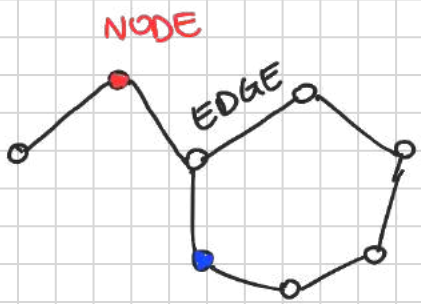
Algorithms

DATE: 22/2/22
LOL



ALGORITHMS
↓
mathematical procedure that allows us to tell what's the shortest path from the red point to the blue point.

- OPTIMISATION PROBLEMS
  - → we will use optimized algoriths.
  - → we will be looking for the most optimal solutions.

DETERMINISTIC ALGORITHM → doesn't make random decisions.

- OPTIM. PROBLEM (shortest path, minimum spanning tree)

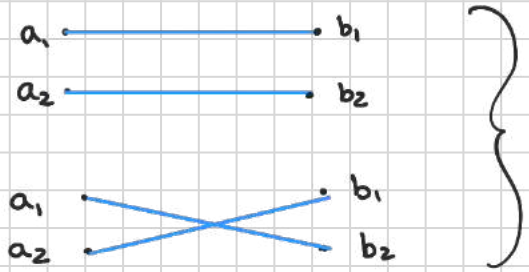
SPANNING TREE → graph with no cycles.

 → shortest path

- ALGORITHM TECHNIQUES
  - greedy algorithms
  - dinamic programming
  - divide - et - impera (divide - and - conquer)

"STABLE MATCHING" PROBLEM

COMPANIES (a_n) APPLICANTS (b_n)



POSSIBLE MATCHES BETWEEN COMPANIES AND APPLICANTS

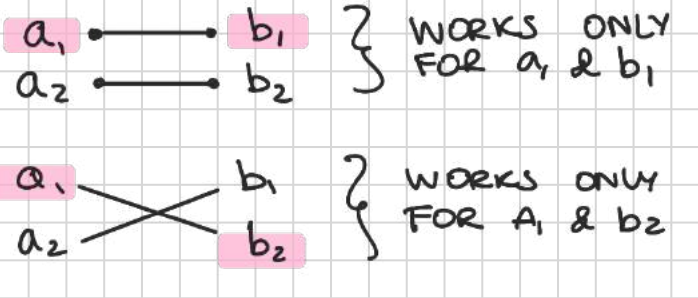
b_1	PREFERS	a_1	to	a_2
b_2	=	a_2	to	a_1
a_1	=	b_1	to	b_2
a_2	=	b_2	to	b_1

IF WE CONSIDER THESE CASES, EVERYONE WILL BE SATISFIED

prefers

$a_1 : b_1 > b_2$
$a_2 : b_1 > b_2$

$b_1 : a_1 > a_2$
$b_2 : a_1 > a_2$



UNSTABLE MATCHING

Def: Let A and B be two sets of cardinality $|A| = |B| = n$, with $A \cap B = \emptyset$:

A perfect matching between A and B is a pairing of the elements of A with the elements of B. That is, $M \subseteq [\{a,b\} \mid a \in A \wedge b \in B]$

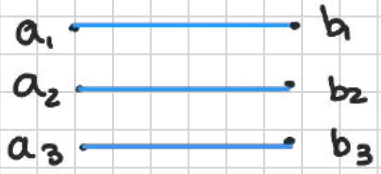
M is a perfect matching if and only if (IFF)
$\forall a \in A$ there exists exactly one $b \in B$ such that $\{a, b\} \in M$ (AND viceversa)

↓ example

One company will have only one applicant and viceversa.

$A = \{a_1, a_2, a_3\}$

$B = \{b_1, b_2, b_3\}$



$M = \{\{a_1, b_1\}, \{a_2, b_2\}, \{a_3, b_3\}\}$

M is a matching IFF $\exists$ bijective function

$f: A \to B \quad \{\{a, f(a)\} \mid a \in A\} = M$

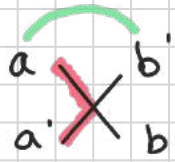
NOT A MATCHING (b_1 left alone, b_2 has 2 matches)

STABLE MATCHING

Let A, B be two sets, $|A| = |B| = n$, $A \cap B = \emptyset$ suppose that each $a \in A$ has a preference order on B and, likewise, each $b \in B$ has a preference order on A.

Given a perfect matching M of A and B, we say that M is unstable if it contains two pairs $\{a, b\}, \{a', b'\} \in M$, such that:

- a PREFERS b' to b AND
- b' PREFERS a to a'



A MATCHING M is STABLE, IF ITS NOT UNSTABLE (bruh...)

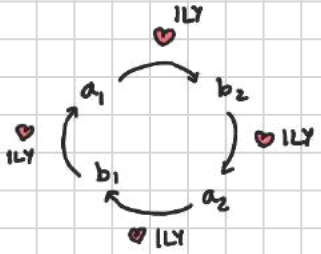
Questions:

1. Does a stable matching always exist?
2. How can we find a stable matching (if it exists)?

ANOTHER CASE :

$a_1 : b_1 > b_2$
$a_2 : b_2 > b_1$

$b_1 : a_2 > a_1$
$b_2 : a_1 > a_2$



INTUITIVE ALGORITHM

- Suppose $\underline{a}$ is unmatched, let $\underline{a}$ propose an engagement to his most preferred $\underline{b} \in B$
  - says "YES" IF b is unmatched,
  - says "NO" if b is matched with a' that she likes better than "a"
  - says "YES" if b is matched with "a'" and she likes "a" more than "a'"
- $\underline{a}$ NEVER ASKS the same b more than once.

(slowly dying .....)

Algorithms

DATE : 24/02/2022

L = LEMMA
P = PROOF

GALE - SHAPELEY ALGORITHM

- Initially, each $a_i \in A$, and each $b_j \in B$, is FREE
- While there exists some FREE a_i that has not yet proposed to each $b_j \in B$
  - Let a_i be a FREE person that has not proposed to each $b_j \in B$
  - Let $B' \subseteq B$ be the set of b_j such that a_i has not yet proposed to
  - Let $b_j \in B'$ be the person from B' that a_i likes the most

$a_1 : b_1 > b_2 > b_3$
$B' = \{b_2, b_3\}$
} b_2 is the most preferred in the B' set

  - IF b_j is FREE :
    - MATCH UP a_i and b_j // a_i & b_j get engaged
    - a_i and b_j are not free anymore
  - ELSE :
    - Suppose that b_j is engaged to a_k
    - IF b_j likes a_k more than a_i
      - a_i REMAINS FREE
    - ELSE :
      - the match between b_j and a_k is broken
      - a_i and b_j are matched up
      - a_k becomes FREE
- RETURN THE FINAL LIST OF "MATCHES" AS THE MATCHING.

NOTE: This still doesn't define whether the final matching is perfect. Proof needed.

TECHNICAL OBSERVATIONS

L1: Each $b \in B$ remains matched / engaged from the first time she gets a proposal until the end of the execution

↑

PROOF: When "b" gets the first proposal, she becomes engaged. (Since it's her first time, NO REFUSE)

From then onwards she might get other proposals. She might either:

- accept
- reject

SHE WILL ALWAYS BE ENGAGED

- If she accepts one, she'll switch partners (but she'll remain engaged);
- If she rejects, she'll keep her previous partner.

L2: The engaments of the generic $b \in B$ get better (from her perspective) over the time

↑

P: b changes partner only if she gets a proposal from a better then her current one.

MONOTONE PROPERTY (keeps getting better)

THE A SIDE HAS A DIFFERENT FATE

L3: For each sequence of proposals made by a decreases in quality over time.

↑

P: TRIVIAL (by the algorithm's def)

NOTE: Theorem → more important than "lemma"

Theorem(T): The algorithm terminates after at most n^2 iterations.

↑

Proof (P): Each a_i can propose to at most $|B| = n$ people from B. In each iteration of the algorithm, some a_i proposes to some $b_j \in B$ that he had not yet proposed to earlier. ↓

Therefore, there can be at most $|A| = |B| = n^2$ proposals, and iterations

---

In general, people look for some "quantities" to bound the runtime of an algorithm

L4: IF $a \in A$ is FREE at some point in the execution, then there must exist some $b \in B$ to which a has not yet made a proposal.

↓ TECHNICAL OBS.

↑ (THIS LEMMA LEADS TO THE FACT THAT WE WILL HAVE A PERFECT MATCHING IN THE END)

P: BY contradiction, suppose that, at some point, $a^* \in A$ is FREE and he has proposed to everyone from B

By L1, each $b \in B$ remains engaged from the first proposal she gets, until the end. Thus, for a^* to remain FREE after $n = |B|$ proposals, it must be that, at the time of his last proposals, each $b \in B$ was engaged.

But, recall that $|A| = |B| = n$.

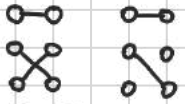
For each $b \in B$ to be engaged, it must be that each $a \in A$ must be engaged. Thus, it is IMPOSSIMPLE that a^* is FREE.

THIS is A PROOF BY CONTRADICTION

- Basically, you start from an assumption and end up contradicting

L5: The algorithm returns a perfect matching

P: When we match a to b, if b was already matched, then we break up the current engagement of b. Thus, the current matches from matching



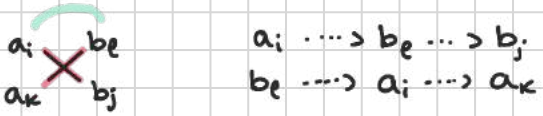
PERFECT MATCHING MATCHING

Suppose, by contradiction, that in the end $a \in A$ is FREE. Then, a has proposed to each $b \in B$. But, this contradicts L4. Thus, in the end, no $a \in A$ is FREE and, also, no $b \in B$ is FREE. ($|A| = |B|$ and the returned structure is a matching). Thus, the algorithm returns a perfect matching.

THEOREM: The algorithm returns a stable matching

PROOF: By L5, the algorithm returns a perfect matching (each person is matched to exactly one other person).

By contradiction, suppose that there exist two pairs in M, $\{a_i, b_j\}$, $\{a_k, b_\ell\}$ that are UNSTABLE



Then, a_i prefers b_ℓ to b_j, and b_ℓ prefers a_i to a_k. By the algorithm, a_i's last proposals was to b_j

Now, let us consider two cases:

- a_i did not propose to b_ℓ before b_j.
  Then, a_i did not ever propose to b_ℓ (b_j is a_i's last proposal). But then, a_i prefers b_j to b_ℓ CONTRADICTION
- a_i proposed to b_ℓ before b_j.
  Then, since b_ℓ ended up with a_k, and since L2 entails that b_ℓ's partners improve over time, it must be that b_ℓ prefers a_k to a_i.
  CONTRADICTION

Therefore, no unstable pairs exist -
Thus, M is a stable matching.

DEF: Let us say that $b_j \in B$ is a valid match for $a_i \in A$ if $\exists$ stable matching M such that $\{a_i, b_j\} \in M$

DEF: Let best(a_i), for $a_i \in A$, be the valid match $b^* \in B$ of a_i, that a_i likes the best (best partner from a_i's perspective)

THEOREM: The G-S algorithm returns $M = \{\{a_i, \text{best}(a_i)\} \mid a_i \in A\}$

THEOREM: // // // // $M = \{\{b_j, \text{worst}(b_j)\} \mid b_j \in B\}$

(This is a bit unfair since one gets the best and the others get the worst, all in the stable matching)

Algorithms

DATE : 1/3/2022

CONTINUE of Previous Lecture

PROOF OF THEOREM A

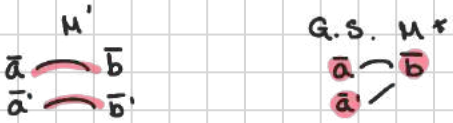
By contradiction, suppose that some "a_i" ends up being matched to a partner other than best(a_i) in M^*

Since the a's propose in decreasing order of preference, there must be a time when some "$\bar{a}$" gets rejected by one of its valid matches "$\bar{b}$" (rejection can happen right after a proposal or when "$\bar{b}$" accepts some other proposal)

Let the pair $\{\bar{a}, \bar{b}\}$ be the first pair (during the execution of G-S algorithm) that is valid , and such that $\bar{b}$ rejects $\bar{a}$.

When this rejection happens, $\bar{b}$ will be paired up with some other $\bar{a}'$ that she likes better than $\bar{a}$

Now, since $\{\bar{a}, \bar{b}\}$ is a valid pair there must exist a stable matching M' such that $\{\bar{a}, \bar{b}\} \in M'$.
Since M' is a stable matching, it must match $\bar{a}'$ to some $\bar{b}'$, $\{\bar{a}', \bar{b}'\} \in M'$



Then observe that $\{\bar{a}', \bar{b}'\}$ is a valid pair.
Given that $\{\bar{a}, \bar{b}\}$ was the first valid pair with a rejection, it must be that $\bar{a}'$ was not rejected by a valid partner before $\bar{a}'$ gets engaged with $\bar{b}$ — 1st claim

Now, since $\bar{b}'$ is a valid partner of $\bar{a}'$ ($\{\bar{a}', \bar{b}'\} \in M'$), — 3rd claim
and since $\bar{a}'$ proposes in decreasing order of preference, it must be that $\bar{a}'$ prefers $\bar{b}$ to $\bar{b}'$ — 2nd claim

Since $\bar{b}$ prefers $\bar{a}'$ to $\bar{a}$ (she rejected $\bar{a}$ to be with $\bar{a}'$) and since $\{\bar{a}, \bar{b}\}$, $\{\bar{a}', \bar{b}'\} \in M$. It holds that $\{\bar{a}, \bar{b}\}, \{\bar{a}', \bar{b}'\} \in M'$ is an instability of M' thus M' is unstable. This is a contradiction.

PROOF OF THEOREM B

Suppose that $\exists \{\bar{a}, \bar{b}\} \in M^*$ s.t. $\bar{a} \neq \text{worst}(\bar{b})$.

Then, there exists a stable matching M' s.t. $\{\bar{b}, \bar{a}\} \in M'$ and $\bar{b}'$ likes $\bar{a}'$ less than $\bar{a}$

Suppose that in M', $\bar{a}$ is matched with some $\bar{b}' \neq \bar{b}$ ($\{\bar{a}, \bar{b}'\} \in M'$).

By theorem A, we know that $\bar{b}$ is the best valid partner of $\bar{a}$ thus, $\bar{a}$ likes $\bar{b}$ more than $\bar{b}'$ (or thm A would not hold). that is $\bar{a}: \bar{b} > \bar{b}'$ and $\bar{b}: \bar{a} > \bar{a}'$ but, $\{\bar{a}, \bar{b}\}$. $\{\bar{a}, \bar{b}'\} \in M'$.
Thus, = = form an instability of M', which is then unstable. CONTRADICTION

G-S has given us the opportunity to discuss about a number of questions one encounters when studying algorithm problems.

1. Formulated the problem in a mathematical precise way
2. Ask questions about the mathematical problem
3. Design an efficient algorithm for your problem
4. Prove that your assumption is correct, and bound its runtime

---

EFFICIENCY

Def (??) "An algorithm is efficient if, after being implemented, it runs quickly on real input instances."

(not a formal definition)

"QUICKLY": "quick" on which hardware? with which programming language? with which implementation? (each of these questions significantly changes the runtime)

We would like our definition to be independent of details

REAL INPUT INSTANCES : What does it mean? How to define them?

To prove something about an algorithm, we need mathematical definitions.

