Lecture 1

Algorithms via Convex Optimization

CPSC 368/516, Spring 2023

Nisheeth Vishnoi

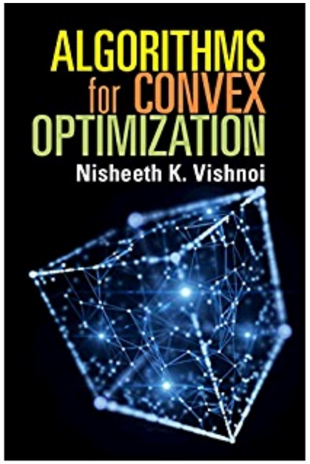
Yale

Administrative Stuff

- Tuesday: 9.25 – 11.15

- Professor: Nisheeth VISHNOI
  - 10 Hillhouse, Room 227
  - Appointment by email

- TA: Anay Mehrotra
  - a.mehrotra@yale.edu
  - Office hours: Thursday 4-5 PM? (on Zoom)



- Course will be largely based on the book:

https://convex-optimization.github.io/

- CANVAS – everyone must register!

Content

- **Turing machines**
- **Part I – Convexity**
  - Basics of calculus, linear algebra, probability, …
  - Convexity
  - Convex programming and efficiency
- **Part II – 1st-order methods for convex optimization (with applications)**
  - Gradient descent (flows/cuts)
  - Mirror descent and multiplicative weights method (matching)
- **Part III – Second-order/advanced methods (with applications)**
  - Newton's method
  - Interior point methods (linear programming, flows)
  - Ellipsoid methods (submodular functions, counting)

Content

- **Mathematical:** Significant experience in mathematical problem solving, writing proofs. Must solve homework problems, write them up (ideally in latex) and submit

- **Prerequisites:** Calculus, linear algebra, and probability, or permission of the instructor

- **What this course is not?**
  - *A first course in proofs/discrete mathematics*
  - *An introduction to machine learning*

- **End Goal:** Prepare you for **mathematical** research in theoretical computer science, optimization, and machine learning

Grading - Undergraduates

- **Problem sets –** 40% (~ 8 problem sets/4 graded)
- **Exam 1 –** 30% (Week of March 6)
- **Exam 2 –** 30% (Week of April 24)

Grading - Graduates

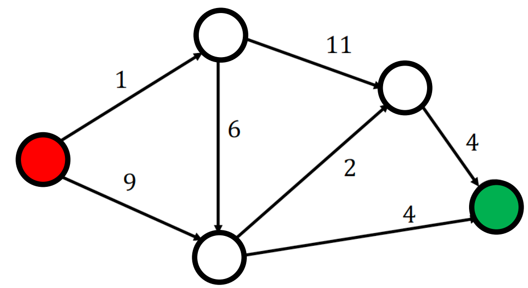
- **Problem sets** – 30% (~ 8 problem sets/4 graded)
- **Exam 1** – 30% (Week of March 6)
- **Exam 2** – 30% (Week of April 24)
- **Additional work** – 10%

Discrete problems in TCS/Optimization

- **Shortest path**

  Input: Graph $\boldsymbol{G = (V, E)}$, source $\boldsymbol{s}$, and sink $\boldsymbol{t}$

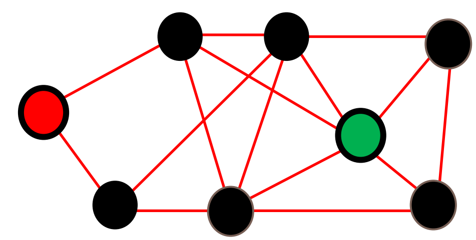
  Output: Shortest "path" from $\boldsymbol{s}$ to $\boldsymbol{t}$



- **s-t-Max Flow**

  Input: Graph $\boldsymbol{G = (V, E)}$, source $\boldsymbol{s}$, and sink $\boldsymbol{t}$

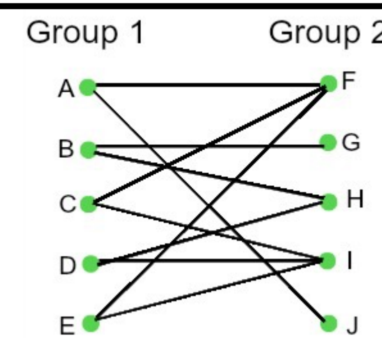
  Output: Maximum "flow" from $\boldsymbol{s}$ to $\boldsymbol{t}$ such that at most 1 unit flow per edge



- **Bipartite Matching**

  Input: Graph $\boldsymbol{G = (L, R, E)}$

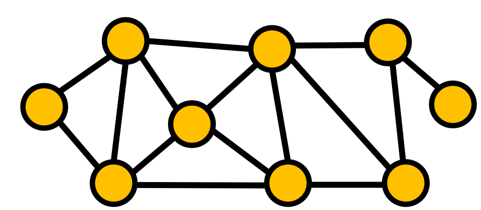
  Output: Decide if G has a perfect matching



- **Count spanning trees**

  Input: Graph $\boldsymbol{G = (V, E)}$

  Output: Count the number of spanning trees in $\boldsymbol{G}$



Old and new approaches

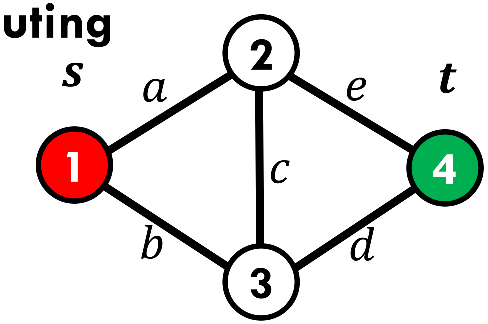
- **Old Idea:**
  - Formulate an optimization problem over discrete variables
  - Use combinatorial/discrete optimization methods

- **New approach:**
  - Formulate a (convex) formulation over continuous domains
  - Use continuous methods (convex optimization)
  - Prove correctness, establish precise running time guarantees

- **Why?**
  - Big data – old algorithms may be slow
  - Combination of this idea with tools such as linear solvers have led to fastest known algorithms for *nearly all* discrete optimization problems
  - **Added benefits:**
    - Learn methods important in many areas (e.g., ML)

The s-t-maximum flow problem

s-t-maximum flow problem captures many **discrete optimization problems**, e.g., generalizes **bipartite matching, scheduling, routing**



Input: 1) Undirected graph $G = (V, E)$, $n := |V|$, $m := |E|$

2) Source and sink $s, t \in V$, $s \neq t$

Vertex-edge incidence matrix $B \in \mathbb{R}^{n \times m}$

$\forall i \in E$, direct $i := (u, v)$, B has a column $b_i := e_u - e_v$

$$B = \begin{bmatrix} -1 & -1 & 0 & 0 & 0 \\ +1 & 0 & -1 & 0 & -1 \\ 0 & +1 & +1 & -1 & 0 \\ 0 & 0 & 0 & +1 & +1 \end{bmatrix}$$

