IT-216 - Design and analysis of algorithms

$x \leftarrow$ n digits no.
$y \leftarrow$ n " no.

multiply x and y.

x * y

5 × 6 = 30

25 × 11 = 275

5 6 7 8 9 1 0 2
× 1 2 3 5 6 9

$x = 5\ 6\ 7\ 8$

$y = x\ 1\ 2\ 3\ y$

$2\ 2\ 7\ 1\ 2$

$1\ 7\ 0\ 3\ 4\ x$

$1\ 1\ 3\ 5\ 6\ x\ x$

$5\ 6\ 7\ 8\ x\ x\ x$

$7\ 0\ 0\ 6\ 6\ 5\ 2$

3
3
2

Q: How many premitive operations are there?

sigle digit multiplication and addition are allowed.

Each row takes $\rightarrow 2n$ } total n rows takes $- 2n^2$

rows $- n$

Adding two rows takes $\rightarrow 2n$ operations.

Adding n rows takes $\rightarrow (n-1) \cdot 2n \cdot \leq 2n^2$

Total operations $\rightarrow 2n^2 + 2n^2 \leq 4n^2$

$\rightarrow$ constant $* n^2$

Algorithm designer's mantra

can we do better?

$x = a\,\underline{56}\,\underline{78}\,b$

$y = \underline{12}\,\underline{34}$

$c \quad d$

step 1: compute ac ← $T(n/2)$

step 2: compute $b \cdot d$ ← $T(n/2)$ $2 \cdot n/2$

step 3: compute $(a+b)\cdot(c+d)$ ← $2 \cdot (2\frac{n}{2}) + T(n/2)$

step 4: compute step 3 − step 1 − step 2 ← constant · n

step 5: 10^4 step 1 $+ 10^2$ step 4 + step 2 ← constant · n.

Ans 7006652 (H.W)

$T(n)$ ← total number of operations to multiply two n digits number.

overall time:

$T(n) = 3T(n/2) + O(n)$

constant * n

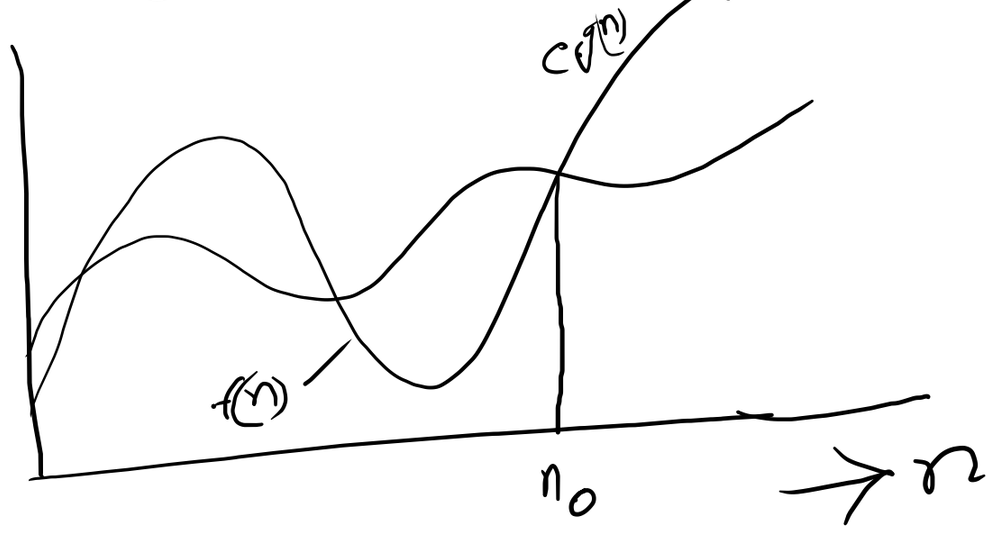
less than $2n^2$

Asymptotic notations

Big 'O' notation (upper bound)

Question When is $f(n) = O(g(n))$?

$f(n) = O(g(n))$ if there exists constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0$



$$x = 10^{n/2} a + b$$

$$y = 10^{n/2} c + d$$

$$x \cdot y = (10^{n/2} a + b)(10^{n/2} c + d)$$

$$= 10^{n} ac + 10^{n/2}(ad + bc) + bd$$

$$(a+b)(c+d) - ac - bd$$

Exm

$2n^2 = O(n^3)$

$c = 1$
$n_0 = 1$

$0 \le f(n) \le c\, g(n) \quad \forall\, n \ge n_0$

$2n_0^2 \le 1 \cdot n_0^3$

$2 \le 1$ ✗

$c = 1$
$n_0 = 2$

$2n^2 = O(n^3)$

$2n_0^2 \le 1\, n_0^3$

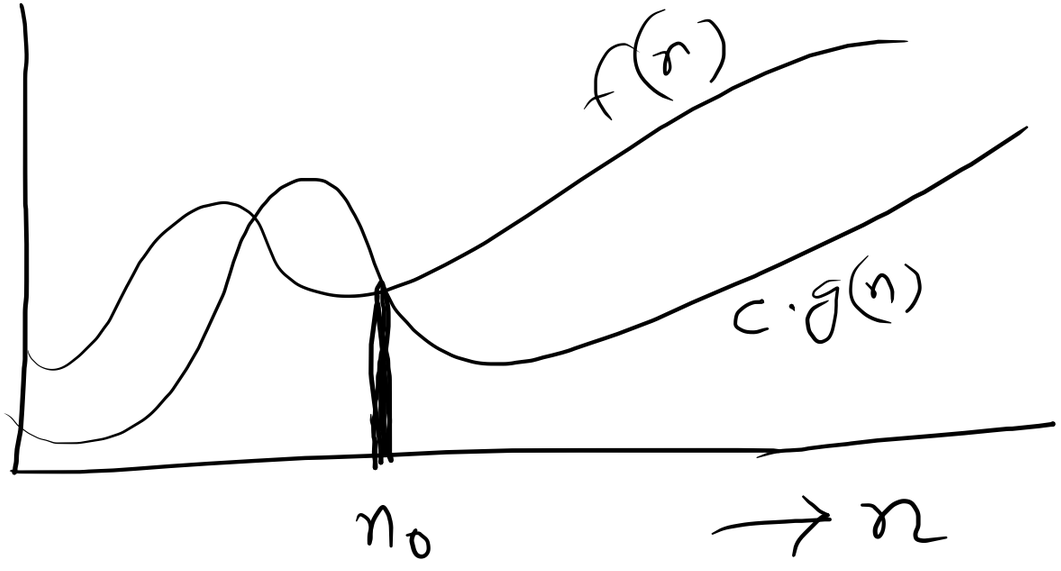
$2 \cdot 2^2 \le 1 \cdot 2^3$

$8 \le 8$ ✓

Big omega (Ω) notation

lower bound.

- $f(n) = \Omega(g(n))$ if there exists constants $c > 0, n_0 > 0$ such that $0 \leq c \cdot g(n) \leq f(n)$ $\forall$ $n \geq n_0$



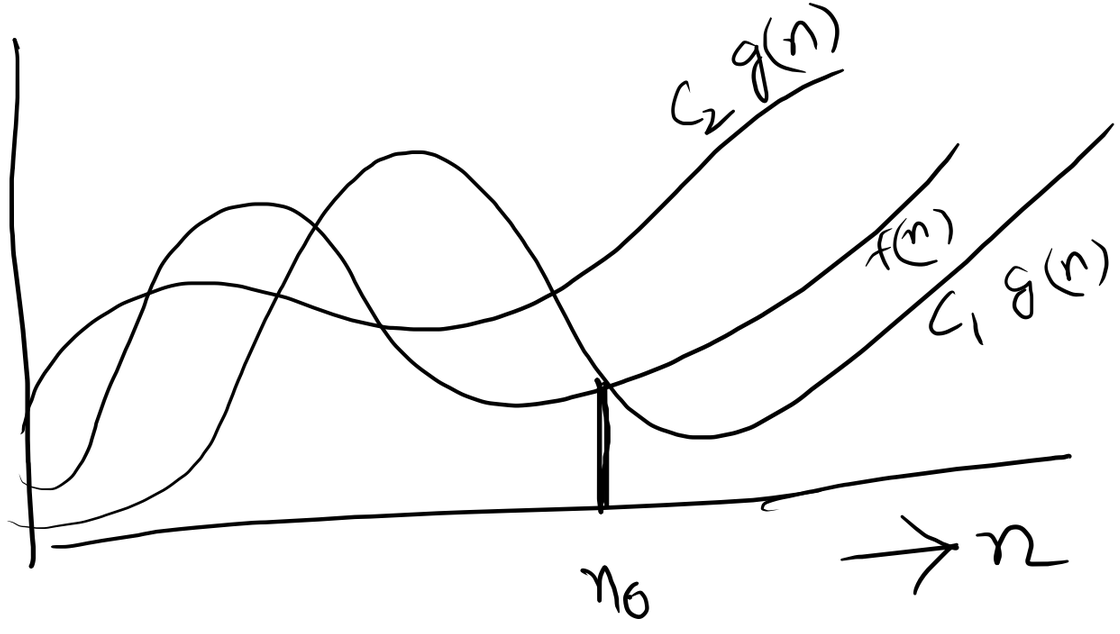
Ex $5n^2 + 3n = \Omega(n^2)$

Theta notation

Tight bound

$f(n) = \theta(g(n))$ if $\exists$ constants $c_1 > 0, c_2 > 0, n_0 > 0$ such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$$



Ex: $3n+2 = \theta(n)$

small 'o' and small ω.

$$f(n) = o(g(n))$$

$$0 \leq f(n) < c\, g(n)$$

$$f(n) = \omega(g(n))$$

$$0 \leq c\, g(n) < f(n)$$

$T(n) = 3T(\frac{n}{2}) + constant*n$ called recurrence equations.

Solving recurrences

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

- Analyse the running time of divide and conquer algorithm.

Methods to solve recurrences

1. Substitution method
2. Recursion tree method
3. The master method

Substitution method

This is the most general method

1. Guess the form of the solution
2. verify it by induction
3. Solve some constants.

Ex^m $T(n) = 4T(\frac{n}{2}) + n$

[Assume $T(1) =$ constant $\theta(1)$]

- Guess $O(n^3)$
- Assume that $T(k) \leq ck^3$ for $k < n$

we need to prove, $T(n) \leq cn^3$ by induction.

$$T(n) = 4T(\tfrac{n}{2}) + n$$
$$\leq 4 \cdot c \cdot (\tfrac{n}{2})^3 + n$$
$$= \tfrac{c}{2} n^3 + n$$
$$= cn^3 - \underbrace{\left[\tfrac{c}{2} n^3 - n\right]}_{\text{residual}}$$

$\underbrace{\quad}_{\text{required}}$ derived

$T(n) \leq cn^3$ whenever $\frac{c}{2} n^3 - n > 0$

this is true when $c \geq 2$, $n \geq 1$

so $T(n) = O(n^3)$

Guess: $O(n^2)$

I.H. $T(K) \le cK^2$ for $K < n$

$$T(n) = 4T(\tfrac{n}{2}) + n$$
$$\le 4 \cdot c \cdot (\tfrac{n}{2})^2 + n$$
$$= cn^2 + n$$
$$= cn^2 - [-n]$$

required — residual

H.W. Try solving examples

Strengthen

I.H. $T(K) \le c_1K^2 - c_2K$ for $K < n$

$$T(n) = 4T(\tfrac{n}{2}) + n$$
$$\le 4 \cdot c_1(\tfrac{n}{2})^2 - 4c_2\tfrac{n}{2} + n$$
$$= c_1n^2 - c_2n - [c_2n - n]$$

desired — required

$T(n) \le c_1n^2 - c_2n$ whenever $c_2n - n \ge 0$

$\Rightarrow c_2 \ge 1$

$T(n) = O(c_1n^2 - c_2n)$

$= O(n^2)$

Recursion tree method

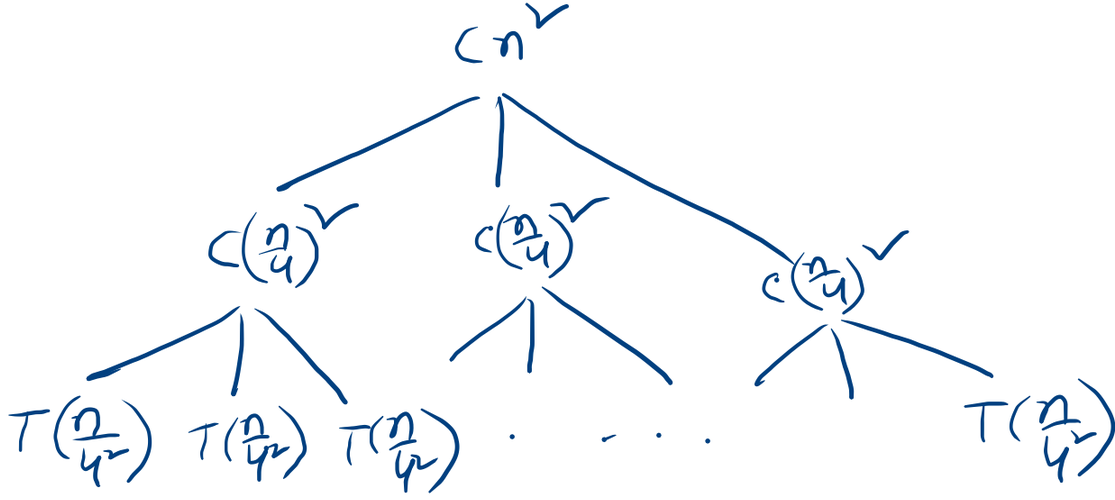
- A recursion tree models the cost (time) of a recursive execution of an algorithm.
- It is an intution of the running time of an algorithm
- It can be use as a guess for substitution method.

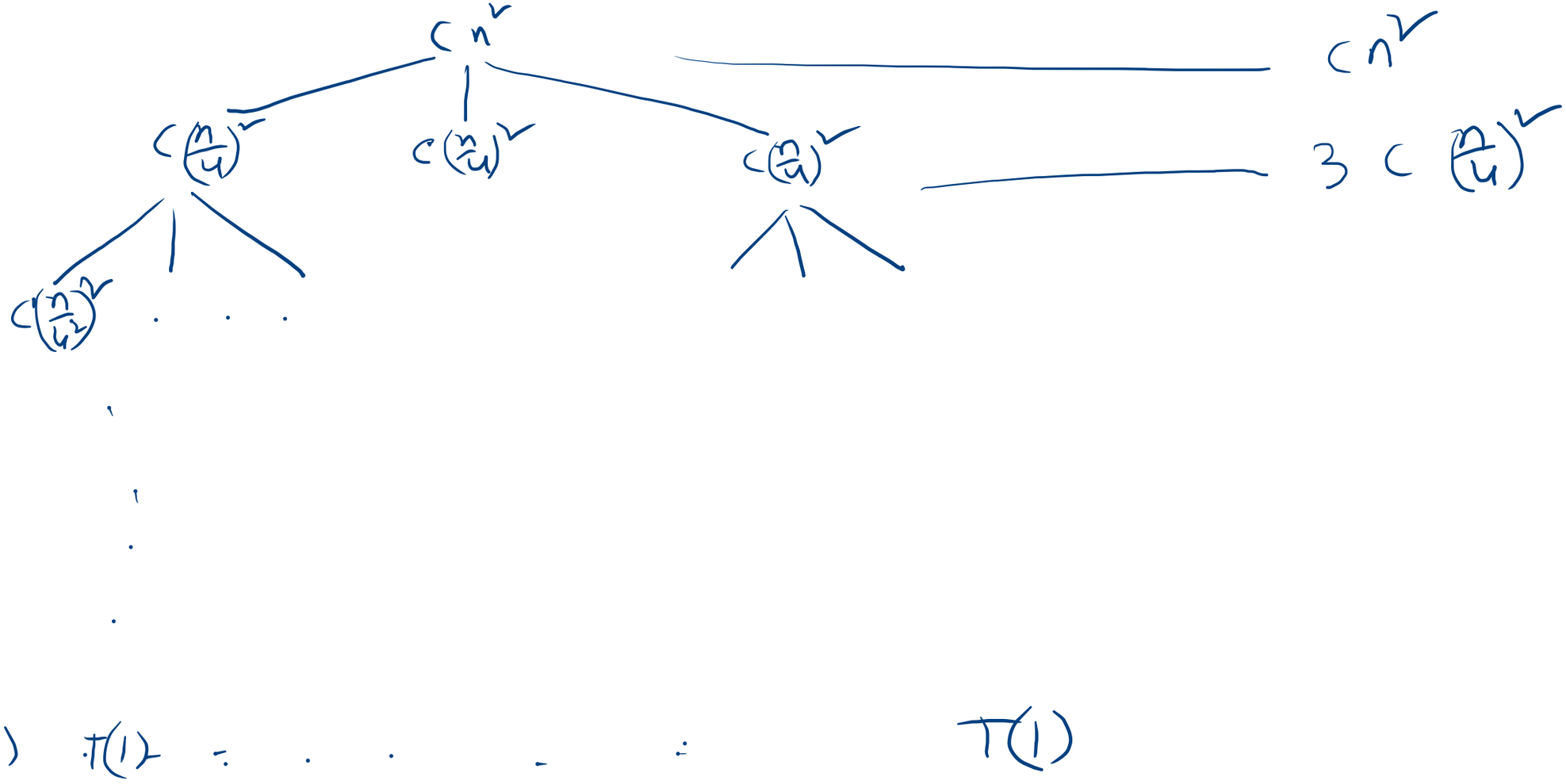
Guess the solution using recursion tree method.

$$T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n^2)$$

$$\equiv \quad T(n) = 3T\left(\frac{n}{4}\right) + cn^2 \quad \text{for some constant } c.$$

$T(n)$ •




cn^2
cn^2
$c(\frac{n}{4})^2$
$c(\frac{n}{4})^2$
$c(\frac{n}{4})^2$
$3c(\frac{n}{4})^2$
$c(\frac{n}{16})^2$
T(1) T(1) T(1)

cost at i-th depth

nodes in i-th depth $\rightarrow 3^i$

cost of each node at depth $i \rightarrow c \cdot \left(\frac{n}{4^i}\right)^2$

cost at depth $i \Rightarrow 3^i \cdot c \cdot \left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i c\, n^2$

Height: Let i be the height

$$\frac{n}{4^i} = 1 \Rightarrow i = \log_4 n$$

cost at last level $\rightarrow 3^{\log_4 n} \cdot T(1)$

Total cost of the tree.

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i c n^2 + 3^{\log_4 n}$$

$$\leq \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i c n^2 + n^{\log_4 3}$$

$$= \frac{1}{1 - \frac{3}{16}} c n^2 + n^{\log_4 3}$$

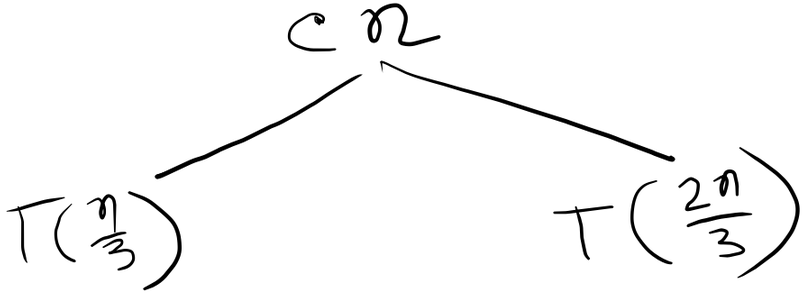
$$= O(n^2)$$

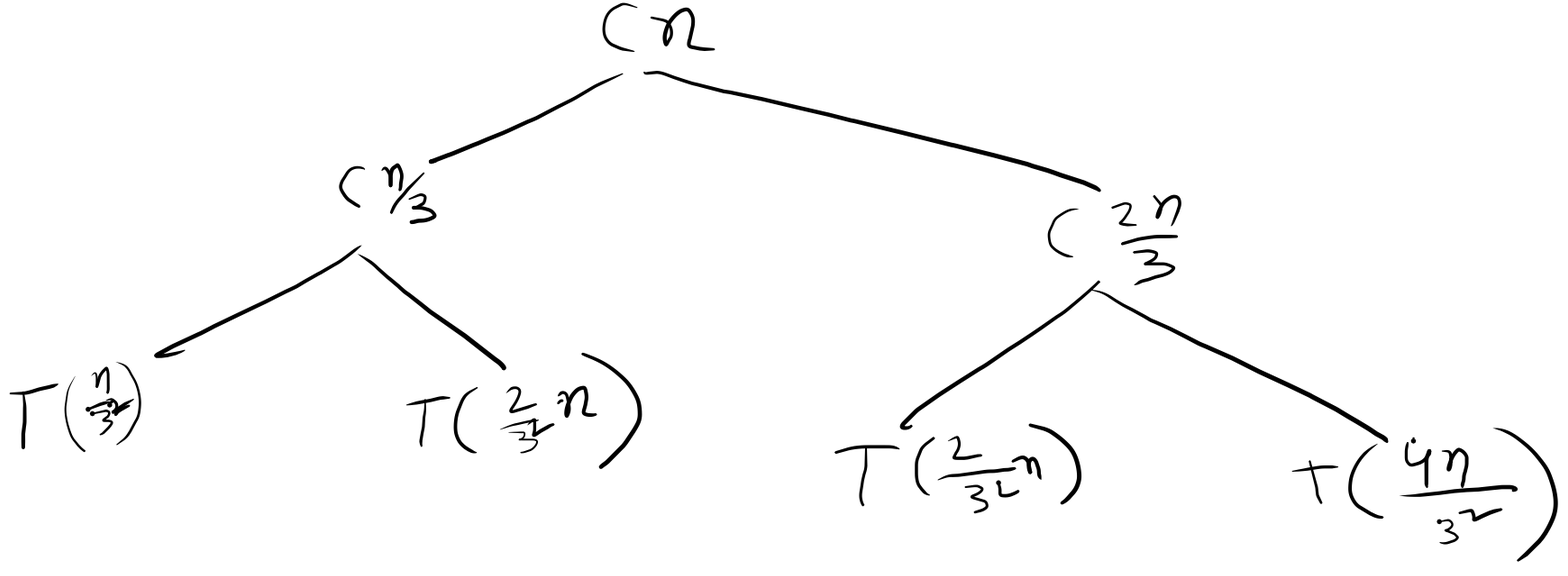
Recursion tree method

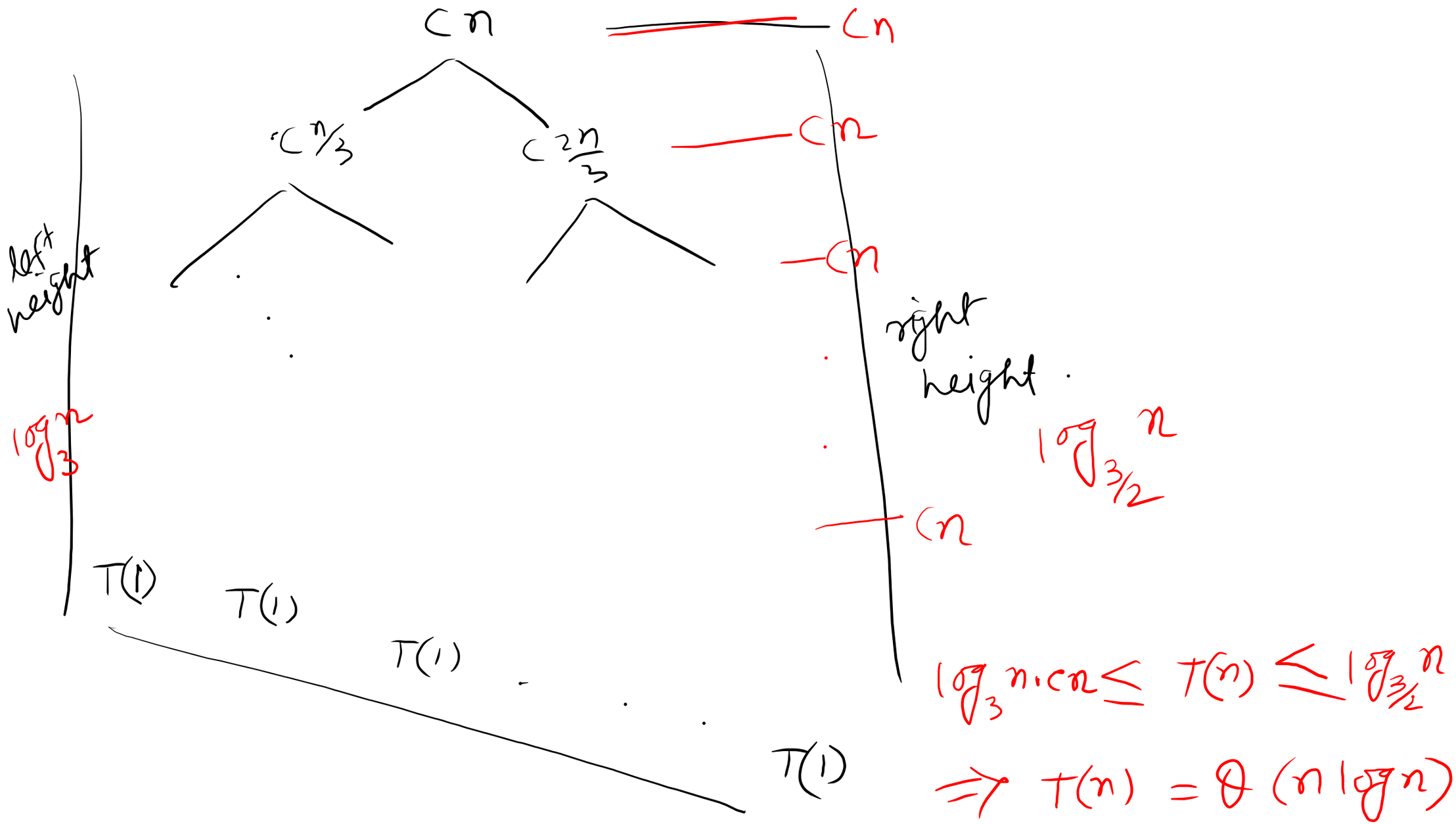
Ex^n $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + cn$

Find $T(n)$ using recursion tree.