WORST-CASE ANALYSIS

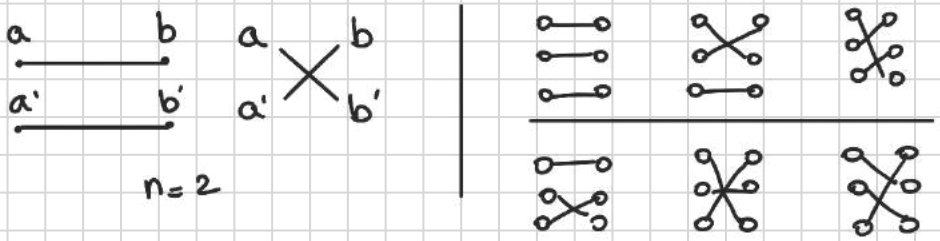
We want the runtime of an algorithm to be bounded in terms of the worst possible input. For instance, in the case of stable matching, we proved that G-S algorithm uses at most n^2 iterations ($n = |A| = |B|$)

Suppose, for instance that we consider inputs in which $|A| = |B| = N$, for $N = 2n$

$$N^2 = (2n)^2 = 4n^2$$

Doubling n changes the runtime by a factor 4

The brute-force algorithm tried each of the n! matches



What would happen to the brute-force algorithm if we doubled the number of people?

$$N! = (2n)! =$$

$$= 2n(2n-1)(2n-2)\dots(n+1)\cdot n(n-1)\cdot 2\cdot 1$$
$$= 2n(2n-1)\dots(n+1)\,n!$$
$$\geq (n+1)^n \cdot n! \geq 2^n \cdot n!$$

Def: An algorithm having a runtime $\leq c \cdot n^d$, where c and d are constants, on inputs of size n is said to be a polynomial time (POLYTIME) algorithm

If I have a polytime algorithm running in time $c \cdot n^d$ on inputs of size n, and I run it on inputs of size $N = b \cdot n$,

$$c \cdot N^d = c(nb)^d = c \cdot n^d \cdot b^d$$

then the blow-up in the runtime is going to be a constant b^d

Algorithms

DATE : 03/03/2022

We talked last time about "polynomial time" algorithm.

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

WALL-CLOCK RUNTIMES WITH A PROCESSOR PERFORMING 1 MILLION OPERATORS / s

Def: An algorithm has a (worst-case) runtime of $f(n)$ if the algorithm never makes more than $f(n)$ operations when run on inputs of size n.

So: How can we determine the (worst-case) runtime for an algorithm?
How can we say that an algorithm has a runtime $\leq f(n)$?

Given the varying nature of computational devices (and of their CPUs), it's very hard to count exactly the number of a given algorithm, or of a given piece of code.

```
DEF INC(A):
    RETURN A+1

INC(10
```

ALL OF THIS CAN BE DONE WITH C OPERATIONS

- "10" will be pushed onto the stack
- INC will be executed
- INC will add to its namespace a variable of name "A"
- The value "10" will be popped out of the stack, and will be assigned to A.
- You have to take the value of A, sum 1 to that value, and you have to return the result.

no way to tell how many operations the CPU does

It's easy to observe that there exists a constant $c > 1$ such that, that piece of code can be run with c elementary operations. It's hard to determine c.

ELEMENTARY OPERATIONS ⟶ CPU runs in one clock/s

UNPRECISE DEFINITION which might be true for older CPUs

The real definition won't be defined because there are many factors to take in consideration

NOTE: if we throw away constants, we won't be able to distinguish which algorithm is faster.

Suppose that, somehow, we determine that - in this case (code above) - $c = 27$.

```
DEF F(n):
  x=0
  FOR i in range(n):
     x= INC(x)
  return x
```

```
DEF INC(A):
  RETURN A+1
```

} 27 OPER.

RUNTIME $f(n)$ → $\frac{n27 + c'n + c''}{n(27 + c') + c''}$

we have to throw away const.

```
y=0
FOR i IN RANGE(n)
   y+= f(i)
```

⟶ $n(n(27 + c') + c'') + c'''$

O, Ω, Θ - ASYMPTOTIC RUNTIMES

To avoid getting stuck into endlessly many constants, we just disregard constants.

For an algorithm A, let $T(n)$ be its runtime (worst-case) on inputs of size n.

What we aim to do is to represent a runtime $T(n) = 1,35n^2 + 0,16n + 250$ with it's most representative term, n^2

DEF : IF $T(n)$ and $f(n)$ are non-negative, increasing, functions, we say that "$T(n)$" is $O(f(n))$

IF $\exists\ c, n_0 > 0$, such that $\forall n \geq n_0$, $T(n) \leq c\, f(n)$.

$T(n) = n$ $\quad$ $f(n) = n^3 - n^2$

$n > 1$

$T(1) = 1$ $\quad$ $f(1) = 0$

$T(n) = O(f(n))$

$T(n) \leq O(f(n))$

$T(n) \in O(f(n))$

Suppose that an algorithm takes time $T(n) = pn^2 + qn + r$, $p, q, r \geq 0$ and $p \neq 0$

Then $\forall n \geq n_0 \triangleq 1$, IT holds that $qn \leq qn^2 = n(qn)$

IT $\Rightarrow$ $\Rightarrow$ $r \leq rn^2$.

Thus, $T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p+q+r)n^2$

If we take $c = p+q+r$, we then have that $f(n) = n^2$

$T(n) \leq c \cdot f(n) \quad \forall n \geq n_0$

$T(n) = O(f(n)) = O(n^2)$

$T(n) = pn^2 + qn + r$, $p, q, r \geq 0$, $p \neq 0$

Is it the case that $T(n) = O(n^3)$?

$T(n) \leq (p+q+r)n^2 \leq (p+q+r)n^3$ $\quad$ $(\forall n \geq n_0 = 1)$

$c = p+q+r$

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

How can we prove that $O(n^2)$ is tighter to $T(n)$ than $O(n^3)$?

DEF: IF $T(n)$ and $f(n)$ are non-negative, increasing, functions, we say that "$T(n)$ is $\Omega(f(n))$" IF $\exists\ c, n_0 > 0$ such that $\forall n \geq n_0$, $T_n \geq c \cdot f(n)$.

$T(n) = pn^2 + qn + r \qquad p, q, r \geq 0$ AND $p \neq 0$

$qn \geq 0 \cdot n = 0$

$r \geq 0$

$T(n) \geq p \cdot n^2 \qquad c = p \qquad n_0 = 0$

$$T(n) \geq c f(n) \qquad f(n) = n^2$$

THUS $\quad T(n) = \Omega(f(n)) = \Omega(n^2)$

$$T(n) \geq \Omega(n^2)$$

If $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$, we write that $T(n) = \Theta(f(n))$.

$f(n) = \Theta(T(n))$

$T(n) \geq \Omega(f(n))$
$T(n) \leq O(T(n))$

Since, $T(n) = pn^2 + qn + r$

$T(n) \leq O(n^2)$, $T(n) \leq O(n^3)$

$T(n) \geq \Omega(n^2)$

$T(n) = \Theta(n^2)$

TRANSITIVITY

L: IF $f = O(g)$ and $g = O(h)$, then $f = O(h)$
IF $f = \Omega(g)$ and $g = \Omega(h)$, " $f = \Omega(h)$.

P: $f = O(g)$ implies that $\exists\ n_0, c > 0$ s.t. $f(n) \leq c \cdot g(n)$ $\forall n \geq n_0$

$g = O(h)$ " " $\exists\ n_0', c' > 0$ s.t. $g(n) \leq c' h(n)$ $\forall n \geq n_0'$

Then, $\forall\ n \geq \max(n_0, n_0')$ IT HOLDS THAT :

$f(n) \leq c\,g(n)$ and $g(n) \leq c'h(n)$

$f(n) \leq cg(n) \leq c(c'h(n)) = (c \cdot c') \cdot h(n)$

THUS, FOR $c'' = c \cdot c'$ and for $n_0'' = \max(n_0, n_0')$
IT HOLDS THAT $f(n) \leq c'' \cdot h(n) \quad \forall n \geq n_0''$

THUS $f(n) = O(h(n))$

The Ω case is symmetric, so we skip it $\square$

C: If $f(n) = \Theta(g(n))$ AND $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

SUM OF FUNCTIONS

L: If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.

P: $\exists\ n_0, n_0', c, c'$ S.T. $f(n) \leq c \cdot h(n) \quad \forall n \geq n_0$ AND
$g(n) \leq c \cdot h(n) \quad \forall n \geq n_0'$

BUT THEN, $f(n) + g(n) \leq c \cdot h(n) + c' h(n) = (c + c') h(n)$ IF
$n \geq \max(n_0, n_0')$

WITH $c'' = c + c'$ and $n_0'' = \max(n_0, n_0')$, we get
$f + g = O(h)$.

$f(n) = n \log_2 n$

$g = f = h = f$

$g(n) = f(n) + f(n)$

$f(n) + f(n) = O(f(n))$.

$= O(n \cdot \log_2 n)$

$f(n) + f(n) + f(n) = O(n \cdot \log n)$

$$\sum_{i=1}^{\sqrt{n}} f(n) = \sqrt{n} \cdot f(n) = O(n^{3/2} \log n)$$

L: Let $K \geq 1$ BE A CONSTANT.

IF $F_1 = O(h), f_2 = O(h), \ldots, f_K = O(h)$ then $f_1 + f_2 + \ldots + f_K = O(h)$

$O(100 \cdot n) = O(n)$

Let $T(n) = a_0 + a_1 n + a_2 n^2 + \dots + a_d n^d$, for some $d \geq 1$, with $a_d \neq 0$

$$T(n) = 3 + n^2 + 15n^3$$

we say that $T(n)$ is a polynomial of degree d.

L: If $f(n)$ is a polynomial of degree "d" with non-negative coefficients, then $f(n) = O(n)$

P: $f(n) = \sum_{i=0}^{d} (a_i n^i)$. If we let $f_i(n) = a_i n^i$, we have that $f(n) = \sum_{i=0}^{d} f_i(n)$.

But, then, $f_i(n) \leq |f_i(n)| = |a_i| n^i$

$$f(n) = \sum_{i=0}^{d} \leq \sum_{i=0}^{d} (|a_i| n^i) = \sum_{i=0}^{d} O(n^i)$$

(since d is a constant) $\leq \sum_{i=0}^{d} O(n) - O(n)$

L: For each $b > 1$, and for each $x > 0$, it holds $\log_b n = O(n^x)$

$$\log n \leq O(n^{0.001})$$

$$\log_a n = \frac{\log_b n}{\log_b a} \Rightarrow \log_a n = \left(\frac{1}{\log_b a}\right) \log_b (n)$$

$$\log_a n = \Theta(\log_b n)$$

L: For each $r > 1$, and for each $x > 0$ it holds $n^x = O(r^n)$

$$n^{100} \leq O(1{,}01^n)$$

$2^n = O(3^n)$ $\qquad$ $3^n \neq O(2^n)$

a^n vs b^n $\qquad$ $a = 2 \quad b = 3$

$a = c \cdot b$ $\qquad$ $c = \frac{2}{3}$

$a^n = (c \cdot b)^n = c^n \cdot b^n$

IF $c > 1 \Rightarrow c^n \to \infty$

IF $c < 1 \Rightarrow c^n \to 0$

IF $c > 1$ $\quad a^n = O(b^n)$

IF $c < 1$ $\quad a^n = \Omega(b^n)$

Algorithms

DATE : 08/03/2022

GALE - SHAPELEY ALGORITHM

- Initially, each $a_i \in A$, and each $b_j \in B$, is FREE
- while there exists some FREE a_i that has not yet proposed to each $b_j \in B$
  - Let a_i be a FREE person that has not proposed to each $b_j \in B$ (1)
  - Let $B' \subseteq B$ be the set of b_j such that a_i has not yet proposed to (2)
  - Let $b_j \in B'$ be the person from B' that a_i likes the most

$a_1 : b_1 > b_2 > b_3$
$B' = \{b_2, b_3\}$

b_2 is the most preferred in the B' set

  - IF b_j is FREE : (3)
    - MATCH UP a_i and b_j // a_i & b_j get engaged
    - a_i and b_j are not free anymore
  - ELSE :
    - Suppose that b_j is engaged to a_k (3)
    - IF b_j likes a_k more than a_i (4)
      - a_i REMAINS FREE
    - ELSE :
      - the match between b_j and a_k is broken
      - a_i and b_j are matched up
      - a_k becomes FREE

O(1)

① Identify a free a_i

② Identify the b_j that is highest in a_i's preference list, that a_i hasn't yet proposed to

③ Given b_j, determine whether she's free, and - if not - identify her current partner a_k.

④ Determine whether b_j likes a_i more than a_k

To implement each of the 4 operations in $O(1)$ time, we use the following data structures:

Ⓐ APREF, A $n \times n$ array, such that APREF[i][e] contains the index of the e^{th} most preferred b_j in a_i's ranking.

EX: if $a_i : b_{i_1} > b_{i_2} > \cdots > b_{i_n}$ then APREF[i][e] = i_e

$e_3 : b_2 > b_1 > b_3$

APREF[3][1] = 2
APREF[3][2] = 1
APREF[3][3] = 3

Ⓑ NEXT, an array of size n, such that next[i] contains the rank of the next b_j in a_i's list, that a_i is going to propose to.

At the outset, next[i] = 1 $\forall i$.

When a_i is about to propose, he'll propose to b_j for j = APREF[i][next[i]]; after having proposed, next[i] += 1

↳ thus ② can be solved in $O(1)$

Ⓒ CURRENT, an array of size n, such that current[i] contains the index j of the a_j that b_i is currently engaged to; if b_i is currently free, then CURRENT[i] = None (-1) ← simbolo per dire None

Thus, ③ can be solved in $O(1)$ time.

Ⓓ RANKING, an $n \times n$ array, such that RANKING[j][i] is the rank of a_i in b_j's preference list.

If b_j : $a_{j_1} > a_{j_2} > \cdots > a_{j_n}$, then RANKING[j][$j_\ell$] = ℓ

EXAMPLE : if b_3 : $a_2 > a_1 > a_3$ then

RANKING[3][2] = 1
RANKING[3][1] = 2
RANKING[3][3] = 3

To decide whether b_j likes a_k more than a_i it is sufficient to check whether
RANKING[j][k] < RANKING[j][i]

Thus, ④ can be solved in O(1) time

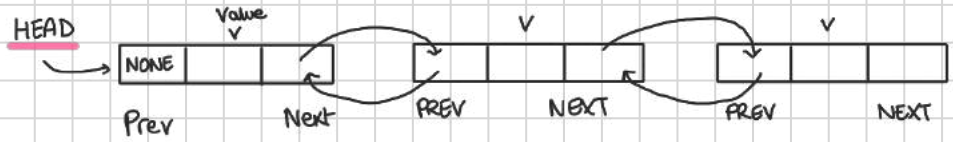
So, ②, ③ and ④ can be implemented in O(1) time, if one has arrays Ⓐ, Ⓑ, Ⓒ, Ⓓ. (It is easy to check that Ⓐ, Ⓑ, Ⓒ, Ⓓ can be built $O(n^2)$ time.)

So, the only thing that remains to be done is to solve / implement ① in O(1) time. If we can do that, the total runtime G-S becomes :

$$O(n^2) + \underbrace{n^2}_{\text{nº of iterations}} \cdot (\underbrace{O(1)}_{①} + \underbrace{O(1)}_{②} + \underbrace{O(1)}_{③} + \underbrace{O(1)}_{④}) = O(n^2)$$

① IDENTIFY A FREE a_i
HOW TO DO IT IN O(1) TIME?

To do this, we're going to use doubly-linked lists.

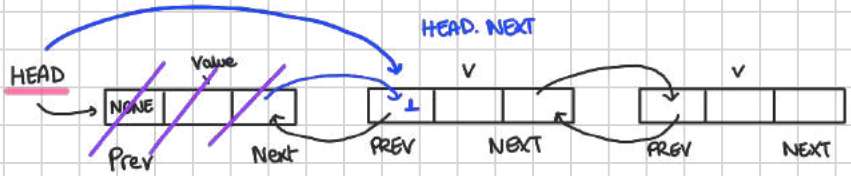


To get the first element of the list, I can just run.

HEAD. V.