B

Output: **s-t-flow** $x: E \to \mathbb{R}$ satisfies

1) **flow conservation:** for all $j \in V \setminus \{s, t\}$, $\langle e_j, Bx \rangle = 0$

2) **feasibility:** for all $i \in E$, $|x_i| \leq 1$ (**capacity 1**)

Problem: Find a feasible s-t-flow x that maximizes the flow out of s: $|\langle e_s, Bx \rangle|$

Fact: There exists an integral s-t-maximum flow $x_i \in \{-1, 0, 1\}$

B is **totally unimodular** $\Rightarrow$ every sq. submatrix A of B satisfies $\det(A) \in \{-1,0,1\}$

Many combinatorial algorithms: Ford-Fulkerson, Edmonds-Karp, Dinic, …

For the s-t-maximum flow problem with **capacity $U \in \{1, 2, \ldots\}$**:

[Goldberg and Rao, 1998]: An $\tilde{O}(m \min(n^{2/3}, m^{1/2}) \log U)$ time **exact** algorithm for s-t-maximum flow. E.g., when $m = O(n)$ and $U = O(1)$, running time is $O(m^{1.5})$

Convex programming (continuous) approach for maxflow

s-t-maximum flow **reduces** to: Given $F \in \mathbb{R}$ find an s-t flow $\boldsymbol{x}$ of value at least F (F can be found in $O(\log m)$ steps using binary search)

Idea 1: s-$t - F$ flow is the same as finding a **point** in

$$\underbrace{\{x \in \mathbb{R}^m : Bx = F(e_s - e_t)\}}_{(K_1)\ x \text{ is } s\text{-}t\text{-flow } F} \quad \cap \quad \underbrace{\{x \in \mathbb{R}^m : |x_i| \leq 1, \forall i \in [m]\}}_{(K_2)\ x \text{ satisfies "capacities"}}$$

K_1 and K_2 are **convex sets**—they are defined by **linear equalities/inequalities**

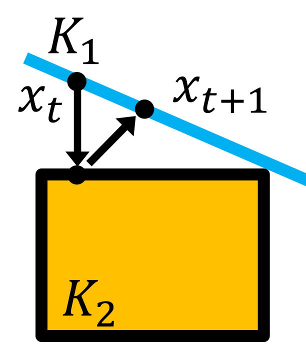
Idea 2: Formulate as convex program. E.g.,

1) Find $x \in K_1$ that minimizes "distance" to K_2
2) Find $x \in K_2$ that minimizes "distance" to K_1

Both are **convex programs** (i) K_1, K_2 are **convex,** (ii) **distance to convex sets** is **convex**

Idea 3 [Lee-Rao-Srivastava, 2013]: Consider **nonlinear convex program**

$$\min_{x \in \mathbb{R}^m} \ \operatorname{dist}(x, K_2)$$
$$\text{s.t.,} \quad x \in K_1,$$



where $\operatorname{dist}(x, K_2)$ is the (squared) **Euclidean distance** between x and K_2

How do we solve the above convex program?

First-order methods for minimizing convex fns

Roughly, family of iterative methods: each step moves in direction of **negative gradient**

Theorem: Given $\varepsilon > 0$, **convex** function $f: \mathbb{R}^m \to \mathbb{R}$, and **access to gradients** of f the following **gradient descent methods** make $\boldsymbol{O(T)}$ **calls** to the gradients of f and output a point $x \in \mathbb{R}^m$ such that

$$f(x) \le f(x^\star) + \boldsymbol{\varepsilon} \qquad (x^\star \text{ – optimal point})$$

Where

- **Gradient descent** assumes that f is **L-Lipschitz continuous** and has $T = O(L\varepsilon^{-1})$
- **Mirror-descent** assumes that **norm of gradient** of f is $\le \boldsymbol{G}$ and has $T = O(G^2\varepsilon^{-2})$
- **Accelerated GD** assumes that f is **L-Lipschitz continuous** and has $T = O\left(\sqrt{L\varepsilon^{-1}}\right)$

[Lee-Rao-Srivastava, 2013] use **accelerated GD** to give:

An $\tilde{O}(mn^{1/3}\varepsilon^{-1/3})$ time algorithm that for any $\varepsilon > 0$, $F \in \mathbb{R}$ outputs a s-t-flow of value $\ge (1-\varepsilon)F$. E.g., when $m = O(n)$, runtime $O(m^{4/3})$ *(beats Goldberg-Rao)*

It can be converted to an **exact algorithm**, but requires $\boldsymbol{\varepsilon \approx O(1/F)}$
For general capacity graphs, **F can be large** – so the running time is **slow**..

Problem: There is convex, L-Lipschitz cont. f for which any GD-method has $T = \Omega(\sqrt{L\varepsilon^{-1}})$

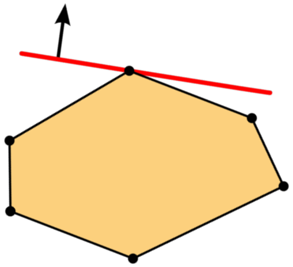
Can we develop algorithms whose runtime scales as $O\left(\log \varepsilon^{-1}\right)$?

Linear programming approach to maxflow

s-t-maximum flow is also **special case** of linear programming:

(i) objective is to **maximize F**

(ii) subject to **linear equality/inequality constraints**



Linear program: Given matrix $A \in \mathbb{R}^{n \times m}$, constraint vector $b \in \mathbb{R}^n$, a cost vector $c \in \mathbb{R}^m$, solve:

$$\min_{x \in \mathbb{R}^m} \langle c, x \rangle \quad \text{s.t.} \quad Ax = b \text{ and } x \geq 0$$

Combinatorial algorithms for s-t-maximum flow **rely** on

- **max-flow min-cut theorem**,
- **integrality** of s-t-maximum flow

Linear prog. duality generalizes the max-flow min-cut theorem; e.g., **[Farkas, 1902]**

Dual of the above program:

$$\max_{y \in \mathbb{R}^n} \langle b, y \rangle, \text{ s.t. } \quad A^\top y \geq c$$

Theorem: For any matrix $A \in \mathbb{R}^{n \times m}$, constraint vector $b \in \mathbb{R}^n$, a cost vector $c \in \mathbb{R}^m$, if both primal and dual programs are **feasible**, then their **optimal values are equal**

But general linear programs—among other properties—do **not** guarantee **integrality**!

How to solve linear prog. in time polynomial in the bit-complexity of A, b, c?