$T(n)$







$\log_3 n \cdot cn \leq T(n) \leq \log_{3/2} n \cdot cn$

$\Rightarrow T(n) = \Theta(n \log n)$

The master method

This method applies to the recurrences of the form

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

$a \geq 1$, $b > 1$ and f is asymptotically positive

Three common cases based on comparing $f(n)$ and $n^{\log_b a}$

case 1: $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for $\epsilon > 0$

$f(n)$ grows polynomially slower than $n^{\log_b a}$ by a factor n^{ϵ}

solution: $T(n) = \theta\left(n^{\log_b a}\right)$

case 2 $f(n) = \theta\left(n^{\log_b a} \lg^k n\right)$

Solution: $T(n) = \theta\left(n^{\log_b a} \cdot \lg^{k+1} n\right)$

case 3: $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$

Addition condition

$af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$

regularity condition

Solution: $T(n) = \theta\left(f(n)\right)$

Exᵐ i) $T(n) = 4T(\frac{n}{2}) + n$

$a = 4, \quad b = 2, \quad n^{\log_b a} = n^{\log_2 4} = n^2$

$f(n) = n$

$f(n) = O(n^{2-1}) = O(n^{\log_b a - 1})$

case 1 of master method applies here

solution is $T(n) = \theta(n^2)$

ii) $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

$$T(n) = \theta(n^2 \lg n)$$

iii) $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

$$n^3 = \Omega\left(n^{2+1}\right)$$

$$af\left(\frac{n}{b}\right) \leq cf(n) \quad \text{for some } c < 1$$

$$4\left(\frac{n}{2}\right)^3 \leq c \cdot n^3$$

$$\frac{n^3}{2} \leq cn^3 \qquad c \geq \frac{1}{2}$$

any value of c where, $.5 \leq c < 1$ satisfies regularity condition

solution $T(n) = \theta(n^3)$

iv) $T(n) = 4T(\frac{n}{2}) + \frac{n^2}{\lg n}$

$f(n) = \frac{n^2}{\lg n}$ $\qquad$ $n^{\log_b a} = n^2$

Master method cannot applied here ✗

$f(n) = n^2$ → upper bound

$f(n) = n$ ⇒ lower bound.

> your solution lies between these values.

H/W. Try examples

Algorithm design techniques

Divide and conquer paradigm

1. Divide the problem (instance) into subproblems.
2. Conquer recursively solving the subproblems.
3. Combine the solutions of the subproblems.

Ex^n Binary search problem

Defn: Given an array of n sorted numbers and an number K verify whether K is in the array or not.

simple algorithm:

Traverse the array one by one and compare each element

running time:- O(n)

can we do better?

Divide : check for the middle element

conquer : Recursively search one subarray

combine : Trivial.

```
BinarySearch (A, K, low, high)                          — T(n)
 if low > high                                          — θ(1)
      return no
 else                                                   — θ(1)
      mid = ⌊(low + high)/2⌋                            — θ(1)
      if K == A[mid]                                    — θ(1)
           return yes
      else if K > A[mid]                                — θ(1)
           BinarySearch (A, K, mid+1, high)             — T(n/2)
      else                                              — θ(1)
           Binarysearch (A, K, low, mid-1)              — T(n/2)
return no                                               — θ(1)
```

total time

$T(n) = T(\frac{n}{2}) + \theta(1)$

master method

$T(n) = \theta(\lg n)$

class code: hljk4tm

Integer multiplication

Primary school algorithm takes $O(n^2)$ time

Karatsuba algorithm

```
karatsuba(x, y)                                          ——— T(n)
 if (n == 1)                                             ——— θ(1)
     return x * y                                        ——— θ(1)
 else                                                    ——— θ(1)
     a, b = first and second half of x                   ——— O(n)
     c, d = "    "    "    "    "  of y                  ——— O(n)
     compute p = a + b                                   ——— O(n)
     "       q = c + d                                   ——— O(n)
     ac = karatsuba(a, c)                                ——— T(n/2)
     bd = karatsuba(b, d)                                ——— T(n/2)
     pq = karatsuba(p, q)                                ——— T(n/2)
     adbc = pq - ac - bd                                 ——— O(n)
     return 10^n ac + 10^(n/2) adbc + bd                 ——— O(n)
```

overall

$T(n) = 3T(\frac{n}{2}) + cn$

$T(n) = \theta(n^{\log_2 3})$

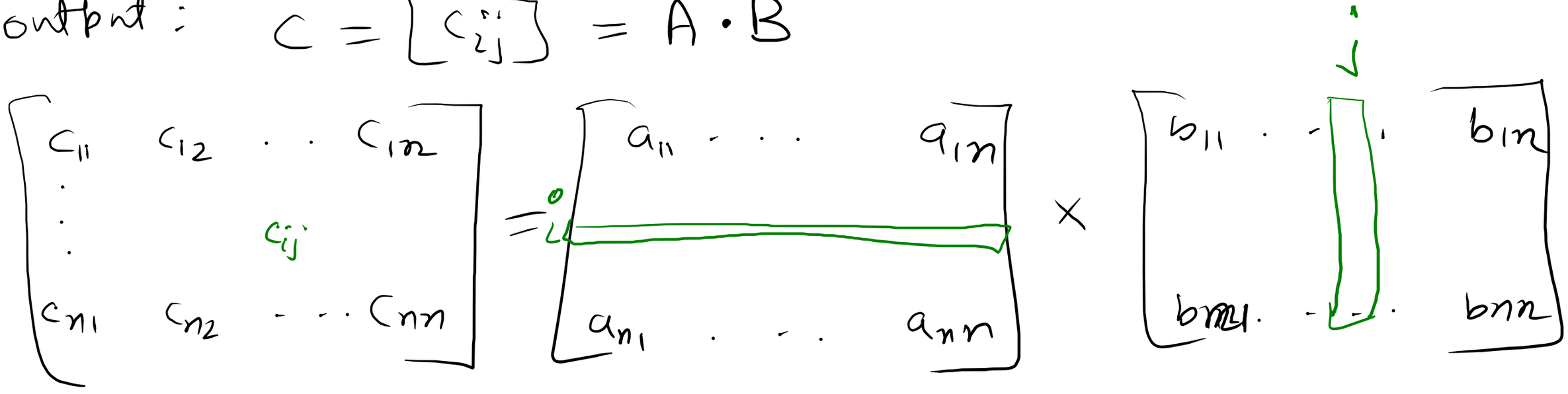
$\log_2 3 \approx 1.59$

$T(n) = \theta(n^{1.59})$

matrix multiplication

Input: $A = [a_{ij}]$, $B = [b_{ij}]$

output : $C = [c_{ij}] = A \cdot B$



$$c_{ij} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

$$= \sum_{k=1}^{n} a_{ik} b_{kj}$$

```
mat-mul (A, B)
 for i = 1 to n
      for j = 1 to n
           c_ij = 0
           for k = 1 to n
                c_ij = c_ij + a_ik · b_kj

 return C
```

Time: $O(n^3)$

can we do better?

Divide and conquer

$$C = AB$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$C_{11} = A_{11} B_{11} + A_{12} B_{21}$

C_{12}

C_{21}

C_{22}

```
mat_mul (A, B)
if (n == 1)
    c11 = a11 b11
else
    partition A into A11 A12 A21 A22
        "     B  "  B11 B12 B21 B22
    c11 = ————————
    c12 = ————————
    c21 = ————————
    c22 = ————————
    return C
```

Total time

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

$$T(n) = O(n^3)$$

No improvement

Idea: need to reduce # of recursive multiplication.

Strassen's algorithm (1969) It uses 7 multiplications.

$$E_1 = A_{11}(B_{12} - B_{21})$$
$$E_2 = (A_{11} + A_{12})B_{22}$$
$$E_3 = (A_{21} + A_{22})B_{11}$$
$$E_4 = A_{22}(B_{21} - B_{11})$$
$$E_5 = (A_{11} + A_{12})(B_{11} + B_{22})$$
$$E_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$E_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} E_5 + E_4 - E_2 + E_6 & E_1 + E_2 \\ E_3 + E_4 & E_5 + E_1 - E_3 - E_7 \end{bmatrix}$$

Total time:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

$$T(n) = O(n^{\log_2 7})$$

$$\approx O(n^{2.808})$$

Progress of the algorithm

$O(n^3)$ — standart

$O(n^{2.808})$ — Strassen 1969

$O(n^{2.796})$ — Pan (1978)

$O(n^{2.522})$ — Schonhage (1981)

$O(n^{2.517})$ — Romani (1982)

$O(n^{2.496})$ — Coppersmith and Winograd (1982)

$O(n^{2.479})$ — Strassen (1986)

$O(n^{2.376})$ — Coppersmith and Winograd (1989)

$O(n^{2.374})$ — Stothers (2010)

$O(n^{2.3728642})$ — V Williams (2011)

$O(n^{2.3728639})$ — Le Gall (2014)

.

.

.

Powering a number

Problem compute a^n where $n \in \mathbb{N}$

Algorithm: multiply a n times.

Time $O(n)$

can we do better?

Divide and conquer

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{n/2} \cdot a^{n/2} \cdot a & \text{" " " odd} \end{cases}$$

$T(n) = 2T(\frac{n}{2}) + \theta(1)$

$\theta(n)$

```
Power (a, n)
 if (n == 1)
      return a
 else
   tmp = Power(a, n/2)
    if n is even
         return tmp * tmp
    else
         return tmp * tmp * a
```

$$T(n) = T\left(\frac{n}{2}\right) + \theta(1)$$

$$T(n) = \theta(\lg n)$$

Ti

Greedy algorithm design paradigm

- It constructs a solution by considering one step at at a time
- At each step it chooses the locally best solution
- In some cases it constructs a globally best solution by repeatedly choosing the locally best option.

Advantages vs challenges

Advantages: Simplicity :- Easy to describe
Efficiency :- efficiently implemented

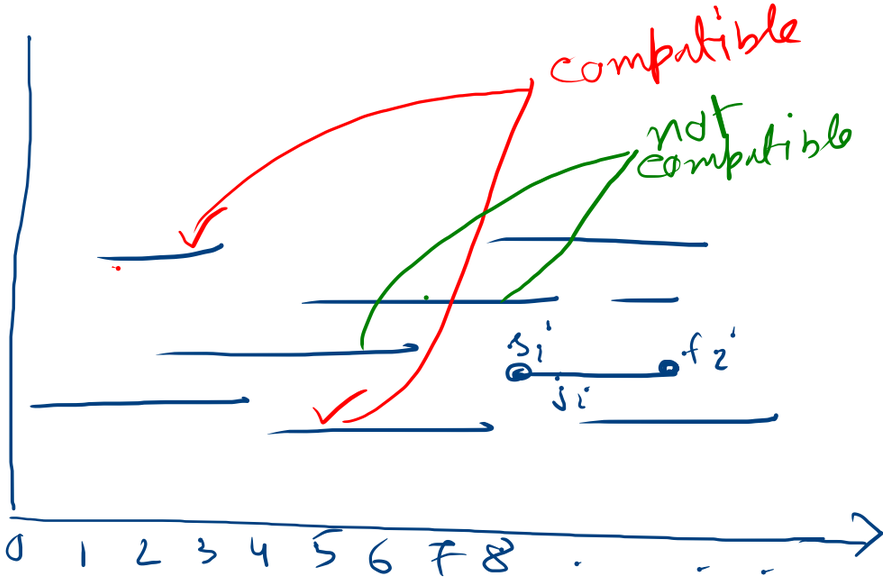
challenges: Hard to design: once you have the right greedy approach design greedy algorithm is easy.

Hard to verify: The correctness often requires critical arguments.

Activity selection Problem / interval scheduling problem
maximum independent set problem in intervals given on a line.

Input: n jobs $J = \{j_1, j_2, \ldots, j_n\}$

each job $j_i \in J$, start time s_i
finish time f_i



Feasible solution :-

Two jobs are **compatible** if they do not overlap.

Find a subset of jobs that are mutually compatible

objective: maximise the size of the mutually compatible set of jobs.

Pick the job which finishes earlier.

Greedy rule.

- Select jobs one after another using some rule

Rule 1 → Earliest start time

Rule 2 :- Smallest job first

Rule 3 :- Smallest conflict job first

Rule 4: Earliest finish time

Rule 1: not optimum

Rule 2:

H. W.

Rule 3:

H. W.

Rule 4: Earliest finish time

Greedy rule:

- Initially J be the set of jobs and A be an empty set
- While J is not empty
  - choose a job $j \in J$ that has the smallest finish time
  - Add j to A
  - Delete all jobs from J that are not compatible with j
- Return A

$O(n \log n + n)$

$= O(n \log n)$

correctness :-

claim: A is a feasible solution.

Proof straightforward.

claim: A is optimum

A ← set of jobs return by the algorithm

opt ← largest set of pairwise non-overlapping jobs.

what we have to prove?

A must be as large as opt

$$|A| = |opt|$$

$A = \{A_1, A_2, \ldots, A_K\}$
$Opt = \{O_1, O_2, \ldots, O_m\}$
} Assume A and Opt are sorted.

A = ——— ——— —— ———

Opt = ——— ——— ——— ——— ———

Question: What is the relation between K & m

Answer: $K \leq m$

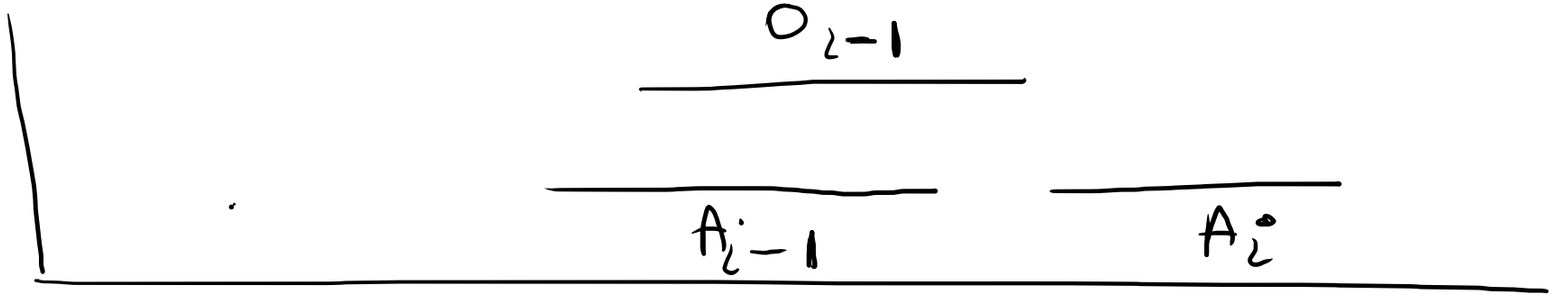
our aim :- ~~K~~ = m

claim for every $i \leq K$
A_i finishes not later than O_i

Proof prove using induction

Base case:- $i = 1$ it is true A_1 finishes not later than O_1

I.H: A_{i-1} finishes not later than O_{i-1}



If O_i finishes before A_i then it would overlap with A_{i-1} that means it overlap with O_{i-1}

$\Rightarrow \Leftarrow$

Main claim A is the optimum solution.

Proof we need to prove $k = m$.

If $k = m$ we are done.

Assume that $k < m$.

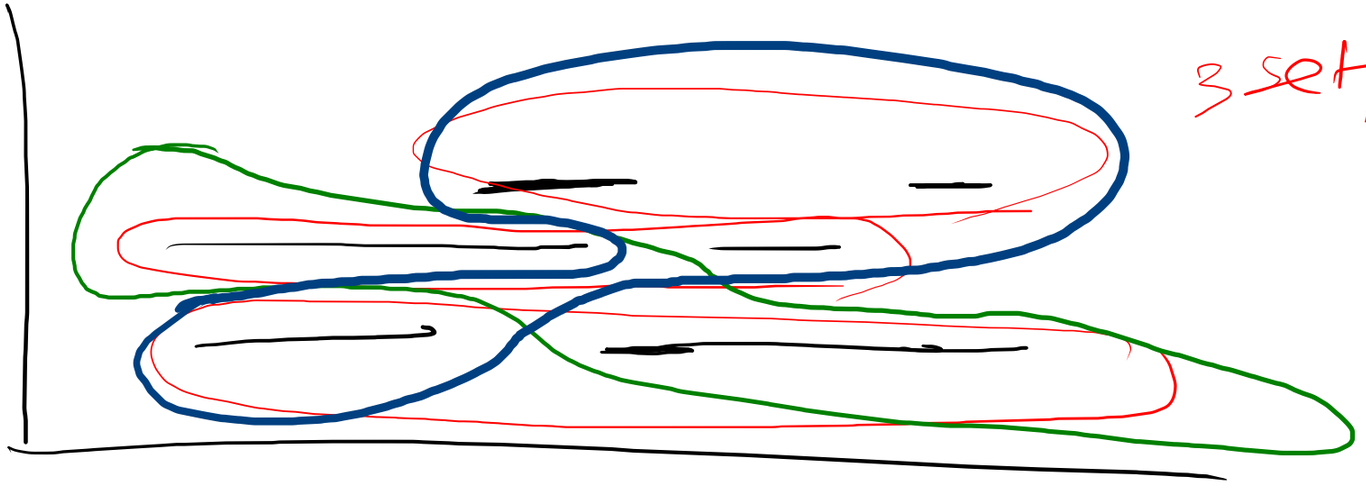
O_{k+1} starts after O_k and consequently A_k we could add O_{k+1} in A and obtain a bigger solution. $\Rightarrow \Leftarrow$

Interval partitioning problem

Input: A set of n jobs $J = \{J_1, J_2, \ldots, J_n\}$ where each job J_i has a start time s_i and a finish time f_i.

Output: A partition of the jobs in J into K sets such that each set of jobs are mutually compatible.

Objective: minimize K



3 sets $K=3$

green, blue } 2 sets $K=2$

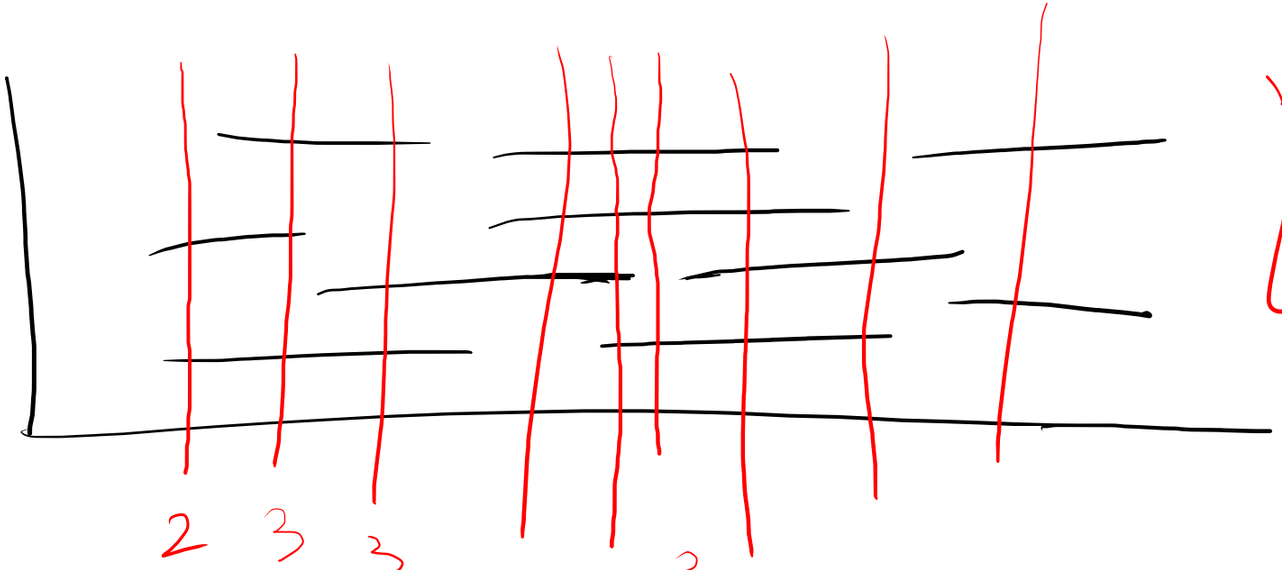
Situations where the problem appears

i) you have a set of fixed jobs and want to schedule them using minimum resources.

ii) There are a set of lectures that are schedule in some classrooms.
one wants to schedule the lectures using minimum classrooms.

Depth of a set of intervals

The maximum number of intervals that contain any given time



depth = 4

observation: Number of sets must be $k \geq$ depth

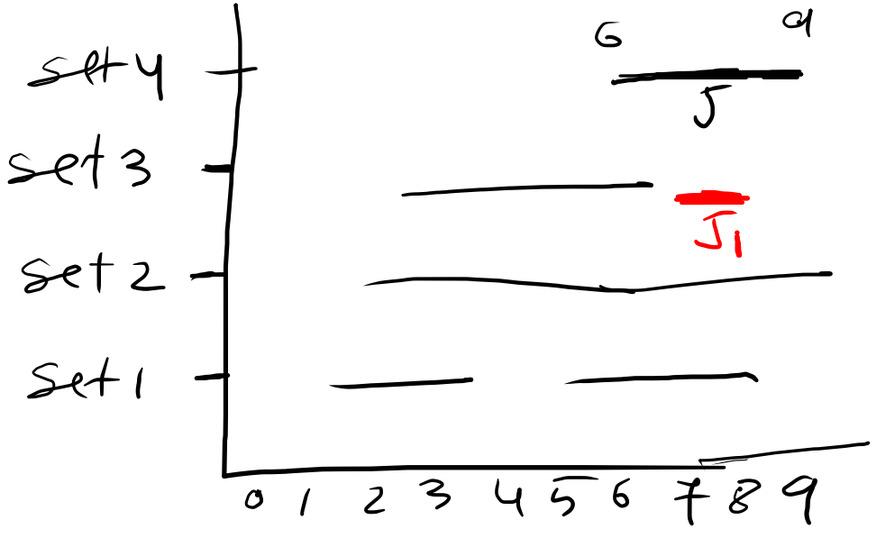
Question: What about k = depth ??

$\Rightarrow$ schedule is optimum.

Target: — Need a solution with depth number of sets.

We design greedy algorithm

- consider the jobs in some order — Which order?
- Assign each job to an available set — Which set?
  - If all the ~~sets~~ are not available then create a new set.



j → 6 — 9

J_1 → 7 - 8

rule 1 : Earliest finish time ✗

rule 2 : shortest interval first ✗

rule 3 : Fewest conflict ✗

rule 4 : Earliest start time. ✓

H.W. counterexamples

Earliest start time

Input $(n, (s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n))$

- Sort the jobs in increasing order of their start time. $s_1 \le s_2 \le s_3 \le \cdots \le s_n$ — $O(n \log n)$
- depth 0 — $\theta(1)$
- for $i = 1$ to n — $O(n)$ times
  - if J_i is compatible with some job in any set — $O(n)$
    - assign J_i in such set.

Priority queue $O(\log n)$

  - else
    - create a new set — depth + 1 — $\theta(1)$
    - schedule J_i in depth + 1 set
    - depth = depth + 1
- return the schedule.

Total time $O(n^2)$

improved $O(n \log n)$

Correctness

claim: The algorithm never schedule two incompatible jobs in the same set.

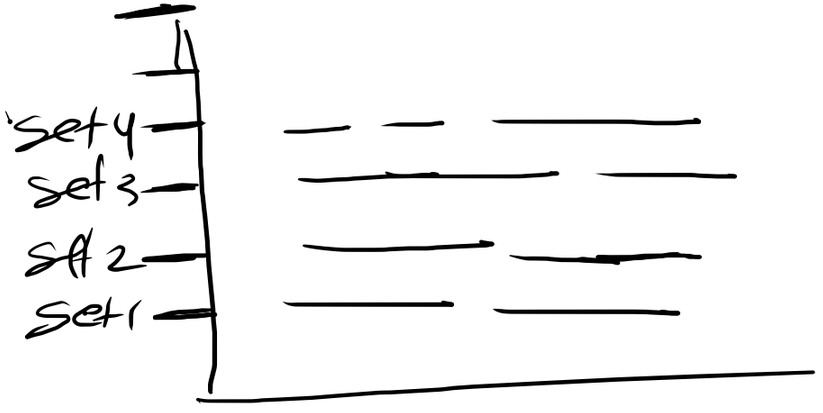
Straightforward.