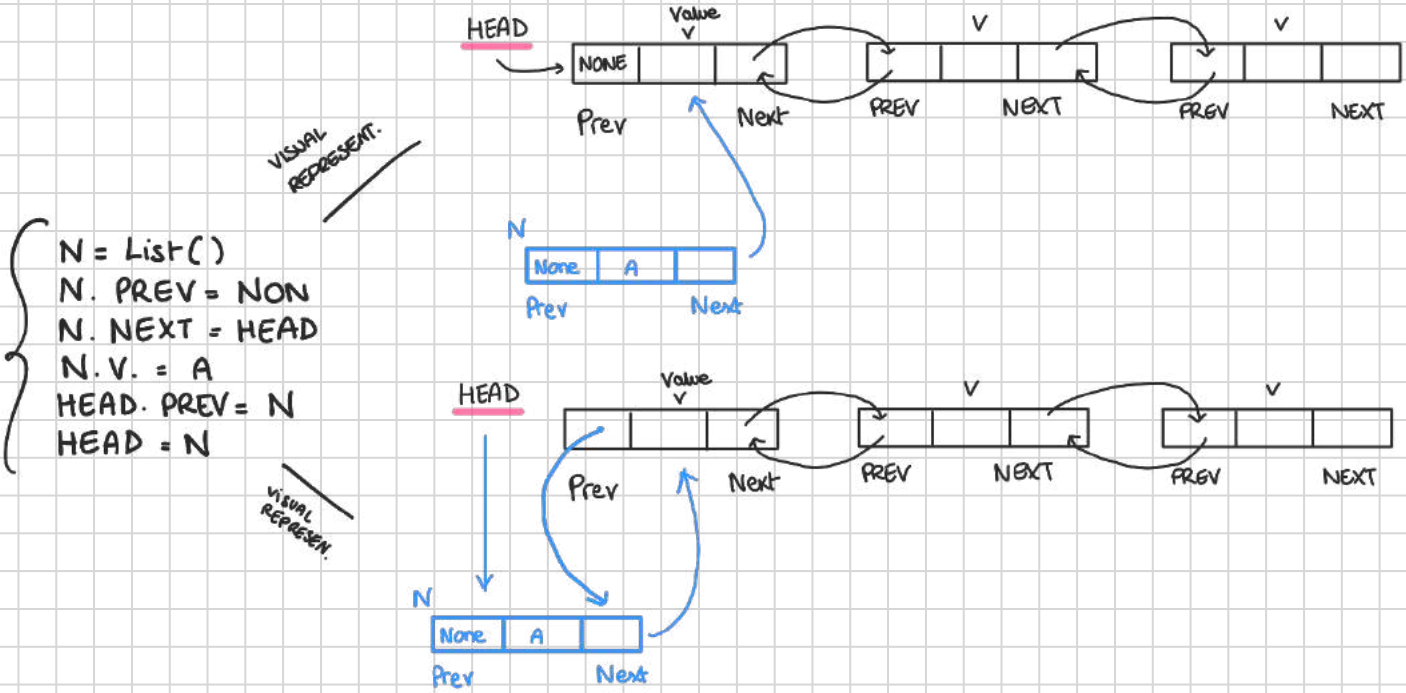
To remove the first element from the list

HEAD = HEAD. NEXT
HEAD. PREV = NONE



Now, we only need to attach to the list some a_i that is rejected by b_j.

Whenever b_j has to choose between a_i and a_k, the "UNCHOSEN" one (let us say it is a_t for $t \in \{i, k\}$) will be added back to the list.



THM: G-S can be implemented to run in $O(n^2)$ time.

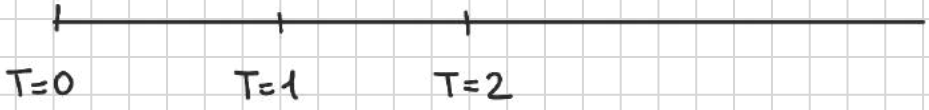
Algorithms

DATE: 10/03/2022

GREEDY ALGORITHMS

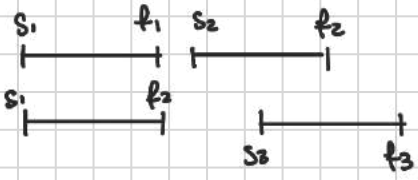
INTERVAL SCHEDULING PROBLEM

1 RESOURCE (CPU, CLASSROOM, CAR)



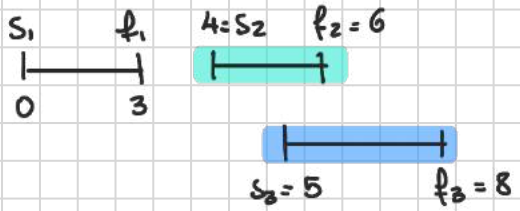
Each job i has a starting time s_i, and an ending time f_i

$$I = \{(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)\}$$



s = 2

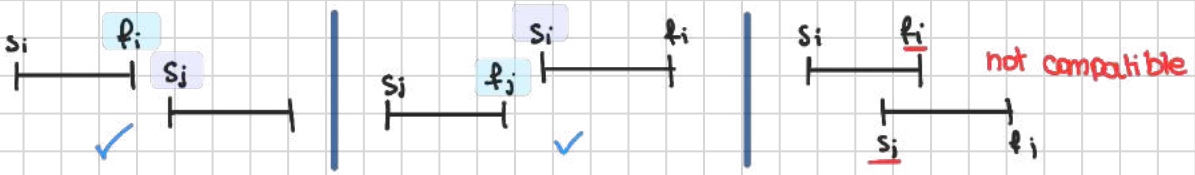
s = 2

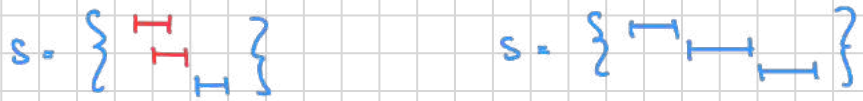


if you choose one, you have to cancel out another

Q: How to find a longest set of "compatible" jobs to be scheduled?

DEF: Let us a consider a set of jobs S. Now, $S \subseteq I$ and it's compatible if $\forall (s_i, f_i), (s_j, f_j) \in S$ and $(s_i, f_i) \neq (s_j, f_j)$, it holds that $\min(f_i, f_j) < \max(s_i, s_j)$





S is incompatible

S is compatible

PRECISE PROBLEM DEFINITION

P: Given an input I (a set of slots I), what is a longest subset $S \subseteq I$ of pairwise compatible jobs?

INPUT, INSTANCES

```
SELECT_M (I):                                   ← INPUT, INSTANCES
    S = [ ]
    WHILE |I| ≥ 1 :
        PICK (s_j, f_j) ∈ I according to rule M
        N = { (s_i, f_i) | (s_i, f_i) ∈ I AND s.t. (s_i, f_i)
              and (s_j, f_j) ARE INCOMPATIBLE }
        I -= N          // I and N are sets
        S.APPEND((s_j, f_j)) // appending compatible intervals
    RETURN S
```

IT CONTAINS ALL THE REMAINING INTERVALS INCOMPATIBLE WITH (s_j, f_j)

L: $\forall$ rule M, $SELECT_M$ returns a set of compatible jobs/intervals, hence, it returns a FEASIBLE (ammissibile) solution

P: Let (s_j^*, f_j^*) be the jth interval selected by the algorithm. (That is, the interval time the algorithm selects in the jth iteration of its loop).

Likewise, (Allo stesso modo)

- Let $I_0 \triangleq I$ be the set of input intervals, (another way to say "uguale =")
- Let $S_0 = [\,]$ be the value of S, before the loop begins;
- let I_j be the value of the variable I at the end of the jth iteration of the loop;
- let S_j be the value of S at the end of jth iteration.

Usiamo l'indice generico j perché vogliamo sapere i valori di I ed S in diversi momenti del ciclo

STRICT SUPERSET

Then, $I_0 \supsetneq I_1, \supsetneq I_2 \supsetneq \dots \supsetneq I_A = \emptyset$, if the loop runs for A iterations.

$A \supsetneq B \iff A \supset B$
$\Updownarrow$
$(B \subseteq A \text{ AND } B \neq A)$

This is because the set N has to contain at least the interval that the rule M has just selected. In particular, then, that interval will be removed from I.

The set I will be decreased of one or more intervals at every iteration.

N will never be empty because there will always be an interval (s_i, f_i) that is incompatible with itself

$S_{j+1} = S_j \cup \{(s^*_{j+1}, f^*_{j+1})\}, \forall j = 0, \dots, t-1$

$\subsetneq$ = STRICT SUBSET

$\emptyset = S_0 \subsetneq S_1 \subsetneq S_2 \subsetneq \dots \subsetneq S_t$

claim

C: Each interval in I_j is compatible with each interval in S_j, $\forall j \geq 0$

P: IF $j=0$, then $S_j = S_0 = \emptyset$. Thus, $\forall A \in I_j$ "$\forall B \in S_j$" A and B are compatible. This claim is true but "empty". (There is nothing in S_j)

Suppose that the claim holds for j. Let's prove it for $j+1$. That is, we assume that $\forall A \in I_j \ \forall B \in S_j$ A and B are compatible.
We would then just like to prove that $\forall A \in I_{j+1}$, $\forall B \in S_{j+1}$, A AND B are compatible. Recall that $S_{j+1} = S_j \cup \{(s^*_{j+1}, f^*_{j+1})\}$.
Thus, if we prove that (s^*_{j+1}, f^*_{j+1}) is compatible with each $A \in I_{j+1}$, we are done.

Now, $I_{j+1} \subseteq I_j$ and the generic interval of I_j is in I_{j+1} IFF (IF and only if) it is compatible with (s^*_{j+1}, f^*_{j+1}).

Thus, $\forall A \in I_{j+1}$ is compatible with (s^*_{j+1}, f^*_{j+1}), the inductive step is then proved.

We now show that each of $S_0, S_1, S_2, \dots$ is a set of compatible jobs. Since the last of the S_j's is going to be returned, the algorithm returns a set of compatible jobs.

By induction,

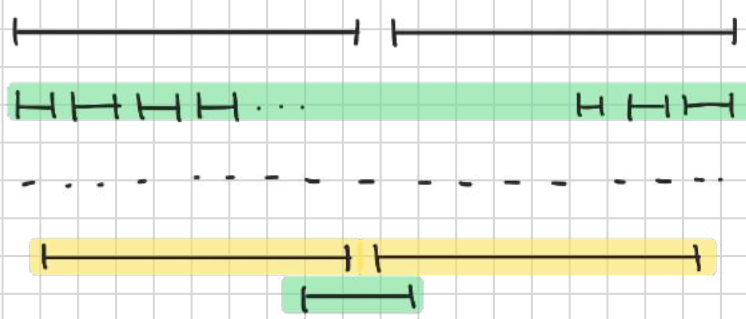
- S_0 is vacuously compatible;
- Assume that the claim holds for S_j. We prove it for $S_{j+1} = S_j \cup \{(s^*_{j+1}, f^*_{j+1})\}$, with $(s^*_{j+1}, f^*_{j+1}) \in I_j$. By the previous claim, each interval in S_j is compatible with each interval in I_j. Thus, $(s^*_{j+1}, f^*_{j+1}) \in I_j$ is compatible with each $B \in S_j$. Moreover, by induction, any two intervals $A, B, \in S_j$. $A \neq B$ are compatible. Thus, S_{j+1} is compatible.

SELECT$_M$ (I) : → INPUT, INSTANCES

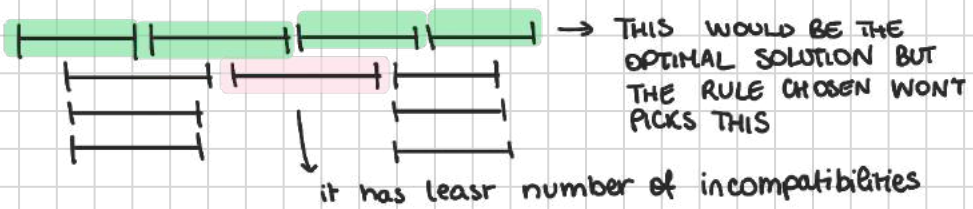
```
S = [ ]
WHILE |I| ≥ 1 :
    PICK (s_j, f_j) ∈ I according to rule M
    N = {(s_i, f_i) | (s_i, f_i) ∈ I AND S.T. (s_i, f_i)
        and (s_j, f_j) ARE INCOMPATIBLE }
    I -= N        // I and N are sets
    S.APPEND((s_j, f_j))
RETURN S
```

HOW TO CHOOSE A RULE M?

- M = "PICK ONE OF THE SHORTEST INTERVALS" ✗



- M = "PICK ONE OF THE INTERVALS HAVING THE SMALLEST NUMBER OF INCOMPATIBILITIES" ✗



M = "PICK THE INTERVAL THAT BEGINS SOONEST"

∎

- M^* = "PICK AN INTERVAL THAT ENDS SOONEST" ✓

L: SELECT_{M^*} returns an optimal solution.

P: Let $O \subseteq I$ be an optimal solution (O is compatible, $|O|$ is as large as possible).

CARDINALITY NOT LOL

WITHOUT LOSS OF GENERALITY: W.L.O.G., $O = \{J_1, \dots, J_n\}$. Let us denote the ending time of interval A with $f(A)$, and its starting time with $s(A)$.

REPHRASE: AGAIN, W.L.O.G, we can assume that $f(J_i) < s(J_{i+1})$ $\forall i = 1..m-1$

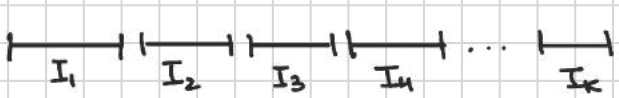
SUPPOSE we sort the J_i's by their ending time.
$f(J_1) < f(J_2) < f(J_3) < \dots < f(J_n)$

Then it must be that $s(J_1) < f(J_1)$. Moreover, $f(J_1) < s(J_2)$ - if this does not hold J_1 and J_2 are incomp., and thus O is not a solution. Thus, by induction,

$$s(J_1) < f(J_1) < s(J_2) < f(J_2) < s(J_3) < f(J_3) < \dots < s(J_m) < f(J_m).$$

Let S be the solution by SELECT_M. By the previous lemma, S is compatible. Then, W.L.O.G., $S = \{I_1, I_2, \dots, I_K\}$ such that

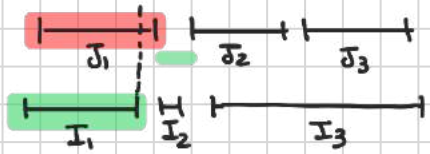
$$s(I_1) < f(I_1) < s(I_2) < f(I_2) < \dots < s(I_K) < f(I_K)$$



We would like to prove that $K \geq m$. (our algorithm is no worse than the optimum), $|S| \geq |O|$.

CLAIM ← **C**: $f(I_1) \leq f(J_1)$

P: By our rule M^*, I_1 is going to be one of the intervals that end soonest. Thus, $f(I_1) \leq f(J_1)$.



Thus, if $O = \{J_1, J_2, \dots, J_m\}$ was an optimal solution, then $O_1 = \{I_1, J_2, J_3, \dots, J_m\}$ is also an opt. sol.

Recall that $S = \{I_1, I_2, \dots, I_k\}$

The solution O_1 is optimal and shares its first interval with our greedy solution.

C: For each $0 \le i \le k-1$, $\exists$ Optimal solution O_{i+1} S.T. $O_{i+1} = \{I_1, I_2, \dots, I_{i+1}, J_{i+2}, J_{i+3}, \dots, J_m\}$ and $f(I_{i+1}) \le f(J_{i+1})$

P: We already proved the claim for $i=0$.
Let us assume to have proved the claim for i. We prove it for $i+1$

Thus, $f(I_{i+1}) \le f(J_{i+1})$. Our algorithm picks the next interval among those that are compatible with $\{I_1, \dots I_{i+1}\}$ and that ends soonest. - Let us call this interval I_{i+2} whichever interval J_{i+2} is, it must be compatible with $I_1, I_2, \dots, I_{i+1}$ then $f(I_{i+2}) \le f(J_{i+2})$.