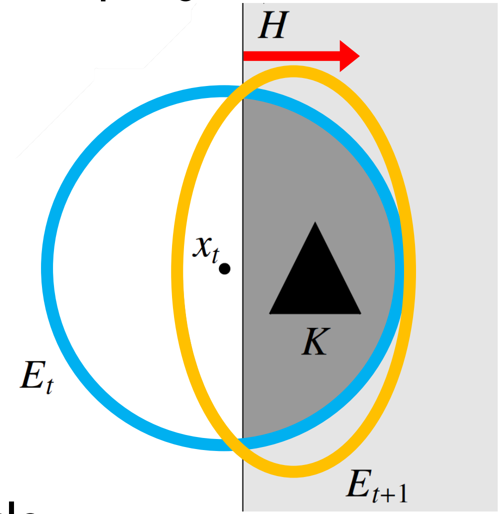
Ellipsoid method: LP is in P

[Khachiyan, 1979] A "geometric" algorithm to check **feasibility** of linear programs

- Along with **binary search**, gives an algorithm to **solve** a linear program

Requires: Separation oracle for $K \coloneqq \{x: Ax = b, x \geq 0\}$

- Input: A point $x \in \mathbb{R}^n$
- Output: YES if x is in K, otherwise
  - A **certificate**–hyperplane H–separating x and K



Input: An Ellipsoid E containing K

At each iteration, guess the **center of E** as a point in K

Then, **update E** based on the **response** of the separation oracle

Key points: At each iteration

- the **volume** of E **reduces** sufficiently
- solves one **linear system**

Theorem: A $\mathbf{poly}(L)$ iteration algorithm for solving linear programs, where L is the bit-complexity of (A, b, c). In **each iteration**, the algorithm makes **one call** to the separation and takes additional **$\mathbf{poly}(L)$ time**

But for s-t-maximum flow it is **slower** than [Goldberg and Rao, 1998]

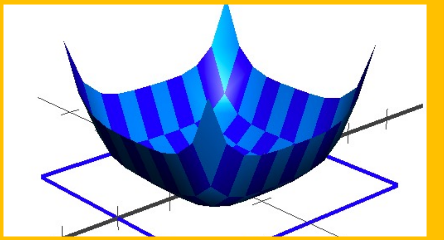
Interior point methods: Faster LP algorithms

[Karmarkar, 1984] A **faster** algorithm for linear programming than Ellipsoid method

Main idea: "Convert" LP to an **unconstrained convex prog.** using **barrier functions**

Barrier function (Informal): A **convex** fn that is finite in interior of set and increases to infinity as one approaches the boundary

Example: For $Ax \leq b$, $F(x) := -\sum_i \log(b_i - \langle A_i, x\rangle)$



[Renegar, 1988] Combined the barrier-approach with **Newton's method**—a **second order optimization method**—to improve the running time

Input: A barrier function $F(x)$, and second-order oracle of $F(x)$

Main step: Minimize $\eta\langle c, x\rangle + F(x)$, for fixed $\eta > 0$ (also change η over time)

Theorem: A $\underline{\boldsymbol{O(\sqrt{m} \cdot L)}}$ step algorithm for solving linear programs, where $\underline{L}$ is the bit-complexity of (A, b, c). In each step, the algorithm solves an $m \times m$ linear system

For s-t-maximum flow:

Theorem: **[Lee and Sidford, 2014]** An $\tilde{O}(mn^{1/2} \cdot \log^2 U)$ time algorithm for s-t-maximum flow problem. E.g., for any $m > n$ it is **faster** than $\tilde{O}(m^{1.5})$

Recently **[Chen, Kyng, Liu, Peng, Probst, Sachdeva 2022]** running time to $\tilde{O}(m)$!

Ellipsoid method for convex programs

Problem: Given **convex set** $K \subseteq \mathbb{R}^m$ and **convex function** $f: \mathbb{R}^m \to \mathbb{R}$: $\min_{x \in K} f(x)$

Ellipsoid method can used to solve the most general convex programs

Theorem: $\mathrm{poly}((T_K + T_f) \cdot m \cdot \log \varepsilon^{-1})$ time algorithm that outputs $x \in K$, s.t.

$$f(x) \leq f(x^\star) + \varepsilon,$$

where T_K and T_f are the running time of separation oracle for K and first-order of f

Implies efficient algorithms for comb. problems; e.g., via submodular minimization

A **submodular (set-)function** $f: 2^{[m]} \to \mathbb{R}$ satisfies: For sets $S \subseteq T \subseteq [m]$ and $i \in [m]$,

$$f(S \cup \{i\}) - f(S) \geq f(T \cup \{i\}) - f(T)$$

Problem: Given submodular function $f: 2^{[m]} \to \mathbb{R}$ find its minimizer: $\mathrm{argmin}_{S \subseteq [m]} f(S)$

Applications:

- Originated in discrete optimization, e.g., minimum s-t-cut in graphs
- Machine learning: Arises in objectives for data summarization, influence maximization

Theorem: There is an algorithm that, given oracle access to a submodular function f, and $\varepsilon > 0$, outputs $S \subseteq [m]$ such that

$$f(S) \leq f(S^\star) + \varepsilon,$$

where $S^\star$ is minimizer of f. The algorithm makes $\mathrm{poly}\left(m, \log(\varepsilon^{-1})\right)$ queries to f

Applications: Max-entropy distributions

Convex programming for (approximately) counting discrete objects

Counting problem: Given $G = (V, E)$, compute the number of spanning trees of G.

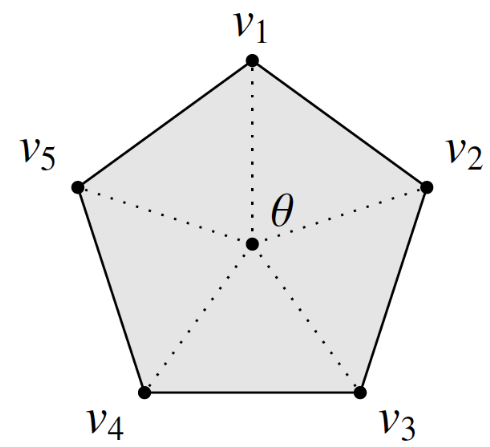
Let $\mathcal{T}_G$ be the set of all spanning trees of G

Let P_G be spanning tree polytope, i.e., the convex hull of all spanning trees in $\mathcal{T}_G$

Optimization problem: Given $G = (V, E)$ and $\theta \in P_G$, write θ as a convex combination of the vertices of P_G so that the probability distribution corresponding to the convex combination maximizes the **Shannon entropy**:

$$\min_p - \sum_{T \in \mathcal{T}_G} p_T \log p_T \quad (1)$$

$$\text{s.t. } \sum_{T \in \mathcal{T}_G} p_T v_T = \theta, \sum_{T \in \mathcal{T}_G} p_T = 1, p_T \geq 0 \ \forall\, T \in \mathcal{T}_G$$



Connection: If θ is the average of the vertices in P_G, then the value of Prog. (1) is $\log|\mathcal{T}_G|$

