南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Algorithm Design and Analysis (H)

CS216

Instructor: Shan CHEN (陈杉)

chens3@sustech.edu.cn

(slides edited from Prof. Shiqi Yu)

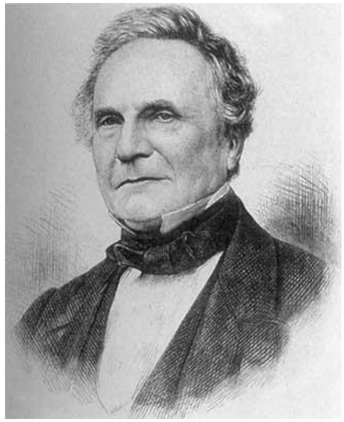
PEARSON
Addison Wesley

1. Computational Tractability

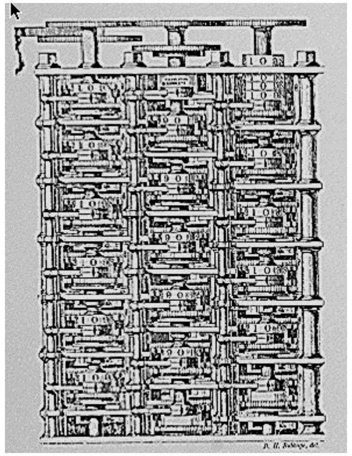
"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - *Francis Sullivan*

Computational Tractability

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*



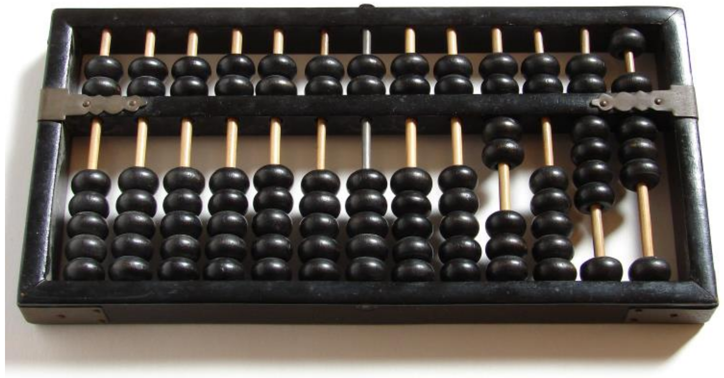
Charles Babbage
(1864)



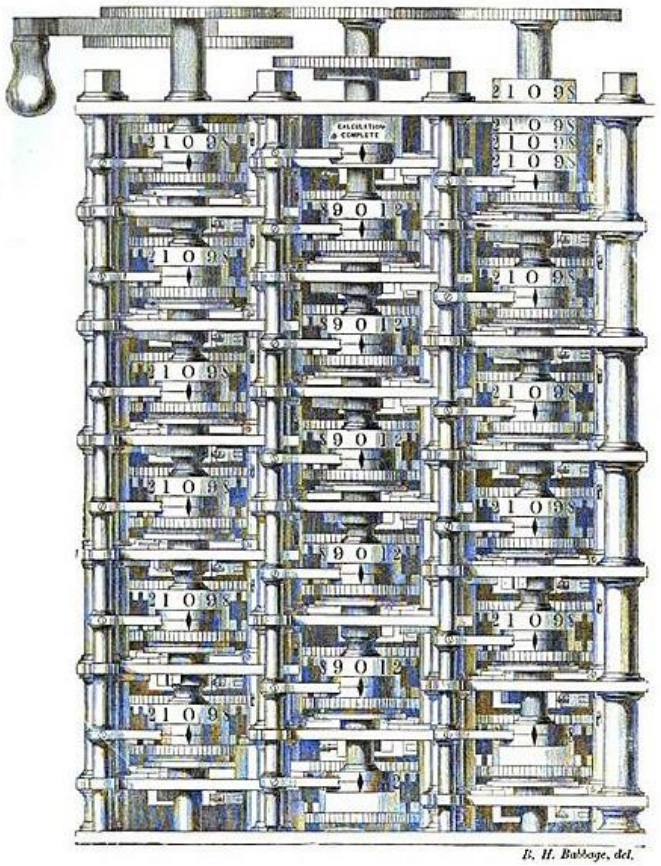
Analytic Engine
(schematic)

Computers

- A computer is a ~~digital electronic~~ machine that can be programmed to carry out sequences of arithmetic or logical operations (computation) automatically.
- Components
  - Calculation capability
  - Storage
  - Instructions



A working model of Babbage's Difference Engine. It was designed in the 1820s by Charles Babbage. It is an automatic mechanical calculator designed to tabulate polynomial functions. It operated on 6-digit numbers and second-order differences and was intended to operate on 20-digit numbers and sixth-order differences.

