$k \geq 1, b > 1, d \geq 0, w \geq 0$

$$T(n) = k \cdot T(n/b) + O(n^d)$$

$$= \begin{cases} O(n^d) & \text{if } k < b^d \\ O(n^d \log n) & \text{if } k = b^d \\ O(n^{\log_b k}) & \text{if } k > b^d . \end{cases}$$

$$T(n) = kT(n/b) + O(n^d \log^w n)$$

$$\begin{cases} O(n^d \log^w n) & \text{if } k < b^d \\ O(n^d \log^{w+1} n) & \text{if } k = b^d \\ O(n^{\log_b k}) & \text{if } k > b^d \end{cases}$$

D&C: Int multi

- Let's stare at this identity again:

$$xy = (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$$
$$= ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd$$

- **Think:**
  - Could we write $ad + bc$ in terms of ac (t_1), bd (t_3),
  - and *something else* that only uses ***one multiplication*** (not two)?

$$ad + bc = (a+b)(c+d) - ac - bd$$

D&C: Closest Pair(nlogn)

Sorted by x	Sorted by y

ClosestPair$(P_1, \ldots, P_n, P'_1, \ldots, P'_n)$: // $n \geq 3$ pts in the plane
if $n \leq 3$ then return min dist among P_1, P_2, P_3 // base case
$(L, R) \leftarrow$ partition points by $P_{n/2}$ // split by median x-coordinate
$\delta_1 \leftarrow$ **ClosestPair**(L (sorted by x), L (sorted by y)) // min dist on left
$\delta_2 \leftarrow$ **ClosestPair**(R (sorted by x), R (sorted by y)) // min dist on right
$\delta_3 \leftarrow$ min distance in δ-strip // details in notes
return $\min\{\delta_1, \delta_2, \delta_3\}$

DP: Runtime = Size of table * time to fill everv entrv

DP: Task selection(nlogn if sorted time)

OPT has **Jn** (use it!) ⇦ **OTV(J₁, …, Jn) = val(Jn) + OTV(J₁, …, Ji)**

Ji is the last interval that doesn't overlap with **Jn**

OPT doesn't have **Jn** (lose it!) ⇦ **OTV(J₁, …, Jn) = OTV(J₁, …, Jn-1)**

"Optimal Task Value" i.e. the value of the optimal solution

DP: Longest Incresing subseq(n^2)

Define $CLIS(S[1..N])$ to be the longest of the increasing subsequences of $S[1..N]$ that contain $S[N]$.
Define L(i) to be the length of $CLIS(S[1..i])$
Then, we have:

$$L(i) = 1 + \max\{L(j) : 0 < j < i \text{ and } S[j] < S[i]\}$$

DP: LCS(MN)

- **Def**: $LCS(i, j)$ = *length* of a LCS of $X[1..i]$ and $Y[1..j]$.
  - $i = 0$ means X is the empty string
  - $j = 0$ means Y is the empty string
- **Goal**: return $LCS(m, n)$
- We have: $LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS(i-1, j-1) & \text{if } X[i] = Y[j] \\ \max\begin{cases} LCS(i-1, j), \\ LCS(i, j-1) \end{cases} & \text{if } X[i] \neq Y[j] \end{cases}$

DP:0-1(nW)

- **The recurrence:**

$$Knapsack(\{t_1, \ldots, t_n\}, W) = \max \begin{cases} Knapsack(\{t_1, \ldots, t_{n-1}\}, W) \\ v_n + Knapsack(\{t_1, \ldots, t_{n-1}\}, W - w_n) \end{cases}$$

Valid only when $W \geq w_n$

- **The base cases:**
  - $Knapsack(\emptyset, W') = 0$ for all W'
  - $Knapsack(\{t_1, \ldots, t_i\}, 0) = 0$ for all i

DP:Longest Pallindromic substring

$$PAL(i,j) = \begin{cases} X[i] = X[j] & j \leq i+1 \\ X[i] = X[j] \textbf{ and } PAL(i+1, j-1) & j > i+1 \end{cases}$$

GraphDP: Singlesource Shortest path(BF)

Definition

- $dist^{(i)}(s,t)$ = "i-hop distance from s to t"
  shortest length of an $s \to t$ path using **exactly i edges**,
  or ∞ if there's no such path
- $dist^{(\leq i)}(s,t)$ = "at-most-i-hop distance from s to t"
  shortest length of an $s \to t$ path using **at most i edges**