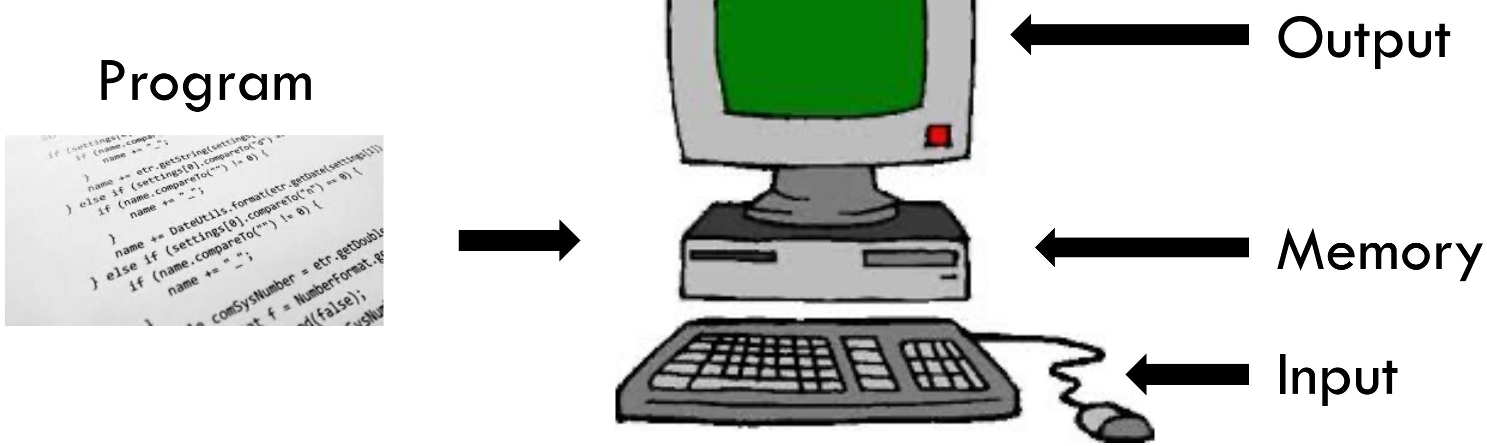
Prog. (1) is convex – however, it has **exponentially** many variables

- e.g., for the complete graph Prog. (1) has $|\mathcal{T}_G| = n^{n-2}$ variables

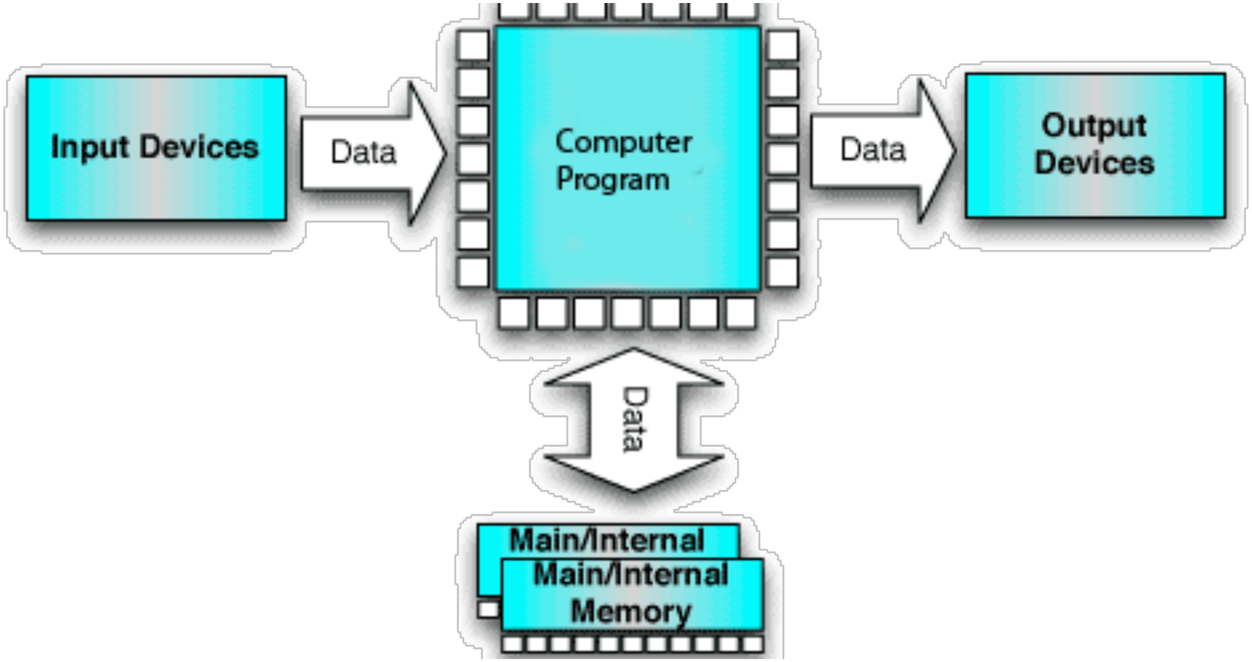
The dual of Prog. (1) has n variables and can be efficiently solved using the Ellipsoid method **[Singh and Vishnoi, 2014] [Straszak and Vishnoi, 2019]**

Turing Machines

What is a computer?