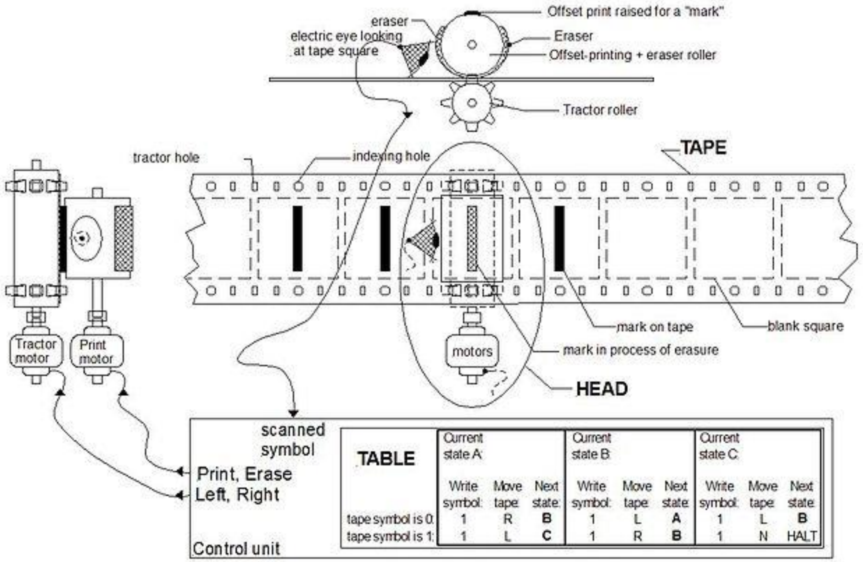


Computers

- Boolean algebra by George Boole
- A master’s thesis by Claude Elwood Shannon: Electrical applications of Boolean algebra could construct any logical numerical relationship.
  - Mechanical-> Electrical

Turing Machine

- Fundamental problems considered by Alan Turing:
  - ➢ Are all math problems solvable?
  - ➢ What problems can we solve in finite steps?
  - ➢ For such problems, can we design a machine that terminates with a solution?

