*

- **Logarithmic time.** Running time proportional to logarithm of input size.

- **Binary search.** Search a given number in a sorted array of size *n*.
  - Terminates within $O(\log_2 n)$ steps

Linear Time: $O(n)$

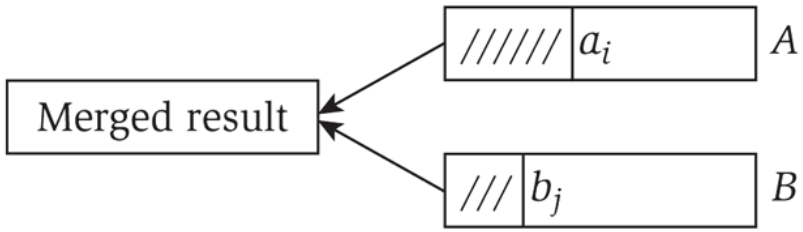
- **Linear time.** Running time proportional to input size n.

- **Computing the maximum.** Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max = a1
for i = 2 to n {
    if (ai > max)
        max = ai
}
```

Linear Time: $O(n)$ – Another Example

- **Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_n$ into one sorted list.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else append b_j to output list and increment j
}
append remainder of the nonempty list to output list
```

- **Claim.** Merging two lists of size n takes $O(n)$ time.
- **Pf.** After each comparison, the length of output list increases by 1.

Linearithmic Time: *O(n log n)*

- **Linearithmic time.** Arises in divide-and-conquer algorithms.

- **Sorting.** Mergesort and Heapsort are sorting algorithms that perform $O(n \log n)$ comparisons.

Quadratic Time: $O(n^2)$

- **Quadratic time.** Enumerate all pairs of elements.
- **Closest pair of points.** Given n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is the closest.
  - $O(n^2)$ solution: try all pairs of points.

```
min = (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n {
   for j = i+1 to n {
      d = (xi - xj)^2 + (yi - yj)^2
      if (d < min)
         min = d
   }
}
```

← no need to take square roots

better solution in later sections

- **Remark.** $\Omega(n^2)$ seems inevitable, but this is just an illusion.

Cubic Time: $O(n^3)$

- **Cubic time.** Enumerate all triples of elements.
- **Set disjointness.** Given n sets $S_1, \ldots, S_n$, each of which is a subset of $1, 2, \ldots, n$, is there some disjoint pair of these sets?
  - $O(n^3)$ solution: for each pairs of sets, check if they are disjoint.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
        determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
        report that Si and Sj are disjoint
   }
}
```

Polynomial Time: $O(n^k)$

- **Independent set of size k.** Given a graph and a constant k, are there k nodes such that no two nodes are joined by an edge?
  - $O(n^k)$ solution: enumerate all subsets S of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
}
```

  - Check whether S is an independent set is $O(k^2)$
  - Number of k element subsets is $O(n^k / k!)$
  - Overall: $O(k^2 n^k / k!) = O(n^k)$

still polynomial time for k = 17, but not practical

Exponential Time: $O(c^n)$

- **Independent set of maximum size.** Given a graph, what is maximum size of an independent set?
  - $O(n^2 2^n)$ solution: Enumerate all subsets.

```
S* = empty set
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* = S
}
```

Factorial Time: *O(n!)*

- **Factorial time.** Enumerate all permutations of length *n*.

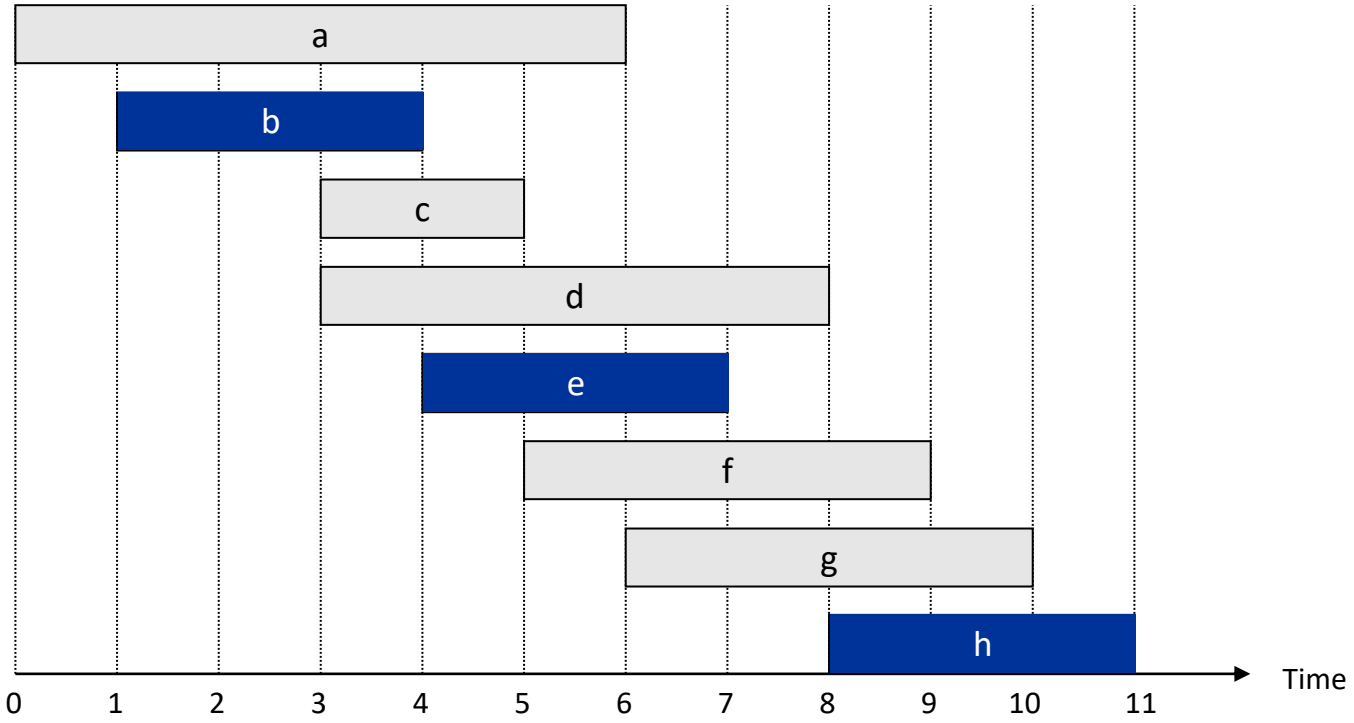
- **Traveling salesman problem (TSP).** Given a set of *n* cities, with distances between all pairs, what is the shortest tour that visits all cities?
  - *O(n!)* solution: search all *(n - 1)!* tours (salesman starts and ends at the same city)