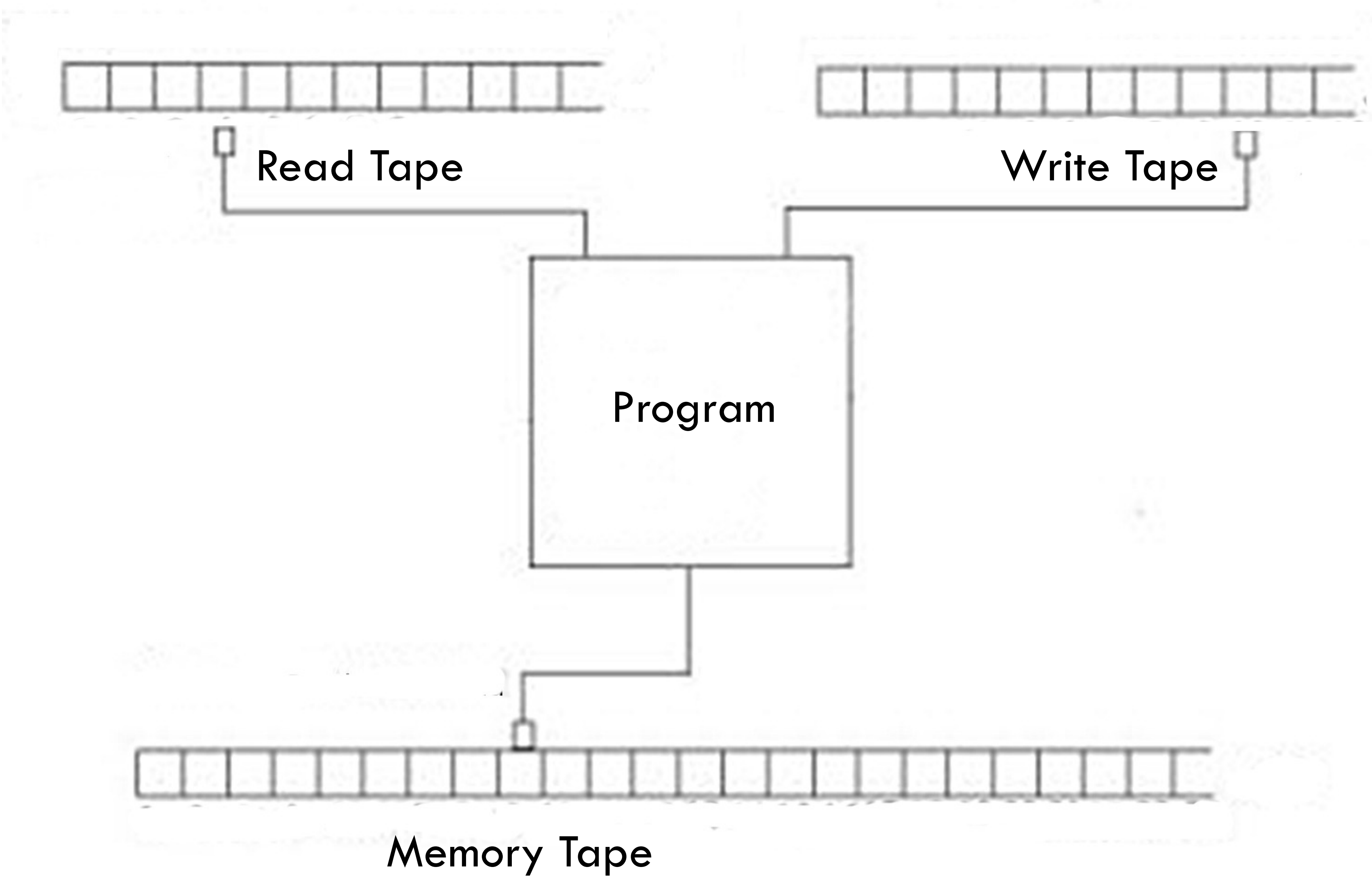


Abstractly ..

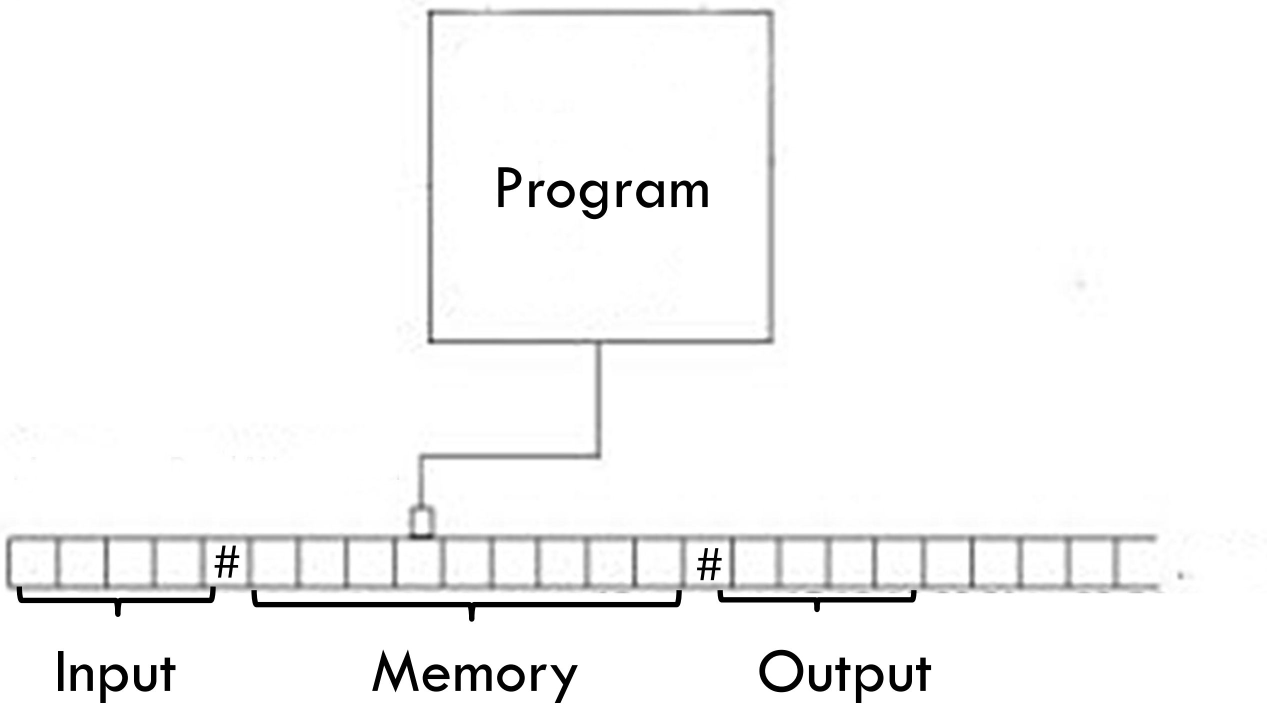


Input Devices
Data
Computer
Program
Data
Output
Devices
Data
Main/Internal
Main/Internal
Memory

More abstractly ..



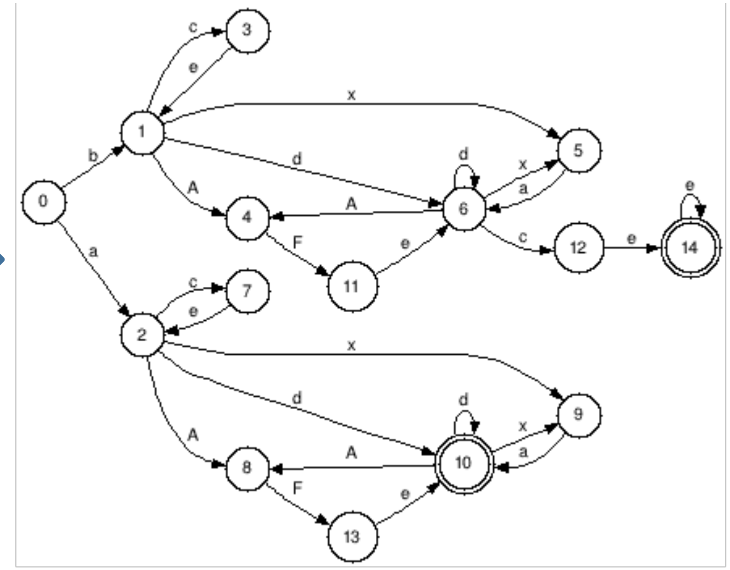
Single tape seems enough ...