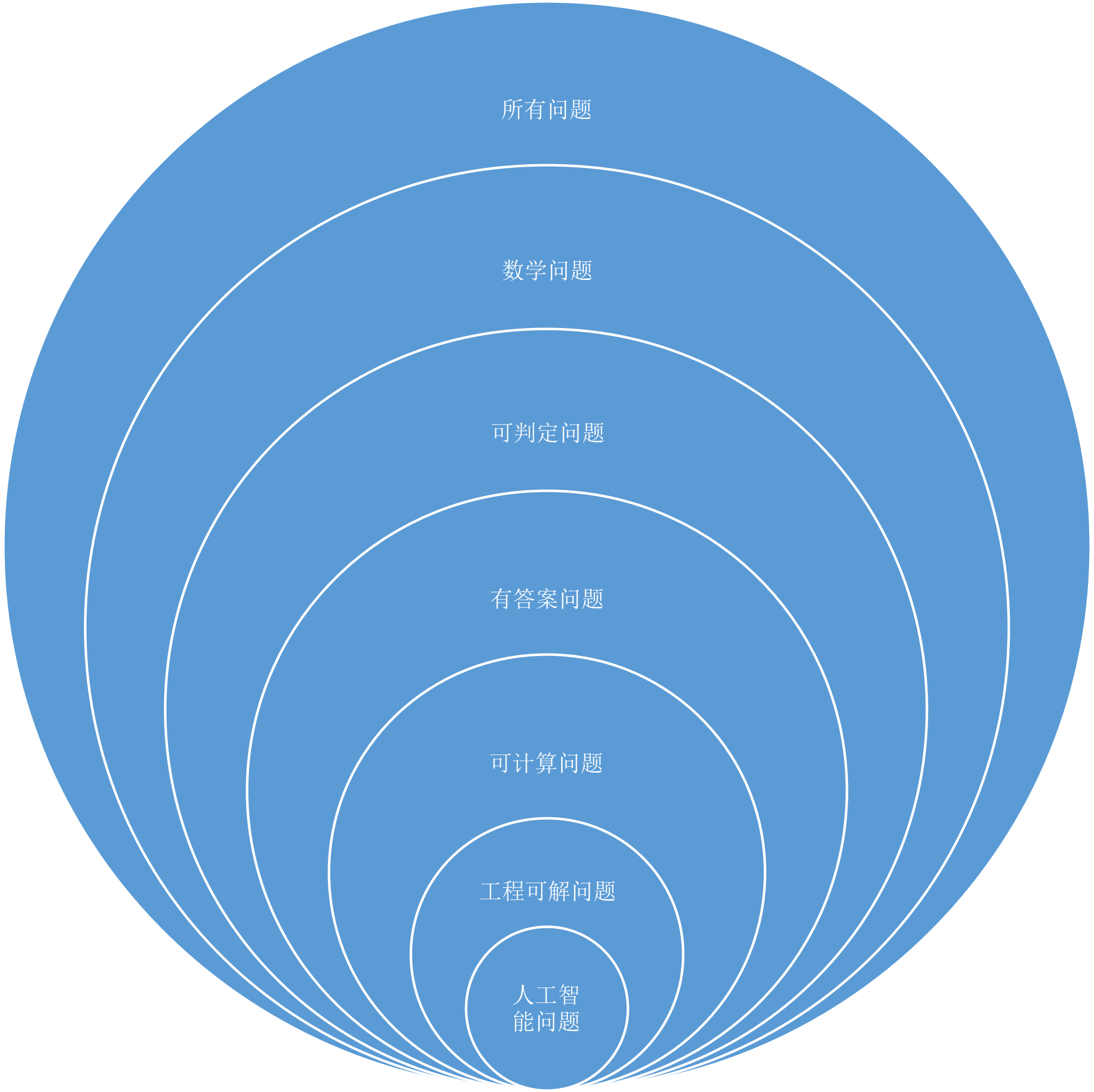


A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

吴军，计算之魂，人民邮电出版社, 2021

Limits of Artificial Inteligence

- Not all problems are math problems:
  - Completeness, consistency, effective axiomatization
- Math problems
- Decidable problems
- Solvable problems
- Computable problems
  - by Turing machines
- Engineering solvabe problems
  - computational complexity
- Artificial inteligence problems



吴军，计算之魂，人民邮电出版社, 2021

Computational Complexity

- How do we measure the efficiency of an algorithm?
  - We human beings are NOT good at perception of numbers.
  - One could measure efficiency as a function of the algorithm's input size.
- Computational complexity
  - Time complexity: number of primitive computation steps
  - Space complexity: number of memory units

Worst-Case/Average-Case Analysis

- **Worst-case running time**. Obtain bound on largest possible running time of algorithm on input of a given size N.
  - Generally captures efficiency in practice.
  - Draconian view, but hard to find effective alternative.

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

Polynomial-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 time or worse for inputs of size N. ← n ! for stable matching with n men and n women
  - Unacceptable in practice.
- **Desirable scaling property**. If the input size is increased by a constant factor C_1, the algorithm should only slow down by some constant factor C_2.

> There exist constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c N^d$ steps.

- **Def**. An algorithm is polynomial-time if its running time is bounded by cN^d.
  - The above scaling property holds: choose $C_2 = C_1^d$.

Worst-Case Polynomial-Time

- **Def.** An algorithm is efficient if its worst-case running time is polynomial.
- **Justification**: It really works in practice!
  - In practice, the polynomial-time algorithms that people develop almost always have small constants and small exponents.
  - Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
- **Exceptions**:
  - Some poly-time algorithms do have high constants and/or exponents, and are 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.

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

2. Asymptotic Order of Growth

Asymptotic Order of Growth

- The worse-case running time of an algorithm on an input of size n is measured by a function $T(n)$.
- **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 $T(n) \leq c \cdot f(n)$.
- **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 $T(n) \leq c \cdot f(n)$.
- **Tight bounds**. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.
- **Ex:** $T(n) = 32n^2 + 17n + 32$.
  - $T(n)$ 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 $\Theta(n^3)$.

Notation

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

- **Meaningless statement**. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.
  - Statement doesn't "type-check".
  - Use Ω for lower bounds.

Properties

- **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 $f = \Theta(h)$.

- **Additivity/Multiplicativity.**
  - 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**. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.
- **Polynomial time**. Running time is $O(n^d)$ for some constant d independent of n.