Thus defining $O_{i+2} = \{I_1, I_2, \dots, I_{i+1}, I_{i+2}, J_{i+3}, \dots, J_m\}$ results in an optimal, feasible, solution (I_{i+2} is compatible with each of $I_1, \dots, I_{i+1}$ and since $f(I_{i+2}) \le f(J_{i+2})$, it is also compatible with $J_{i+3}, J_{i+4}, \dots, J_n$. ■

Thus, we know that the solution $O_k = \{I_1, I_2, \dots, I_k, J_{k+1}, \dots, J_k\}$ is optimal. (DOESN'T EXIST)

Since the greedy algorithms stops exactly when there are no more compatible jobs with the ones it selected, it must be that $J_{k+1}, J_{k+2}, \dots, J_m$ DON'T EXIST

Thus, $O_k = \{I_1, \dots, I_k\}$, so that $m = k$, and the greedy solution is optimal ■

GREEDY ALGOS proofs hypothesize the existence of some optimal solution O
As the greedy algo progresses one shows that the existing hypothetical optimal solution can be turned into an optimal solution that matches all the choices made by the greedy algo so far.

Thus, in the end, greedy can be shown to return an optimal solution.

In other words, you should always be able to prove that the current (partial) solution of the greedy algo can be extended to a full optimal solution.

NOTE: Remember that whenever we want to study an algorithm, there are essentially three things we want to do:

1. We want to prove that whatever it returns is a valid solution
2. We want to prove ideally that the solution is optimal
3. We want to prove that the runtime of the algorithm is small

Algorithms

DATE: 15/03/2022

GREEDY ALGORITHMS

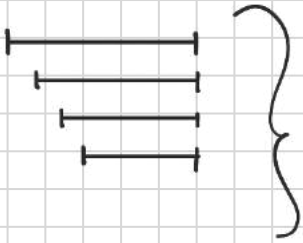
SELECT$_M$ (I): → INPUT, INSTANCES

```
S = [ ]
WHILE |I| ≥ 1 :
O(|I|)  PICK (s_j, f_j) ∈ I according to rule M
        N = {(s_i, f_i) | (s_i, f_i) ∈ I AND S.T. (s_i, f_i)
            and (s_j, f_j) ARE INCOMPATIBLE}
        I = N        // I and N are sets   O(|I|)
O(1)    S.APPEND((s_j, f_j)) // appending compatible intervals
RETURN S
```

IT CONTAINS ALL THE REMAINING INTERVALS INCOMPATIBLE WITH (s_j, f_j)

M = "PICK THE INTERVAL THAT ENDS SOONEST"

IF n is the number of input interwals



with this input one iteration is sufficient.

with this input n, the iterations are necessary

OBS: The number of iterations of the while loop is $\leq n$.

P: At least one interwal is removed from I in the generic iteration

With simple data structure ("I" is just a linked list), the generic iteration takes:

$$O(|I|) + O(|I|) + O(1) = O(\max(|I|, |I|, 1)) = O(|I|)$$

Let $I_0 = I$ be the input set of intervals ($|I| = |I_0| = n$ intervals)

Let I_i be the value of I after the ith iteration of the while loop.

$|I_0| = n$

we always cut AT LEAST one interval at every iter.

$|I_0| > |I_1| > |I_2| > \dots > |I_t| = 0$ (IF THE LOOP ITERATES t TIMES)

$|I_0| \geq |I_1| + 1$

$|I_1| \geq |I_2| + 1$

$\vdots$

$|I_{t-2}| \geq |I_{t-1}| + 1$

$|I_{t-1}| \geq |I_t| + 1 = 1$

$|I_t| = 0$

$|I_{t-3}| \geq |I_{t-2}| + 1 \geq 2 + 1 = 3$

$|I_{t-j}| \geq j$

$1 + 2 + 3 + 4 + 5 + 6$

$\frac{n}{2}(n+1)$

$$\sum_{i=1}^{n} i = \frac{(n+1)\cdot n}{2} = \frac{n^2+n}{2} = O(n^2)$$

$$n^2 \geq \frac{n^2+m}{2}$$

This implies that t cannot be bigger than n.

$$\text{Total runtime} = \sum_{i=0}^{t-1} O(|I_i|) \leq O(|I_0|) + O(|I_1|) + \dots +$$

$$O(|I_{t-1}|) \leq O(n + (n-1) + (n-2) + \dots + 2 + 1) =$$

$$= O\left(\sum_{i=1}^{n} i\right) = O(n^2)$$

FASTALG (I):

$O(n \log n)$ ↑ RUNTIME OF MERGE-SORT — Sort the intervals in I increasingly by finishing time.

— Let $I = \{I_1, \dots, I_n\}$ with $f(I_1) \leq f(I_2) \leq f(I_n)$

we set this so the first test is always true / it works also with empty interval

O(1) - Set $T \leftarrow -\infty$, $S = \emptyset$

- For $i = 1, \dots, m$ (STARTING TIME OF I_i)
  - IF $s(I_i) \geq T$
    $S = S \cup \{I_i\}$
    $T = f(I_i)$

O(1)

- RETURN S

Let $T \leftarrow f(I_1)$, $S \leftarrow \{I_1\}$

For $i = 2, 3, \dots, n$

EX: Prove that fastalg returns an optimal solution to interval scheduling.
(HINT: show that it behaves like $SELECT_H$ with H = "EARLIEST TO FINISH")

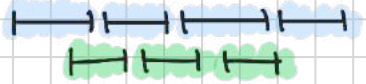
L: The runtime of fast alg. is $O(n \log n)$

P: $O(n \log n) \cdot T$ $n \cdot O(1) \leq O(n \log n)$

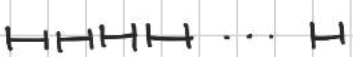
INTERVAL PARTITIONING

We are given a set of intervals I.

We aim to schedule each interval on the minimum possible number of resources

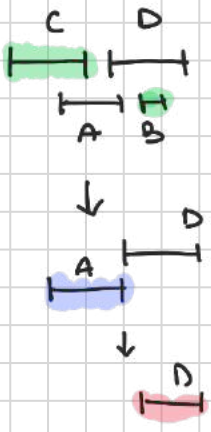


TO SCHEDULE EACH INTERVAL, I NEED 2 RESOURCES



HERE, ONE RESOURCE IS SUFFICIENT

H^* = "EARLIEST TO FINISH"



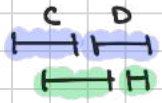
$SELECT_{H^*}$

$\{C, B\}$

$\{A\}$

$\{D\}$

$\rightarrow$ $SELECT_{H^*}$ APPLIED GREEDILY USES 3 RESOURCES

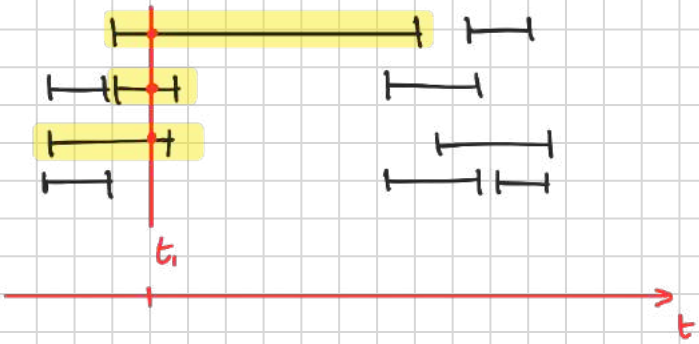


BUT 2 RESOURCES ARE OPTIMAL!

① Find the min. number of resources

② Find a schedule for them

EXAMPLE:



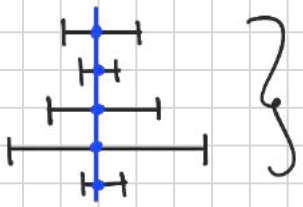
How can we run those 3 intervals at the same time if we don't have 3 resources ?

DEPTH(I') = 3

max number of intervals that can run at the same time

DEF: DEPTH (I) is the minimum integer d S.T. $\forall t \in \mathbb{R}$

$$|\{ I_i \mid I_i \in I \wedge t \in I_i \}| \leq d$$



I'

DEPTH (I') = 5

NOTE: the depth will represent the minimum resources we need but we will have to prove it first.

DEF: Let OPT(I) be the minimum number of resources to schedule each interval in I.

L: $OPT(I) \geq DEPTH(I)$

P: There must exist a time t when exactly DEPTH (I) intervals are running at the same time.
At time t, we then need DEPTH (I) resources to schedule all the intervals: $OPT(I) \geq DEPTH(I)$. ■

Algorithms

DATE: 17-03-2022

ALG(I):

- Let d be the depth(I) (n^2)
- Sort the $|I| = n$ intervals increasingly by their starting time. $O(n \log n)$
- Let $I = \{(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)\}$ with $s_1 \leq s_2 \leq s_3 \leq \dots \leq s_n$
- For $j = 1$ to n
  - $L \leftarrow \{1, 2, \dots, d\}$
  - For $i = 1$ to $j-1$

    if (s_i, f_i) is incompatible with (s_j, f_j):

    $L \leftarrow L - \{e(i)\}$
  - IF $|L| \geq 1$:

    Let $a \in L$

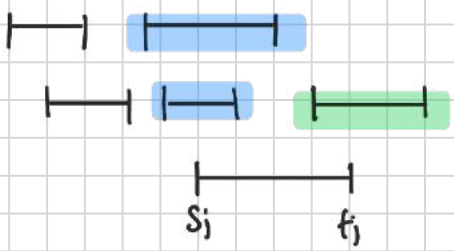
    Set $e(j) = a$
  - ELSE

    FAIL
- RETURN THE LABELING $e(1), e(2), \dots, e(n)$

L2: The algorithm never fails.

P: Consider the generic iteration of the outer loop.
Let j be the value of that loop's index j in this iteration.
Let S_j be the set of intervals that the algorithm considered before the jth interval (s_j, f_j) and that (11) end after f_j.

$$S_j = \{(s_i, f_i) \mid i \leq j-1 \text{ and } f_i \geq s_j\}$$



S_j is the set of intervals that we have already labelled and that interfere with (s_j, f_j).

Each interval in S_j will have a label that the inner loop will remove from L. No other label will be removed from L

Thus, after the inner loop,

$$|L| \geq d - |S_j|,$$

since L started with d labels.

CLAIM: $|S_j| \leq d - 1$

P: Each interval in S_j passes through the s_j, and also comes before (s_j, f_j) in the ordering.
Then $(s_j, f_j) \notin S_j$.
Suppose, by contradiction, that $|S_j| \geq d$.
Then, since $(s_j, f_j) \in S_j$, the set

$$T = S_j \cup \{(s_j, f_j)\} \text{ has cardinality } |T| \geq d+1$$

But, each interval in T passes through the s_j.

Then, $\text{DEPTH}(I) \geq |T| \geq d+1$. This contradicts $d = \text{DEPTH}(I)$ ■

But, then, after the loop, $|L| > d - |S_j| \geq 1$. Thus, $L \neq \emptyset$ and the algorithm doesn't fail ■

L3: The algorithm returns a valid labeling.

P: Suppose that (s_i, f_i) and (s_j, f_j) are overlapping intervals, with $i < j$.
Then, the label of (s_i, f_i) is chosen before the label of (s_j, f_j).
Since (s_i, f_i) and (s_j, f_j) are overlapping, the label we assign to (s_j, f_j) cannot be the label of (s_i, f_i) – indeed we removed that label from L, before picking a label for (s_j, f_j) from L.

T: The algorithm returns an optimal solution.
(or valid)

P: No feasible solution can have fewer than $\text{DEPTH}(I)$ resources (L1)
The solution returned by the algorithm uses $\text{DEPTH}(I)$ resources (L2), and it is FEASIBLE/VALID (L3) ■

ALG(I) :

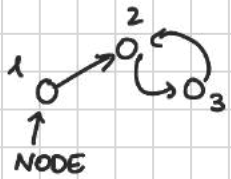
- Let d be the depth(I)
- Sort the $|I| = n$ intervals increasingly by their starting time.
- Let $I = \{(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$ with $s_1 \leq s_2 \leq s_3 \leq \dots \leq s_n$
- For $j = 1$ to n
  - $L \leftarrow \{1, 2, \dots, d\}$
  - For $i = 1$ to $j-1$
    IF (s_i, f_i) is incompatible with (s_j, f_j): // if they overlap
    $L \leftarrow L - \{e(i)\}$
  - // INVARIANT : $|L| \geq 1$
  - Let $a \in L$
  - Set $e(j) = a$
- Return the labeling $e(1), e(2), \dots, e(n)$

comment to remind $|L| \geq 1$

---

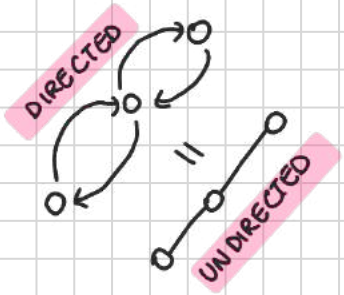
v* starting point
w* finish point

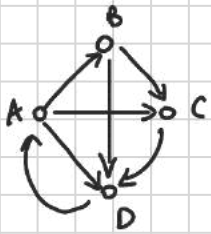
A (directed) graph $G(V, E)$ is composed of a set of vertices V and of a set of arcs E, with $E \subseteq \{(v, w) \mid v \neq w$ and $v, w \in V\}$



$V = \{1, 2, 3\}$

$E = \{(1,2), (2,3), (3,2)\}$

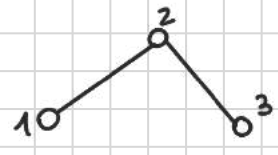




$V = \{A, B, C, D\}$

$E = \{(A,B), (B,C), (C,D), (D,A), (A,D), (A,C), (B,D)\}$

An undirected graph $G(V, E)$ is composed of a set of vertices V and of a set of edges E, with $E \subseteq \{(v, w) \mid v \neq w$ and $v, w \in V\}$

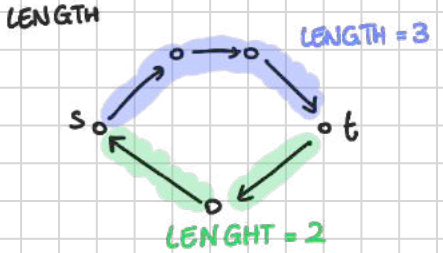


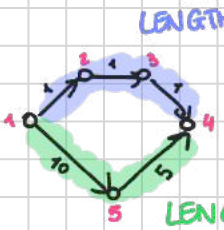
$V = \{1, 2, 3\}$

$E = \{(1,2), (2,3)\}$

A weighted directed graph is a directed graph $G(V,E)$ with a function $\ell: E \to \mathbb{R}_{\geq 0}$

(LENGTH)





(10+5 TOO MUCH PATH
1+1+1 QUICKER)

LENGTH = 15 (holy shit....)

$V = \{1,2,3,4,5\}$

$E = \{(1,2), (2,3), (3,4), (1,5), (5,4)\}$

$\ell(1,2) = 1$

$\ell(2,3) = 1$

$\ell(3,4) = 1$

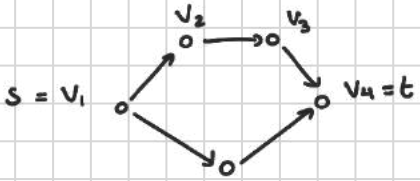
$\ell(1,5) = 10$

$\ell(5,4) = 5$

} LENGTH OF PATHS

HOW TO FIND THE SHORTEST PATH FROM A NODE "s" TO A NODE "t" IN THE GRAPH?

DEF: Given $G(V,E)$ a path from $s \in V$ to $t \in V$ is a sequence of nodes $s = v_1, v_2, v_3, \dots, v_k = t$ such that $(v_1, v_2) \in E, (v_2, v_3) \in E, \dots, (v_{k-1}, v_k) \in E$.