4. Five Representative Problems on Independent Set

Interval Scheduling

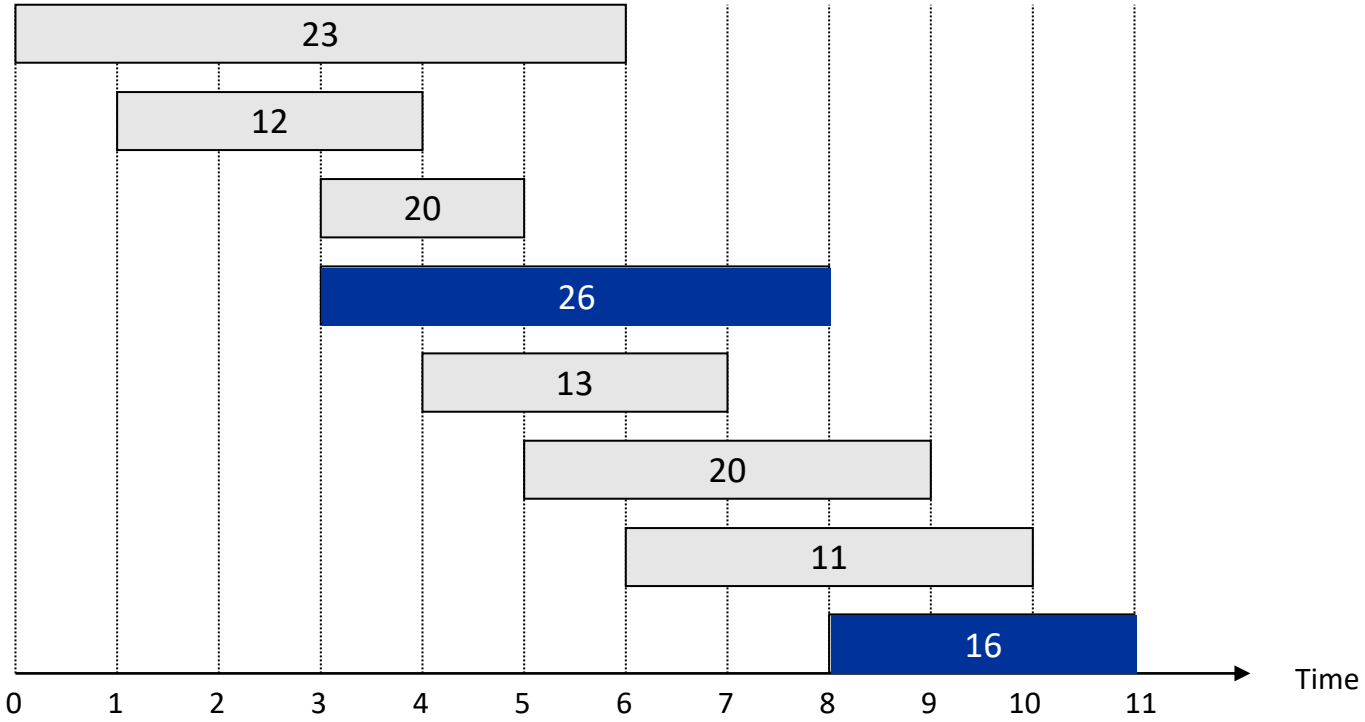
- **Input.** Set of jobs with start times and finish times.
- **Goal.** Find maximum-size subset of mutually compatible jobs.