- **Logarithms**. $O(\log_a n) = O(\log_b n)$ for any constants a, b > 0.

can avoid specifying the base

- **Logarithms**. For every x > 0, $\log n = O(n^x)$.

log grows slower than every polynomial

- **Exponentials**. For every r > 1 and every d > 0, $n^d = O(r^n)$.

every exponential grows faster than every polynomial

3. A Survey of Common Running Times

Sublinear Time: O(log n)

- **Logarithmic time**. Running time is 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)

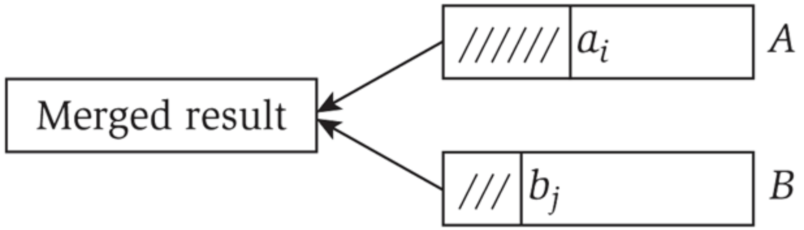
- **Linear time**. Running time is proportional to input size.

- **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)

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




```
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.

O(n log n) Time

- **Linearithmic time**. Arises in divide-and-conquer algorithms.
- **Sorting**. Mergesort and Heapsort are sorting algorithms that perform O(n log n) comparisons.
- **Largest empty interval**. Given n timestamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?
  - **O(n log n) solution**. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive timestamps.