Lemma:
In n-node graph without neg-length cycles,
$dist^{(\leq n-1)}(s,t) = dist(s,t)$

$$-\ dist^{(i)}(s,v) \leftarrow \min_u dist^{(i-1)}(s,u) + \ell(u,v)$$

GraphDP: All pair BF

$$dist^{(\leq i)}(s,t) = \min_x dist^{(\leq i/2)}(s,x) + dist^{(\leq i/2)}(x,t)$$

GraphDP: Floyd

Definition

$dist^{[i]}(s,t)$ is the "middle-restricted distance:"
Shortest length of an $s \to t$ path that
only uses $\{v_1, \ldots, v_i\}$ as intermediate vertices
(but s, t can be anything)

$$dist^{[k]}(s,t) = \min \begin{cases} dist^{[k-1]}(s,t) \\ dist^{[k-1]}(s,v_k) + dist^{[k-1]}(v_k,t) \end{cases}$$

- Bellman-Ford (naïve method):
  - $O(mn^2)$ time

  $|E||V|^2$

- Bellman-Ford (with path-doubling):
  - $O(n^3 \log n)$ time

  $|V|^3 |\log V|$

- Floyd-Warshall (next):
  - $O(n^3)$ time

  $|V|^3$

Definition 34 (Tree #1) An undirected graph G is a *tree* if it is connected and acyclic (i.e., has no cycle).

A graph is *connected* if for any two vertices there is a path between them. A *cycle* is a nonempty sequence of adjacent edges that starts and ends at the same vertex.

Definition 35 (Tree #2) An undirected graph G is a tree if it is *minimally connected*, i.e., if it is connected, and removing any single edge causes it to become disconnected.

Definition 36 (Tree #3) An undirected graph G is a tree if it is *maximally acyclic*, i.e., if it has no cycle, and adding any single edge causes it to have a cycle.

A **decision problem** can be thought of as

$$f : \Sigma^* \to \{No, Yes\}$$

or equivalently as a **language**

$$L \subseteq \Sigma^*$$

$L = \{x \in \Sigma^* : f(x) = Yes\}$ $\quad f(x) = \begin{cases} \text{Yes if } x \in L \\ \text{No if } x \notin L \end{cases}$

Let M be a DFA, using alphabet Σ.

M accepts some strings in Σ^* and rejects the rest.

Definition: $L(M) = \{x \in \Sigma^* : M \text{ accepts } x\}$

注意，
problem size
可以 infly large

Called the "language decided by M".

If L is a language,
we say M decides L if L(M) = L.

A deterministic finite automaton is a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

① Q is a nonempty finite set of states,
② Σ is an alphabet,
③ $\delta : Q \times \Sigma \to Q$ is the state-transition function,
④ $q_0 \in Q$ is the start state,
⑤ $F \subseteq Q$ is the set of accepting states.

即：graph 的 abstraction. 有几个 states，哪个 start，哪些 acc，使用什么 alphabet，states 间怎么 transit.

- **Regular Expression**
  - A regular expression = finite expression using the string notations
    - Start from finite alphabet.
    - Compose them using **concantenation, alternation "|", or Kleen star "*"**
  - Examples:
    - $L(\epsilon) = \{\epsilon\}$
    - $L(ab*|ba*) = \{a, ab, abb, \ldots\} \cup \{b, ba, baa, \ldots\}$
- **Theorem** (RegExp = DFA):
  - L is defined by regular expression ⟺ L is decided by a DFA
- These languages are called **regular languages.**

* A Turing machine is a 7-tuple:
  $M = \langle Q, \Gamma, \Sigma, \delta, q_{start}, q_{accept}, q_{reject} \rangle$
* Q = set of **states**
* Σ = the **input alphabet** (typically {0,1} but not always)
* $\perp$ = the **blank symbol**
* Γ = the **tape alphabet** where generally $\Gamma = \Sigma \cup \{\perp\}$
* $q_{start} \in Q$, = the **initial state**
* $F = \{q_{accept}, q_{reject}\} \subseteq Q$, = the set of **final states** (one accepting state and one rejecting state)
* $\delta : (Q \backslash F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ = the **transition function**

(现 state, symbol (read)) ⟼ (下一个 state, symbol (write), pointer 左/右 移?)