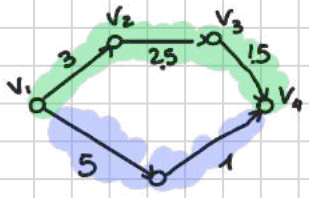
$\pi = v_1, v_2, v_3, v_4$ is a path from $v_1 = s$ to $v_4 = t$

$v_1 v_3 v_4$ is not a path

NEIN

DEF: If $G(V,E)$, ℓ is a weighted graph and if $\pi = v_1, v_2, \dots, v_k$ is a path in $G(V,E)$, then the length of π is:

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_{i+1})$$



$\pi = v_1, v_2, v_3, v_4$

$\ell(\pi) = 3 + 2{,}5 + 1{,}5 = 7$

$\pi = v_1, v_5, v_4$

$\ell(\pi) = 5 + 1 = 6$

PROBLEM: Given a weighted graph $G(V,S), \ell$, and given $s, t \in V$. What is the length of the shortest path from s to t?

E. W. DIJKSTRA'S ALGO $(G(V,E), \ell, s)$ # figures out the shortest path

// the algorithm will use the set S to denote the
// set of nodes it visited so far.
// Moreover, $\forall u \in V$, $d(u)$ will be set to the length of a
// shortest path from "s" to "u".

distance ↖ $S \leftarrow \{s\}$

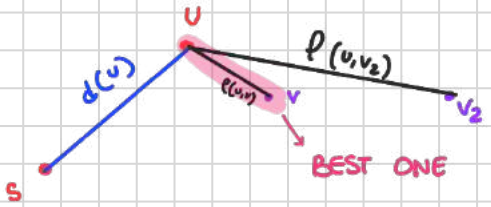
$d(s) = 0$

WHILE $S \neq V$: # visiting all vertices

- SELECT A NODE $v \in V - S$ (a node yet not visited) THAT CAN BE REACHED DIRECTLY FROM SOME NODE IN S, AND FOR WHICH

$$d'(v) = \min_{\substack{(u,v) \in E \\ u \in S}} (d(u) + \ell(u,v)) \text{ is MINIMUM}$$

- $S \leftarrow S \cup \{v\}$
- $d(v) = d'(v)$



RETURN d

BECAUSE

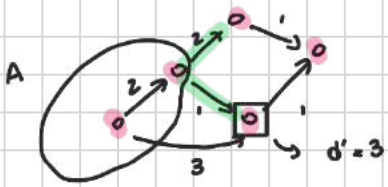
$d(u) + \ell(u,v) < d(u) + \ell(u,v_2)$

↑

THIS IS THE MINIMUM

Algorithms

DATE : 22/03/2022



DIJKSTRA'S ALGO (G(V,E) , s) :

$S \leftarrow \{s\}$ // S is the set of nodes visited so far

$d(s) = 0$ // d(v) will contain the length of a shortest path from s to v

$P_s = [s]$ // P_v is a shortest path from s to v

WHILE $S \neq V$: (set of vertices) (≤ n iterations)

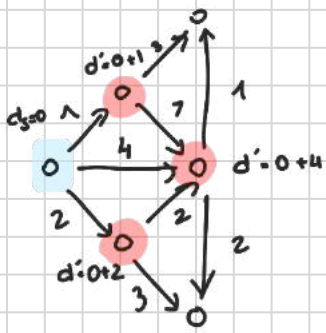
$O(n^2)$

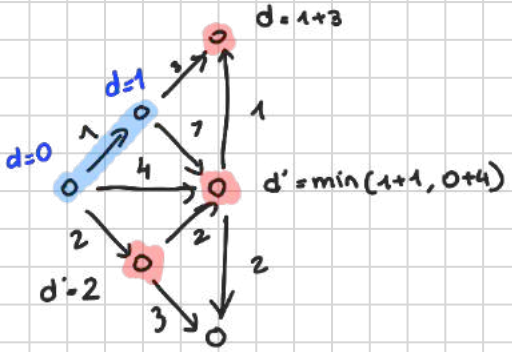
- $T = \{ w \mid w \in V-S$ AND THERE EXISTS $v \in S$ S.T. $(v,w) \in E \}$ ($O(n)$)
- $\forall w \in T$, Let $d'(w) = \min_{\substack{(u,w)\in E \\ u \in S}} (d(u) + \ell(u,w))$ ($O(n \cdot |S|)$)
- LET $v \in T$ be a node of minimum $d'(v)$ ($O(n)$) (shortest path from source to w)
- LET $u \in S$ be such that $d'(v) = d(u) + \ell(u,v)$ ($O(1)$)

NOTE: There will always be many shortest paths to choose from. The algo just provides one of the many possible solutions.

- $S \leftarrow S \cup \{v\}$
- $d(v) \leftarrow d'(v)$ ($O(1)$)
- $P_v \leftarrow P_u + [v]$

RETURN $d, P_v, P_{v_2}, \ldots, P_{v_n}$





(it doesn't necessarily need to find path from the "last" node)

① S is the set that contains the nodes visited so far.

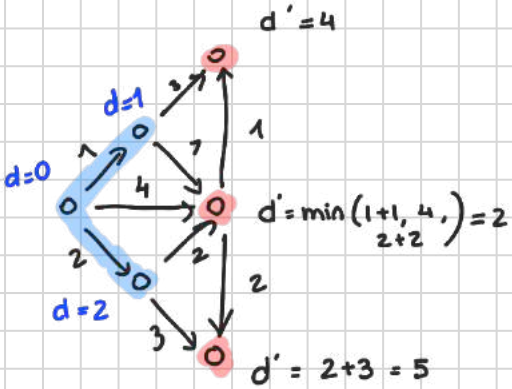
② P[s] → contains all the shortest paths

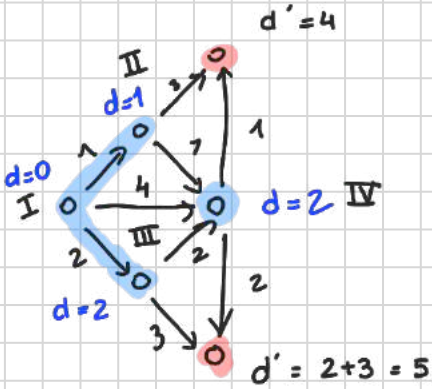
③ while $S \neq V$ → loop will go on until all nodes have been visited so far.

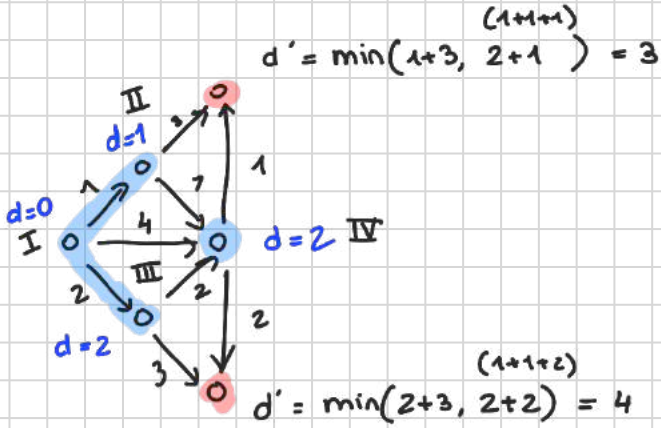
④ T is the SET that contains all the nodes that haven't been visited so far.

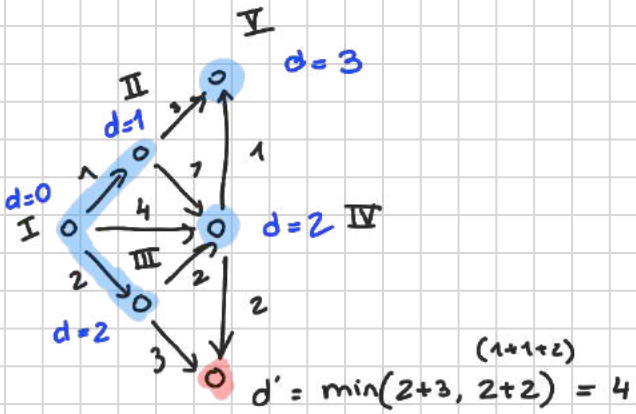
⑤ d'(w) is the minimum distance required to reach a w node by using only one step

⑥ w is a generic node (that hasn't been visited yet) while v is the









L: At any time during the execution of the algorithm, if $u \in S$, then P_u is a shortest path from s to u.

P: We prove the claim by induction on $|S|$.
If $|S| = 1$, then $S = \{s\}$, so $d(s) = 0$ and $P_s = |s|$.
The base case has been proved.

Let us suppose that the claim holds for $|S| = K$.
(we prove it for $K+1$).

In the iteration in which, at it's beginning, S has cardinality K, the algorithm will add some new node v to S, obtaining $S' = S \cup \{v\}$.
Since $v \notin S$, it holds $|S'| = K+1$.

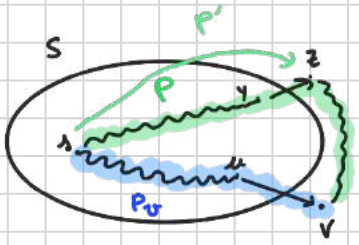
Let $u \in S$ be the node that the algorithm chose to get to v then, $P_v = P_u + [v]$

By the inductive hypothesis, since $u \in S$, P_u is a shortest path to u.

We would like to prove that P_v is a shortest path to v. Clearly, P_v is a path to v.

Consider any other path P from s to v.
Since $v \notin S$, P must leave S at some point.

Let z be the first node of P that is not in S and Let y be the node preceding z in P.



We want to prove that P cannot be shorter than P_v. We will prove this by showing that the prefix P' of P that goes from s to z is not shorter than P_v.

In iteration $k+1$, the algorithm considered adding z to S via the edge (y, z). The algorithm has also rejected this choice, in favor of v.

Thus, there exists no path from s to z that is shorter than P_v.

Since edge-lengths are non-negative, the path P (which includes P') cannot be shorter than P_v.
Thus, P_v is a shortest path from s to v.
and the induction policy has been proved ■

COR: DIJKSTRA'S algorithm finds all the shortest paths from s.
↓
COROLLARY

THUS, shortest paths can be found in polynomial time (we showed $O(n^3)$ time).

Next time, we will introduce a data structure (called "priority queue", or "heap") that will allow us to implement DIJKSTRA'S algo in $O(m \log n)$ time

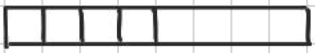
Algorithms

DATE: 24/03/2022

PRIORITY QUEUES / HEAPS

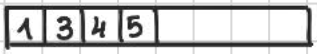
A heap makes it possibee for you to store N (key, value) pairs, where N is to be fixed a priori, in such a way that

- The item with the lowest key can be accessed in O(1) time;
- The item in position "i" can be removed from the heap in O(logn) time;
- A new item can be inserted into the heap in O(log n) time.



ARRAY

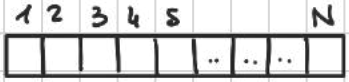
- Removing and adding takes O(1) time
- Finding the minimum takes Ω(n) time

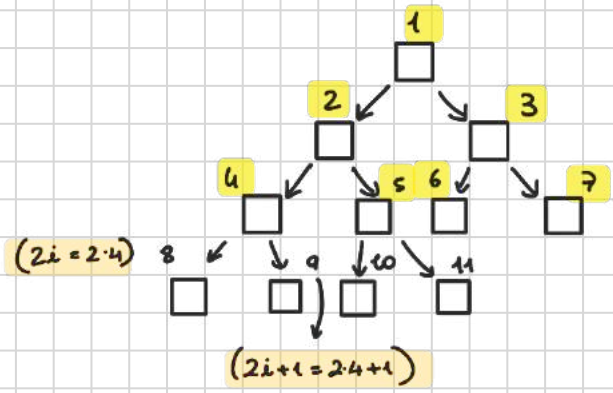


SORTED ARRAY

- Finding the minimum takes O(1) time.
- Remove and adding takes Ω(n) time.

Heaps "interpolate" between the two extremes



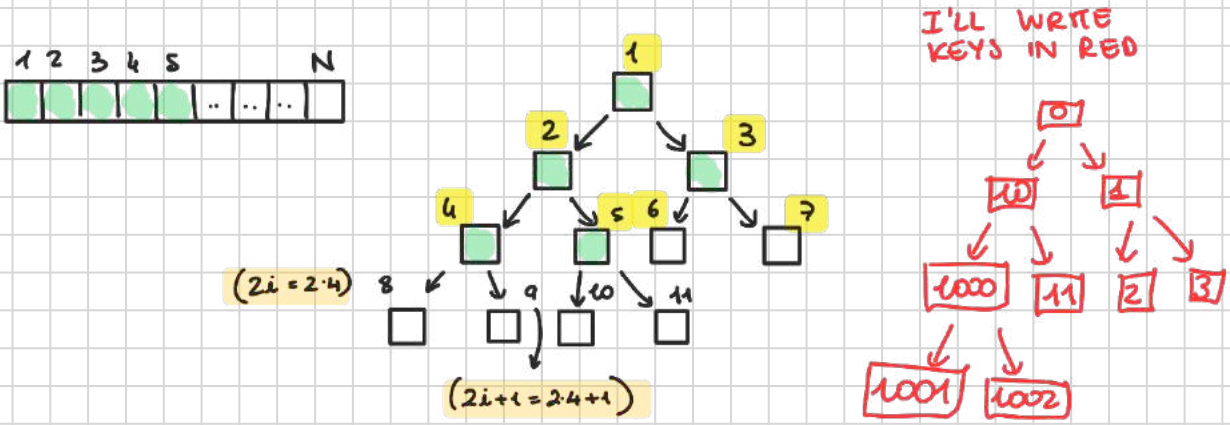


This "logical" representation of the array has a number of useful properties

- The left child of node "i" has index/position $2i \triangleq left(i)$
- The right child of node i has index "$2i+1$" $\triangleq$ right(i)
- The parent of node i has index $\lfloor \frac{i}{2} \rfloor \triangleq parent(i)$

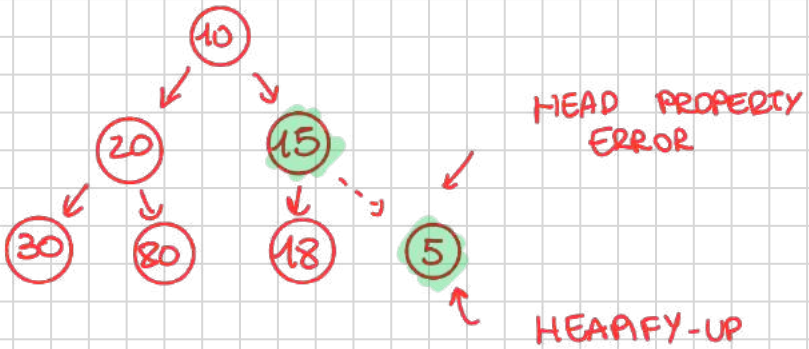
We will be representing this binary tree with an array.