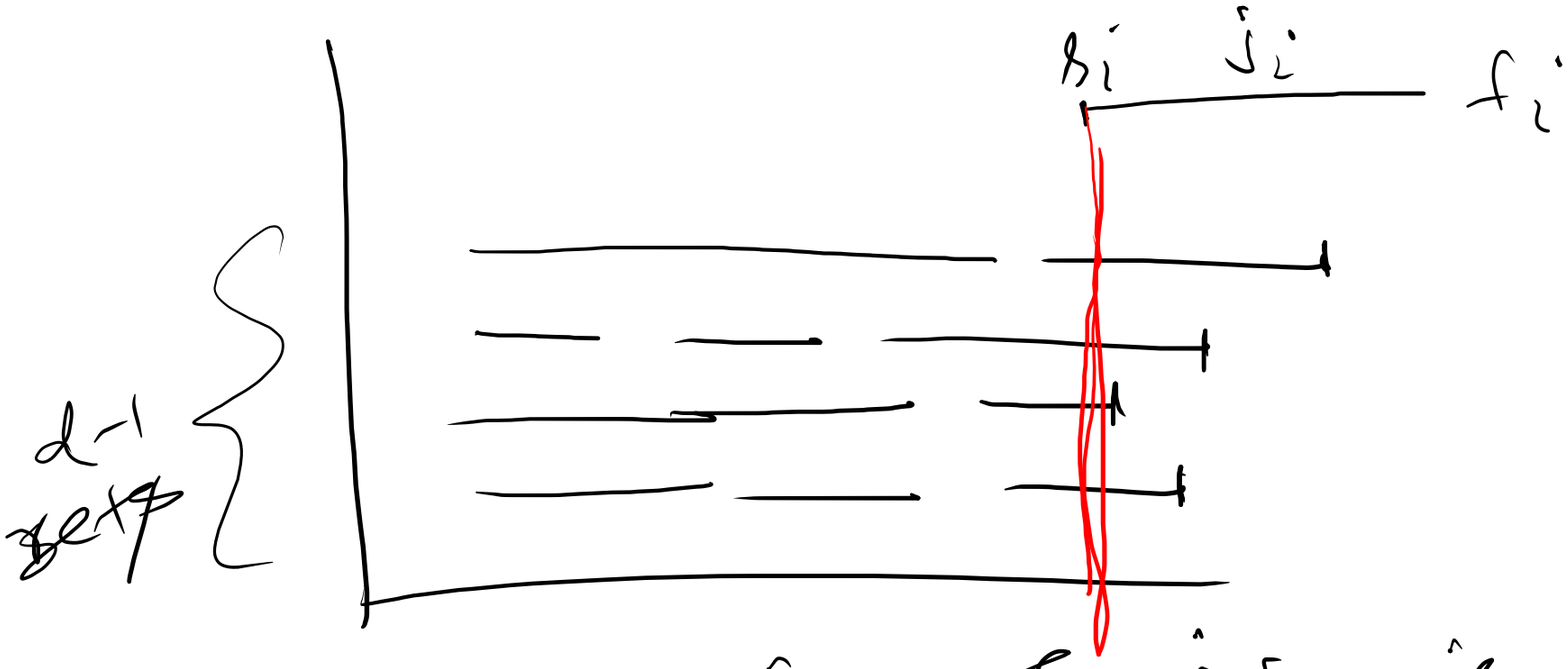
claim: The greedy algorithm is optimal.

Proof: Let d be the no. of sets the algorithm returns.

Question: When the last set ie, the d-th set first used?

The algorithm trying to add J_i^{th} job but it is incompatible with all other $d-1$ sets.





The start time of j_i is in between the start time and finish time of all the last jobs in d-1 sets.

Therefore the algorithm uses d sets only because of ther are d overlapping jobs ie, the depth of the problem is d.

Minimize the maximum lateness

Assume that there is only one processor

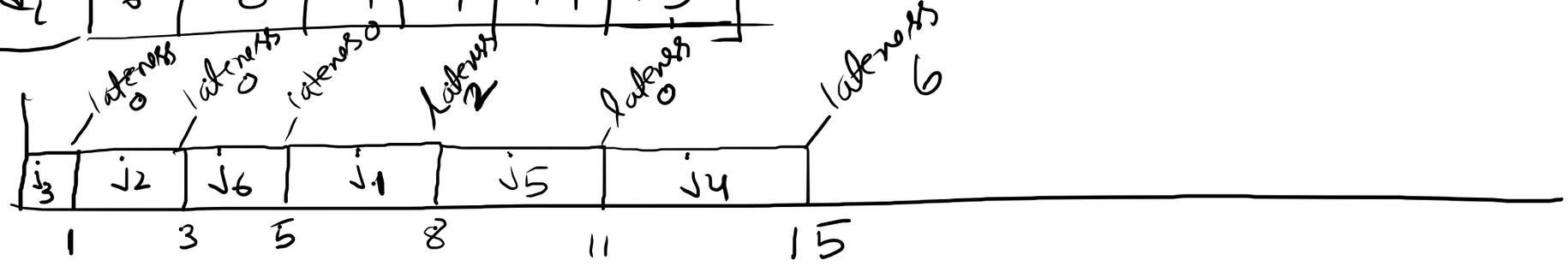
Input: A set J of n jobs $j_1, j_2, \ldots, j_n$ where each job j_i has a processing time t_i and a deadline d_i

output: Schedule the jobs in one processor such that the maximum amount time that any single job is past its deadline is minimized.

objective:

Exm

	j_1	j_2	j_3	j_4	j_5	j_6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15



$j_3 \rightarrow j_2 \rightarrow j_6 \rightarrow j_1 \rightarrow j_5 \rightarrow j_4$

objective: Find a schedule that minimizes the lateness

greedy template:

- consider the jobs in some order Which order ??
- Assign the jobs in this order to the resource.

rule 1: shortest processing time ✗ H.W: counterexample

rule 2: shortest slack time ✗

$d_i - t_i$

rule 3: Earliest deadline first ✓

Algorithm

Earliest deadline first $(n, (t_1, d_1), (t_2, d_2), \ldots, (t_n, d_n))$

- sort the jobs by their deadlines.
- $d_1 \leq d_2 \leq \cdots \leq d_n$ be the order
- $t = 0$
- for $i = 1$ to n

$$s_i = t, \quad f_i = t + t_i$$

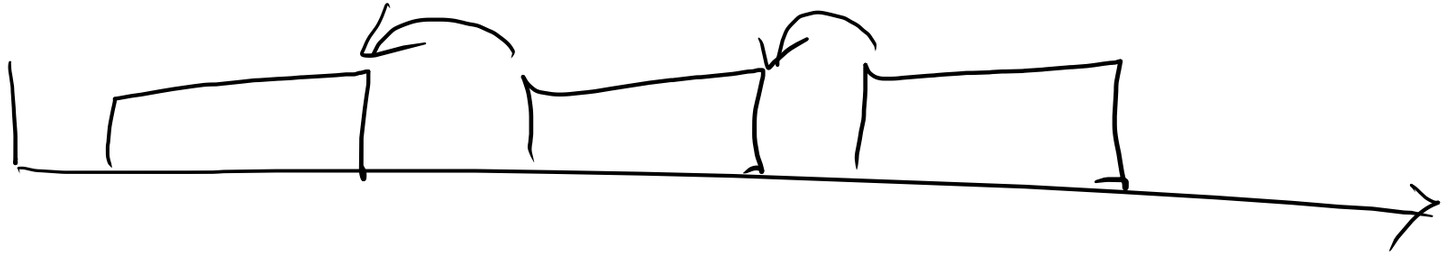
$$t = t + t_i$$

- output intervals: $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$

running time $O(n \log n)$

correctness

observation: There exists an optimum schedule with no idle time.

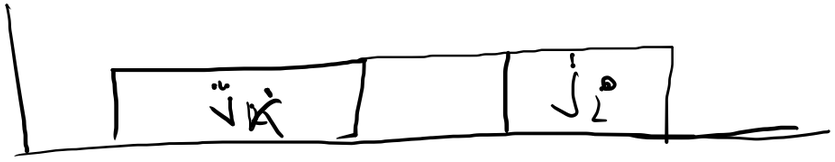


observation about our greedy schedule

EDF algo returns a schedule with no idle time

Definition inversion

An inversion is a pair of jobs j_i and j_k such that $d_i < d_k$ but j_k is scheduled before j_i



However $d_k > d_i$

observation about greedy

EDF does not have any inversion.

Claim If an idle-free schedule has an inversion then it is an adjacent inversion.

Proof j_i and j_K be a closest inversion but not adjacent.

$d_i < d_K$

There are two cases

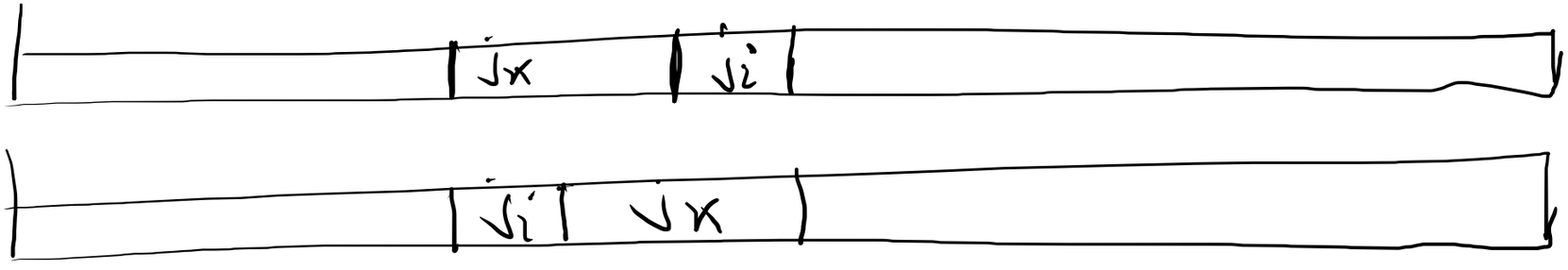
- $d_K > d_\ell \Rightarrow$ j_K and j_ℓ are inverted jobs.

- $d_K < d_\ell \Rightarrow$ $d_i < d_K < d_\ell$

j_ℓ and j_i are inverted jobs. $\Rightarrow\Leftarrow$

claim If we invert two adjacent inverted jobs j_i and j_k then it reduces the number of inversions by 1 and does not increase the maximum lateness.

Proof



- $L \leftarrow$ ~~be~~ the lateness before exchange
  $L' \leftarrow$ " " after exchange.
- For all other jobs other than j_i and j_k their lateness remain same.
- For the i-th job j_i:
  $l_i^{new} \leq l_i^{old}$ as it scheduled befor now.

we want to show, $l_k^{new} \leq l_i^{old}$.

- If the job j_k is late.

$$l_k^{new} = f_k^{new} - d_k \quad \text{def}^n \text{ of lateness}$$

$$= f_i^{old} - d_k \quad \text{as } j_k \text{ finishes}^{now} \text{ at time } j_i \text{ in old schedule.}$$

$$\leq f_i^{old} - d_i^{old} \quad d_i \leq d_k \text{ by def}^n \text{ of inversion}$$

$$= l_i^{old}$$

lateness of j_k can not be more than the lateness of l_i in the old schedule.

current lateness $= \max \{ l_1^{new}, l_2^{new}, \ldots l_k^{new}, l_i^{new}, \ldots l_n^{new} \}$

$$\leq \max \{ l_1^{old}, l_2^{old}, \ldots, l_i^{old}, l_i^{old}, \ldots, l_n^{old} \}$$

- The algorithm produces a schedule with no inversion and no idle-time
- There is an optimum schedule with no inversion and no idle time.
- All schedule with no inversions and idle-free have the same lateness.

=> Greedy EDF is optimum.

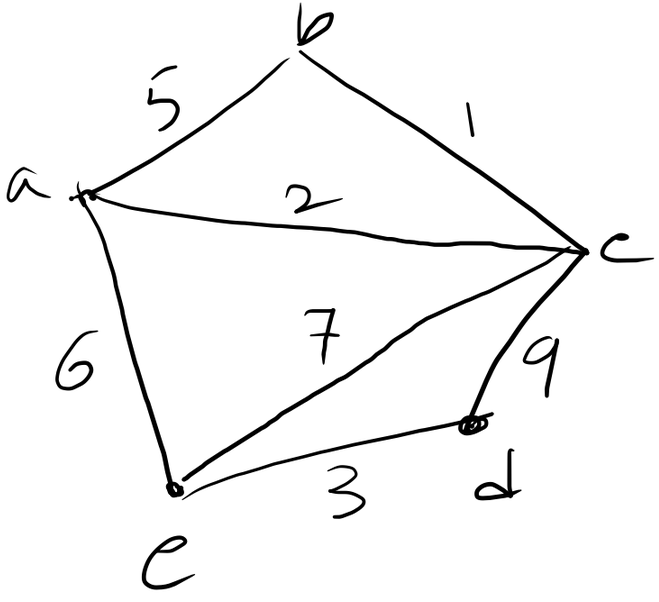
Minimum spanning tree (MST)

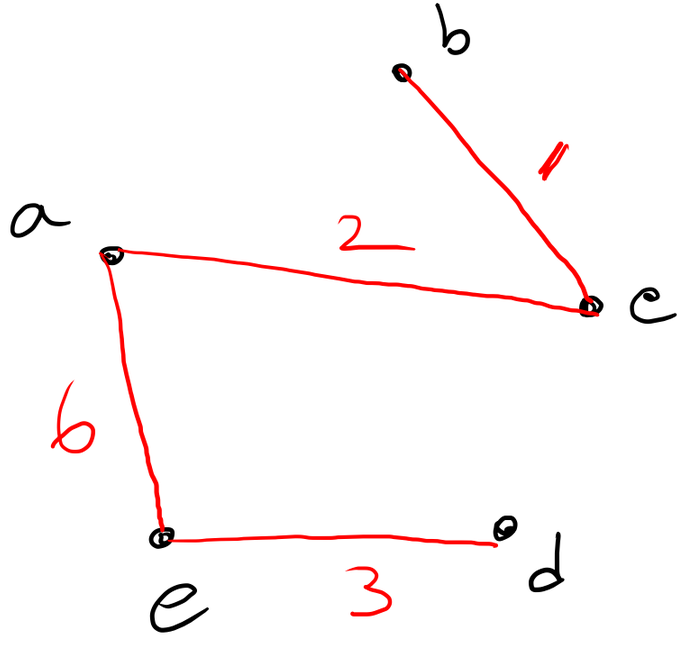
Problem: A connected graph $G(V, E)$ with edge costs.

$$w : E \rightarrow \mathbb{R}^+$$

Feasible solution: A subset $T \subseteq E$ such that $G_T(V, T)$ is connected

objective Total cost of all the edges in T is minimum.





cost of $T = 12$

Greedy algorithm

```
Generic-MST (G, w)
T is empty
while E is not empty
        choose e in E
        if (e satisfies some condition)
                add e to T
Return the set T
```

The graph $G_T = (V, T)$ is MST.

main task: → In what order should the edges be processed?
→ when should an edge be a part of the spanning tree?

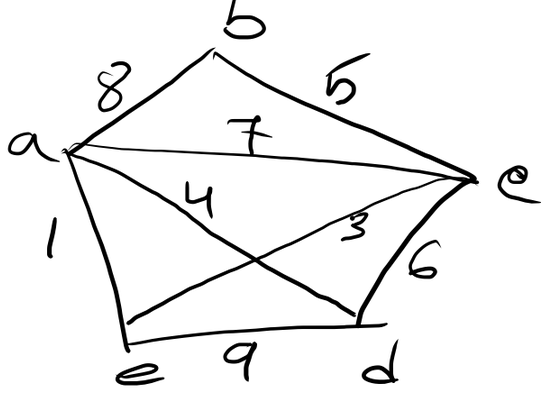
Kruskal's algorithm

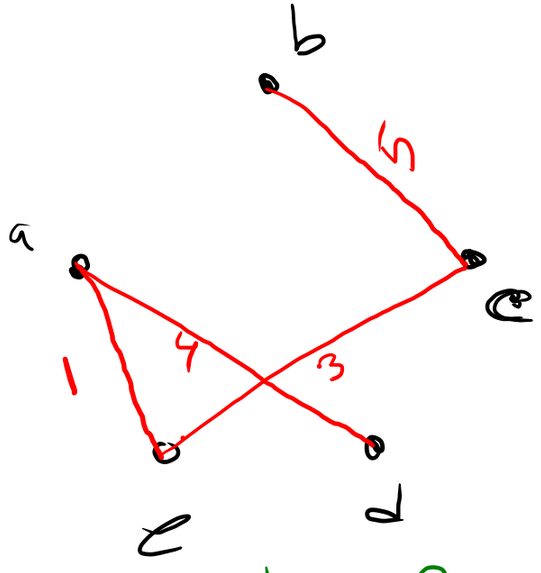
Kruskal-MST(G, w)

- T is empty
- sort the edges of G in non-decreasing order by their weights
- while E is none empty
  - choose $e \in E$ in the above order
    - if (e does not form a cycle with the edges in T)
      - add e to T
  - return the set T.

✓ ✓ ✓ ✓ X X X X
1 3 4 5 6 7 8 9

$T = \{(a,e), (c,e), (a,d), (b,c),$





cost = 13

Prim's algorithm

Prim - MST (G, w)

- T is empty
- $X = \{s\}$ where s is an arbitrary start vertex.
- while there is an edge $e = (u, v)$ where $u \in X$ and $v \notin X$

  $e^* = (u^*, v^*)$ be a minimum such edge

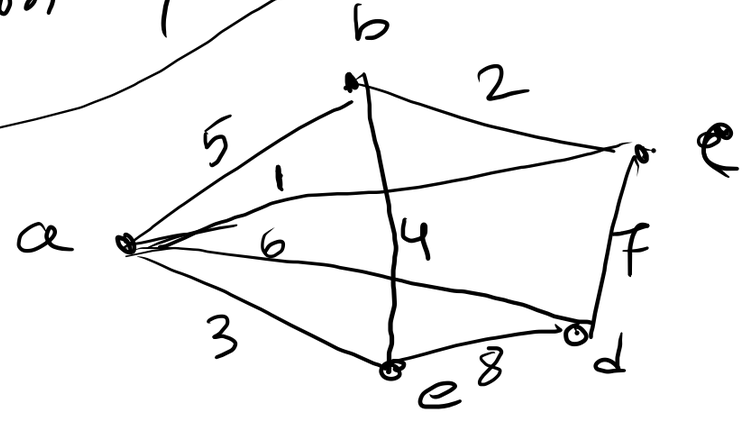
  add v^* in X

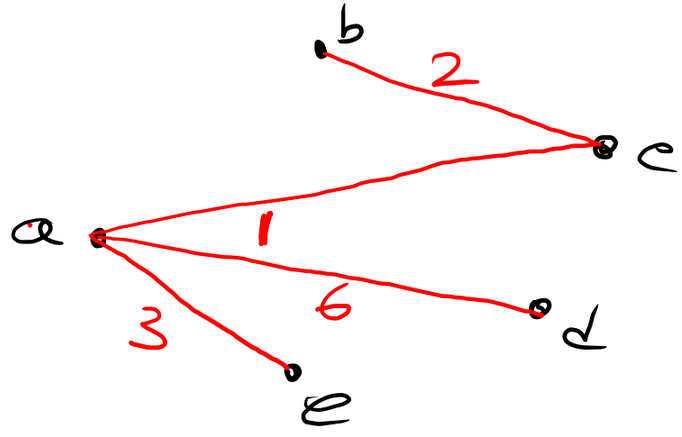
  add e^* in T

return T

$T = \{(a,c), (c,b), (a,e), (a,d)\}$

$X = \{a, c, b, e, d\}$

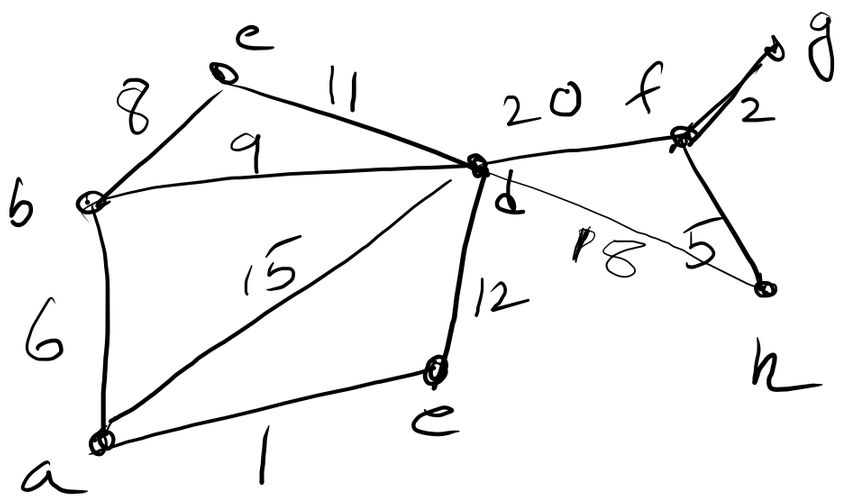


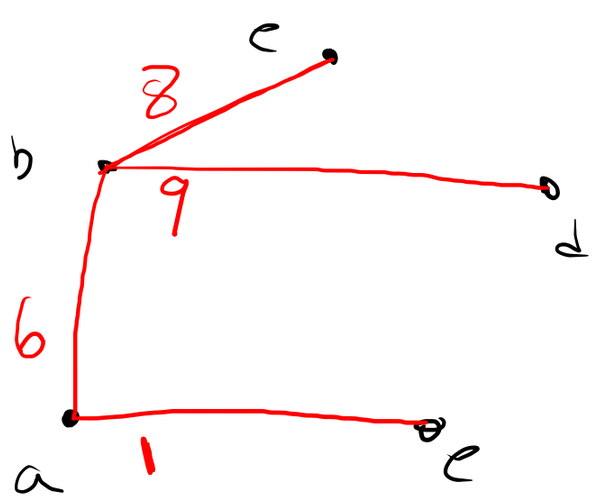


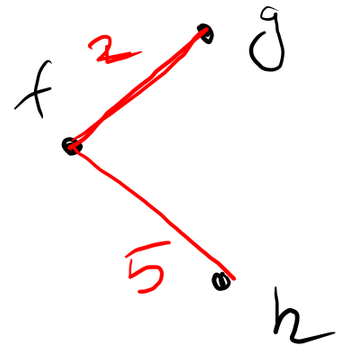
Boruvka's algorithm

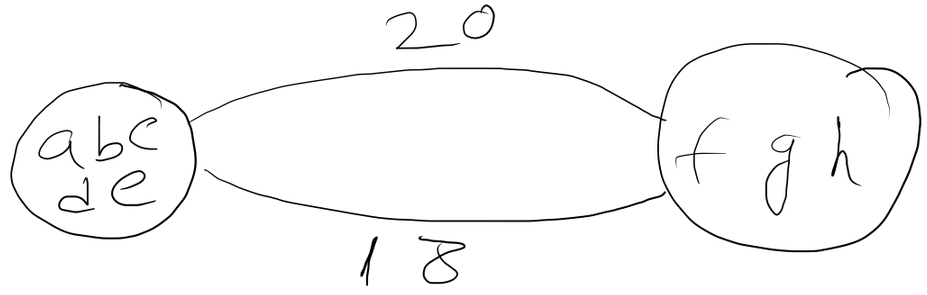
Boruvka-MST(G, w)

- T is empty
  (assume that each vertex is a connected component)
- for each vertex $v \in V$
  - add the edge in T whose weight is minimum over all edges incident on v.
- G' be the graph generated by contracted all the edges of T
- T' be the MST computing recursively on G'
- return the set $T \cup T'$









$T' = (d, h)$ of weight 18

Reverse delete

Reverse-delete-MST(G, w)

- T is E
- sort the edges in E in non-increasing order by their weights.
- while E is not-empty:
  - choose e with largest weight
    - if removing e does not disconnect T
      remove e from T

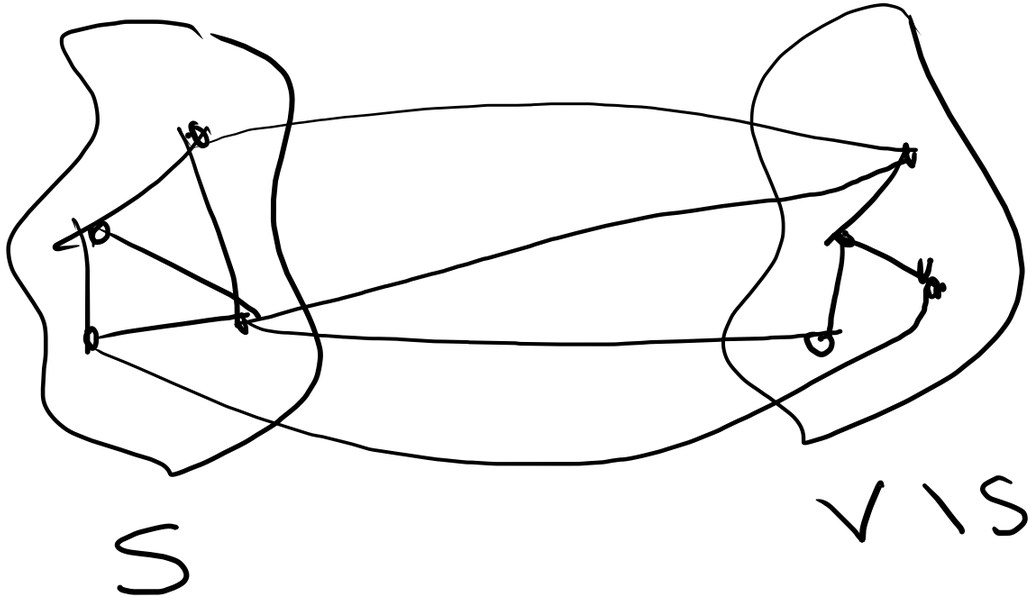
return the set T.