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, …, n, is there some disjoint pair of these sets?
  - **$O(n^3)$ solution**. For each pairs of sets, determine 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, are there k nodes such that no two are joined by an edge?

k is a constant

  - **$O(n^k)$ solution.** Enumerate all subsets 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 = $O(k^2)$.
- Number of k element subsets = $O(k^2 n^k / k!) = O(n^k)$.

e.g., poly-time for k = 17, but not practical

Exponential Time: $O(c^n)$

- **Independent set**. 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!)

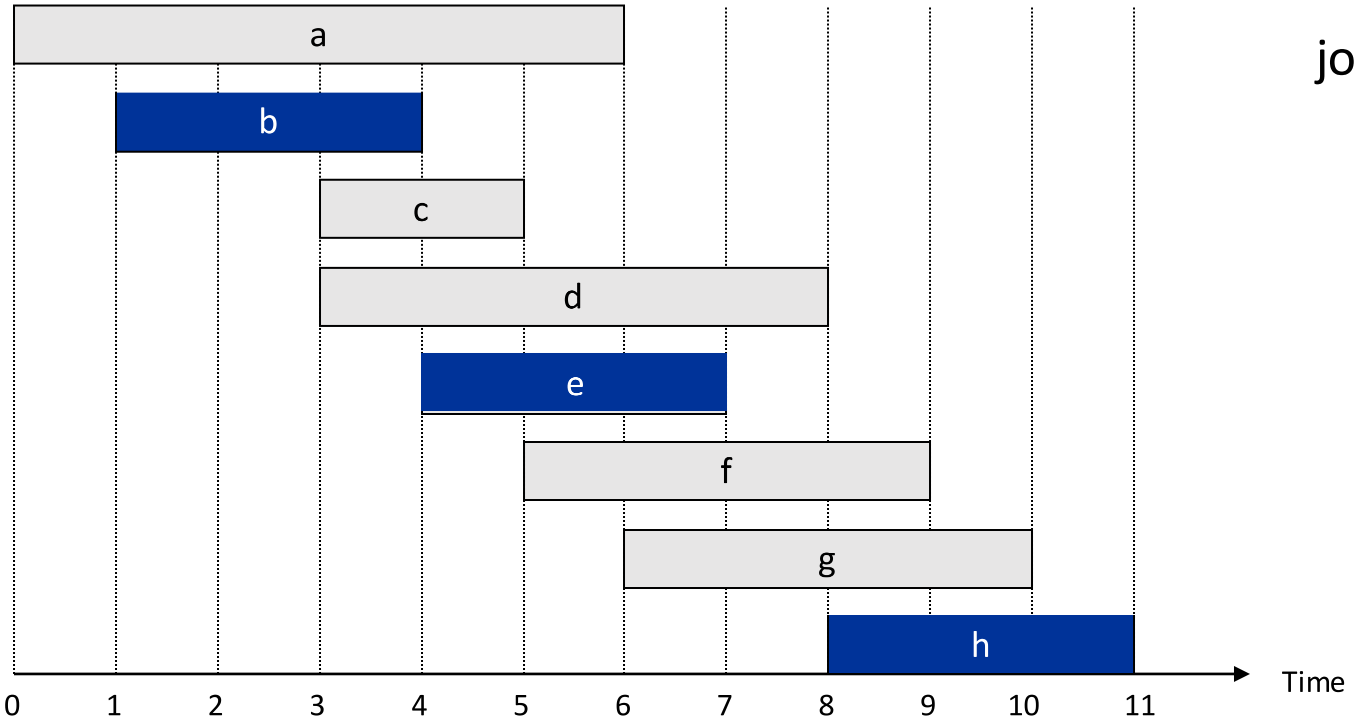
- The function n! grows even more rapidly than c^n for any $c > 1$.
  - ➢ $n! = 1 \times 2 \times 3 \times \ldots \times n$
  - ➢ $c^n = c \times c \times c \times \ldots \times c$
- All perfect matchings for n men and n women: **n!**
- **Traveling salesman problem**. Given a set of n cities, with distances between all pairs, what is the shortest tour that visits all cities?
  - ➢ The salesman starts and ends at the first city: search **(n - 1)!** tours.

4. Five Representative Problems on Independent Set

Interval Scheduling

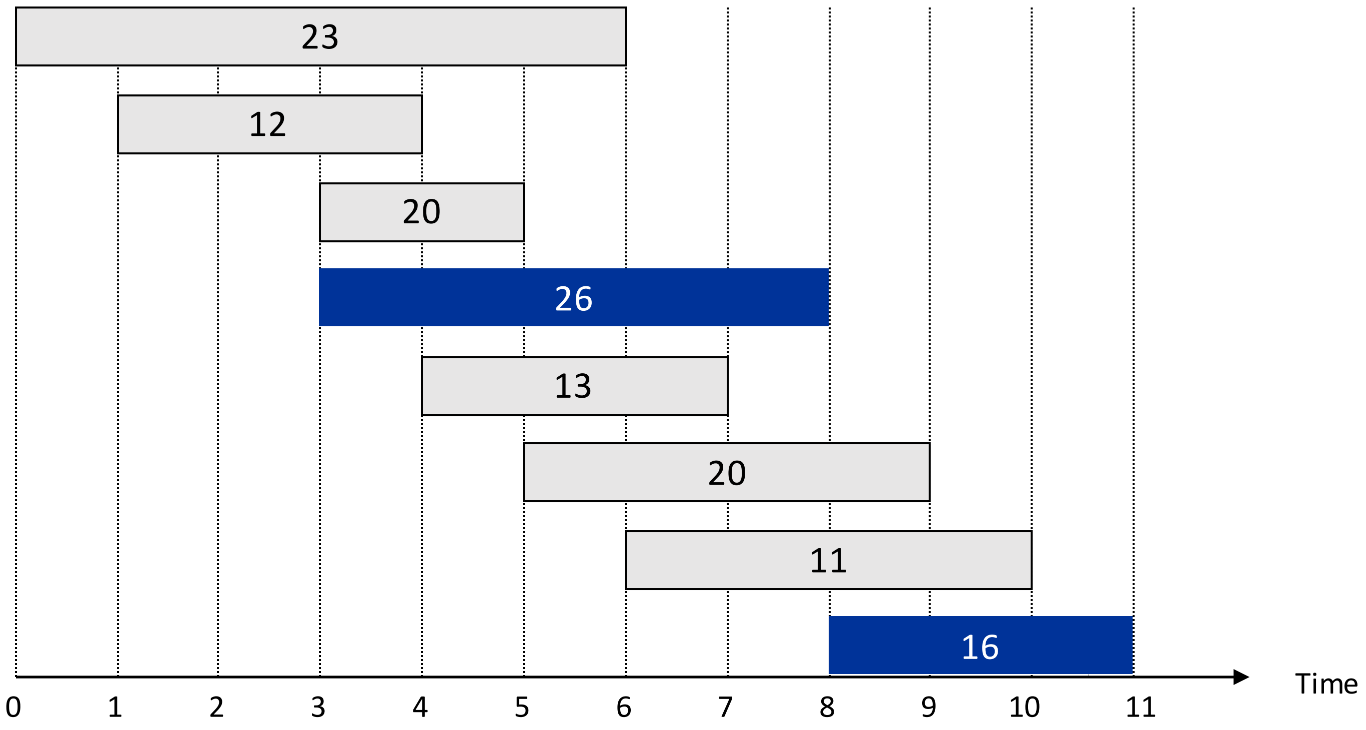