If at some point, the heap only contains $n \leq N$ items then only the first "n" positions of the array will be filled



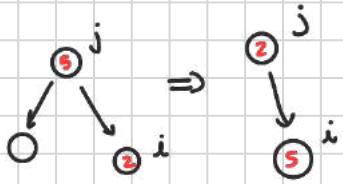
HEAP PROPERTY:

A heap satisfies the HEAP PROPERTY if, for each filled position i in the heap, if "v" is the item in position i, and if w is the item in position parent(i) $(= \lfloor \frac{i}{2} \rfloor)$, then KEY [w] $\leq$ KEY [v]

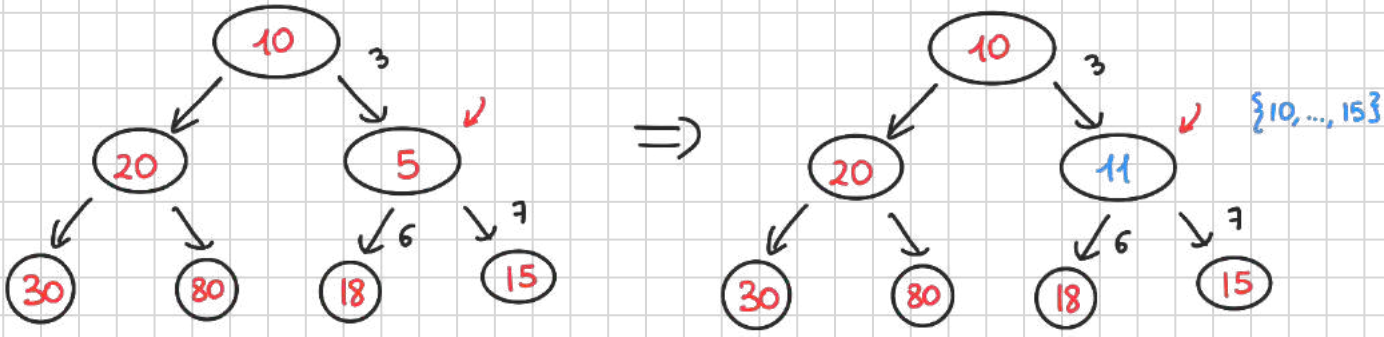


HEAPIFY-UP(H, i)

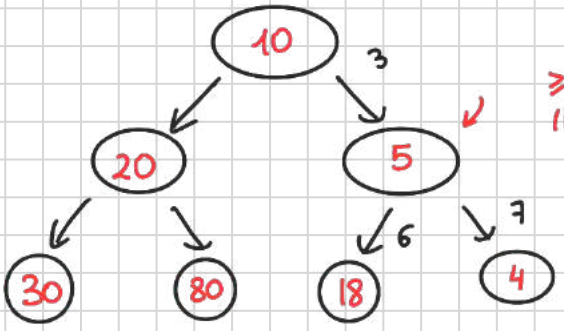
```
// This Function will try to "push up" the error
// in position i

IF i ≥ 2:
    // i has a parent
    j = ⌊i/2⌋
    IF KEY[H[i]] < KEY[H[j]]:
        // the heap property fails at i
        SWAP H[i] with H[j]
        HEAPIFY-UP(H, j)
```



DEF: We say that H is an almost-heap-with-H[i]-too-small if it is possible to increase the key of H[i] to a value $\alpha \geq$ KEY[H[i]], so that the resulting structure has the heap property.



This H has the almost-head-with-H[3]-too small prop.



≥10 AND ≤4
IMPOSSIBLE TO FIX

THIS HEAP DOES NOT HAVE THE ALMOST-HEAP-WITH-H[3]-TOO-SMALL PROPERTY

L: The function HEAPIFY-UP(H, i) fixes the heap property of H (provided it needs fixing), if H has the ALMOST-HEAP-WITH-H[i]-TOO-SMALL PROPERTY.

The runtime of HEAPIFY-UP(H, i) is $O(\log n)$.

P: We prove the claim by induction on i.

If $i=1$, the claim is true since H already has the HEAP-PROPERTY (i has no parent).

If $i \geq 2$, then the algorithm swaps $H[i]$ with $H[j]$ where $j = \text{parent}(i) = \lfloor \frac{i}{2} \rfloor$ if $\text{KEY}[H[i]] < \text{KEY}[H[j]]$.

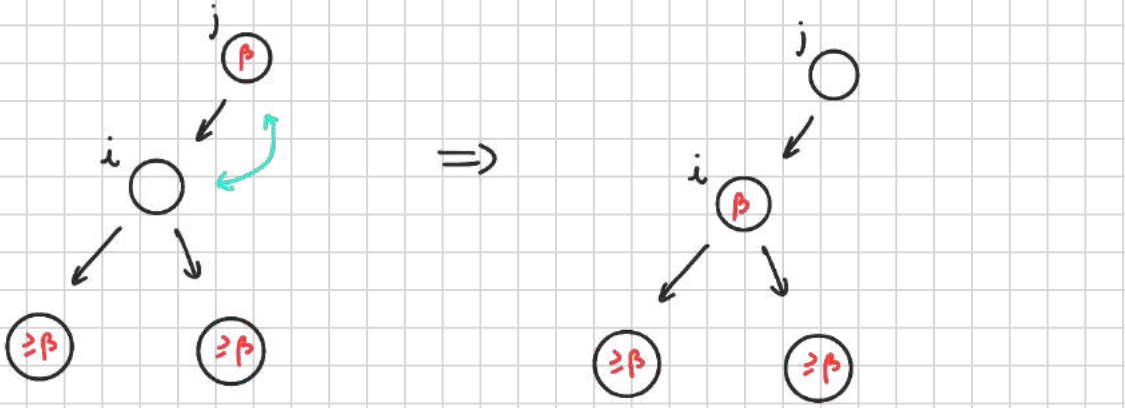
Let β be equal to $\beta = \text{KEY}[H[j]]$, <u>before</u> the swap

Because of the A-H-W-H[i]-T-S property $\exists\, \alpha \geq \text{KEY}[H[i]]$ such that if the key of $H[i]$ becomes α, the head is fixed.

But, then, such a α has to satisfy $\alpha \geq \beta$, i.e., α cannot be smaller than the key of the parent of i, and, moreover, α has to satisfy

$$\alpha \leq \text{KEY}[H[\text{left}(i)]] \text{ AND } \alpha \leq \text{KEY}[H[\text{right}(i)]]$$

Thus, $\beta \leq \alpha \leq \min(\text{KEY}[H[\text{left}(i)]], \text{KEY}[H[\text{right}(i)]])$



Thus, after the swap, left(i) will have a key not smaller than that of i. Same is true for right(i). Since the algorithm made the swap, β was larger than the key of $H[i]$. After the swap, then, the key of i will be larger than the key of $j = \text{parent}(i)$.

Then, after the swap, the heap is going to have the A-H-W-H[j]-T-S PROPERTY.

Since $j = \lfloor \frac{i}{2} \rfloor$, we have $j < i$. And the claim for $j < 1$ is true by induction.

As for the runtime, each call to HEAPIFY-UP takes $O(1)$ time plus, possibly, an extra call to HEAPIFLY-UP.
Since the number of calls is bounded by $O(\log n)$, the height of the tree, the runtime is $O(\log n)$ ■

Using HEAPIFY-UP, we can easily add/insert items to the heap.

ADD (h, v, n)

// n is the number of items currently in the heap

IF $n \leq N - 1$: // then the HEAP has enough space for v

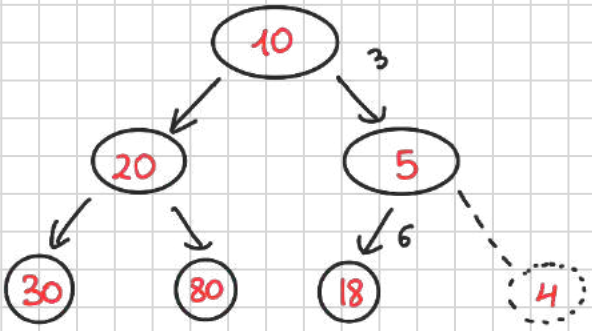
$n += 1$

$H[n] = v$

// now the HEAP might not satisfy the HEAP PROPERTY

// But the HEAP satisfies the ALMOST-HEAP-WITH-H[n]-TOO-SMALL Prop.

HEAPIFY-UP(H, n)



ADD 4

THM : ADD adds an item to the HEAP, while keeping the HEAP Property, in time $O(\log n)$

DEF: We say that H is an almost-heap-with H[i]-too-large if $\exists\ \alpha \leq$ KEY[H[i]] s.t. if we decrease KEY[H[i]] to α, then H satisfies the HEAP property.

HEAPIFY-DOWN(H,i):

```
# i is a position in the heap, 1 ≤ i ≤ n
LET n BE THE CURRENT SIZE OF THE HEAP
IF 2i > n:
    # i has no left, and no right, chiled
    RETURN
ELIF 2i == n:
    # i has onley the left chiled
    j = 2i
ELSE:
    # i has both children
    IF KEY[H[right(i)]] < KEY[H[left(i)]]:
        j = right(i)
    ELSE:
        j = left(i)

IF KEY[H[j]] < KEY[H[i]]:
    SWAP H[i] AND H[j]
    HEAPIFY-DOWN(H,j)
```

Algorithms

DATE: 29/03/2022

```
HEAPIFY-DOWN(H, i):
    # i is a position in the heap, 1 ≤ i ≤ n
    LET n BE THE CURRENT SIZE OF THE HEAP
    IF 2i > n:        // think of the key, not value
        # i has no left, and no right, child
        RETURN
    ELIF 2i == n:
        # i has only the left child
        j = 2i
    ELSE:
        # i has both children
        IF KEY[H[right(i)]] < KEY[H[left(i)]]:
            j = right(i)
        ELSE:
            j = left(i)
    IF KEY[H[j]] < KEY[H[i]]:
        SWAP H[i] AND H[j]
        HEAPIFY-DOWN(H, j)
```

L: The function HEAPIFY-DOWN(H, i) fixes the heap property of H, provided that H has the ALMOST-HEAP-WITH-H[i]-TOO-LARGE property, in time O(log n)

P: We prove the claim by reverse induction (we start from i=n, and we move from i to i-1)

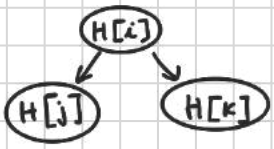
In general, if $i > 2n$, then i has no children, thus the A-H-W-H[i]-T-L property implies the heap property. Thus, the claim holds for $i > 2n$.

Otherwise, $i \leq 2n$. If the algorithm swaps $H[i]$ with $H[j]$, then - at the outset - $\text{KEY}[H[j]] \leq \text{KEY}[H[i]]$ ①

(If the algorithm does not swap, then the heap property holds.)

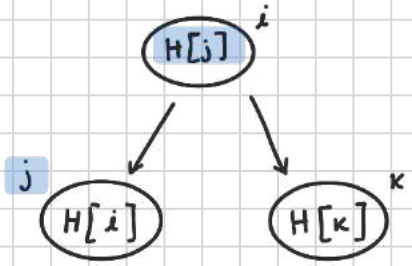
If the algorithm decided to swap $H[j]$ with $H[i]$, then the algorithm chose j as the child of i to consider

Thus, if k is the sibling of j, then $\text{KEY}[H[j]] \leq \text{KEY}[H[i]]$

IF we start from 

, then $\text{KEY}[H[j]] \leq \min(\text{KEY}[H[i]], \text{KEY}[H[k]])$

and the algorithm will swap $H[i]$ and $H[j]$, obtaining



Thus, the heap property now holds at i.

After the swap, the heap satisfies the ALMOST-HEAP-WITH-H[i]-TOO-LARGE property.

Since the number of levels is $O(\log n)$, and since each call takes $O(1)$ time, plus possibly the time for a new call at a lower level, the runtime is $O(\log n)$ ■

REMOVE (H, i) :

```
LET n be the current number of items in H
SWAP H[i] WITH H[n]                                   O(1)
H[n] = NONE                                           O(1)
n -= 1                                                O(1)
IF KEY[H[i]] < KEY[H[parent(i)]]:
    // ALMOST-HEAP-WITH-H[i]-TOO-SMALL
    HEAPIFY-UP(H, i)                                  O(log n)
```

ELIF KEY[H[i]] > min (KEY[H[left(i)]], KEY[H[right(i)]]: O(1)

// ALMOST-HEAP-WITH-H[i]-TOO-LARGE:

HEAPIFY-DOWN(H, i) O(log n)

THM: REMOVE(H, i) removes the item in position i, while keeping the heap property of H, in O(log n) time.

HEAPS:

- ADD(H, v) takes O(log n) TIME;
- FINDMIN(H) = O(1) = ; // the minimum will be in position 1 because of the heap property.

```
DEF FINDMIN(H):
    RETURN H[1]
```

- REMOVE(H, i) = O(log n) = ;
- EXTRACTMIN(H) removes and returns the minimum of the heap

```
DEF EXTRACTMIN(H):
    v = FINDMIN(H)
    REMOVE(H, n)
    RETURN v
```

WITH THESE FUNCTIONS, we can already sort an array in O(n log n) time.

DEF HEAPSORT(V):

LET V BE AN ARRAY OF N ELEMENTS

INITIALISE A HEAP WITH N POSITIONS

FOR $i = 0, 1, .., N-1$

LET x BE S.T. KEY[x] = V[i] AND VALUE[x] = V[i]

ADD(H, V[i])

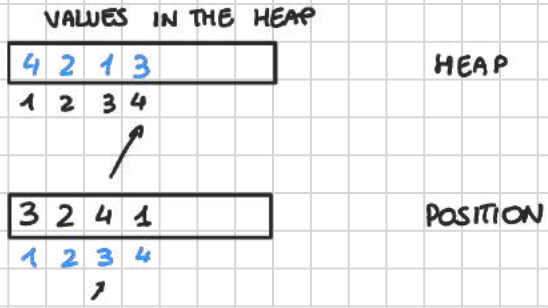
```
FOR i = 0, 1, .. , N-1
    V[i] = KEY[EXTRAMIN(H)]
RETURN V
```

THM: HEAPSORT sorts an array with N items in time $O(N \log N)$

One could want to be able to access items in the heap by value. In order to do so, values should be unique.

Suppose that the values form a subset of $\{1, 2, \dots, N\}$.

IF we use a vector/array V (POSITION) that assigns to the generic value v the position of v in the heap, one gets a good solution.



THUS, POSITION[VALUE[v]] = i IF and only if VALUE[H[i]] = v.

- REMOVEV(H, value) REMOVES the (UNIQUE) ELEMENT of VALUE "value" in time $O(\log n)$

```
DEF REMOVEV(H, v):
    REMOVE(H, POSITION[v])
    POSITION[v] = NONE
```

Observe that, for everything to work out correctly, "Position" should be updated whenever we modify the heap.

We modify the heap in 3 ways:

- ADD AN ELEMENT ;
- REMOVE = = ;
- SWAP TWO = ;

Each of these operations requires no more than updating 2 entries of "position". Thus, the extra work is only O(1) - TIME - The runtime of each of the functions remain same.

Algorithms

DATE: 31/03/2022

v = (key, value)

PRIORITY QUEUES / HEAPS

- ADD (H, v) takes O(logn) time;
- FINDMIN (H) = O(1) // ;

  // returns one item of the heap with smallest key

- REMOVE (H, i) takes O(logn) time;

  // removes item in position i

- EXTRACTMIN (H) takes O(logn) time;

  // removes, and returns, an item with smallest key (that is, the item in position 1)

```
v = FINDMIN(H)
REMOVE(H, 0)
RETURN v
```

- REMOVE V(H, value) takes O(logn) time;

  // removes the item in the heap having value equal to "value" (THIS WORKS ONLY IF ALL VALUES IN THE HEAP ARE DISTINCT)

- UPDATEKEY (H, value, newkey) takes O(logn) time;

  // changes the key of the item in the heap having value "value" to "newkey"

  (THIS WORKS ONLY IF ALL THE VALUES IN THE HEAP ARE DISTINCT)

```
(key, value) = H[POSITION[value]]
REMOVE V(H, value)
ADD (H, (newkey, value))
```

With HEAPSORT we can sort an array of n elements in $O(n \log n)$ time.