Correctness of MST algorithms

- There are many MST finding algorithms
- All of them rely on some basic properties of MST
  - cut property
  - cycle ,,

- Assume that edge costs are distinct.

cut: Partition of vertex set of a graph in S and $V \setminus S$

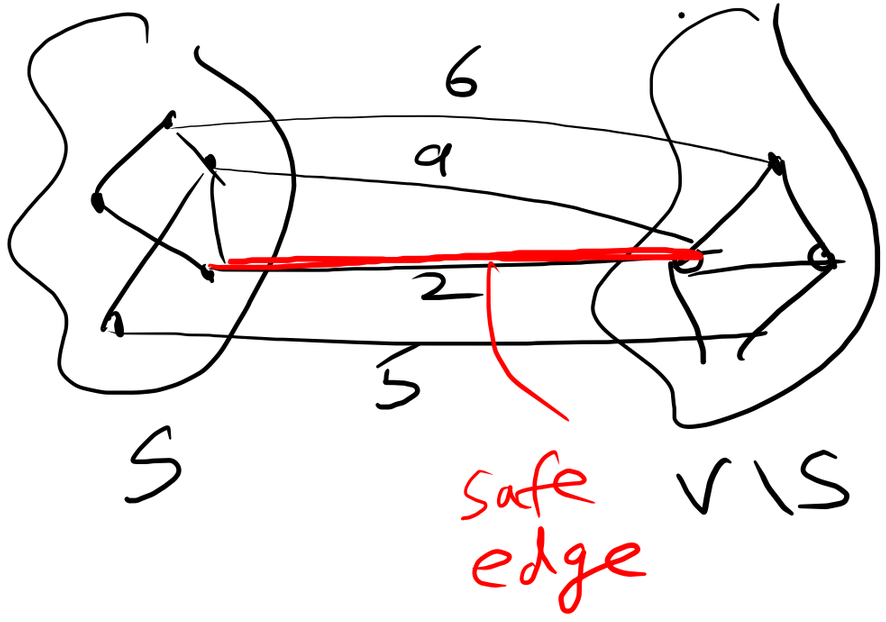


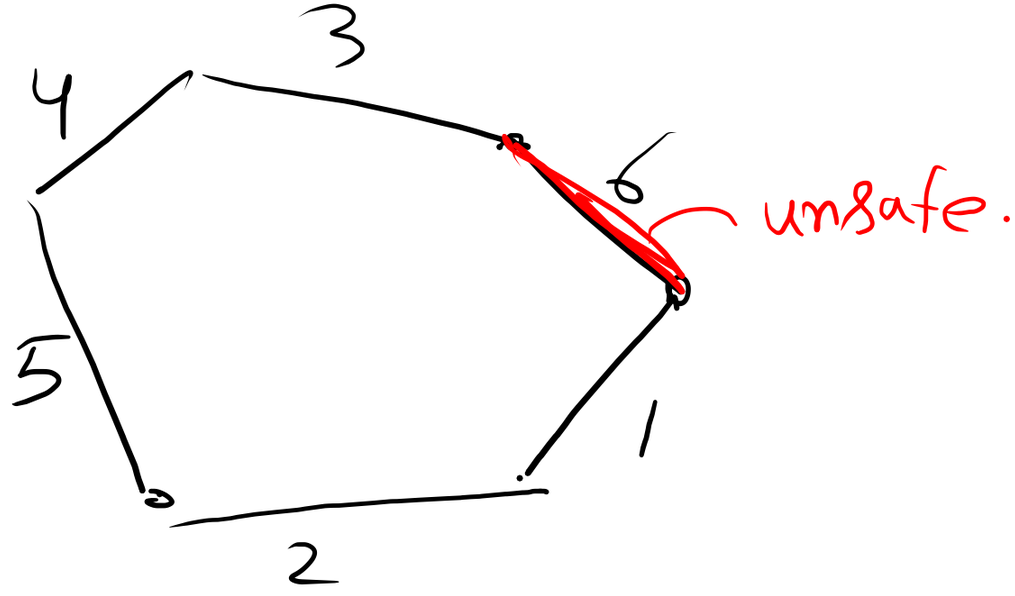
edges in the cut: edges crossing the cut.
one end in S and another in $V \setminus S$

Safe and unsafe edges

safe edge: unique minimum weight edge crossing a cut.

unsafe edge: unique maximum weight edge of a cycle.



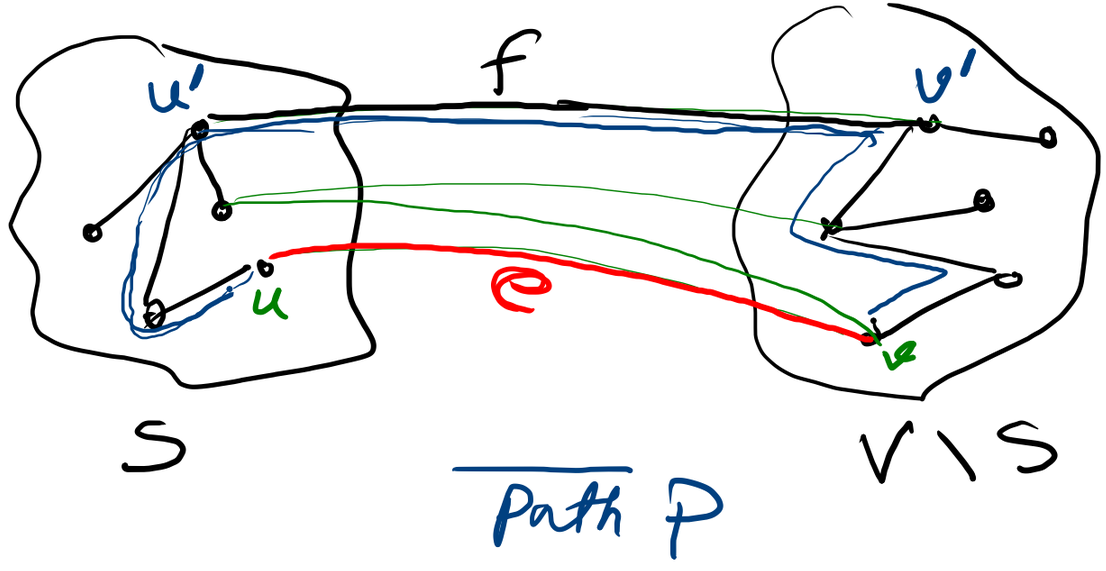


cut property and its correctness

cut property: Let S be any subset of vertices of V and $e=(u,v)$ be the minimum cost edge with one end point in S and the other in $V \setminus S$. Then every MST contains e.

Proof By contradiction.

- G_T be a MST that does not contain e.
- Assume that $u \in S$ then $v \notin S$
- Since G_T is a MST there is a unique path P from u to v.



- $f = (u', v')$
- v' is the first vertex on P (starting from u) in $V \setminus S$
- u' is just befor v'
- consider the graph $G_{T'} = (G_T \setminus \{f\}) \cup \{e\}$

If $G_{T'}$ is a spanning tree → we need to prove.

we have a contradiction due to G_T is a MST

claim $G_{T'}$ is a spanning tree

i) $G_{T'}$ is connected. ✓

ii) $G_{T'}$ is a tree ✓

iii) $G_{T'}$ has lower cost than G_T ✓

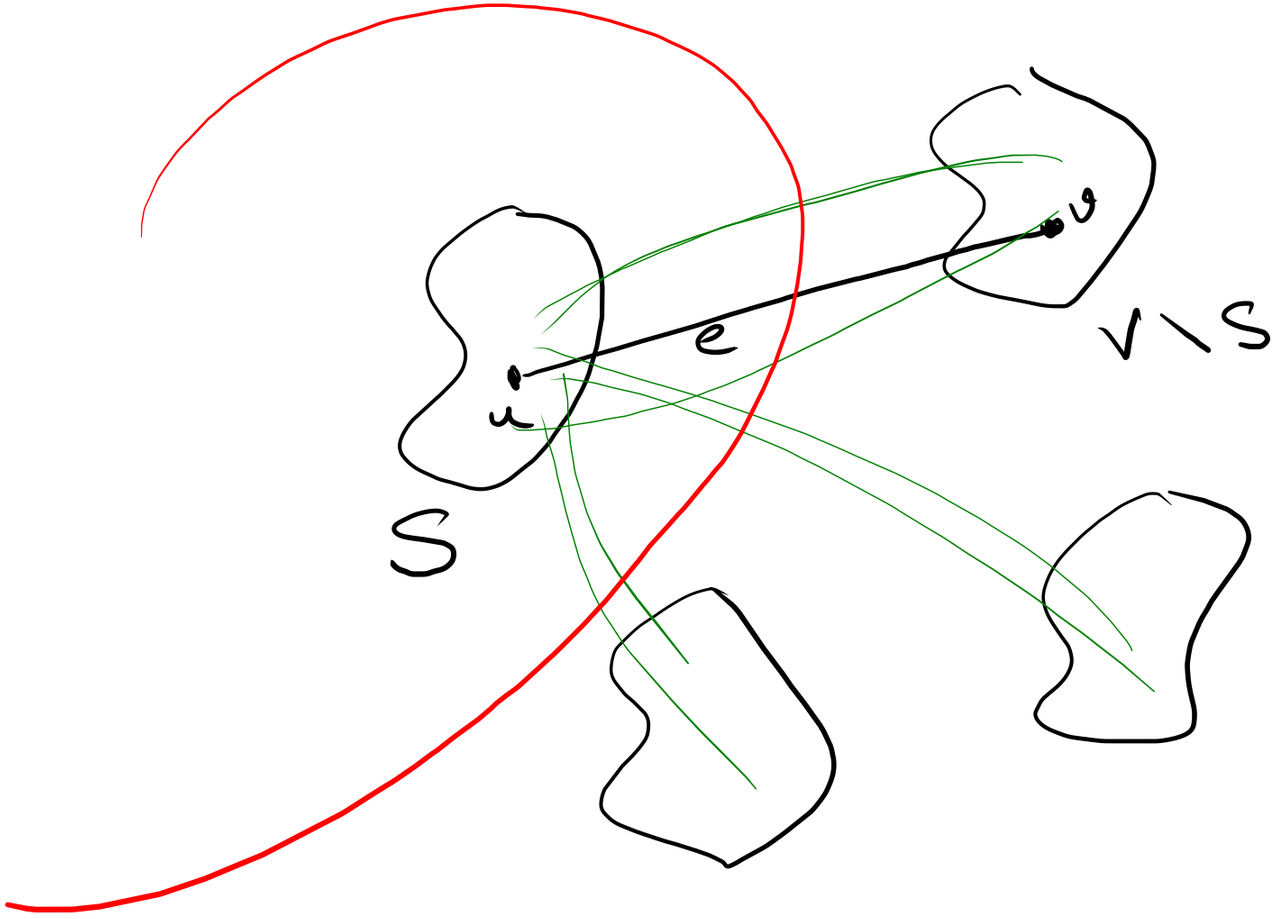
cycle property :—

H.W.

- $e = (u, v)$ be a edge of smallest weight not creating any cycle.

- e does not belong to a single component.

-



Boruvka's algo
Reverse delete } H.W.

Running time

Graph with n vertices
m edges

Kruskal $\rightarrow O(m \lg m + m \cdot n)$

$\rightarrow O(mn)$

Prims $\rightarrow O(mn)$

Boruvka's $\rightarrow T(mn) = T\left(\frac{n}{2} m\right) + O(m)$

Reverse delete $\rightarrow O(mn)$

Kruskal's algorithm : Efficient implementation

Kruskal-MST (G, w)

- T is empty
- Sort the edges based on non-decreasing weights.
- each vertex v is placed in a set ⟸ make set
- while E is not empty
  - choose $e = (u, v) \in E$
  - if u and v belongs to two different sets ⟸ Find(u) ≠ Find(v)
    - add e to T
    - merge the sets containing u and v ⟸ Union(u, v)
- return the set T

Sorting — $O(|E| \log |E|)$

$|V|$ ← makeset operations.

$2|E|$ ← Find(u) " "

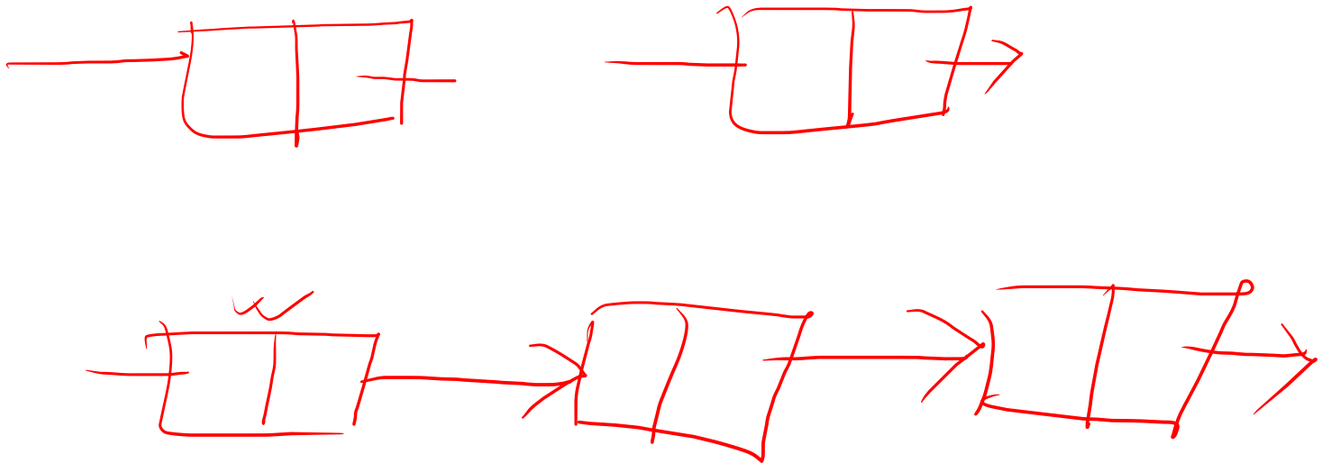
$|V|-1$ ← Union "

Total running time $O(|E| \log |E|) + |E| \log |V|$

If we implement disjoint set operations by union by rank.

then makeset $O(1)$ Union — $O(\log |V|)$

Find(u) — $O(\log |V|)$



Merge .

Prim's algorithm

$\pi(v) \leftarrow$ minimum cost between v and s

$Pred[v] \leftarrow$ vertex just before v

Prim-mst(G, w) — T

- T is empty — $\theta(1)$
- $X = \{s\}$ — $\theta(1)$
- for each vertex $v \neq s$ — $O(|V|)$
  $\pi[v] \leftarrow \infty$, $Pred[v] \leftarrow$ Null — $\theta(1)$
- $\pi[s] \leftarrow 0$ — $\theta(1)$
- create an empty priority queue Q — $\theta(|V|)$
- for each vertex $v \in V$ — $\theta(|V|)$
  Insert (Q, v, $\pi[v]$) — T_{insert}
- while Q is not empty
  $u \leftarrow$ Extract-min (Q)
  for each edge $v \in Adj[u]$
  if $v \in Q$ and $w(u,v) < \pi[v]$
  Decrease-key (Q, v, $\pi[v]$)
  $Pred[v] \leftarrow u$

$O(|V|)$ times

$O(|E|)$ times

$T = O(|V|)\, T_{insert} + O(|V|)\, T_{extractmin} + O(|E|)\, T_{Decrease\text{-}key}$

priority queue	$T_{extract\text{-}min}$	$T_{decrease\ key}$	Total.
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\lg \|V\|)$	$O(\lg \|V\|)$	$O(\|E\| \lg \|V\|)$
Fibonacci heap	$O(\lg \|V\|)$ amortized	$O(1)$ Amortized	$O(\|E\| + \|V\| \lg \|V\|)$ Amortized

Shortest path problem

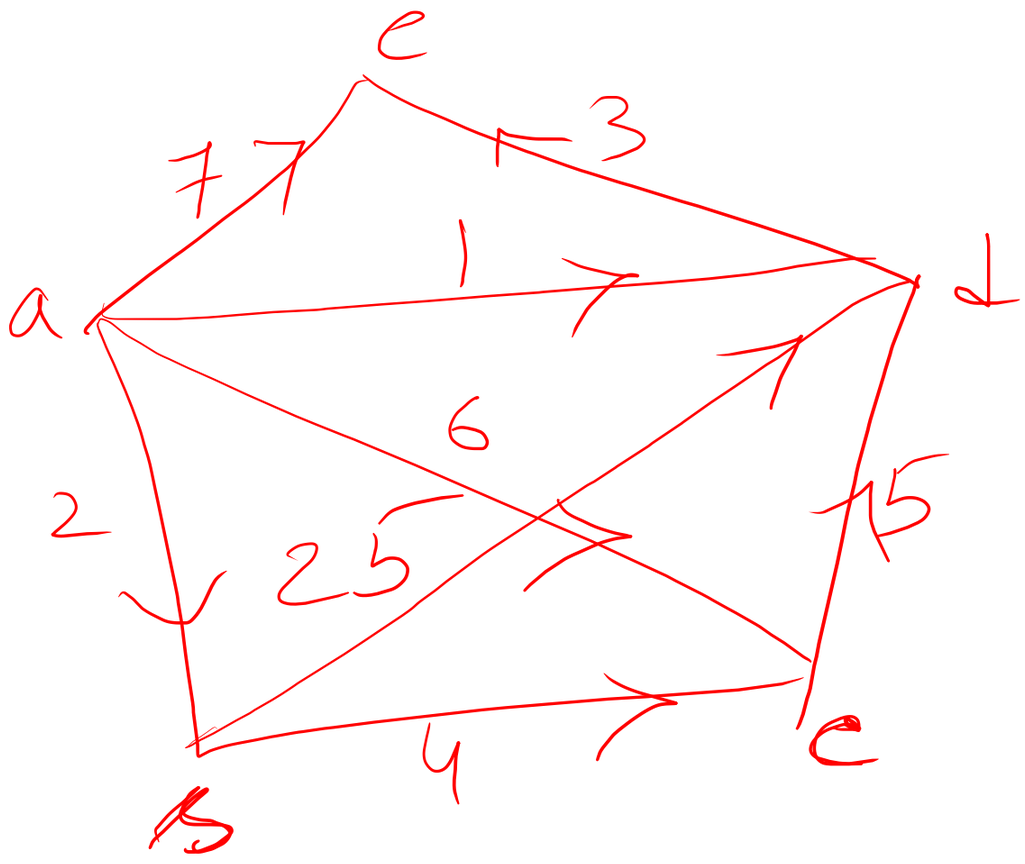
- Given a (directed or undirected) graph $G(V, E)$ with edge costs $w : E \to \mathbb{R}^+$

output: i) Given $s, t \in V$ find shortest path from s to t

ii) Given $s \in V$ find shortest path from s to all other vertices

iii) Find shortest paths from all pairs of vertices.

→ single source shortest path



d = t

s → c → d (=t) cost 9

s → d (=t) cost 25

Shortest path problem

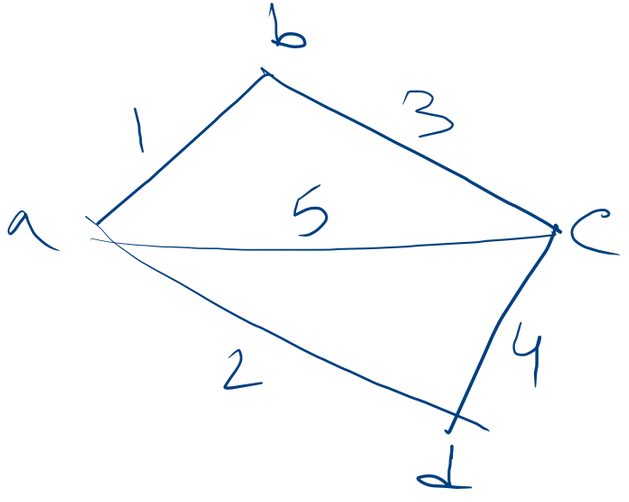
- single source shortest path problem.

  Given s find shortest path from s to all other vertices

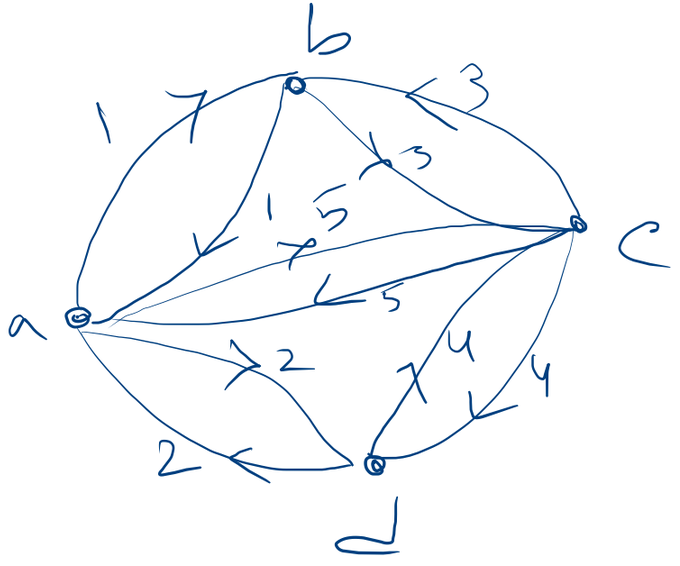
  Given s, t " " " from s to t.

⇒ we will concentrate on directed graphs only.

Why?.



⟹



H.W.

Special cases 1

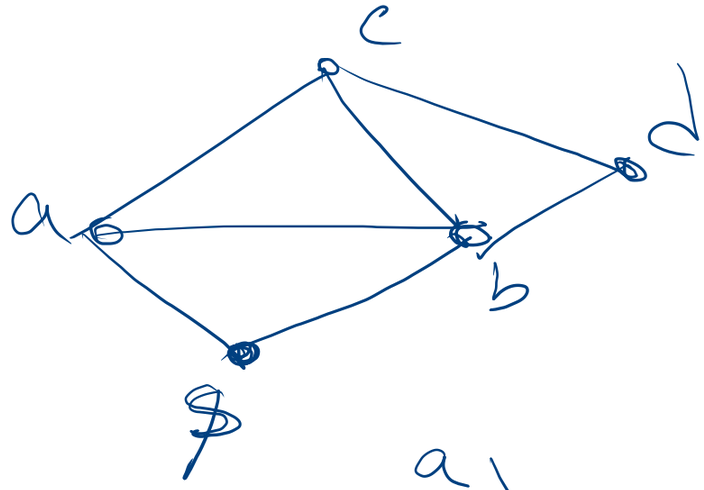
— Assume the edge weights are all 1.

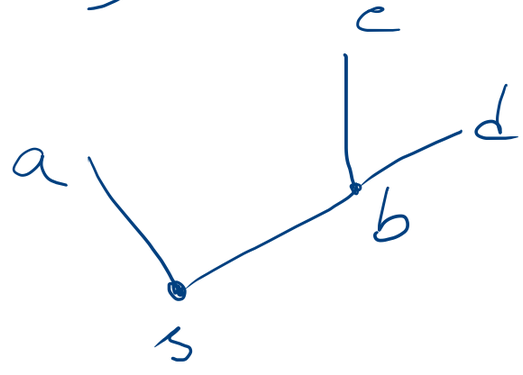
— use BFS algorithm.

Time: $O(|V|+|E|)$

$|V| = n$

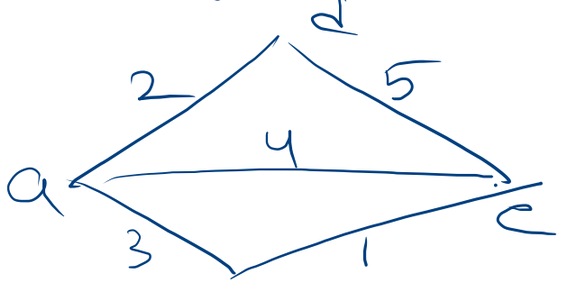
$|E| = m$

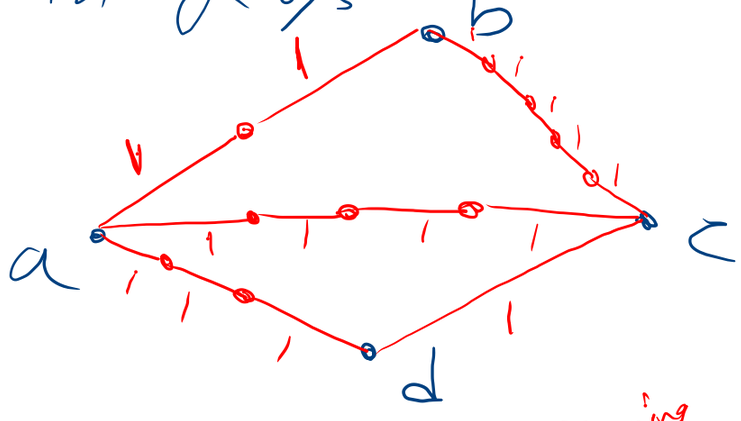




Special case 2

— All edge weights are integers





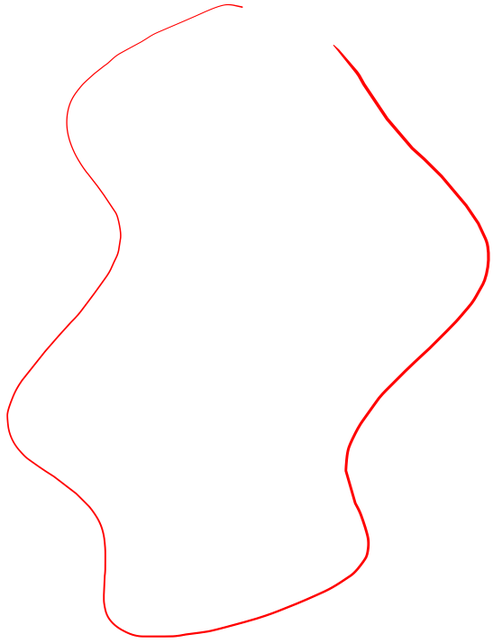
L = maximum Number of edges added in an old edge

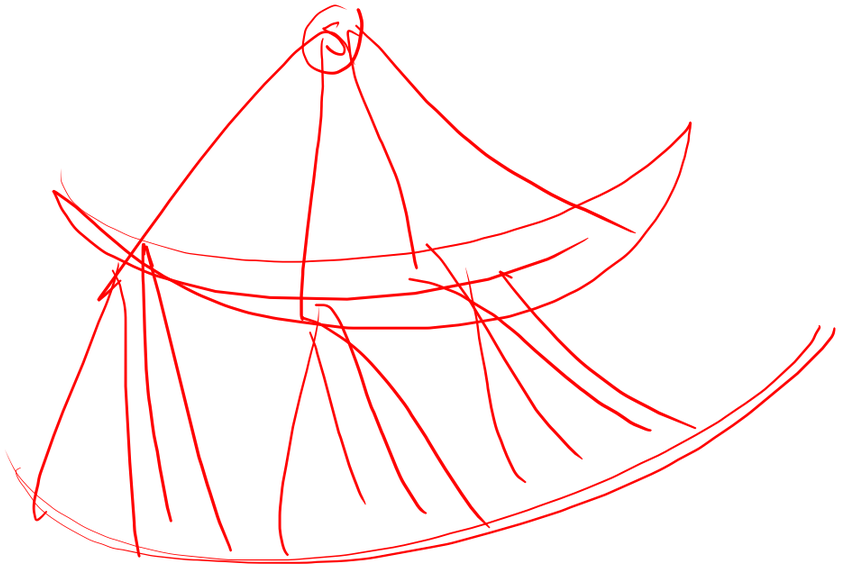
vertices = $mL+n$

edges = mL

running time: $O(mL+n)$

For large L, this algorithm is not efficient.