jobs don't overlap



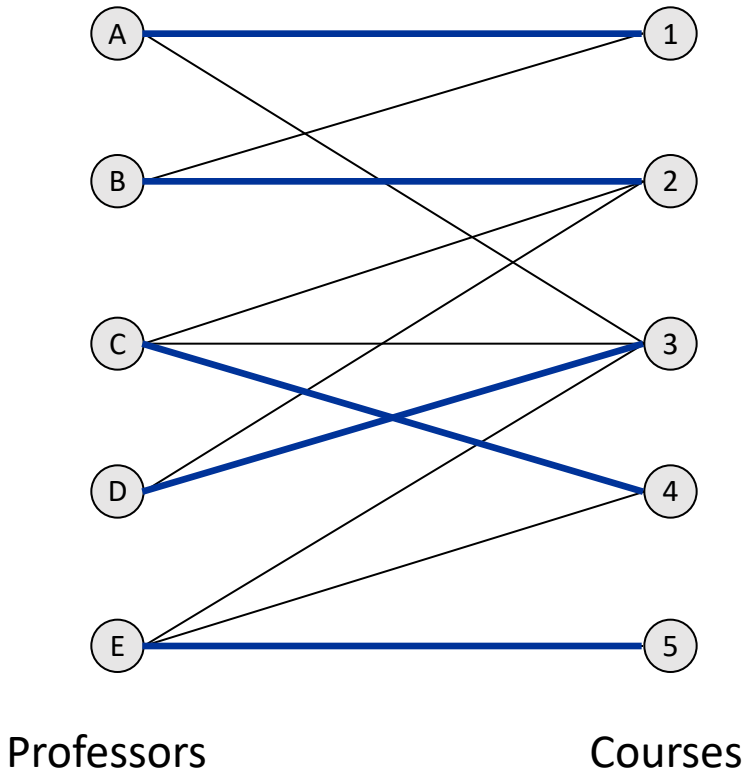
Weighted Interval Scheduling

- **Input.** Set of jobs with start times, finish times, and weights.
- **Goal.** Find maximum-weight subset of mutually compatible jobs.



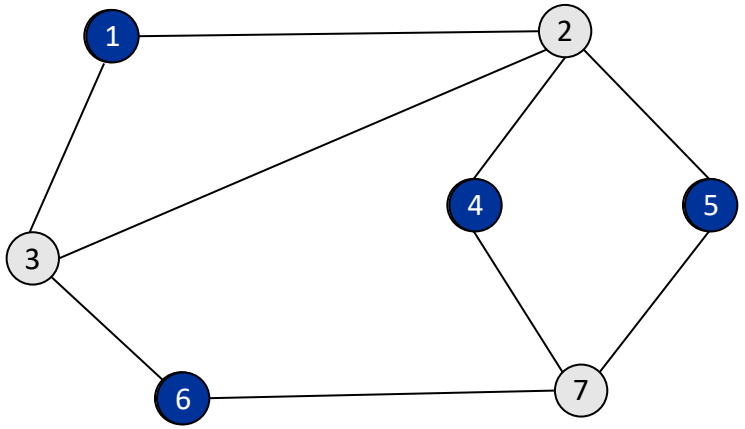
Bipartite Matching

- **Input.** Bipartite graph.
- **Goal.** Find maximum-size matching.



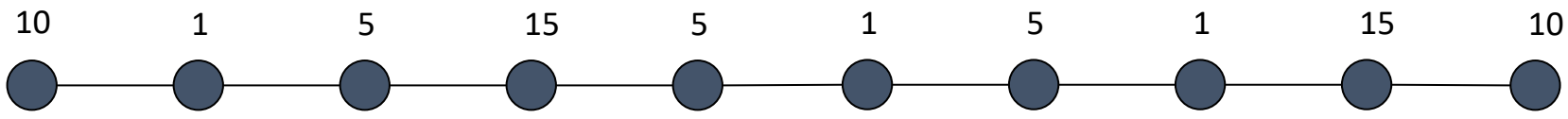
Independent Set

- **Input.** Graph.
- **Goal.** Find maximum-size independent set.



Competitive Facility Location

- **Input.** Graph with weight on each node.
- **Game.** Two competing players alternately select nodes (facility locations). Not allowed to select a node if any of its neighbors has been selected.
- **Goal.** Select a maximum-weight subset of nodes.
- **Q.** Given a bound B, can the second player P_2 get B no matter how the first player P_1 plays?
  - ➢ It is hard to determine if P_2's winning strategy exists; it would even be hard to convince people some strategy is indeed a winning strategy.



P_2 can guarantee 20, but not 25.

Summary of Five Representative Problems

- **Interval scheduling:** *O(n log n)* greedy algorithm.
- **Weighted interval scheduling:** *O(n log n)* dynamic programming algorithm.
- **Bipartite matching:** $O(n^k)$ max-flow based algorithm.
- **Independent set:** *NP*-complete. ← we will learn this in later sections
- **Competitive facility location:** *PSPACE*-complete.

we may not capture this more advanced notion
but see the textbook if you are interested

- All the above are variations on the same theme: independent set

Announcements

- We will learn amortized analysis and some basic lab techniques in our lab class tonight.

- **Lab 1 will be released today and the deadline is March 5.**