DIJKSTRA'S ALGORITHM $(G(V,E), w, s)$

// $V = \{0, 1, 2, \ldots, N-1\}$

① $O(N)$

$O(N)$ INITIALIZE A HEAP OF SIZE N

$O(\log N)$ H.ADD((0, s)) // (KEY, VALUE) WITH KEY = 0 AND VALUE = s

$O(N)$ d = [NONE] * N

$O(1)$ d[s] = 0

} $O(N)$

// IN THE END, d[i] is going to contain the length
// of a shortest path from s to i

FOR EACH OUT-NEIGHBOR u of s:

ADD (H, (H(1, u), 1))

FOR i = 1 TO N-1 (we start from 2 because we already started from 1 node (s))

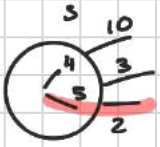
② $O(N \log N)$

$O(\log N)$ NEXT = EXTRACTMIN(H)

$O(1)$ v = NEXT.value

$O(1)$ d[v] = NEXT.key

} THESE ARE RUN FOR N-1 TIMES



③ $O(\deg(v) \log N)$

FOR EACH OUT-NEIGHBOR u OF v:

IF d[u] = NONE: $O(1)$

// we haven't yet selected/visited "u"

distance

dist = d[v] + w(v, u) $O(1)$

IF H[POSITION[u]].KEY > dist: $O(1)$

UPDATEKEY(H, u, dist) $O(\log N)$

} THIS IS RUN FOR degree of v ($\deg(v)$) times

$\deg(v) = 3$

RETURN d

deg - n° nodes/neighbours

(1) (2)

TOTAL RUNTIME $= O(N) + O(N \log N) + \sum_{v \in V} O(\deg(v) \log N) =$

$= O(N \log N) + \left(\sum_{v \in V} \deg(v)\right) \cdot O(\log N) =$

$= O(N \log N) + 2M \cdot O(\log N)$

$= O((N+M) \log N)$

L: IF $G(V,E)$ is a graph, then $\sum_{v \in V} \deg(V) = 2|E|$

P: Recall that $E \subseteq \{\{(u,v)\} \mid u, v \in V, \text{ and } u \neq v\}$

Moreover, the set of neighbours $N(v)$ of "v" is equal to $N(v) = \{u \mid \{u,v\} \in E\}$.

And $\deg(v) = |N(v)|$

Let us also define the set of edges.

Incident on v, $I(v) = \{\{u,v\} \mid v \in \{u,v\} \text{ AND } \{w,v\} \neq E\}$

12, 15
21, 24
35
42, 45
51, 53, 54

$N(2) = \{1,4\}$
$I(2) = \{\{1,2\}, \{2,4\}\}$

Then $\deg(v) = |I(v)|$

$\sum_{v \in V} \deg(v) = \sum_{v \in V} |I(v)| = \sum_{e \in E} 2 = 2|E|$ ■

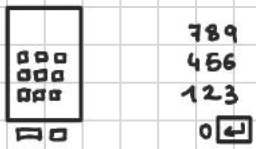
QUESTION:

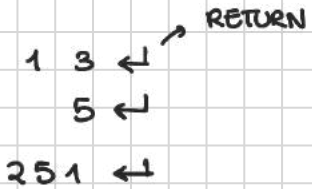
Suppose that V is an array of N elements V[i] is the score that student i got in the algorithm class (scores are "PASS": 1, "INSUFFICIENT": 0, "HONORS": 2)

Sort V increasingly as fast as you can.

Try greedy approach when they ask about interval scheduling

SALOON





Let's say there is a gunfire and you lose your numbers (keys). how can you keep inserting your entries?

If you have 3 keys, then you can write integers in binary (0,1 FOR AN INTEGERY, ↵ TO SPLIT INTEG.)

If you have only 2 keys, I could write in unary.

3 AND 5

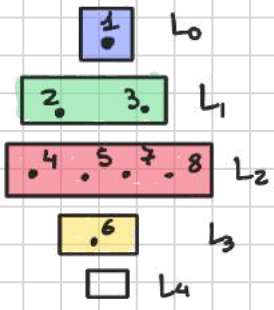
11101111110

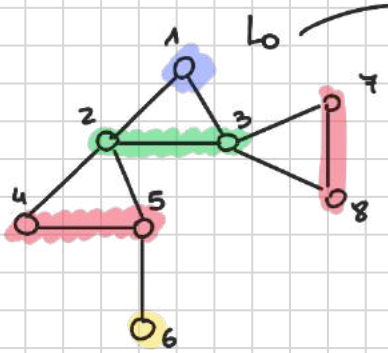
What if you only have the key "zero"?

(Personally, I think we should close down the saloon)

BFS (BREADTH FIRST SEARCH)

DFS (DEPTH = =)





Layer 0
Layer 1 (Green) reachable from L_0
Layer 2 (Red) = from L_1
Layer 3 (Yellow) = = L_2

BFS (G(V,E), s)

```
S ← {s}    // S is the set of visited nodes
L0 ← {s}   // Li will contain all the nodes at DISTANCE
i ← 0      // i from s
```

WHILE TRUE:

$L_{i+1} \leftarrow \emptyset$

WHILE $\exists v \in L_i$ AND $w \in V - S$ S.T. $\{v, w\} \in E$:

$L_{i+1} \leftarrow L_{i+1} \cup \{w\}$

$S \leftarrow S \cup \{w\}$ $O(n+m)$

IF $L_{i+1} = \emptyset$:

BREAK

ELSE:

$i \leftarrow i+1$

RETURN $S, (L_0, L_1, \ldots, L_i, \ldots)$ GREEN BECAUSE ITS OPTIONAL IF YOU WANT TO RETURN THIS OR NOT

NOTE: Works better than DIJSKSTRA but works only with unweighted graphs

DFS (G(V,E), v):

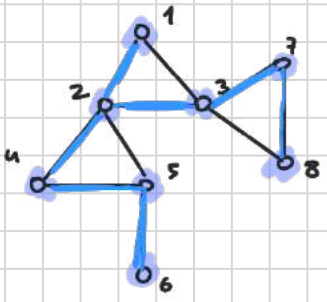
"MARK" v as EXPLORED

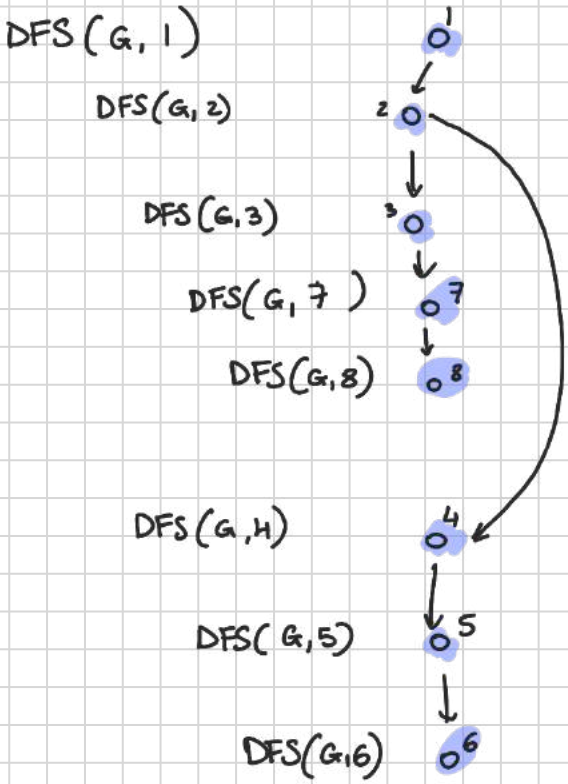
FOR EACH NEIGHBOUR w OF v: $O(n+m)$

IF w HAS NOT BEEN MARKED AS EXPLORED (YET):

DFS (G(V,E), w)

G(V,E)





← we randomly chose 3, we could have chosen 4 too.

since it doesn't have any neighbor to visit, it goes back to 7
Since 7 doesn't have any neighbor, it goes back to 3.
3 has another node unmarked (4) so it goes there.

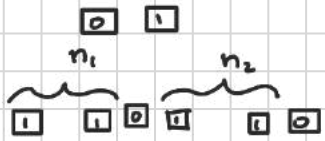
Algorithms

DATE: 05/04/2022

NETWORK DESIGN PROBLEM

[1]

- $n_1, n_2, \dots, n_K, \dots$



THM: IF $n \geq 2$ is an integer, then there exists a unique sequence of prime numbers $2 \leq p_1 < p_2 < p_3 < \dots < p_K$ and a unique sequence of natural numbers $n_1 \geq 1$, $n_2 \geq 1, \dots, n_K \geq 1$ S.T.

$$n = \prod_{i=1}^{K} p_i^{n_i}$$

- $15 = 3 \cdot 5 = 3^1 \cdot 5^1$
  $p_1 = 3 \quad p_2 = 5$
  $n_1 = 1 \quad n_2 = 1$
- $20 = 2^2 \cdot 5^1$
  $p_1 = 2 \quad p_2 = 5$
  $n_1 = 2 \quad n_2 = 1$

$2 = P_1, \quad 3 = P_2, \quad 5 = P_3 \dots$

P_i is the iTH prime number

$p_1^{n_1} p_2^{n_2} \dots p_K^{n_K} = N_K$

$n_1, n_2, n_3, \dots, n_K \Rightarrow$ [1] ... [1]

ADDS $n_{K+1} \Rightarrow p_1^{n_1} p_2^{n_2} \dots p_K^{n_K} p_{K+1}^{n_{K+1}} = N_{K+1}$

$N_{K+1} \geq N_K$

H11 [1] $N_{K+1} - N_K$ TIMES

How many digits to represent $n_1, n_2, \dots, n_K$?

$$\sum_{i=1}^{K} (n_i + 1) = \sum_{i=1}^{K} n_K + K$$

"$n_i + 1$ bits to represent n_i"

"in binary we use $O(\log n_i)$ bits to represent n_i"

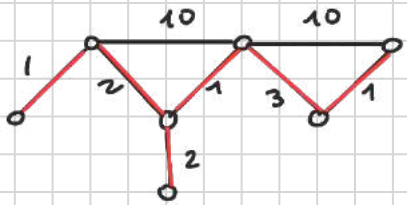
$n_i \xrightarrow{\text{base 2}} b_1, b_2, \dots, b_K$

$0b_1\ 0b_2\ 0b_3\ 0b_K$

$2\log_2 n_i = 2K$
$3\log_2 n_i = 3K$

$2\log_2 n_i$
$\log n_i + O(\log \log n_i)$

NETWORK DESIGN PROBLEM



$G(V,E)$ is a weighted, connected, graph.

We have $V = \{v_1, v_2, \ldots, v_n\}$ locations.

Some pairs of locations, those in E, can be directly linked. Directly linking $\{v_i, v_j\} \in E$ costs $w(v_i, v_j) > 0$.

Assuming G is connected, what is the minimum price for indirectly connecting each pair of nodes of V?

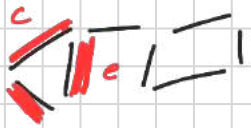
We aim to find subset $T \subseteq E$ of the edges so that:

- $G(V,T)$ is connected, and
- The cost of T, $cost(T) = \sum_{e \in T} c(e)$, is minimum.

L: Let T be an optimal solution to the network design problem. Then, $G(V,T)$ is a tree.

P: By definition, $G(V,T)$ has to be connected. We will show that $G(V,T)$ cannot contain cycles - thus, it has to be a tree (a connected graph with no cycles is a tree).

By contradiction, suppose that $G(V,T)$ contains a cycle C. Let "e" be any edge of the cycle.



$G(V, T - \{e\})$ is also connected,

since any path that went through the edge "e" can be rerouted through $C - \{e\}$.
Thus, $T - \{e\}$ is a valid (FEASIBLE) solution to the network design problem. (you can go from any node to any other node — that is, $G(V, T - \{e\})$ is connected).

The cost of this new solution is:

$$COST(T-\{e\}) = \sum_{e' \in T-\{e\}} c(e') = \left(\sum_{c' \in T} c(e')\right) - c(e) = COST(T) - c(e)$$

Recall that $c(e) > 0$, thus the $cost(T-\{e\}) = COST(T) - c(e) < COST(T)$. Thus, T is not an opt. solution.
CONTRADICTION ■

NOTE: This is also known as minimum spanning tree algo.

Thus, the network design problem is actually asking to find a subtree of $G(V, E)$ of minimum cost, and that connects each pair of vertices.

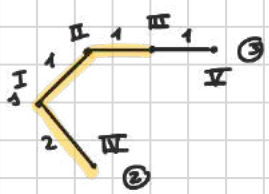
The latter problem is known as MINIMUM SPANNING TREE (or MST)

GREEDY APPROACHES?

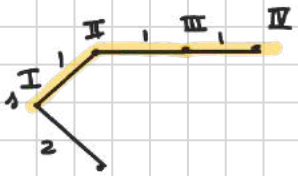
① PRIM'S ALGORITHM.

You start from an arbitrary node s. Let $S \leftarrow \{s\}$.

To select a new edge, we pick one having smallest cost, among those that take us from some node in S to some node in V-S.

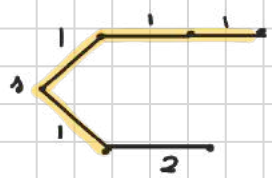


DIJKSTRA



PRIM

chooses edges that don't create cycles.



② KRUSKAL'S ALGORITHM

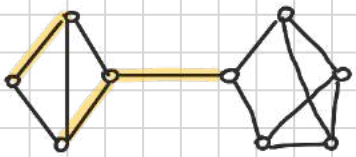
Sort the edges increasingly by cost. Let $T \leftarrow \emptyset$. Scan the list of edges e:

If "e" can be added to T w/o (without) creating cycles, add "e" to T.

③ REVERSE - KRUSKAL

Sort the edges decreasingly by cost. Scan the list of edges "e": IF "e" is part of a cycle (in the current graph), throw "e" away, o/w (otherwise) ADD "e" to T.

When is it "safe" to add an edge to a spanning tree T?

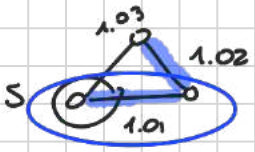


L: Assume that edge cost are pairwise distinct.

Let $\emptyset \subset S \subset V$ be a set of Nodes of $G(V, E)$.

Let $e \in E$ be an edge having smallest cost among the edges having one endpoint in S, and one in V-S.

Then, EACH MST of $G(V, E)$ contains "e".



S is the minimum difference between distinct edges costs in G(V,E)

Algorithms

DATE: 7/4/2022

L: Assume that edge cost are pairwise distinct.

Let $\emptyset \subset S \subset V$ be a set of Nodes of $G(V, E)$.

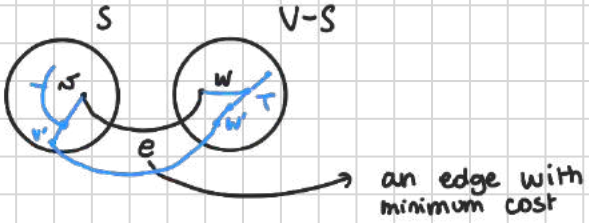
Let $e \in E$ be an edge having smallest cost among the edges having one endpoint in S, and one in $V-S$.

Then, EACH MST of $G(V, E)$ contains "e".

P: Let T be a spanning tree that does not contain the edge "e". We show that T is not a MST.

Let $e = \{v, w\}$, and suppose that $v \in S$ then $w \in V-S$. Since T is a spanning tree, there must exist a path π from v to w in T.

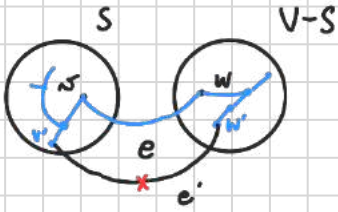
Let w' be the first node of π that is in $V-S$, and let v' be the one node preceding w' in π.



Then, $v' \in S$, o/w w' would not be the first node of π in

Let $e' = \{v', w'\}$.

Let us consider the set $T' = T - \{e'\} \cup \{e\}$.



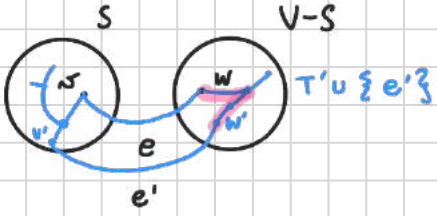
We want to prove that T' is a tree, and that its cost is smaller than the cost of T (this will entail that T is NOT a MST).