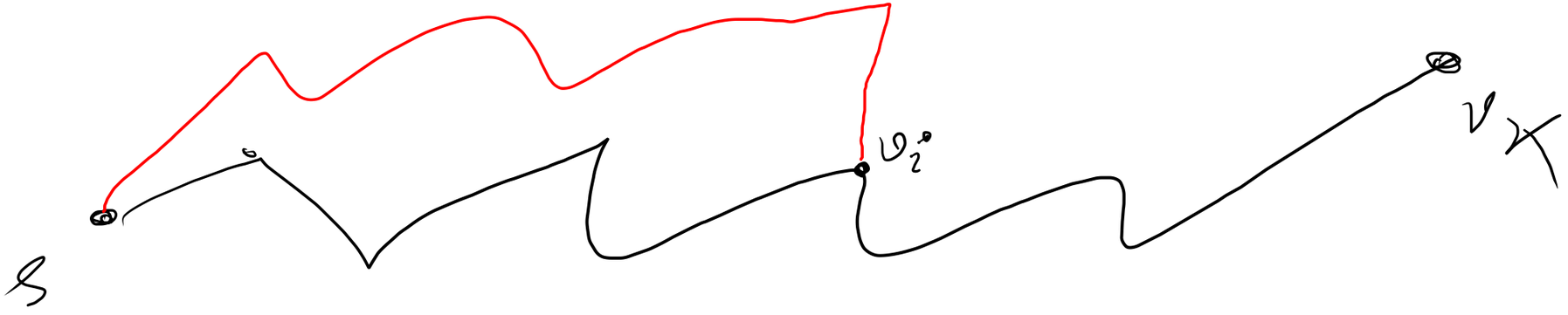




Let G be a directed graph with non-negative edge weights. Let $dist(s, v)$ is the shortest path weight from s to v.

If $s = v_0 \to v_1 \to v_2 \cdots \to v_k$ be the shortest path from s to v_k then for $1 \le i \le k$

- $s = v_0 \to v_1 \to \cdots \to v_i$ is a shortest path from s to v_i
- $dist(s, v_i) \le dist(s, v_k)$



General case

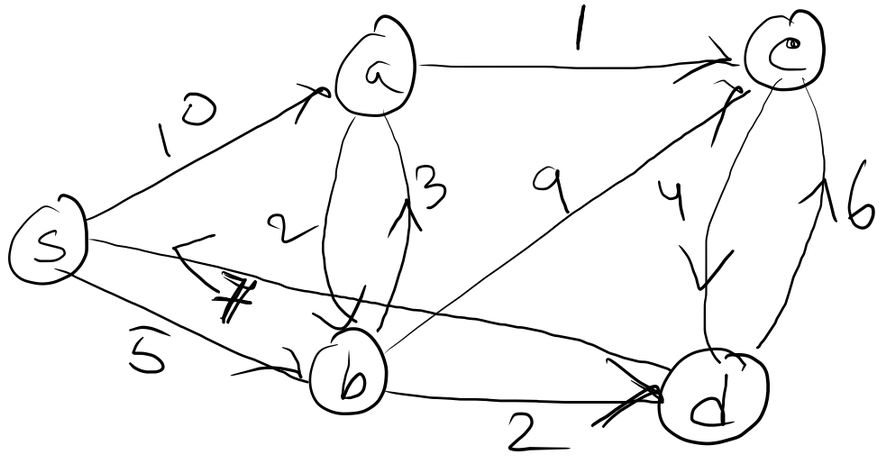
$d[v] \leftarrow$ shortest path distance from s to v
$\pi[v] \leftarrow$ parent of v.

Dijkstra's algorithm

```
Dijkstra (G, w, s)
d[s] = 0
for each vertex v ∈ V \{s}
        d[v] ← ∞ , π[v] ← NIL
S ← ϕ (empty set)
Q ← V   // Q ← a priority queue
While Q ≠ ϕ
    u = Extract-min (Q)
    S = S ∪ {u}
    for each vertex v ∈ Adj[u]
        if d[v] > d[u] + w(u,v)
            d[v] = d[u] + w(u,v)
            π[v] ← u
```

Running time:

$O(|V|) \cdot T_{\text{extract-min}} + O(|E|) T_{\text{decrease-key}}$



Q	d	Π
s	0	NIL
a	∞	NIL
b	∞	NIL
c	∞	NIL
d	∞	NIL

S = { }

Q	d	Π
s	0	NIL
a	10	s
b	5	s
c	∞	NIL
d	∞	NIL

X

s ← Extract-min (Q)

S = {s}

$b \leftarrow$ Extract-min (Q) $\quad S = \{s, b\}$

	Q	d	$\overline{\Pi}$
X	~~s~~	0	~~NIL~~
	a	~~10~~ 8	~~b~~
X	b	5	~~s~~
	c	~~14~~	b
	d	~~7~~	b

Final table

v	d	π
s	0	NIL
a	8	b
b	5	s
c	9	a
d	7	b

Shortest paths:

s to a

weight is 8

s → b → a

Divide and conquer

Sorting algorithms

Sorting: Given a sequence of numbers

$$a_1, a_2, \ldots, a_n$$

output: Return a permutation of the numbers such that

$$a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_n}$$

Ex^m **Input**: 2 1 6 3 9

output: 1 2 3 6 9

Merge Sort

High-level idea

- Divide the array(A) into roughly two equal partitions - L and R
- sort the array L
- sort the array R
- merge the two sorted arrays L & R to get the sorted array A.

Mergesort (A, p, r) — T(n)

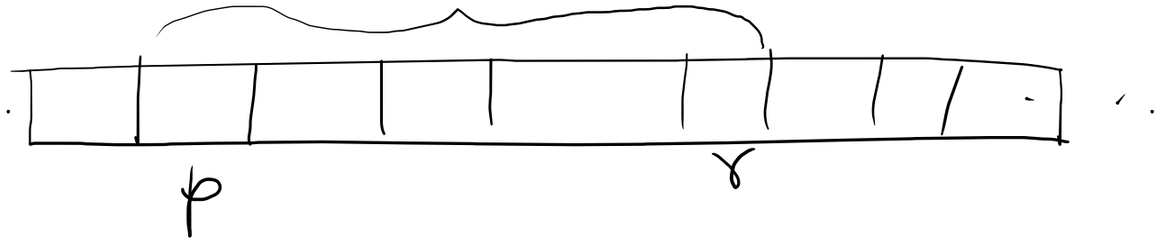
if p < r — Θ(1)

$q = \frac{p+r}{2}$ — Θ(1)

mergesort (A, p, q) — $T(\frac{n}{2})$

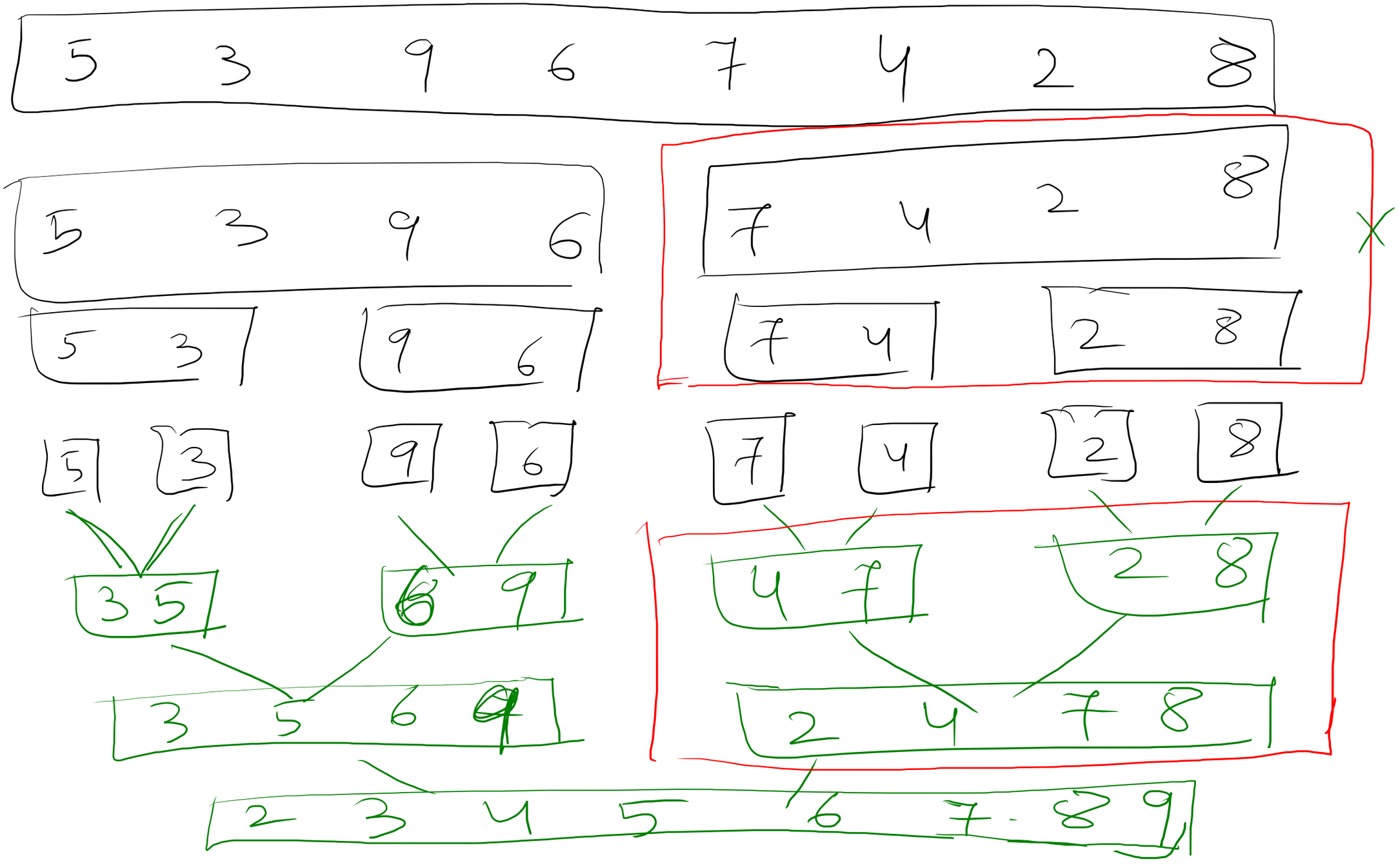
mergesort (A, q+1, r) — $T(\frac{n}{2})$

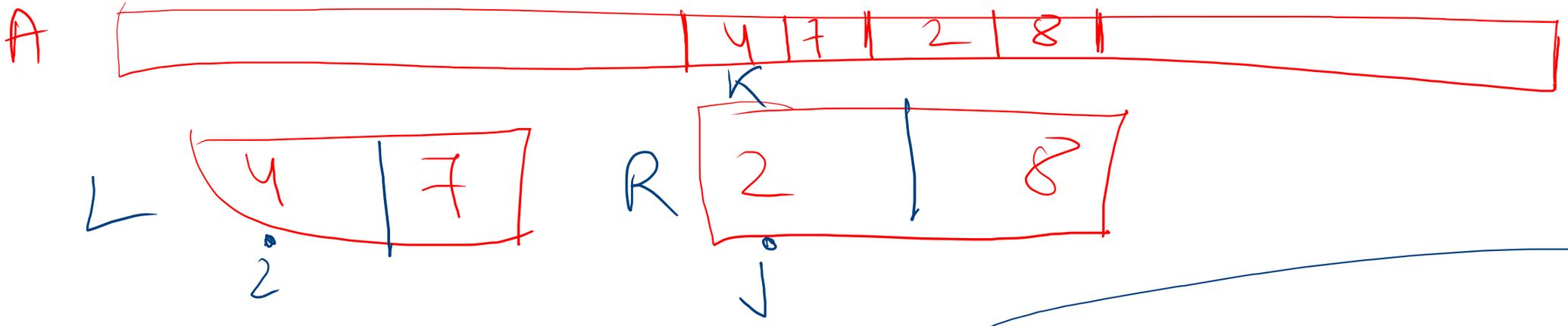
merge (A, p, q, r) — Θ(n)



Total time: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

$\Rightarrow T(n) = \Theta(n \log n)$


5 3 9 6 7 4 2 8
5 3 9 6
7 4 2 8
X
5 3
9 6
7 4
2 8
5
3
9
6
7
4
2
8
3 5
6 9
4 7
2 8
3 5 6 9
2 4 7 8
2 3 4 5 6 7 8 9



if $L[i] \leq R[j]$

$A[k] \leftarrow L[i]$

$k = k+1$

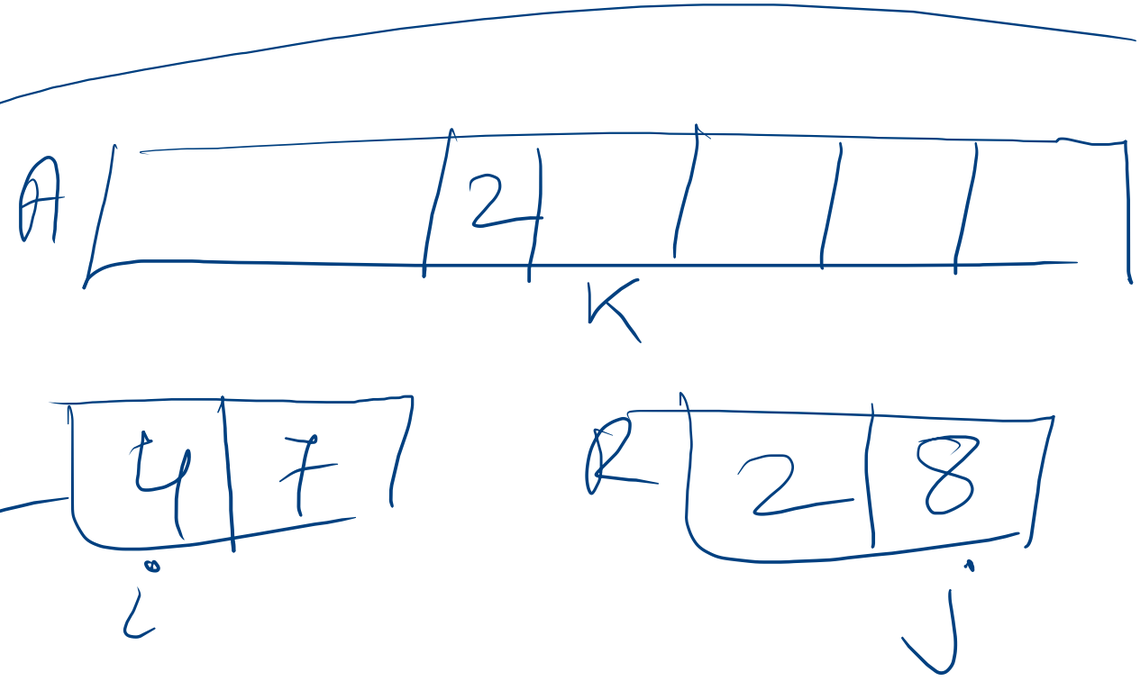
$i = i+1$

else

$A[k] \leftarrow R[j]$

$k = k+1$

$j = j+1$



```
merge(A, p, q, r)
  n1 ← q - p + 1
  n2 ← r - q
  for i = 1 to n1
      L[i] ← A[p + i - 1]
  for j = 1 to n2
      R[j] ← A[q + j]
  L[n1 + 1] ← ∞
  R[n2 + 1] ← ∞
⟹ for k = p to r
      if L[i] ≤ R[j]
          A[k] ← L[i]
          i = i + 1
      else A[k] ← R[j]
          j = j + 1
```

running time :- $O(n)$

At the start of each iteration of the for loop.

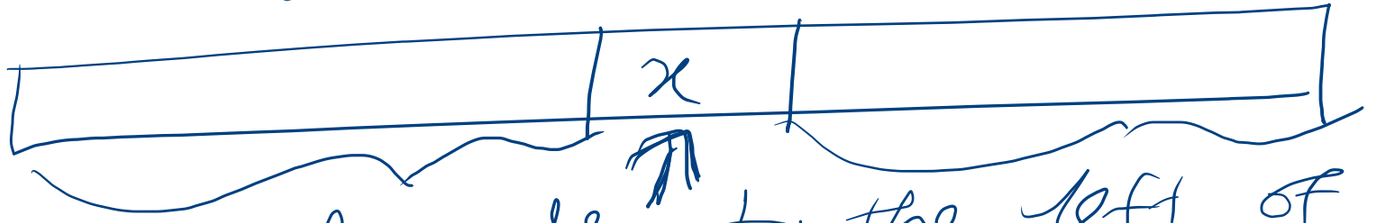
subarray $A[p \ldots k-1]$ contains the $k-p$ sorted elements of L and R

and $L[i]$ and $R[j]$ are the smallest elements of L and L that are yet to copied in A.

Quicksort

highlevel idea

- we choose a suitable element x
- based on the element x we partition the array into two parts:



- all the elements to the left of x are $\leq x$
- all the elements to the right of x are $> x$

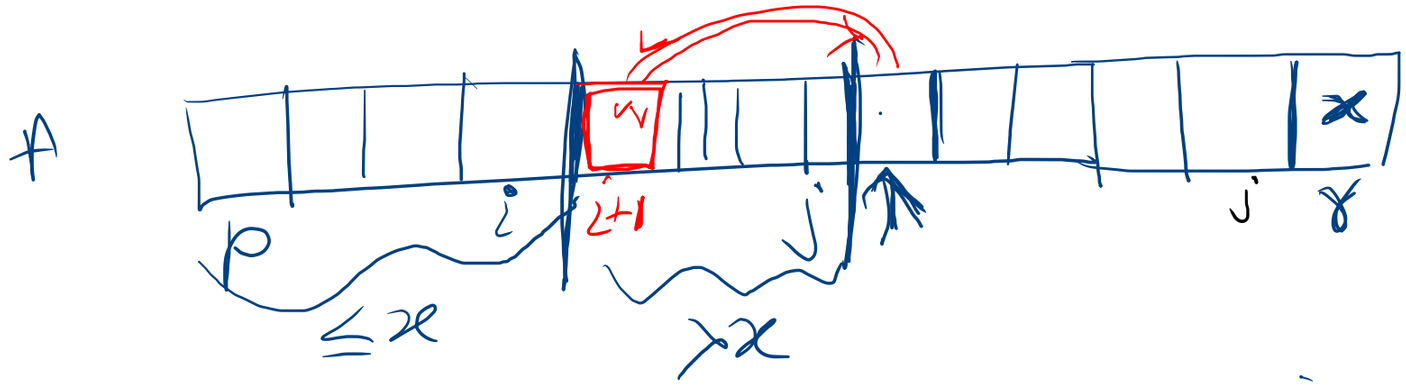
quicksort (A, p, r) ——— T(n)

if p < r ——— θ(1)

q = partition (A, p, r) ——— Θ(n)

quicksort (A, p, q-1) ———

quicksort (A, q+1, r) ———

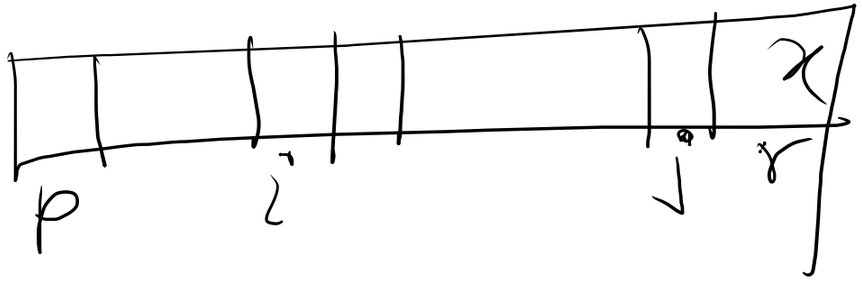


$A[j+1]$ compares with x.

either $A[j+1] \leq x$

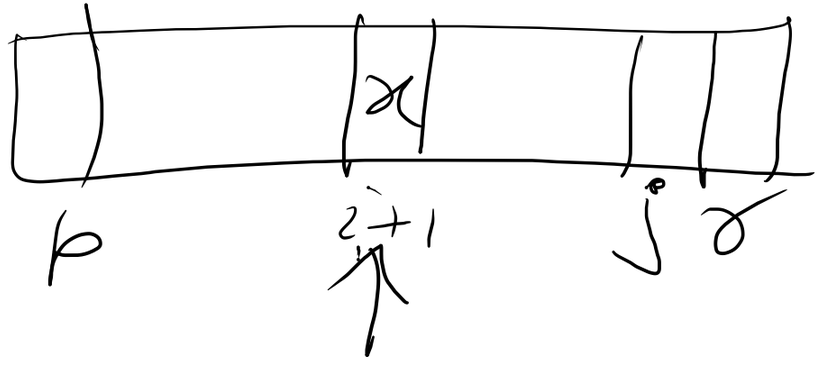
$i = i+1$

swap $A[j+1] \leftrightarrow A[i]$

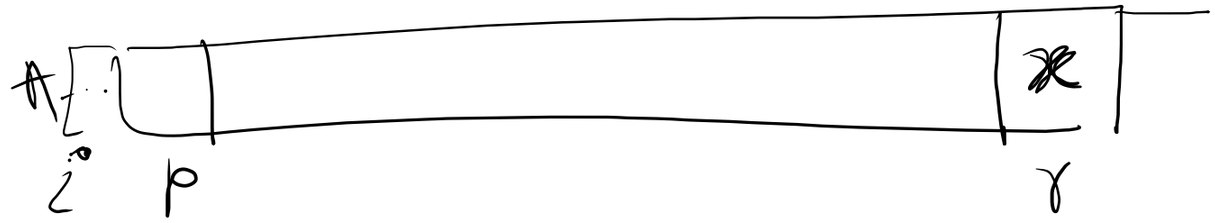


$A[j+1] > x$

$j = j+1$



```
Partition(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
      if A[j] ≤ x
          i = i + 1
          swap A[i] ⟷ A[j]
  swap A[i+1] ⟷ A[r]
  return i+1
```

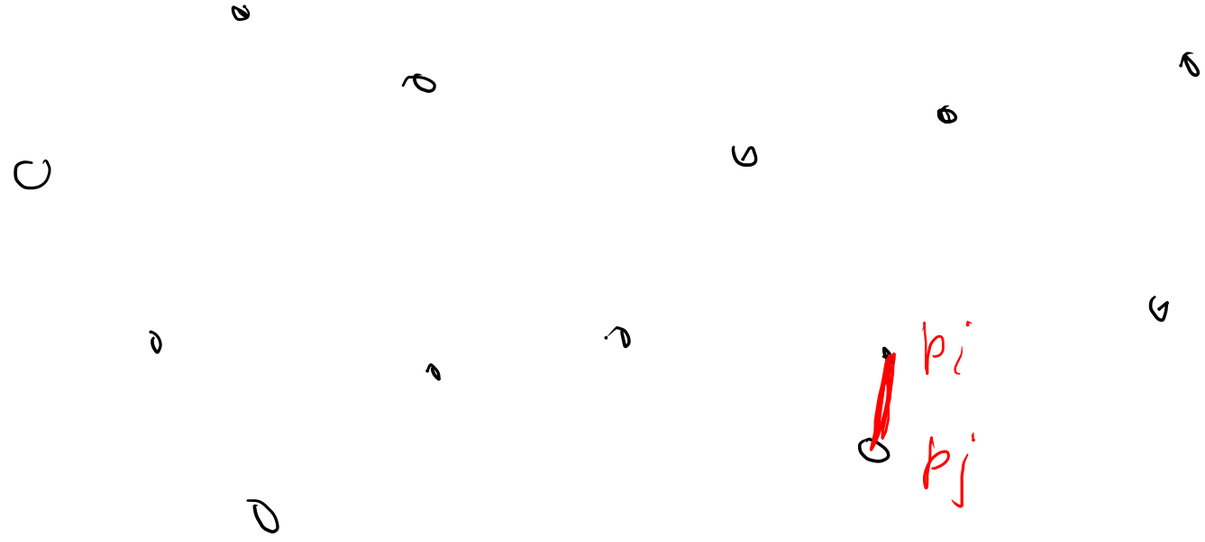


All the elements are same.

closest pair problem

Problem: Input: A set of n points $P = \{p_1, p_2, \ldots, p_n\}$

output: Find a pair (p_i, p_j) such that their distance is minimum.



A naive algorithm

For each pair compute the distance
return the pair with minimum distance.

running time $O(n^2)$

can we do better??