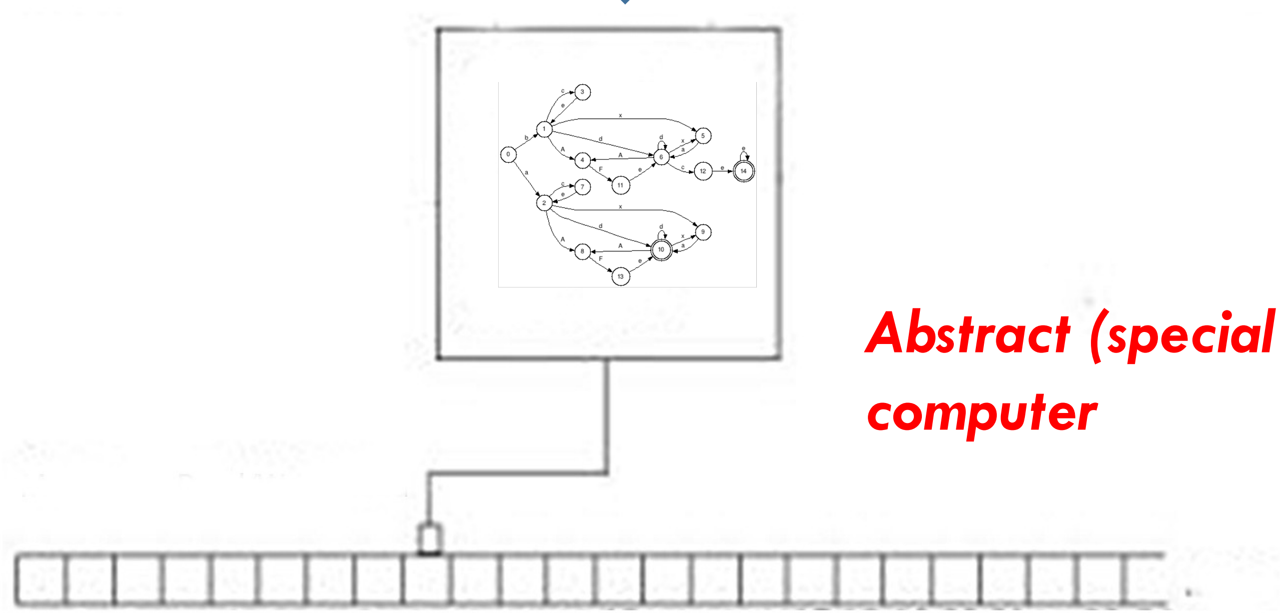


Program vs Finite Automata

```python
def fib_tail(n):
    def fib_tail_rec(a, b, n):
        if n < 1:
            return a
        return fib_tail_rec(b, a + b, n - 1)
    return fib_tail_rec(0, 1, n)


def fib_exponential(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_slow(n - 1) + fib_slow(n - 2)
```

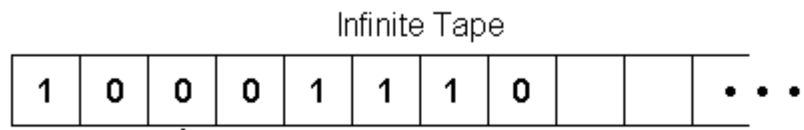




Abstract (special purpose) computer

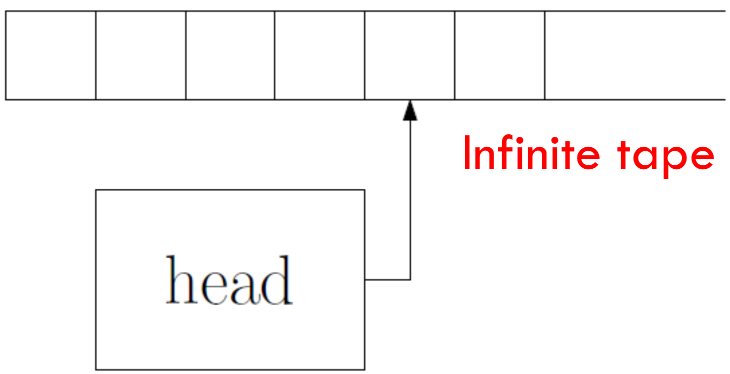
Finite size program, larger and larger instances

Infinite Tape!



And thus the Turing Machine was born.

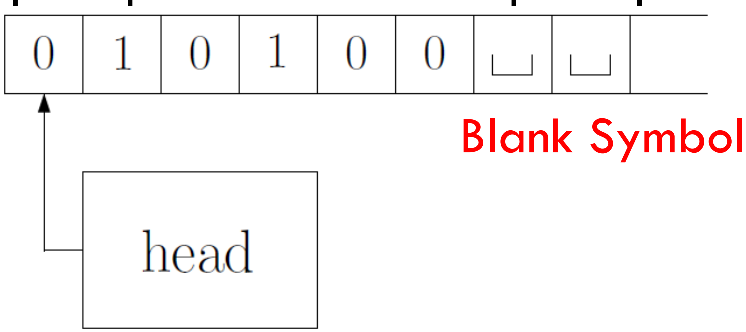
Turing Machine



Head can Read/Write/Move Left/Right/Stay

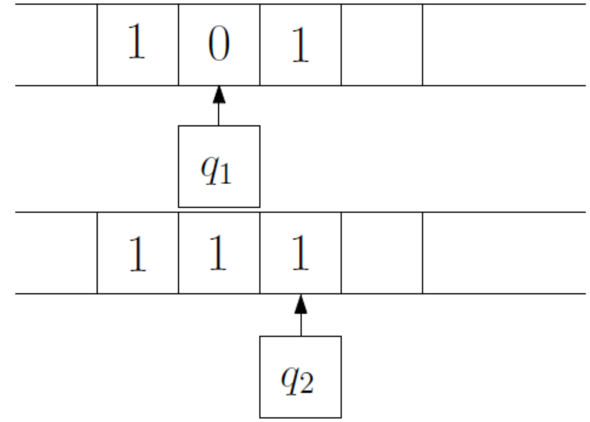
Once it reaches left-most cell, it can't go more left

Tape Alphabet contains Input Alphabet



Example of starting configuration

Head has (finitely many) states



Exactly **one** Accept state

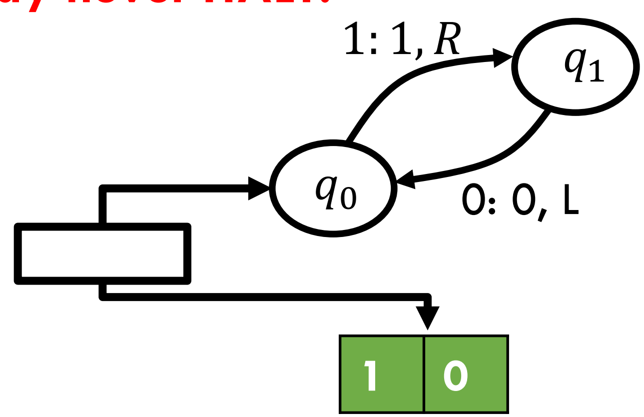
and exactly **one** **Reject** state

Remaining states:

"computation in progress"

May never reach an accept/reject state

May never HALT!