- Input. Set of jobs with start times and finish times.
- Goal. Find maximum-cardinality subset of mutually compatible jobs.

jobs don't overlap



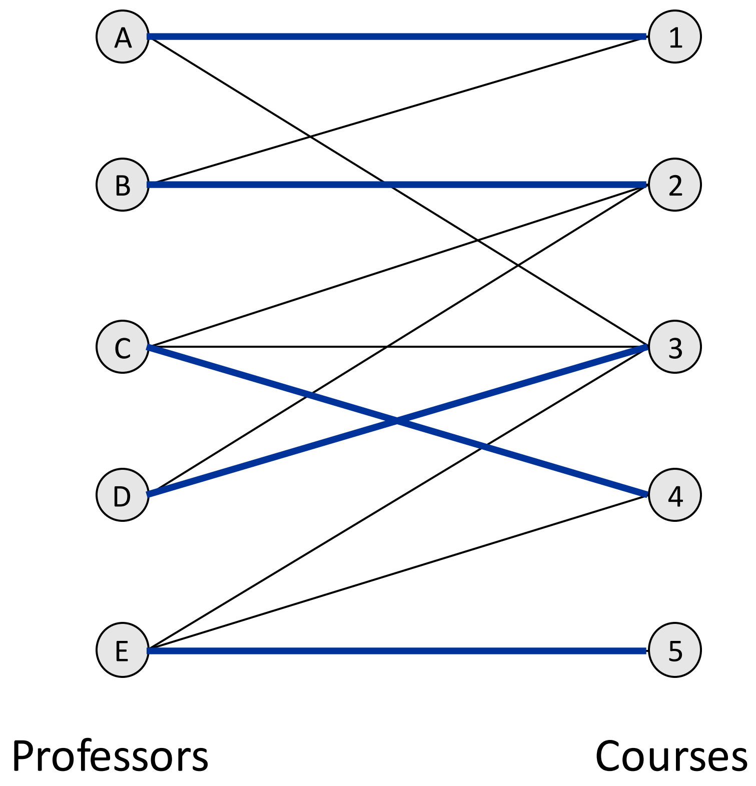
Weighted Interval Scheduling

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



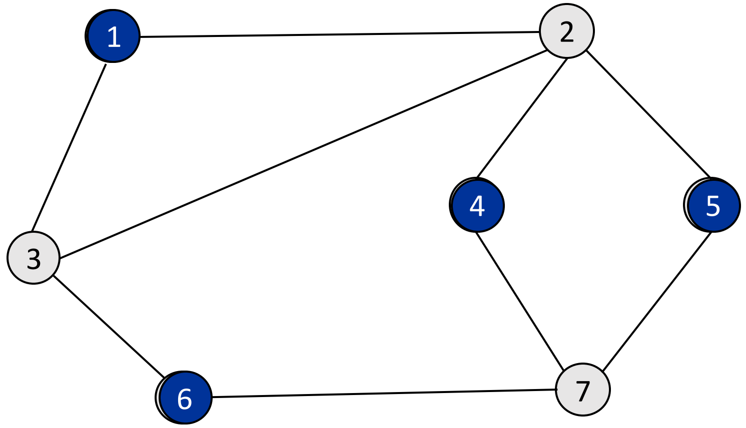
Bipartite Matching

- Input. Bipartite graph.
- Goal. Find maximum-cardinality matching.



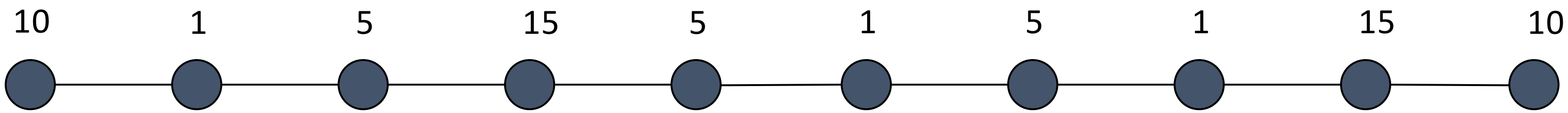
Independent Set

- Input. Graph.
- Goal. Find maximum-cardinality independent set.



Competitive Facility Location

- Input. Graph with weight on each node.
- Game. Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.
- Goal. Select a maximum weight subset of nodes. Given a bound B, can the second player P2 get B no matter how the first player P1 plays?



P2 can guarantee 20, but not 25.

- Not only hard to determine if the winning strategy of P2 exists, it is even hard to convince people P2 has a winning strategy even if we find it.

Five Representative Problems

- Variations on a theme: independent set.

- 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.
- Competitive facility location: PSPACE-complete.