1D-version

points are on a line.

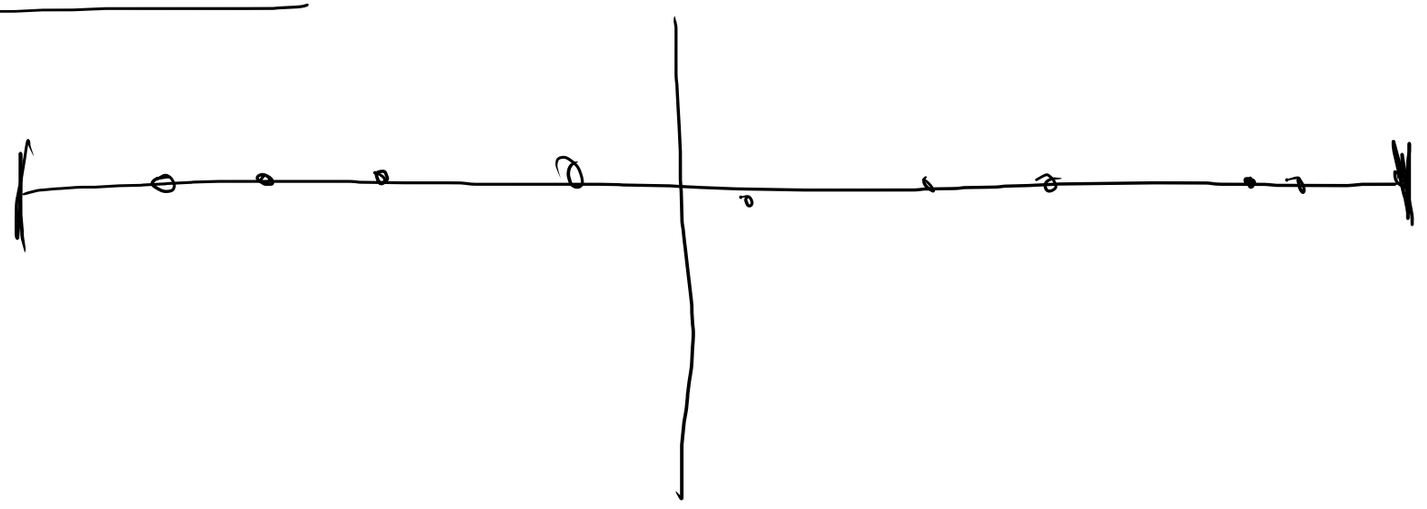
Sort the points.

take distance between consecutive pair.

running time: $O(n \log n) + O(n)$

Applying D & C on 1D version

1 Sort the points



closestpair (P)

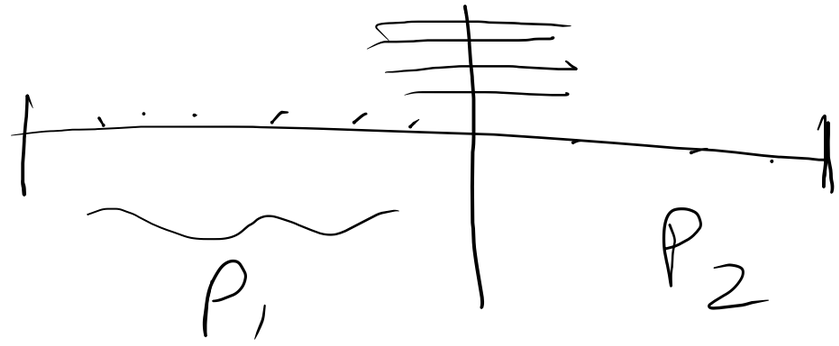
If $|P| = 1$ return $\delta_F = \infty$

If $|P| = 2$ return $\delta_F = |p_2 - p_1|$

otherwise

L = median (P)

Divide P in P_1 and P_2 w.r.t. L



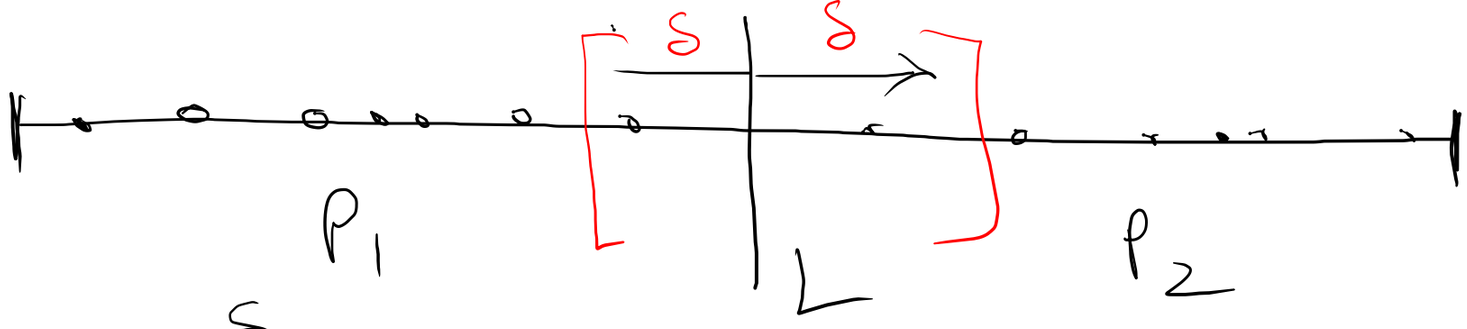
δ_L = closestpair (P_1)

δ_R = closestpair (P_2)

δ_{12} = minimum distance crossing L —— f(n)

return $\delta_F = \min\{\delta_L, \delta_R, \delta_{12}\}$

$T(n) = 2T(n/2) + f(n)$ where $f(n)$



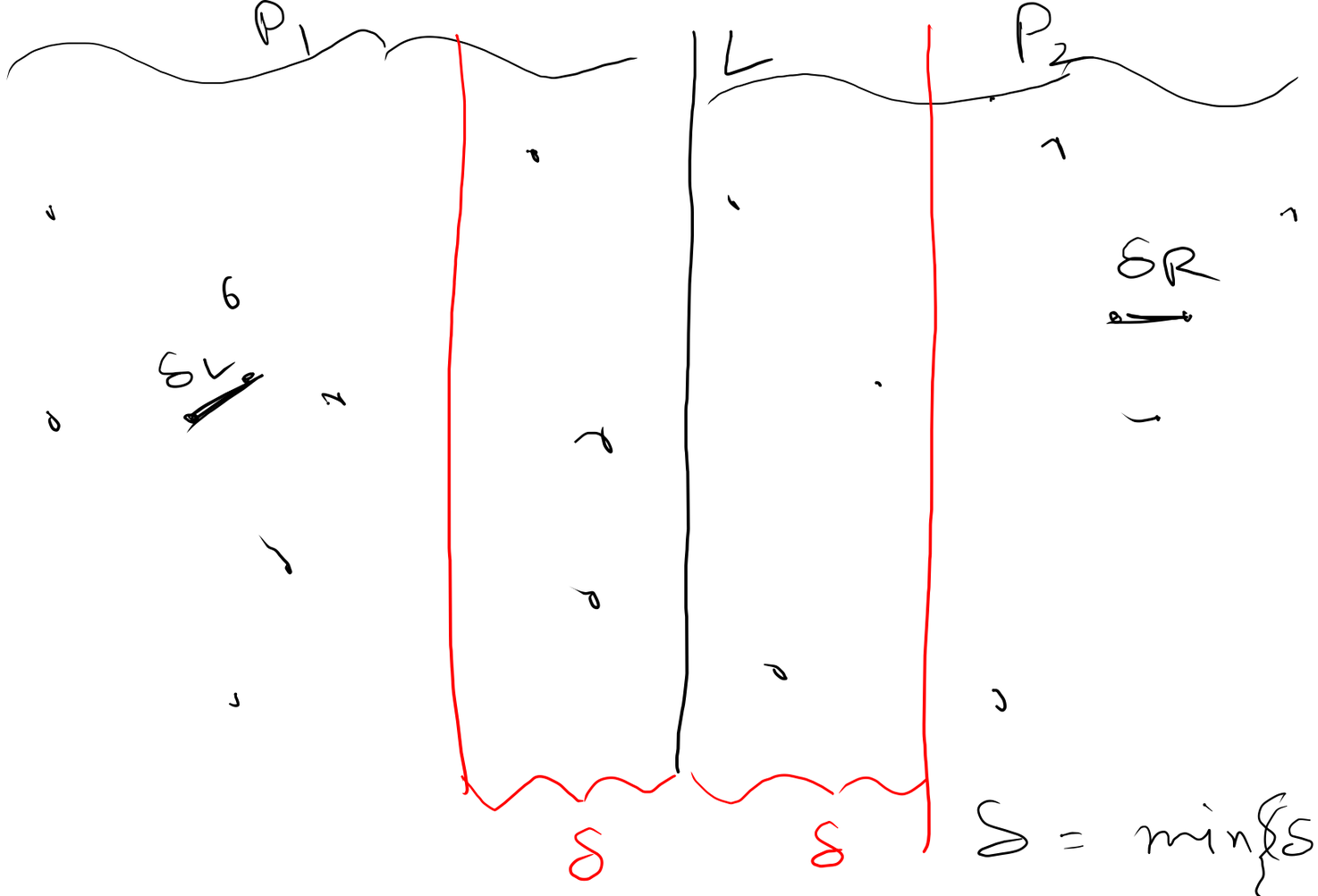
$\delta = \min\{\delta_L, \delta_R\}$

$$T(n) = 2T(n/2) + \theta(n)$$

$$= O(n \log n)$$

2D - version

closest pair 2D (θ)



$\delta_L =$

$\delta_R =$

for each p in P_1 and for each q in P_2

compute their distances

δ_{12} = minimum of them

$\delta_f = \{\delta_L, \delta_R, \delta_{12}\}$

$\delta = \min\{\delta_L, \delta_R\}$

$T(n) = 2\,T(n/2) + O(n^2)$

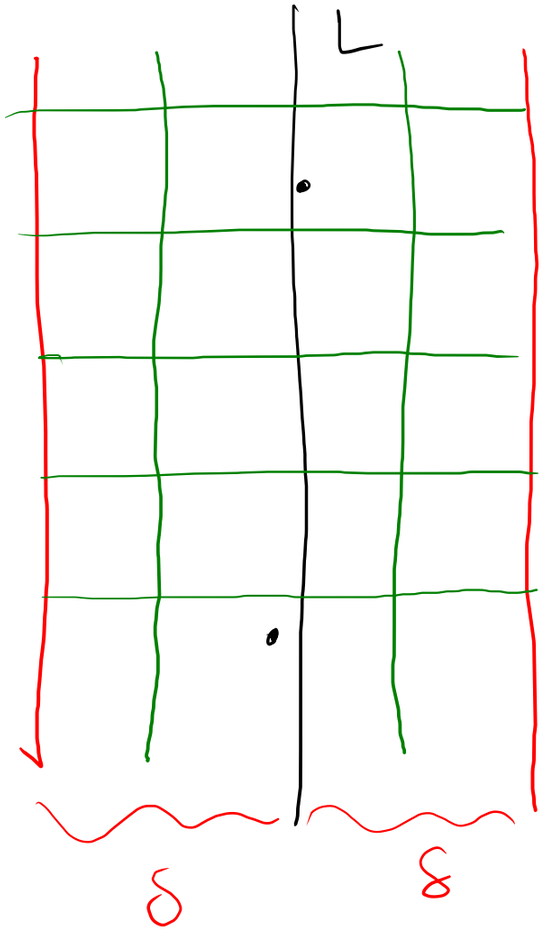
$= O(n^2)$

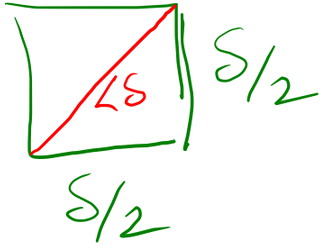
no improvement

Question:

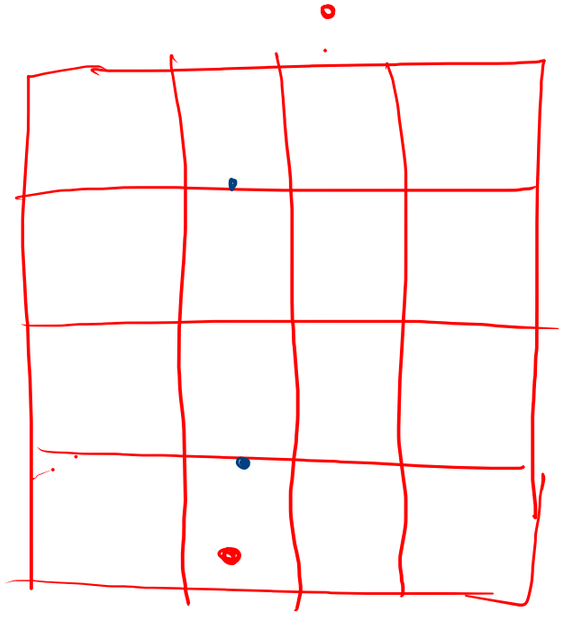
How many points are there in a single box?

Ans: At most 1





$< \delta$



4x4

The final algorithm

P_x: sorted in x order
P_y: " " y order

closestpair2D (P)
construct P_x and P_y — $O(n \log n)$
(p_0^*, p_1^*) = closest-pair-rec (P_x, P_y) — $O(n \log n)$

closest-pair-rec (P_x, P_y) — $T(n)$
= base condition. = $\Theta(1)$

Q: first $n/2$ points in P_x
$R \leftarrow$ remaining points in P_x

Q_x: Q sorted in x direction
Q_y: Q " " y "
R_x: R " " x direction
R_y: R " " y "

construct Q_x, Q_y, R_x, R_y — $O(n)$
(q_0^*, q_1^*) — closestpairrec (Q_x, Q_y) — $T(n/2)$
(r_0^*, r_1^*) — " " " (R_x, R_y) — $T(n/2)$
$\delta = \min \{ d(q_0^*, q_1^*), d(r_0^*, r_1^*) \}$ — $\Theta(1)$
$x^* =$ max x-coordinate of a point in Q — $\Theta(1)$
$L = \{(x, y) \mid x = x^*\}$
$S =$ points in P within δ distance of L — $O(n)$

construct S_y — $O(n)$

for each point $s \in S_y$

compute distances from s to each of the next 15 points in S_y — $O(n)$

let δ_{12} be minimum

$$\delta_{12} = d(q_2^*, r_2^*)$$

If $d(q_1^*, r_1^*) < \delta$ then

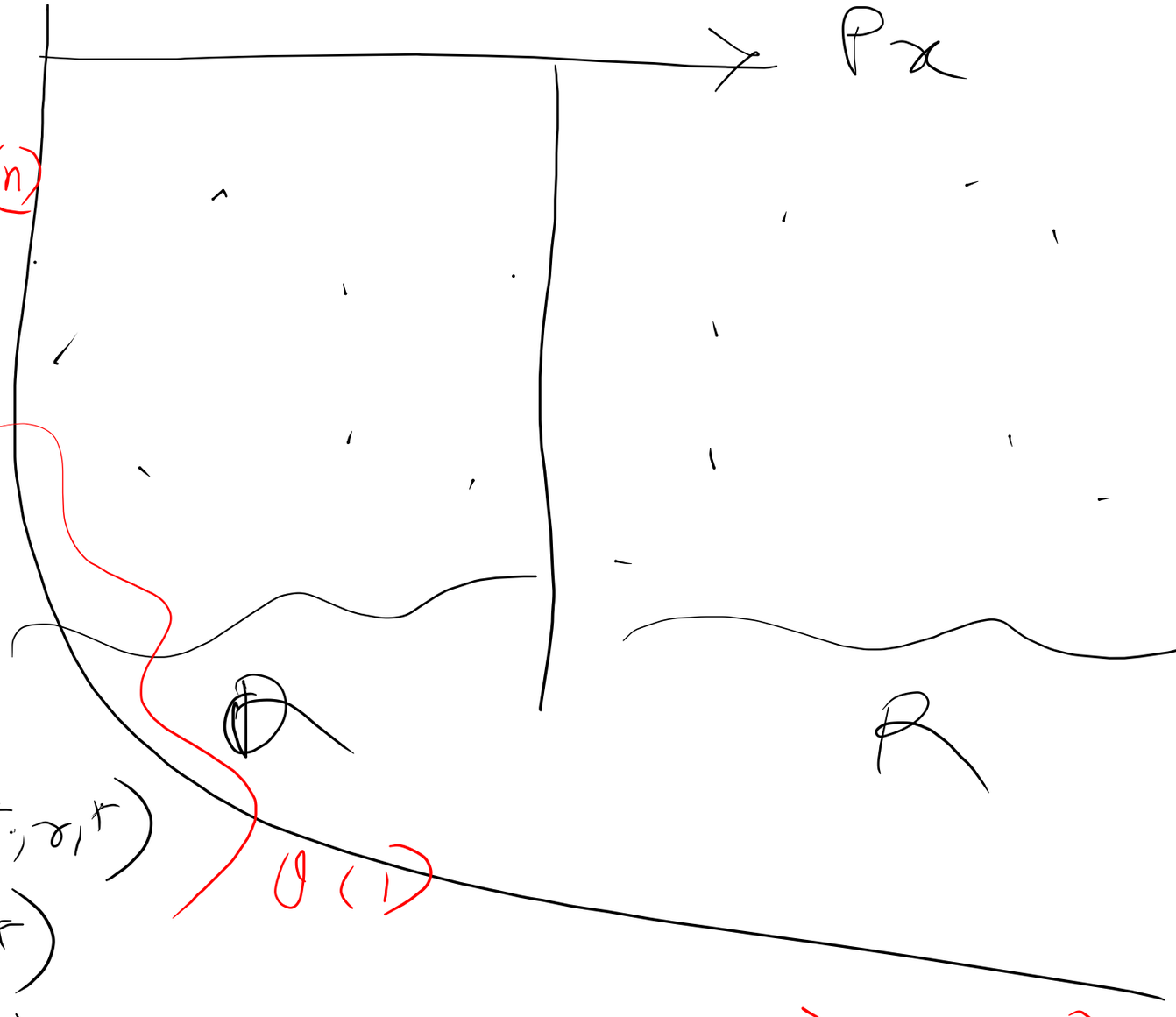
return (q_2^*, r_2^*)

else if $d(q_0^*, q_1^*) < d(r_0^*, r_1^*)$

return (q_0^*, q_1^*)

else

return (r_0^*, r_1^*)



$$T(n) = 2T(n/2) + O(n)$$

$$= O(n \log n)$$

Dynamic Programming

Divide and conquer:

- Divide the subproblem into independent subproblems.
- solve each subproblem independently
- combine the solutions.

Dynamic Programming

- Divide the subproblem into a series of overlapping subproblems.
- solve them ⟸ need extra care
- combine the solutions.

Dynamic Programming: It solves optimization problems.

Main idea

- compute the solutions to the subproblems once
- Store the solution of the subproblems in a table / dictionary
- They can be reused (repeatedly) in a later stage.

⟹ It trades space for time

Rod cutting problem

Input: A rod of length n unit.

A table of prices p_i for $i = 1, 2, \ldots, n$

where p_i is the price of a rod of length i

Goal: Find the maximum revenue

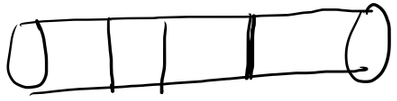
cut the rod into different pieces and sell it.

Ex^m

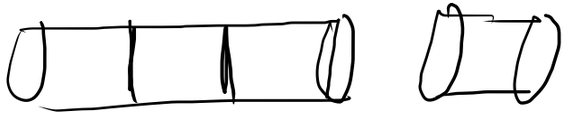
length i	1	2	3	4	5	6	. . . .
price p_i	1	5	8	9	10	17	

The shop owner has a rod of length 4

Solution



9

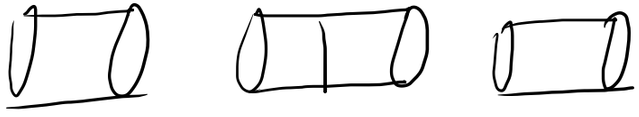


8 + 1

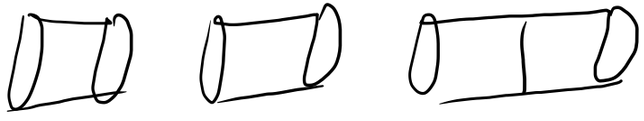
1 + 8

5 + 5

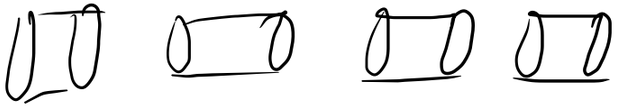
5 + 1 + 1



1 + 5 + 1

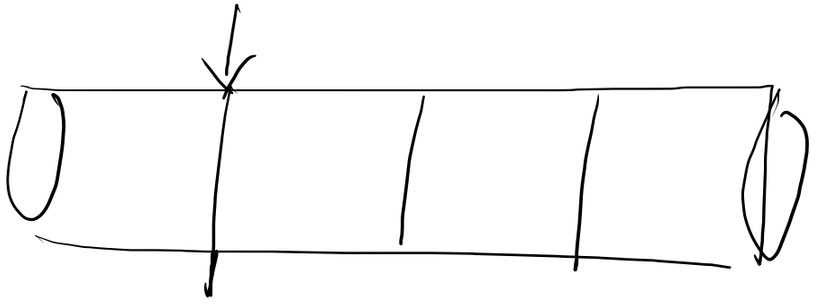


1 + 1 + 5



1 + 1 + 1 + 1

A simple algorithm



- Try all possible cuts of the rod
- Track the best solution.

running time :- $2^{n-1} \approx O(2^n)$

can we do better?

maximum revenue $= r_n$

$r_i \leftarrow$ maximum revenue for a rod of length i

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\}$$

In general:

- making an initial cut into two pieces of length i and $n-i$
- optimally cutting these two pieces
- we donot know ahead of time which initial cut gives the optimum revenue.
- we have to consider all possible value for i and pick that with the maximum revenue.

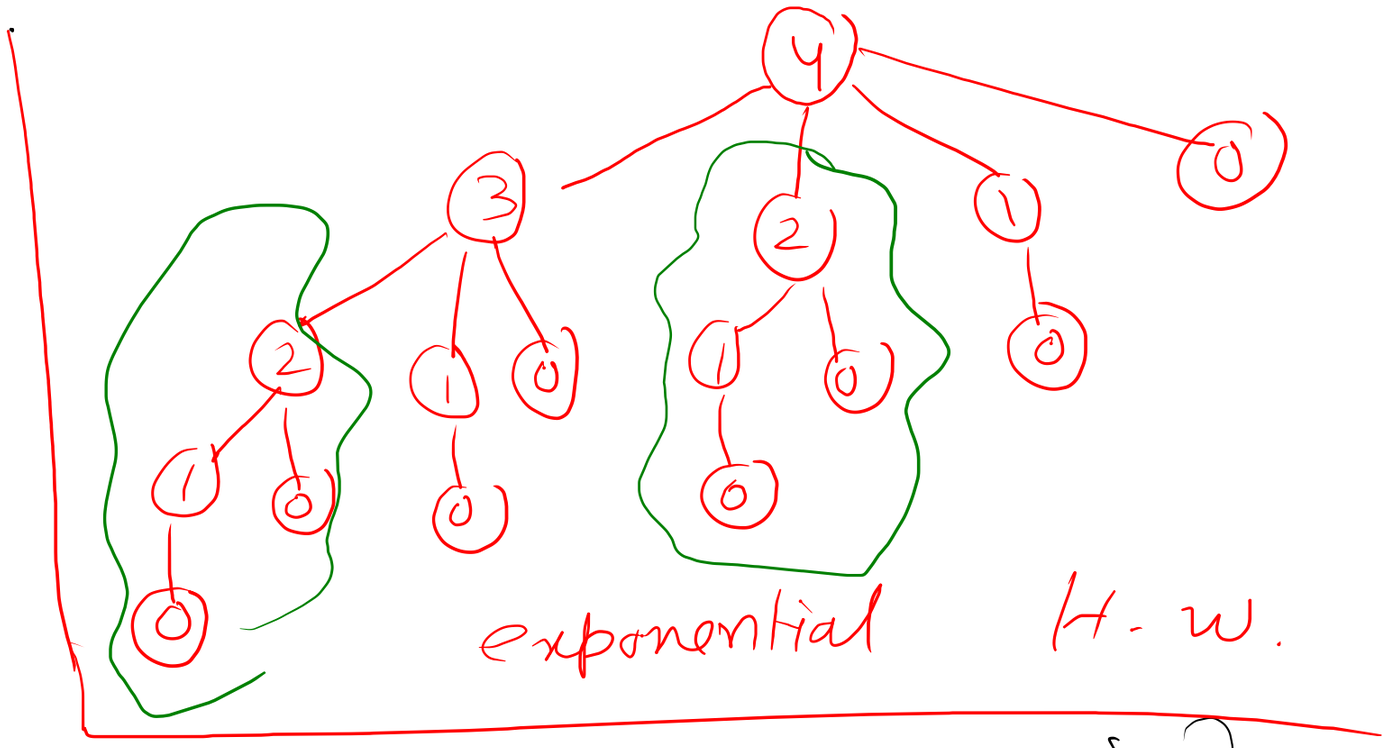
$$r_n = \max \{ p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1 \}$$

Question:
Do we really need two subproblems?

$$r_n = \max_{1 \le i \le n} \{ p_i + r_{n-i} \}$$

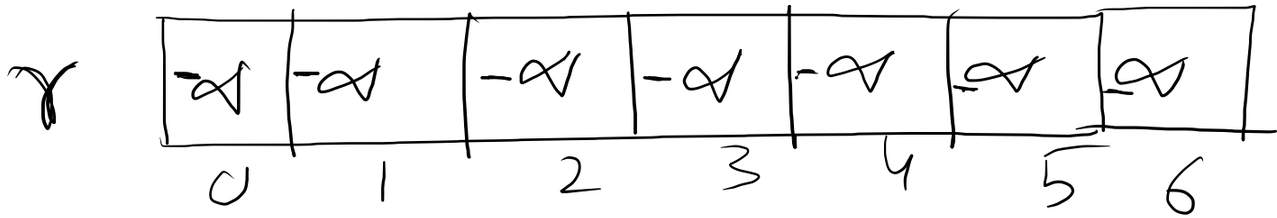
Algorith rod cutting

```
rod-cut (P, n)
  if  n == 0
      return 0

  q = -∞
  for i = 1 to n
      q = max { q, P[i] + rod-cut (P, n-i) }

  return q
```



Problem: several subproblems are called many times.