Formal Definition of a TM

A **Turing Machine** is a 7-tuple, $(Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ q_{accept}, q_{reject})$, where Q, Σ, Γ are **finite** sets and:

1. Q is the set of states,
2. Σ is the input alphabet ***not containing the blank symbol*** $\sqcup$,
3. Γ is the tape alphabet where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Recognizable/Decidable Languages

M **accepts** $w \in \Sigma^\star$ if $\exists\, C_1, C_2, \dots, C_t$ such that

1. C_1 is the starting configuration of M on w
2. $C_i \to C_{i+1}$ is a valid step of the TM (for $i = 1,2,\dots,t-1$)
3. C_t is an accepting configuration

$$L(M) = \{w \in \Sigma^\star : M \text{ accepts } w\}$$

TM M **recognizes** a language $L \subseteq \Sigma^\star$ iff for all inputs $w \in \Sigma^\star$

1. If $w \in L$ then M accepts w and
2. If $w \notin L$ then M either rejects w or never halts

Such languages are called **(Turing)-Recognizable**

TM M **decides** a language $L \subseteq \Sigma^\star$ iff for all inputs $w \in \Sigma^\star$

1. M ***halts*** on w
2. M ***accepts*** w **iff** $w \in L$

Such languages are called **(Turing)-Decidable**

Church-Turing Thesis

Alan Turing

Alonzo Church

Intuitive notion of algorithms	*equals*	*Turing machine algorithms*

Can Turing Machines recognize/decide all languages? NO

Time Complexity

A Decidable Language L

Number of configurations TM needs to reach an accept/reject state on this input

Deciders for L:

Inputs	M_A	M_B	M_C	M_D	M_E	M_F	M_G	M_H	
ε	2	2	5	2	3	4	2	2	⋯
0	2	5	12	2	3	5	12	5	⋯
1	20	12	14	13	8	19	2	9	⋯
00	32	14	18	9	18	3	5	90	⋯
01	12	21	56	8	12	18	18	30	⋯
10	21	22	26	15	11	12	32	15	⋯
11	11	12	25	100	13	48	98	29	⋯
000	320	201	159	201	190	200	180	65	⋯
001	211	208	190	200	189	301	219	82	⋯
010	328	271	214	441	193	208	109	77	⋯
011	227	261	191	201	188	107	211	207	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

How to compare different deciders?

Two Deciders for L

	M_A	
ε	2	2
0	4	12
1	12	
00	15	110
01	22	
10	17	
11	110	
000	49	300
001	34	
010	38	
011	300	
⋮	⋮	

	M_B	
ε	2	2
0	9	10
1	10	
00	40	45
01	30	
10	45	
11	39	
000	73	85
001	77	
010	85	
011	80	
⋮	⋮	

Worst case in this group

Running Time of a TM

Definition: Let M be a TM that halts on all inputs (decider). The ***running time*** or ***time complexity*** of M is the function $t: \mathbb{N} \to \mathbb{N}$ where

$$t(n) = \max_{w \in \Sigma^*; |w| = n} \text{number of steps } M \text{ takes on } w$$

- M runs in time $t(n)$
- n represents the input length

$t(0) = 2$

$t(1) = 10$

$t(2) = 45$

$t(3) = 85$

$\vdots$

ε	2	2
0	9	10
1	10	
00	21	45
01	30	
10	45	
11	33	
000	73	85
001	77	
010	85	
011	80	
$\vdots$	$\vdots$	

Two Deciders for L

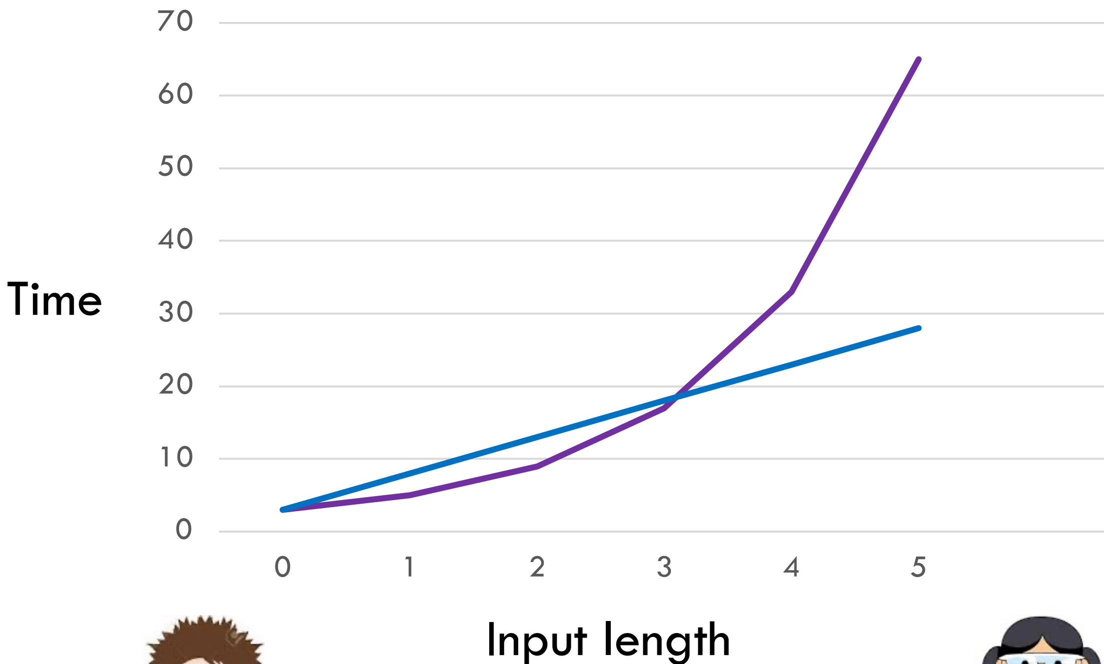
M_A

Length 0	$t_1(0) = 3$
Length 1	$t_1(1) = 5$
Length 2	$t_1(2) = 9$
Length 3	$t_1(3) = 17$
Length 4	$t_1(4) = 33$
Length 5	$t_1(5) = 65$
⋮	⋮

M_B

Length 0	$t_2(0) = 3$
Length 1	$t_2(1) = 8$
Length 2	$t_2(2) = 13$
Length 3	$t_2(3) = 18$
Length 4	$t_2(4) = 23$
Length 5	$t_2(5) = 28$
⋮	⋮

$t_1(n) = 2^{n+1} + 1$ vs $t_2(n) = 5n + 3$



How to compare running time functions?