Definition: A Turing Machine **M** recognizes a language L if it:
1. accepts every string in L, and
2. rejects OR loops for every string not in L.

Definition: A Turing Machine **M** decides a language L if it:
1. accepts every string in L, and
2. rejects every string not in L (and never loops forever)

A *language* L is ***decidable*** if there is a TM that decides L.

Fact: If a language **L** is decidable, then the **complement of L** is also decidable. Why?

Fact: If languages **L1** and **L2** are both decidable, then **L1 ∩ L2** and **L1 ∪ L2** are both decidable. Why?

Theorem: Turing machines can simulate each of these models (after formalization) and all known models of computation.

Lemma 74 *The (infinite) set $\{0,1\}^*$ of binary strings is countable.*

Lemma 76 *The (infinite) set $\mathcal{M}$ of Turing machines is countable.*

Proof 77 The key observation is that any Turing machine M has a *finite* description, and hence can be encoded as a (finite) binary string $\langle M \rangle \in \Sigma^*$ in some unambiguous way. To see this, notice that all the components of the seven-tuple are finite: the alphabets Σ, Γ, the set of states Q and the special states $q_{\text{start}}, q_{\text{acc}}, q_{\text{rej}}$, and the transition function δ. In particular, δ has a finite domain and codomain, so we can encode its list of input/output pairs as a (finite) binary string.

There is an undecidable language $A \subseteq \Sigma^* = \{0,1\}^*$.

	ε	0	1	00	01	10	11	000	⋯
M_0	yes	no	yes	yes	no	no	no	no	
M_1	yes	yes	no	no	no	no	no	yes	
M_2	yes	no	no	no	yes	no	no	yes	
M_3	yes	no	yes	no	yes	yes	no	no	
M_4	yes	no	yes	no	yes	no	no	yes	
M_5	yes	yes	no	no	no	no	yes	yes	
M_6	no	yes	no	no	yes	no	no	no	
M_7	yes	no	no	yes	no	yes	no	yes	

r_0:	0.	1	2	1	5	6	6	
r_1:	0.	2	3	3	9	9	7	
r_2:	0.	4	5	6	7	1	1	⋯
r_3:	0.	3	2	8	9	4	5	
r_4:	0.	3	4	1	7	7	5	
r_5:	0.	4	2	4	3	2	3	
⋮								⋱
r^*:	0.	2	4	7	1	9	4	?

- Specifically, an interpreter **U** takes two inputs: (1) source code **⟨M⟩**, and (2) string **x**.
- **U** simulates the execution of **M** on input **x**:
  - **M** accepts **x** ⇒ **U** accepts (**⟨M⟩**, **x**)
  - **M** rejects **x** ⇒ **U** rejects (**⟨M⟩**, **x**)
  - **M** loops on **x** ⇒ **U** loops on (**⟨M⟩**, **x**)
- This is called the ***Universal Turing Machine*** (and it does exist)

$L_{\text{BARBER}} = \{\langle M \rangle : M \text{ is a TM and } M(\langle M \rangle) \text{ does not accept}\}$.

$L_{\text{ACC}} = \{(\langle M \rangle, x) : M \text{ is a Turing machine and } M(x) \text{ accepts}\}$.

L

$$L_{\text{HALT}} = \{(\langle M \rangle, x) : M \text{ halts on input } x\}$$

$M_{\text{BARBER}}(\langle M \rangle)$:

Run M_{ACC} on $(\langle M \rangle, \langle M \rangle)$.
If M_{ACC} accepts, then **reject.**
If M_{ACC} rejects, then **accept.**

$L_{\text{BARBER}} \leq_T L_{\text{ACC}}$

$M_{\text{ACC}}(\langle M \rangle, x)$:

Run M_{HALT} on $(\langle M \rangle, x)$
If M_{HALT} rejects, **reject**
If M_{HALT} accepts, run M on x
If M accepts, **accept**
If M rejects, **reject**

$L_{ACC} \leq_T L_{HALT}$

Run **M**(**x**) and answer as **M**

$M_{\text{HALT}}(\langle M \rangle, x)$:

Let M_x be a TM that ignores its input and runs **M**(**x**)
Run $M_{\varepsilon\text{-HALT}}(\langle M_x \rangle)$ and answer as $M_{\varepsilon\text{-HALT}}$