Top-down approach




```
memoized-rod-cut(P, n)
  let r[0,..,n] be a new array
  for i = 1 to n
      r[i] ← -∞
  return memoized_rod_cut_aux(P, n)
```

```
memoized-rod-cut-aux(P, n)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else
      q = -∞
      for i = 1 to n
          q = max{ q, P[i] + memoized-cut-rod-aux(P, n) }
  r[n] = q
  return q
```

running time

# of independent subproblems

× time taken without recursive call

Two ways to solve the problem

1. Top-down with memoization
2. Bottom up with tabulation

Rod cutting problem Bottom-up approach.

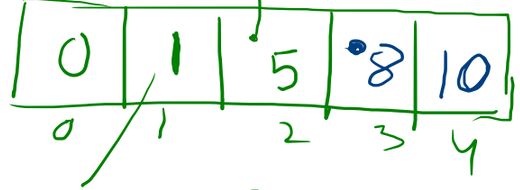
Idea: It solves the subproblems from 0 to n iteratively and store them in a table / dictionary / hash table.

```
Bottom_up_rod_cut(p, n)
let r[0,...,n] be a new array
let s[0,...,n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    q = max_{1≤i≤j} { q, p[i] + r[j-i] }
    s[j] = i // ← this is the i which gives the maximum q
    r[j] = q
return r[n]
```

Running time: $O(n^2)$ — two for loops

i	1	2	3	4
p_i	1	5	8	9

$n = 4$



$p[1] + r[0] = 1$

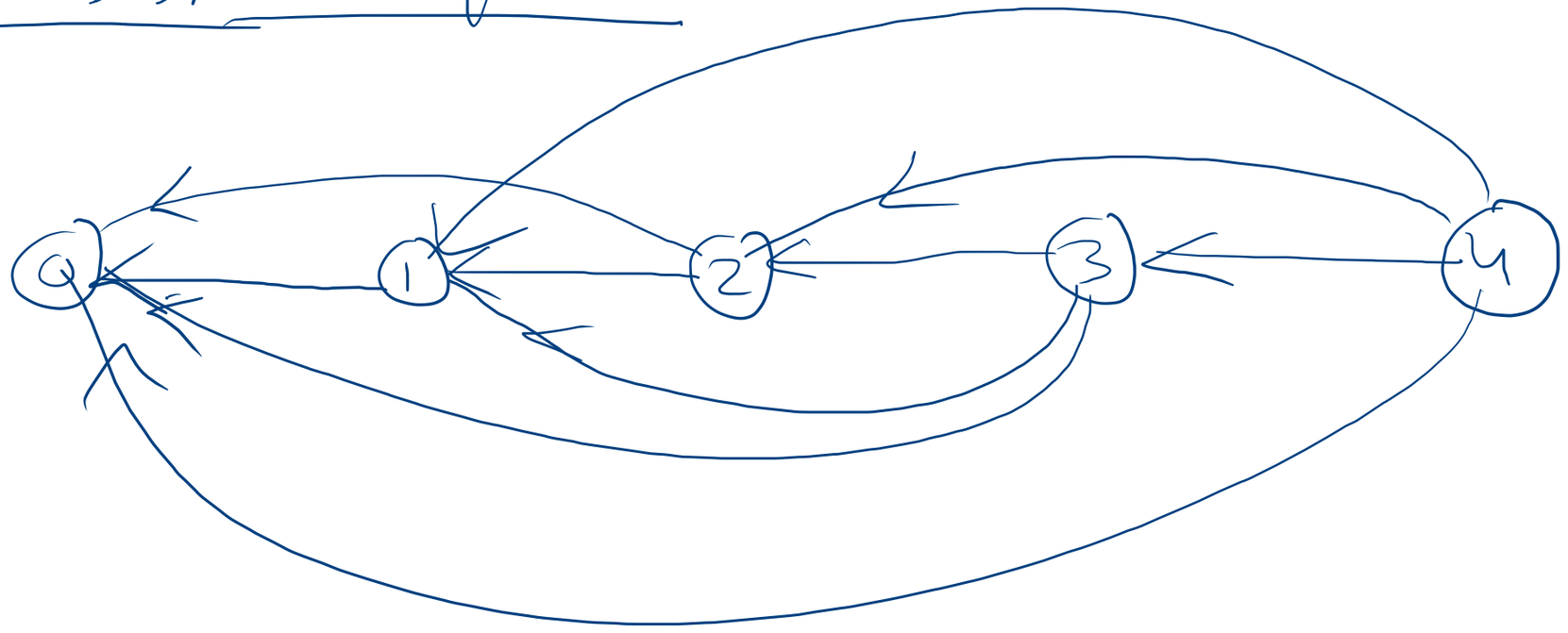
$j = 2$ for $i = 2$ max $\begin{cases} p[1] + r[1] = 2 \\ p[2] + r[0] = 5 \Leftarrow \end{cases}$

$j = 3$ for $i = 3$ $p[3] + r[0] = 8$

$j = 4$ for $i = 2$

s: | 0 | 1 | 2 | 3 | 2 |
0 1 2 3 4

The subproblem graph



construction of a solution

```
Print_rod_cut (P, n)
(r, s) = Bottom-up-rod-cut (P, n)
while n > 0
        Print s[n]
        n = n - s[n]
```

For $n = 4$,

Print gives the solution as 2, 2

General outline of a DP problem

Step 1 :– Structure

Characterize the structure of an optimum solution.
by showing that it can be decomposed into optimum subproblems.

Step 2 Recursive

Recursively define the value of an optimum solution by expressing it in terms of optimum solution for smaller subproblems.

Step 3: Optimum value computation

Two approaches

i) Top-down with memoization

ii) Bottom-up with tabulation.

Step 4: optimum solution computation

Step 4 only requires when one ask for finding an optimum solution.

Some time additional information is maintained during steps 1–3 to easily construct an optimum solution.

Fibonacci number

1, 2, 3, 5, 8, 13, 21, . . .

Step 1 If optimally compute previous 2 terms then you can optimally compute that number. by summing the previous two numbers.

Step 2.

$$F_n = \begin{cases} 1, & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 3 \end{cases}$$

```
fibonacci (n)
let F be an array.
if n = 1
    return 1
if n = 2
    return 2
for i = 3 to n
    F[n] = F[n-1] + F[n-2]
return F[n]
```

```
fib (n)
  if n = 1
      return 1
  if n = 2
      return 2
  for i = 3 to n
      return fib(n-1) + fib(n-2)
```

Logest common subsequence (LCS)

Subsequence:

A B B C B A B

then B C B B is a subsequence

- A sequence whose order is same as in the given sequence.
- They may not be consecutive.

Common subsequence:

X = B A A B C B A B C

Y = A B B C B A C

A B B B C

A B C B A C

longest.

A sequence that is subsequence to both x and y.

ABC is a common subsequence.

Logest: A common subsequence whose length is largest possible.

A simple algorithm

X, Y $\quad |Y| < |X| \quad |Y| = n$, $|X| = m$

consider all possible subsequences of Y

compare with X whether any of the subsequence is common to X

return the largest one.

running time : $O(m \cdot 2^n)$

$X = x_1 x_2 \cdots x_m$

$Y = y_1 y_2 \cdots y_n$

$X_i = x_1 x_2 \cdots x_i$

$Y_j = y_1 y_1 \cdots y_j$

Z be a LCS of X and Y

$Z_K = z_1 z_2 \cdots z_K$

$x_m = y_n$

z_K is a member of LCS of X and Y

Z_{K-1} is a LCS of X_{m-1} and Y_{n-1}

$x_m \neq y_n$

$z_K \neq x_m$

Z_K is a LCS of X_{m-1} and Y_n

$z_K \neq y_n$

Z_K is a LCS of X_m and Y_{n-1}

Recursive definition

$c[i\ j] \leftarrow$ optimum length of an LCS for X_i and Y_j

$$c[i,j] = \begin{cases} 0 & \text{when } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Compute the value of the optimum solution

```
LCS-length (X, Y)
m ← length(X) , n ← length(Y)
let C[0..m, 0...n] be a new table
let B[0..m, 0..n] be a new table.
for i = 1 to m
        C[i,0] = 0
for j = 1 to n
        C[0,j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i = y_j
            C[i,j] = C[i-1,j-1] + 1
            b[i,j] = "↖"
        else if C[i-1, j] ≥ C[i, j-1]
            C[i,j] = C[i-1, j]
            b[i,j] = "↑"
        else
            C[i, j] = C[i, j-1]
            b[i, j] = "←"
return C, b
```

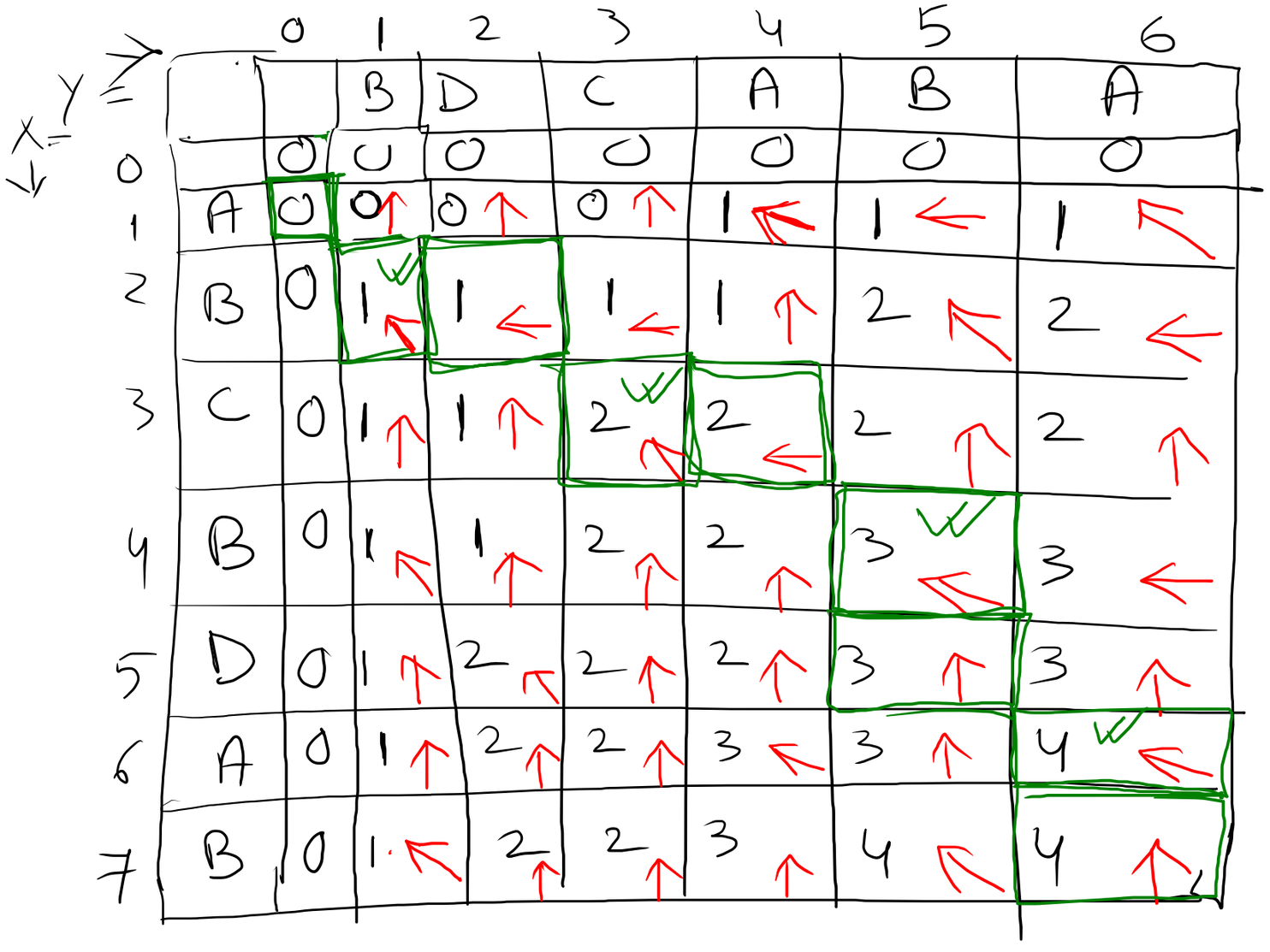
Running time:

$O(mn)$

Ex^m

X = A B C B D A B

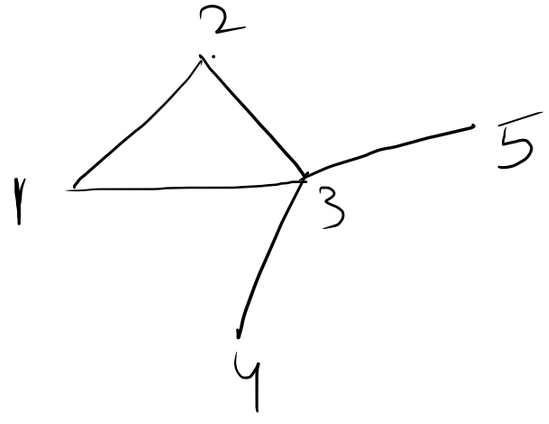
Y = B D C A B A

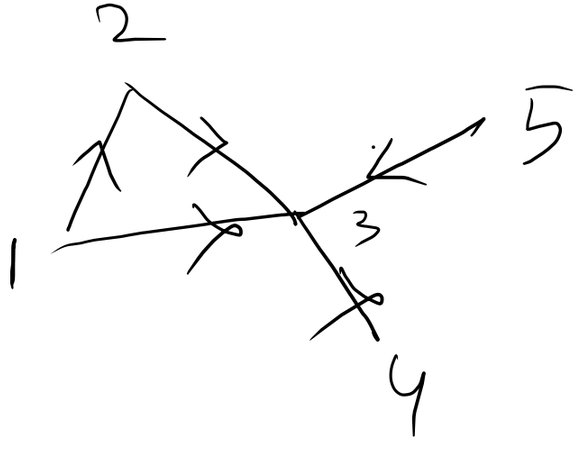


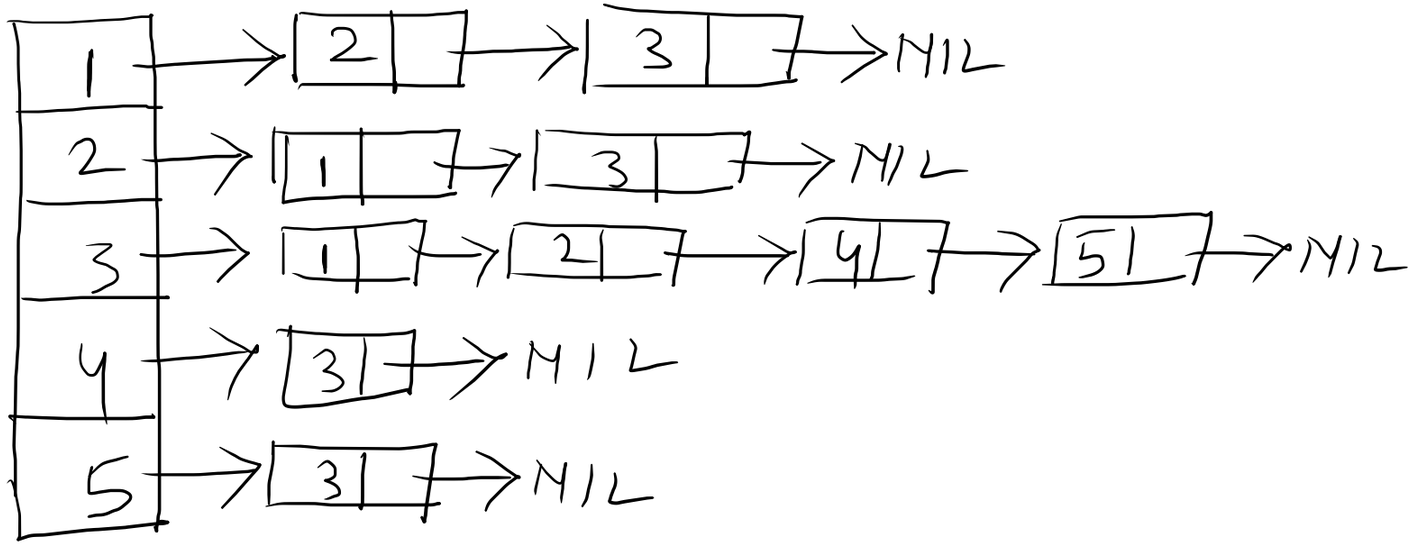
BCBA

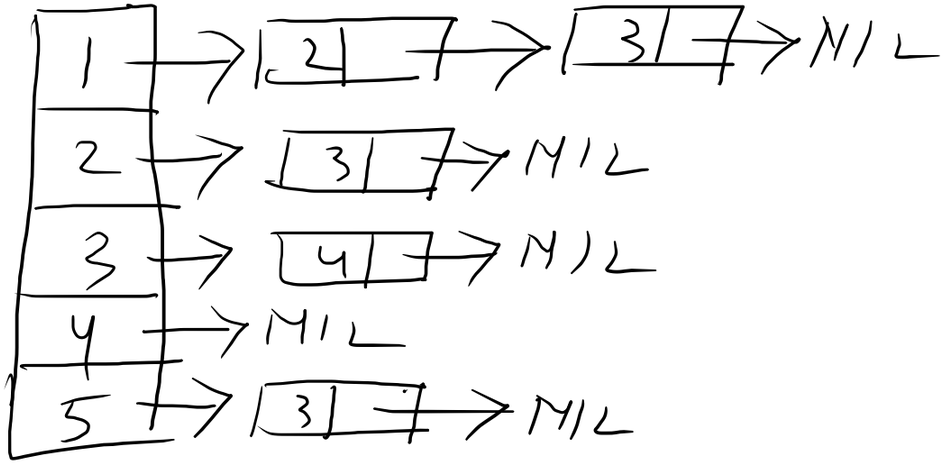
Representations of graphs

Adjacency - List









Storage

$O(|V| + |E|)$

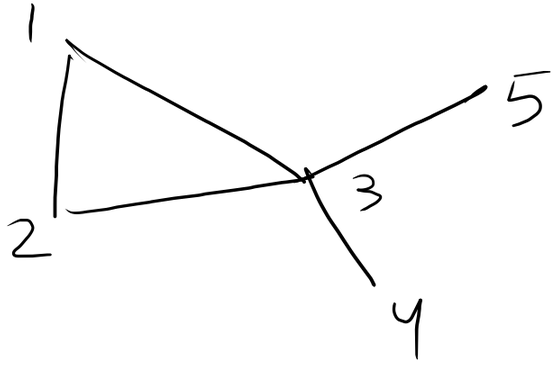
Adjacency matrix

storage: $O(|V|^2)$

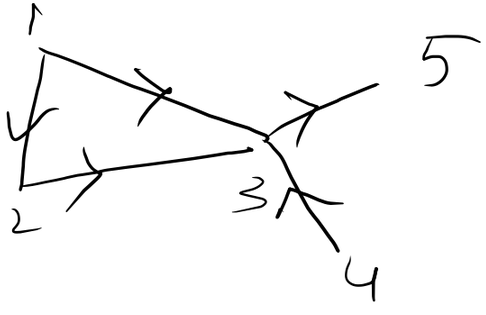
$v_1, v_2, \ldots, v_n$.

$A_{n \times n}$

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is connected with } v_j \\ 0 & \end{cases}$$



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	0	0
3	1	1	0	1	1
4	0	0	1	0	0
5	0	0	1	0	0

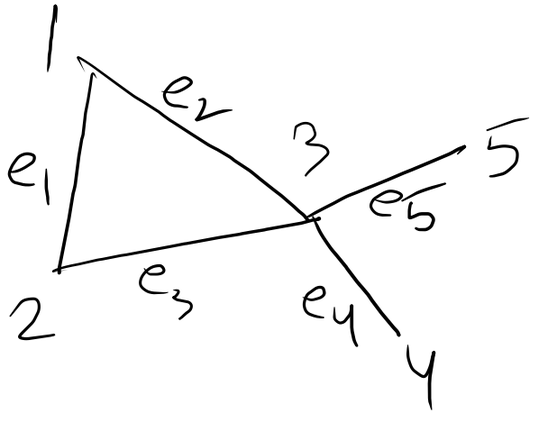


	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	1	0	0
5	0	0	0	0	0

Incidence matrix

$A_{n \times m}$

$$a_{ij} = \begin{cases} 1 & e_i \text{ is incident on } j\text{-th vertex} \\ 0 & \text{otherwise} \end{cases}$$



	e_1	e_2	e_3	e_4	e_5
1	1	1	0	0	0
2	1	0	1	0	0
3	0	1	1	1	1
4	0	0	0	1	0
5	0	0	0	0	1

storage:

$O(|V||E|)$

Graph searching algorithms

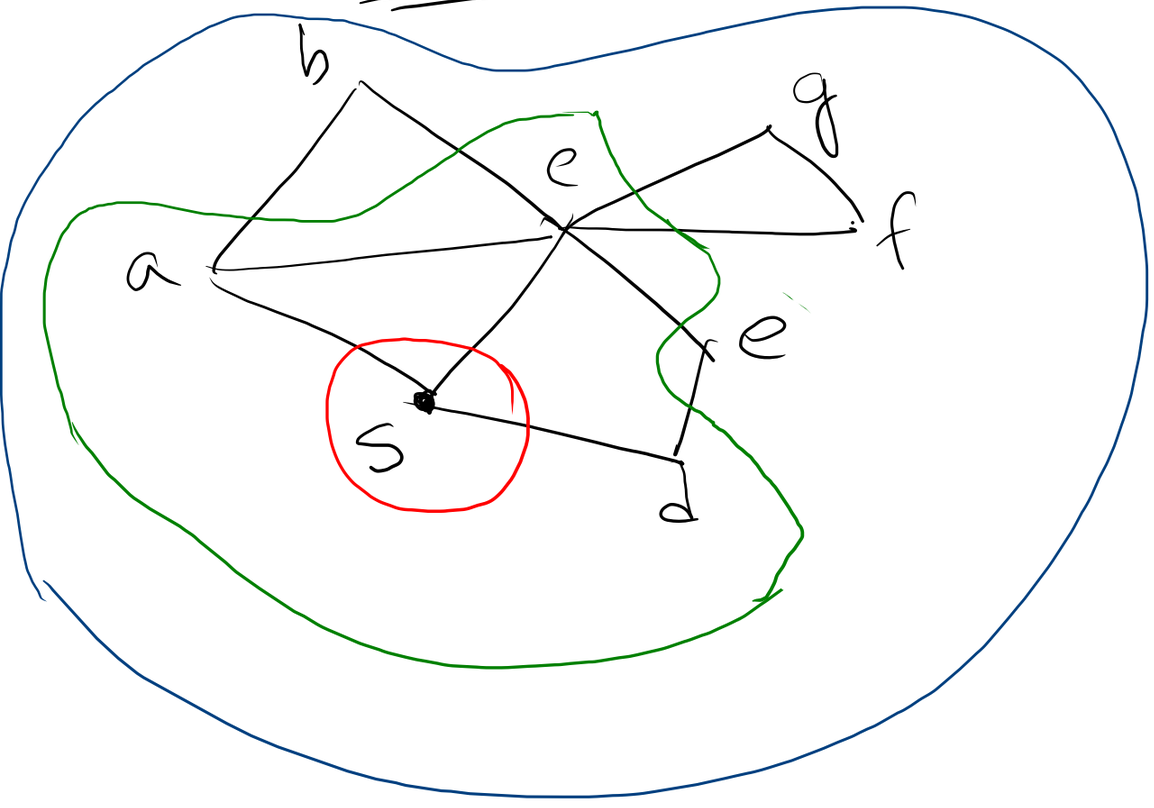
- systematic searching of each edge or vertices of the graph.
- For directed or undirected graph.
- Applications.
  - maze search.
  - connected compond.

Two popular algorithms.

i) BFS

ii) DFS

BFS



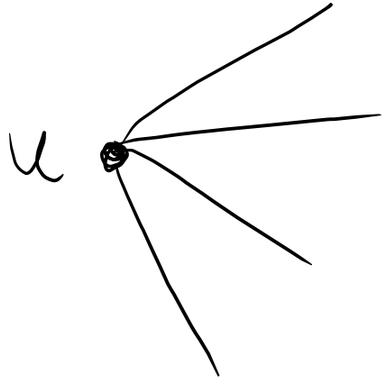
White: This vertex is not yet discovered.

Gray: It is discovered. but not all its neighbours are discovered.