$f_1(n) = 2^n$, $f_2(n) = 5n^3 + 1$, $f_3(n) = 20n + 6$

Big-O and Small-o Notation

Definition (Big-O): Let $f, g: \mathbb{N} \to \mathbb{R}_{\geq 0}$. We say $f(n) = O(g(n))$ if
$\exists \boldsymbol{C} > \mathbf{0}\ \exists n_0$ s.t. $\forall \boldsymbol{n} \geq n_0 \quad \boldsymbol{f(n) \leq C \cdot g(n)}$

Examples:

✓ $5n^3 + 1 \ =?\ O(2^n)$

✗ $5n^3 + 1 \ =?\ O(20n + 6)$

Definition (Small-o): Let $f, g: \mathbb{N} \to \mathbb{R}_{\geq 0}$. We say $f(n) = o(g(n))$ if
$\forall \boldsymbol{c} > \mathbf{0}\ \exists n_0$ s.t. $\forall \boldsymbol{n} \geq n_0 \quad \boldsymbol{f(n) \leq c \cdot g(n)}$

Examples:

✓ $\sqrt{n} \ =?\ o(n)$

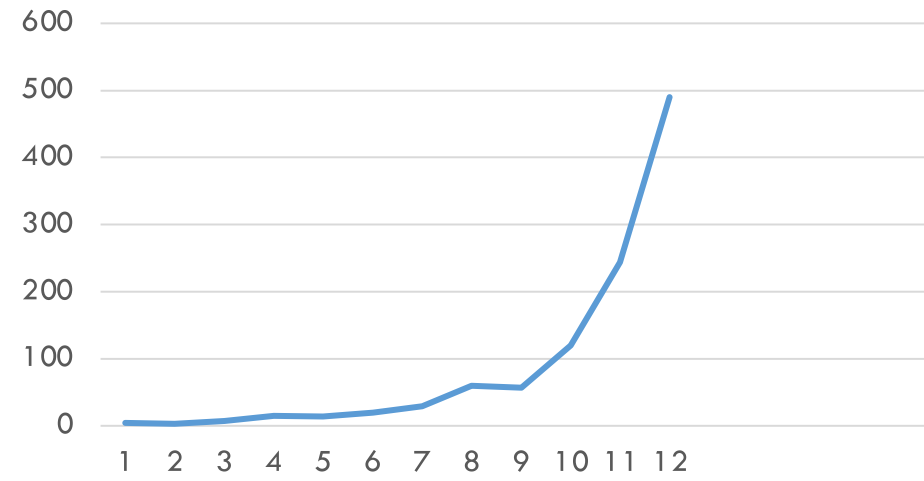
✗ $f(n) \ =?\ o(f(n))$

$$f_1(n) = 2^n, \qquad f_2(n) = 5n^3 + 1, \qquad f_3(n) = 20n + 6$$

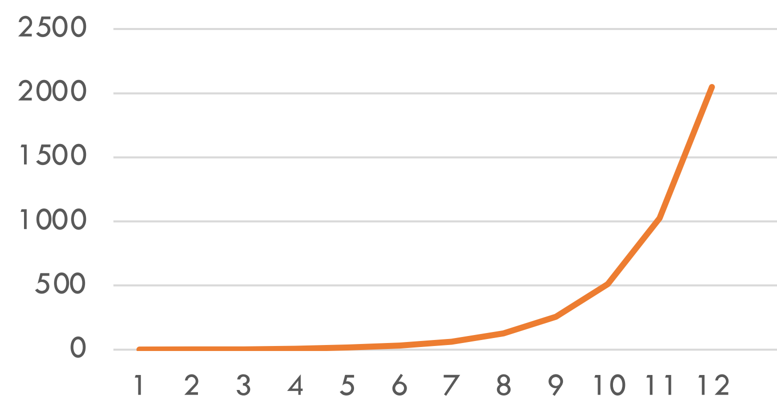
$$f_3(n) = O\big(f_2(n)\big) \quad f_2(n) = O\big(f_1(n)\big)$$

To Summarize ..

M		
2	Length 0	$t(0) = 2$
4, 3	Length 1	$t(1) = 4$
2, 3	Length 2	$t(2) = 3$
⋮, 4, ⋮, 7	Length 3	$t(3) = 7$
⋮, 6, 10, ⋮, 15, ⋮	Length 4	$t(4) = 15$
10, 8, ⋮, 14, ⋮	Length 5	$t(5) = 14$
	⋮	⋮



$$t(n) = O(2^n)$$



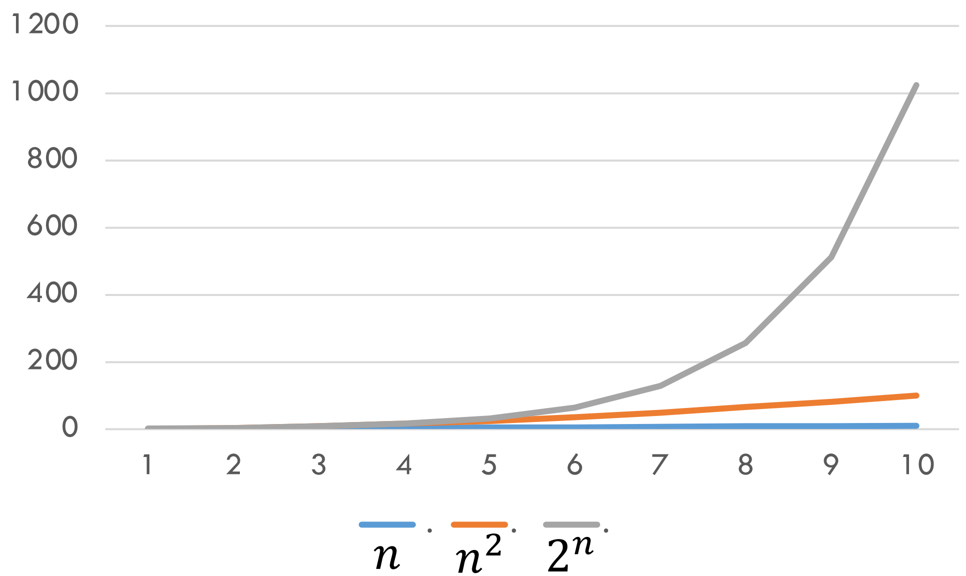
2^n

Time Complexity

Definition: Time complexity class
$\mathrm{TIME}\big(t(n)\big) \coloneqq \{L \subseteq \Sigma^\star | L \text{ is decided by a TM with running time } O(t(n))\}$

- $B \in \mathrm{TIME}(n^2)$

Theorem:
$$\mathrm{TIME}(n) \subseteq \mathrm{TIME}(n^2) \subseteq \cdots \subseteq \mathrm{TIME}\big(2^{\sqrt{n}}\big) \subseteq \mathrm{TIME}(2^n) \subseteq \mathrm{TIME}\big(2^{2^n}\big) \cdots$$



The Complexity Class P and Efficiency

Definition: $\boldsymbol{P}$ is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\boldsymbol{P} = \bigcup_{k=1}^{\infty} \mathrm{TIME}(n^k).$$

For instance: $\boldsymbol{P}$ is the same class of languages for TMs with 2 tapes.

1. $\boldsymbol{P}$ is invariant for all models of computation that are polynomially equivalent to deterministic single-tape TM – robust

2. $\boldsymbol{P}$ roughly corresponds to the class of problems that are realistically solvable – and we focus on such problems in the course