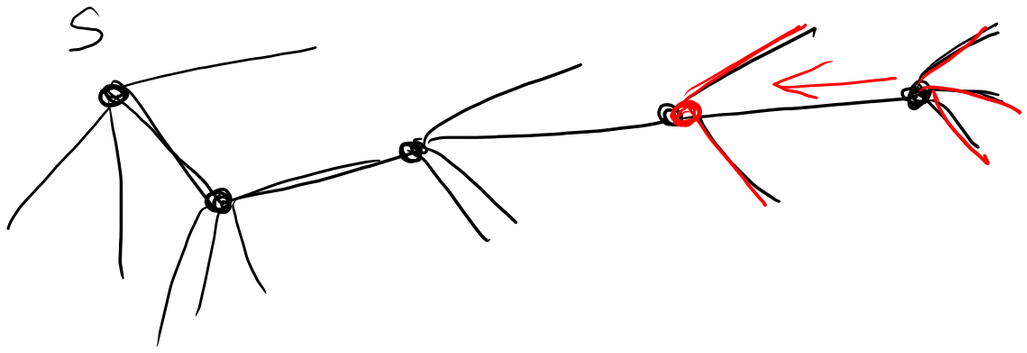
Black: Already discovered and all its neighbours are discovered.

```
BFS (G, s)
for each u ∈ G.V
        u.color = white
        u.dist  = ∞
        u.pred  = NIL
s.color = gray
s.dist = 0
Q = new queue
Q.enqueue(s)
while Q is not empty.
        u = Q.dequeue()
        for v ∈ u.adj
                if v.color = white
                        v.color = gray
                        v.dist = u.dist + 1
                        v.pred = u
                        Q.enqueue(v)
        u.color = black.
```

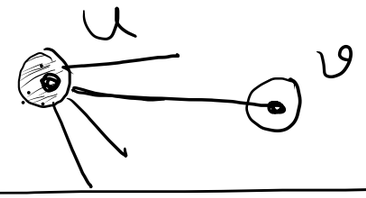
Running time: $O(|V| + |E|)$

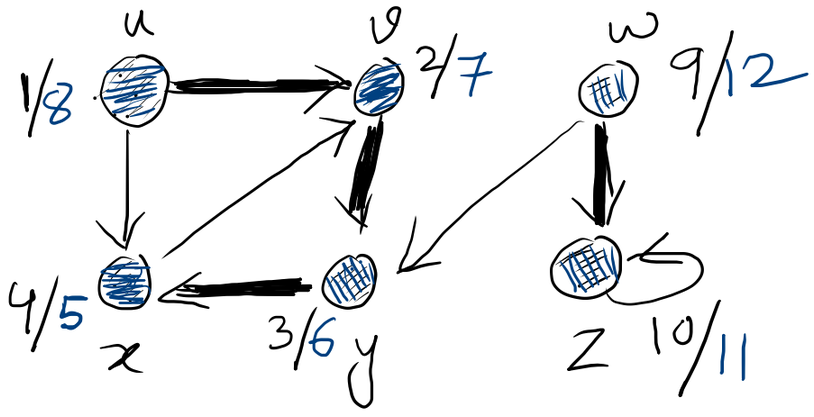


Depth First Search (DFS)



```
DFS(G)
  for each vertex u ∈ G[V]
        u.color = white
        u.Pred = NIL
  time = 0
  for each vertex u ∈ G[V]
        if u.color = white
              DFS-visit(u)

DFS-visit(u)
  u.color = gray
  time = time + 1
  u.starttime = time
  for v ∈ Adj[u]
        if v.color = white
              v.pred = u
              DFS-visit(v)

  u.color = black
  time = time + 1
  u.endtime = time
```





Total time: $O(|V| + |E|)$

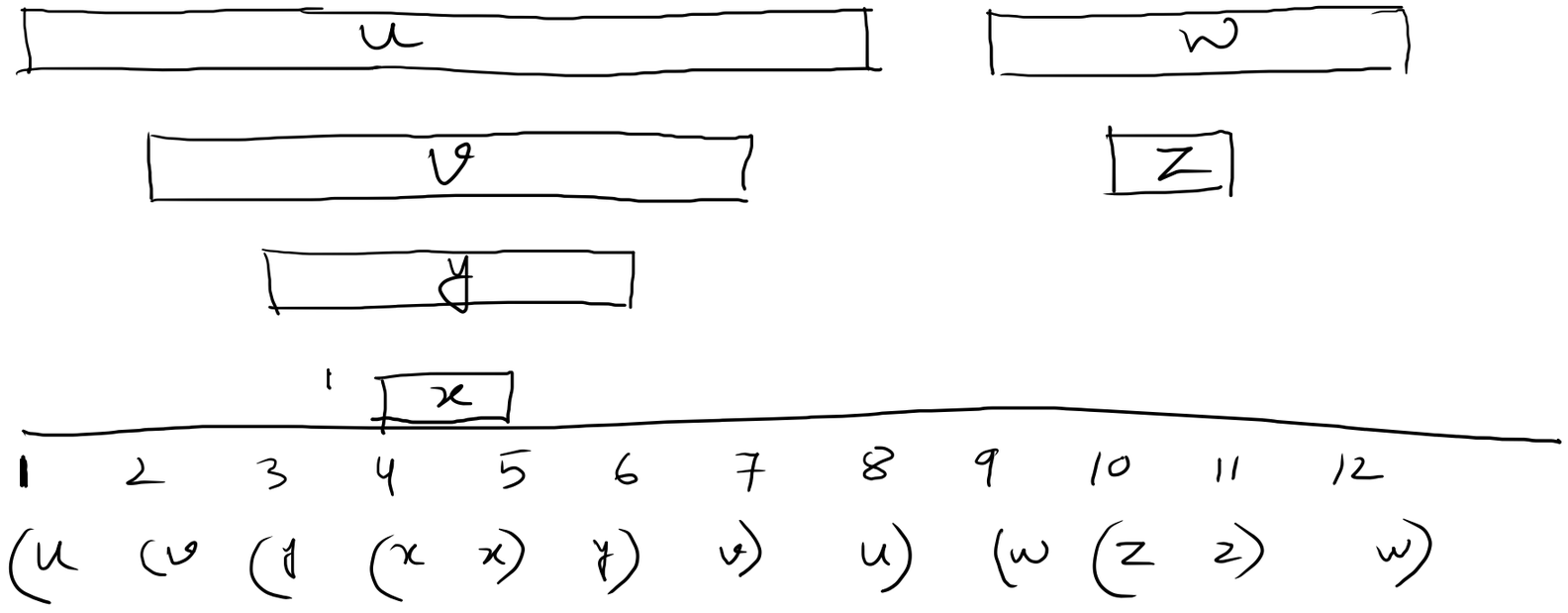
Properties

Parenthesis theorem

(( )) X

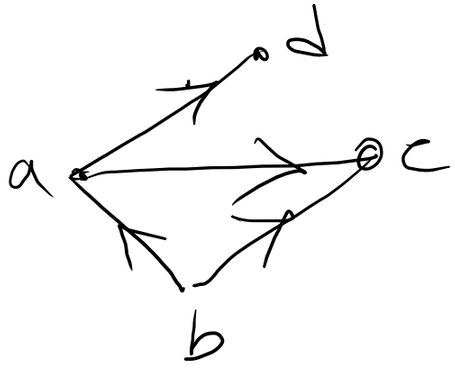
( ) ( ) (( ))

In previous example

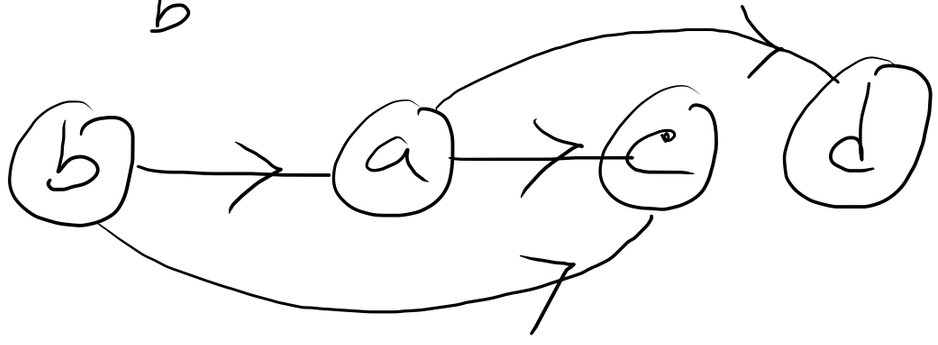


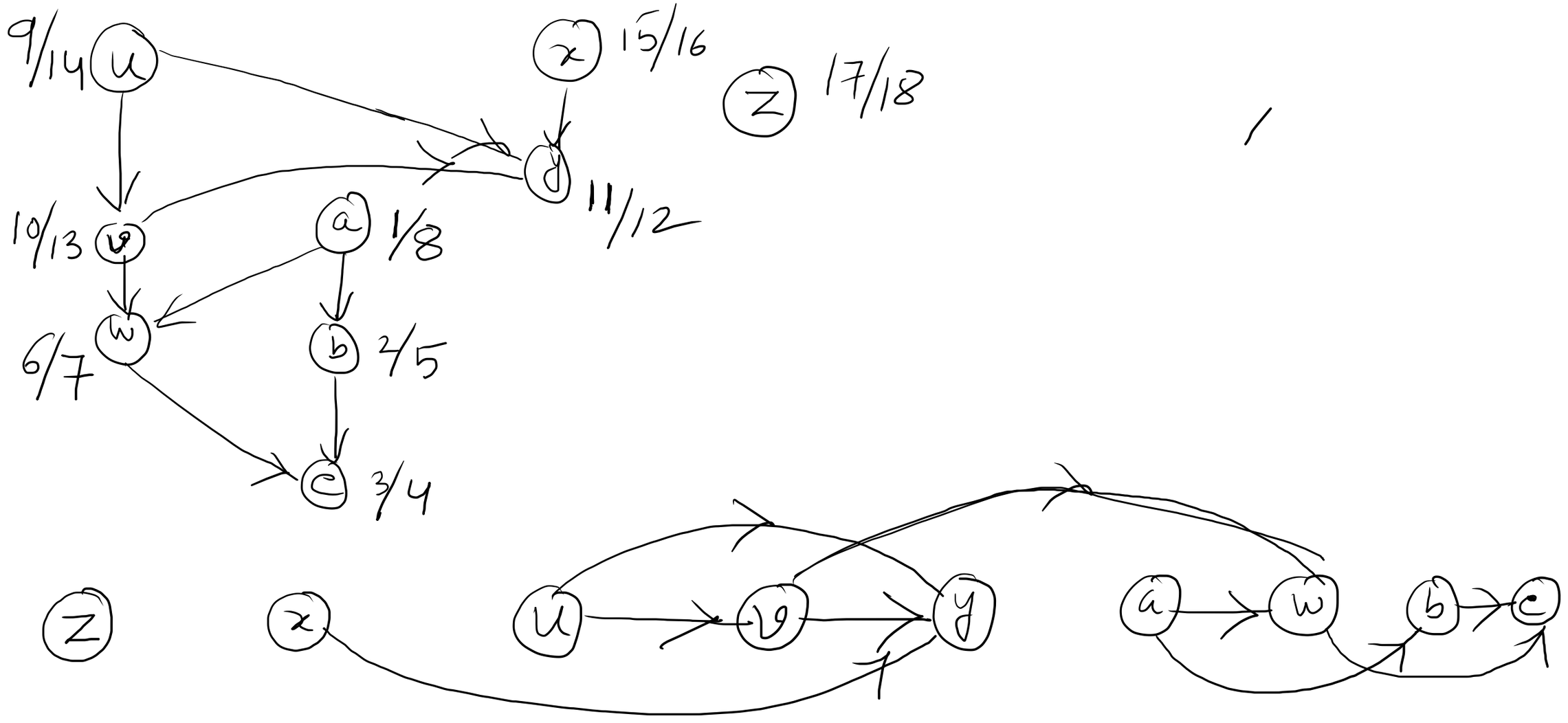
Topological sorting for DAG Directed Acyclic Graph.

A topological sorting of a DAG $G(V,E)$ is a linear order of all its vertices such that if (u,v) be an edge in G then u appears before v in the order.



Topological-sort(G)
- Run DFS(G)
- As each vertex finish, insert it in a front of a queue
- return the queue.




9/14 u
x 15/16
z 17/18
y 11/12
10/13 v
a 1/8
w
6/7
b 2/5
c 3/4
z
x
u
v
y
a
w
b
c

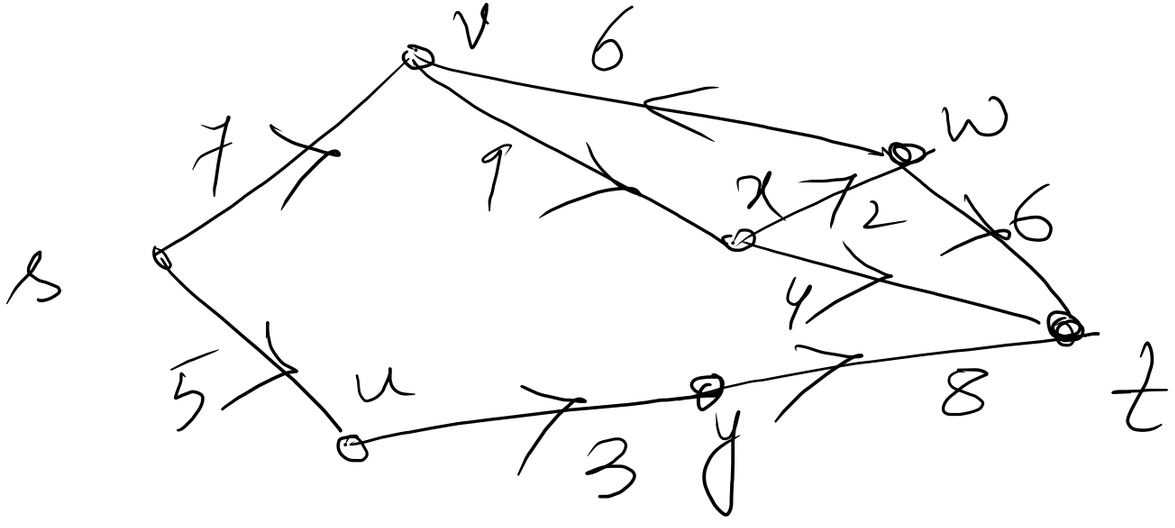
Network-Flow Problem

Flow network : Input: $G = (V, E)$, s, t, $c : E \to \mathbb{R}^+$

A directed graph G

s : source t : sink

For each edge $e \in E$ $c(e) \geq 0$ called the capacity of the edge.



Maximum flow problem

An $s-t$ flow f that satifies the following two conditions / constraints.

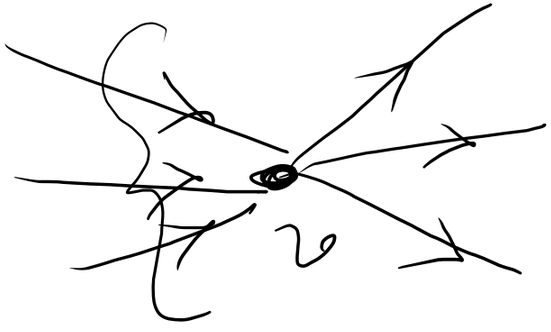
① capacity constraint:

for each edge $e \in E$, $0 \le f(e) \le c(e)$

② flow conservation constraint

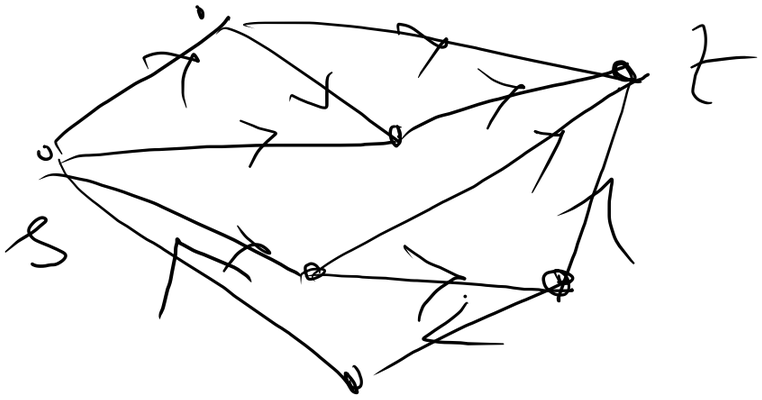
for each vertex $v \in V \setminus \{s, t\}$

$$\sum_{\substack{e \text{ is incident} \\ \text{towards } v}} f(e) = \sum_{\substack{e \text{ is incident} \\ \text{outward } v}} f(e)$$

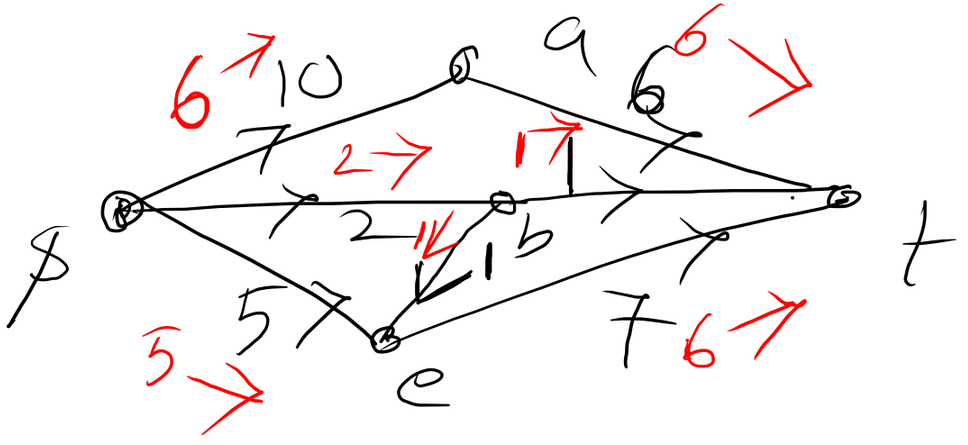


The value of the flow

$$val(f) = \sum_{e \text{ is going out of } s} f(e) - \sum_{e \text{ is going inside } s} f(e)$$

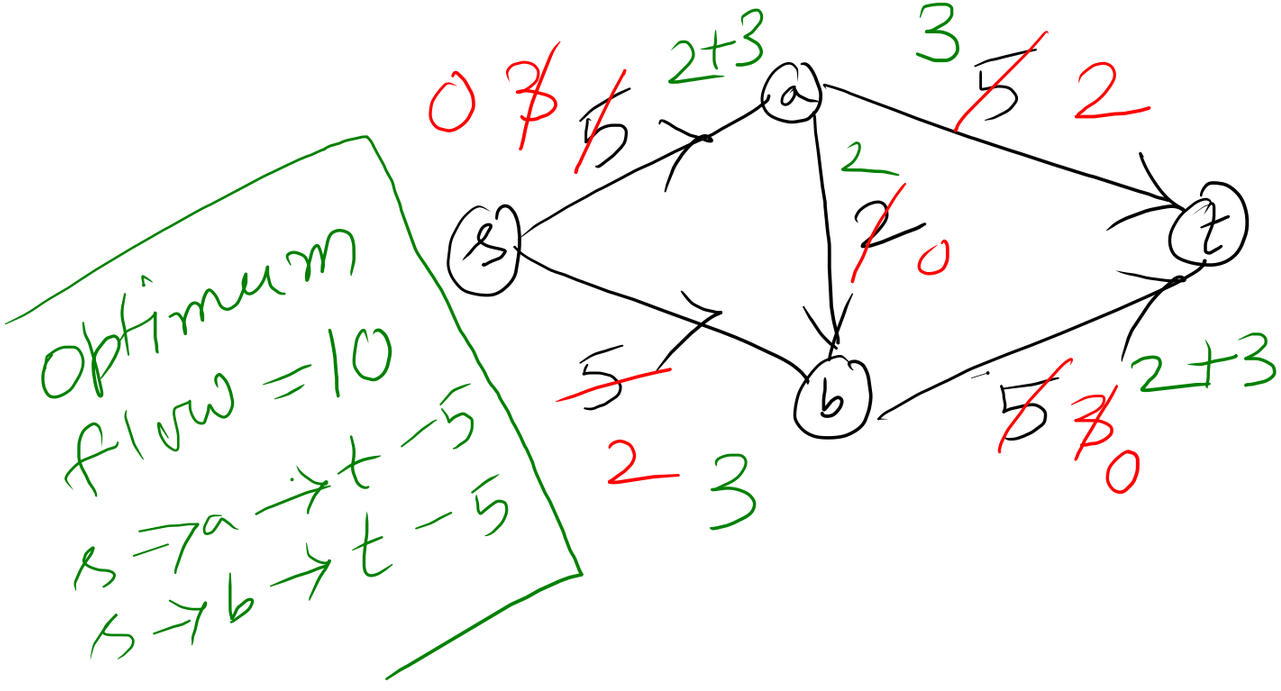


Objective: Find a flow of maximum value.



An algorithm

- for each edge flow is 0 ie $f(e) = 0$
- if there is a $s \to t$ path where $f(e) < c(e)$
- augment flow along the $s \to t$ path
- Repeat until no $s \to t$ path exist.



optimum flow = 10

$s \to a \to t$ - 5

$s \to b \to t$ - 5

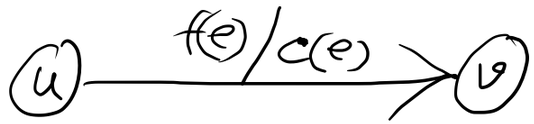
$s \to a \to b \to t \to 2$

$s \to a \to t$ — 3

$s \to b \to t$ — 3

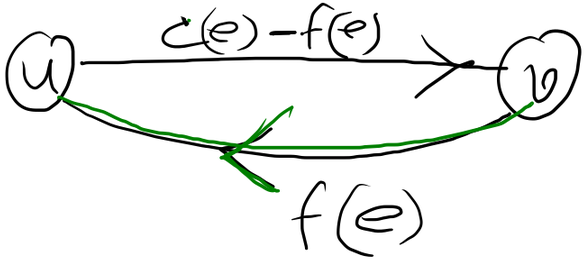
Total flow = 8

Residual network



Residual capacity

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e^{reverse}) & \text{if } e^{reverse} \in E \end{cases}$$



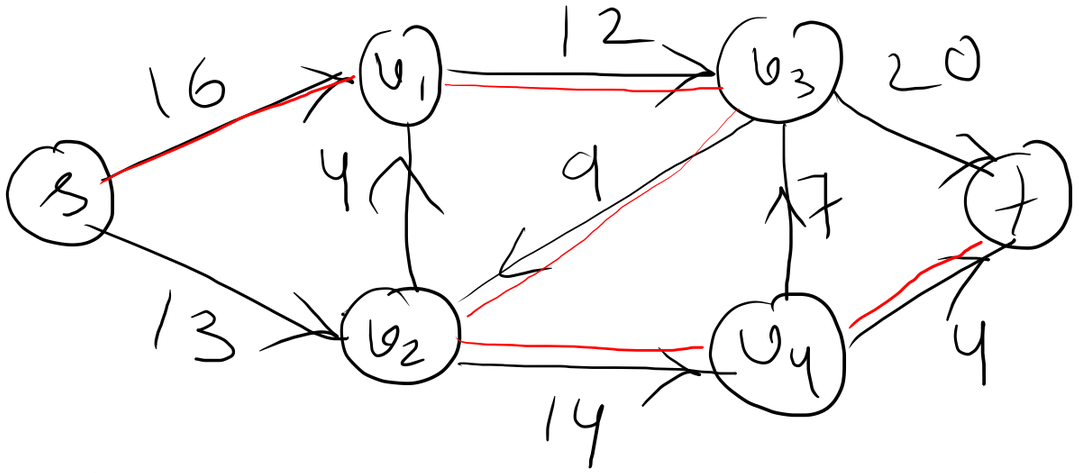
Residual network: $G_f(V, E_f)$, s, t, c_f is residual capacity function.

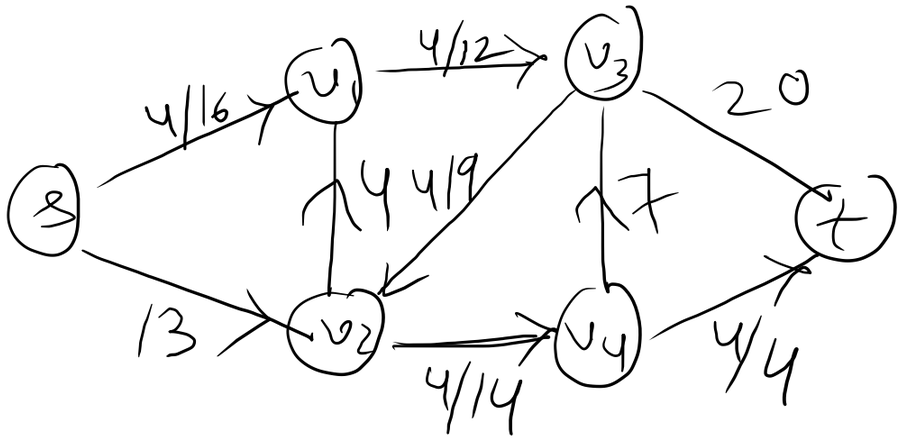
$$E_f = \{ e : f(e) < c(e) \} \cup \{ e : f(e^{reverse}) > 0 \}$$

Ford-Fulkerson algorithm

```
Ford-Fulkerson(G)
    for each edge e ∈ E
        f(e) = 0
    G_f ← residual network of G w.r.t. f
    while there is an s ⇝ t path P in G_f
        f ← augment(G, P, f)
        update G_f
    return f.
```

```
augment(G, P, f)
    δ ← bottleneck capacity in P
    for each edge e in P
        if e ∈ E
            f(e) ← f(e) + δ
        else
            f(e^reverse) = f(e^reverse) − δ
    Return f.
```





$s \to v_1 \to v_3 \to v_2 \to v_4 \to t$