Observe that $G(V, T')$ is connected: $G(V, T)$ is connected (T is a spanning tree), and any path

in $G(V,T)$ that uses the edge e' can be rerouted on edges of T':

- we can first go through the portion of the path that goes from v' to v, then
- we can go through the new edge e,
- we can go from w to w'.

Thus, $G(V,T')$ is connected. It is also acyclic. Indeed, the only cycle in $G(V,T' \cup \{e'\})$ is the one going through "e" and "e'" but e' is not in $G(V,T')$. Thus, $G(V,T')$ does not have cycles.



Thus, $G(V,T')$ is a tree. We prove that the cost of T' is strictly smaller than the cost of T.
$T' = T - \{e'\} \cup \{e\}$. Thus, the $COST(T')$ of T' is equal to the $COST(T)$ of T minus $COST(e')$ plus $COST(e)$:

$$COST(T') = \sum_{f \in T'} cost(f) = \left(\sum_{f \in T} COST(f) \right) - cost(e') + cost(e)$$

but, $COST(e) < COST(e')$ since e is the edge with one endpoint in S, and one in V-S, having the SMALLEST COST in the set of edges having one endpoint in S and one in V-S and, also, e' has one endpoint in S and one in V-S.

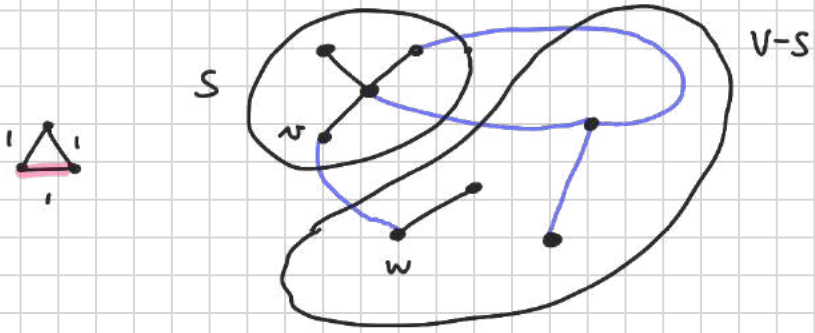
Thus, $COST(e') > COST(e)$ since there are no two edges having the same cost.

Thus, $Cost(T') < COST(T)$, and T is not a MST ■

THM: Assume that edge costs are pairwise different. KRUSKAL Algorithm produces a MST.

P: Suppose that, in a given iteration, Kruskal adds $\{v, w\}$ to T.

Let S be the set of nodes reachable from v, before adding $\{v, w\}$ to T.



Then $v \in S$, and $w \notin S$. Otherwise, the edge $\{v, w\}$ would create a cycle (and, by contradiction, Kruskal never creates cycles).

But then, $v \in S$ and $w \in V-S$. Moreover $\{v, w\}$ is the cheapest edge in the $S, V-S$ cut.

By the previous lemma, then, the edge $\{v, w\}$ is part of each MST. Thus, $T \cup \{\{v, w\}\}$ is still a partial optimal solution. The greedy property is then satisfied.)

Thus, the output of Kruskal algorithm is always a subset of a MST.

We prove that it is also a spanning tree - so that it must be a MST.

Well, Kruskal tries to add <u>each</u> edge "e" and avoids adding "e" only if it induces a cycle.
Thus, given that the $G(V, E)$ is connected, the output of KRUSKAL algorithm will also be connected.
Since it is acyclic, it is a spanning tree ■

THM: Assume that edge costs are pairwise different. KRUSKAL Algorithm produces a MST.

P: Apply the lemma to the sets $S_1 = \{s\}, S_2, S_3, .., S_{n-1}$ that PRIM's algorithm considers.

At each step, PRIM selects the edge that the lemma guarantees to be in a MST ■

L: Assume that edge costs are pairwise distinct.
Let C be the a cycle in G, and let $e \in C$ be the most expensive edge of C.

Then, no MST contains e.

Lemma for REVERSED-KRUSKAL

RIDDLE

Suppose you live in a building with "n" floors. You have a bunch of NOKIA 3310 phones. You would like to test the theory that they're indestructible... or, more seriously, you would like to pinpoint the highest floor i such that throwing a 3310 from the window of floor i guarantees that the 3310 reaches the ground unscathed (so that you can reuse it).

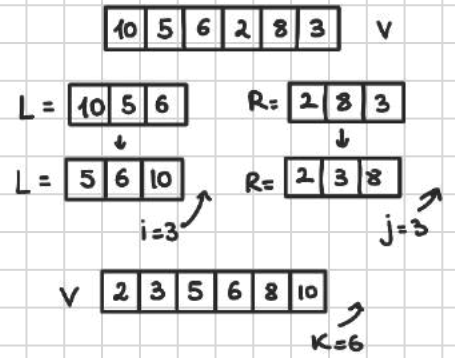
Given that you have k different NOKIA 3310 phones, what is the minimum number of tests you have to perform to find out the magic floor i?

DIVIDE-ET-IMPERA / DIVIDE-AND-CONQUER

- A TECHNIQUE FOR SPEEDING UP ALGORITHMS -

Let's start with the most famous divide-et-impera algo: (recursive)

```
DEF MergeSort (V):
    If len(V) ≤ 1:                      } O(1)
        RETURN V                        }
    L = V[ : len(V)/2 ]                 O(len(V))
    R = V[ len(V)/2 : ]                 O(len(V))
    L = MergeSort(L)                    T(len(V)/2)      RECURSION
    R = MergeSort(R)                    T(len(V)/2)      RECURSION
    i = 0                               }
    j = 0                               } O(1)
    k = 0                               }
    WHILE k < len(V):                   }
        IF i == len(L)                  }
            V[k] = R[j]                 } O(len(V))
            j += 1                      }
        ELIF j == len(R)                }
            V[k] = L[i]                 }
            i += 1                      }
```



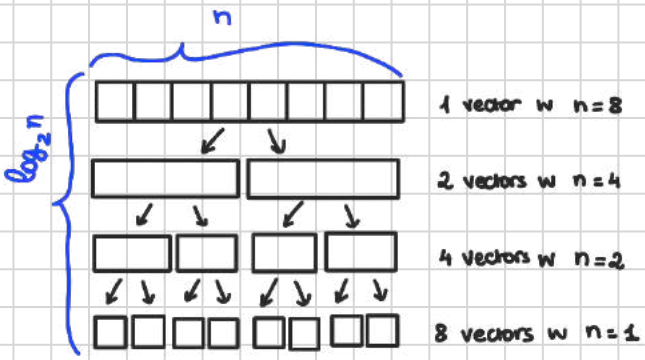
$O(len(V)) + 2T(len(V)/2)$

↑ function T

```
        ELIF L[i] ≤ R[j]:
            V[k] = L[i]
            i += 1
        ELSE:
            V[k] = R[j]
            j += 1
        k += 1
```

O(1)

```
RETURN V
```



$O(n \log n)$

Let $T(n)$ be the worst-case runtime of the algorithm on inputs of SIZE "n".

Let us assume for simplicity that $n = 2^t$ for some int. $t \geq 1$

Observe that

$$\forall n \geq 2: \quad \left.\begin{array}{l} T(n) \leq 2T(n/2) + c_n \\ T(0), T(1) \leq c \end{array}\right\} \quad \text{(FOR SOME CONSTANT } c>0\text{)}$$

This is a recurrence.

$$T(n) \leq \begin{cases} 2T(n/2) + c_n & \text{IF } n \geq 2 \\ c & \text{IF } n = 2 \end{cases} \qquad (R_2)$$

R_2 by itself does not give us a runtime bound like the "usual" ones.

To get the "usual" bounds ($O(n \log n)$ or $O(n)$ or $O(n^2)$) we need to SOLVE the recurrence.

HOW TO SOLVE RECURRENCES ?

VARIOUS APPROACHES

APPROACH 1

Unroll the recurrence for some number of levels, searching for a pattern that we can employ to solve the recurrence.

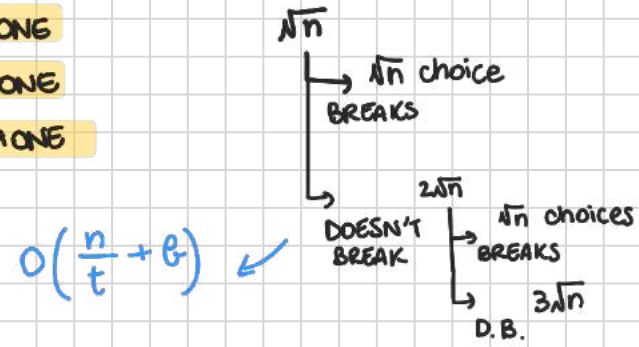
APPROACH 2

Guess the solution, and verify that it works.

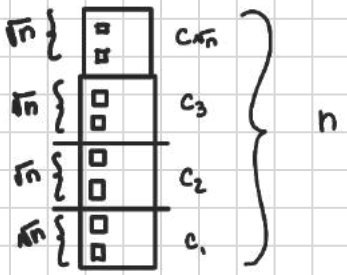
Algorithms

DATE: 12/4/2022

1 PHONE
2 PHONE
3 PHONE



$O\left(\frac{n}{t}+t\right)$

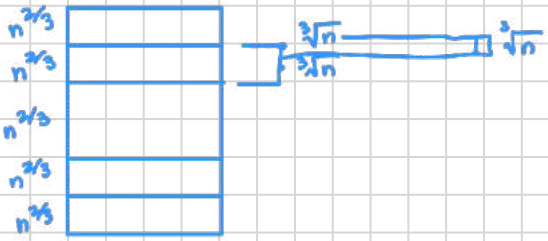


1 PHONE → $O(n)$
2 " → $O(\sqrt{n})$
3 " → $O(\sqrt[3]{n^2})$
K " → $O(K\sqrt[K]{n})$

K-1

$$\frac{n}{2^{K-1}}$$

NOTE: we throw from the last floor of the chunk



$K = \log n$ PHONES $\rightarrow O(K\sqrt[K]{n})$

$$\sqrt[K]{n} = n^{1/K} = \left(2^{\log_2 n}\right)^{1/K} =$$

$$= \left(2^{\log_2 n}\right)^{\frac{1}{\log_2 n}} = 2$$

$$O\left(\log_2 n \cdot \sqrt[\log_2 n]{n}\right) = O(\log_2 n)$$

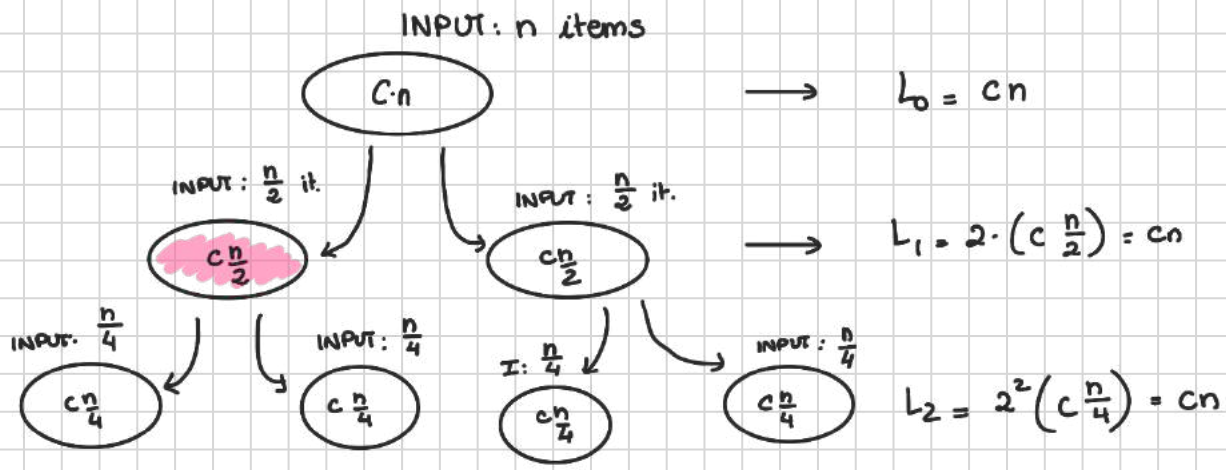
We looked at MERGESORT on $n = 2^t$, for t positive integer, elements.

We proved that the runtime of MERGESORT is $T(n)$, and it satisfies:

$$\exists\, c > 0, \quad T(n) \leq \begin{cases} 2T\left(\frac{n}{2}\right) + c_n & \forall n > 2 \\ T(n) \leq c & n \in \{0,1,2\} \end{cases}$$

APPROACH 1

Unroll the recurrence by looking at its first few steps.



In the iTH level, there will be 2^i calls. The generic call of the ith level is going to take time $\frac{cn}{2^i}$ (plus whatever its own calls will cost).

The total runtime for level "i" is going to be

$$2^i \cdot \left(\frac{cn}{2^i}\right) = cn = O(n)$$

Since there are $\log_2 n$ levels, the total runtime is going to be $O(n \log n)$

APPROACH 2

HOW TO BOUND $T(n)$?

① Guess a particular bound ($T(n) \leq n^2$, $T(n) \leq n$).

Suppose we believe (and would like to check) that $T(n) \leq a \cdot n \log_2 n$, for some constant $a > 0$.

R2

$$\exists\, c > 0, \quad T(n) \leq \begin{cases} 2T(\frac{n}{2}) + c_n & \forall n > 2 \\ T(n) \leq c & n \in \{0, 1, 2\} \end{cases}$$

The case $T(2)$ clearly holds, if $2a \geq c$,

$$T(2) \leq c$$

We want to claim that $T(2) \leq a\, 2 \log_2 2 = 2a$.
The base case holds

Suppose, now, that $n \geq 3$. By induction, we know that $T(m) \leq 2T\left(\frac{m}{2}\right) + c\,m \quad \forall m \leq n-1$.

We want to prove the inequality for n.

$$T(n) \overset{R_2}{\leq} 2T\left(\frac{n}{2}\right) + c_n$$

$$\text{I.H.} \rightarrow \leq 2\left(a\frac{n}{2}\log_2\frac{n}{2}\right) + c_n$$

(Induction Hypothesis)

$$= an(\log_2(n) - 1) + c_n$$
$$= an\log_2 n - an + cn$$
$$= an\log_2 n + (c-a)n$$

$$a \geq c \rightarrow \leq an\log_2 n \quad \checkmark$$

L: Thus, if $a \geq c$, we have that $T(n) \leq an\log_2 n$.

C: $T(n) \leq cn\log_2 n$

R2

$$\exists\, c > 0, \quad T(n) \leq \begin{cases} 2T\left(\frac{n}{2}\right) + c_n & \forall n > 2 \\ c & n \in \{0,1,2\} \end{cases}$$

```
 1 2 3 4 5 +
 6 7 1 9 3 =
-------------
 7 9 5 3 8
```

```
  100110 +
   11011 =
----------
 1000001
```

THE RUNTIME OF THIS ALGORITHM IS $O(n)$ IF YOU SUM UP TWO NUMBERS OF n DIGITS EACH

```
  324 x
  156 =
-------
 1924
1620
324
-------
50524
```

THE RUNTIME OF THIS ALGORITHM IS $O(n^2)$ IF YOU MULTIPLY TWO NUMBER OF n DIGITS EACH.

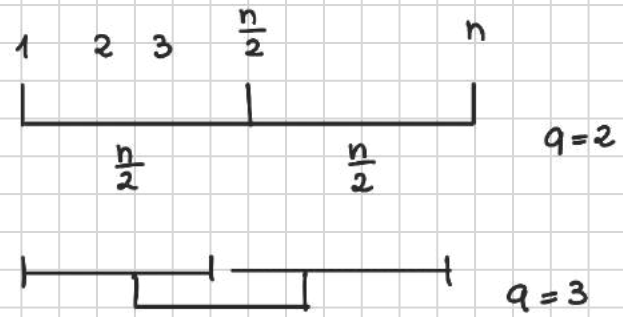
R2

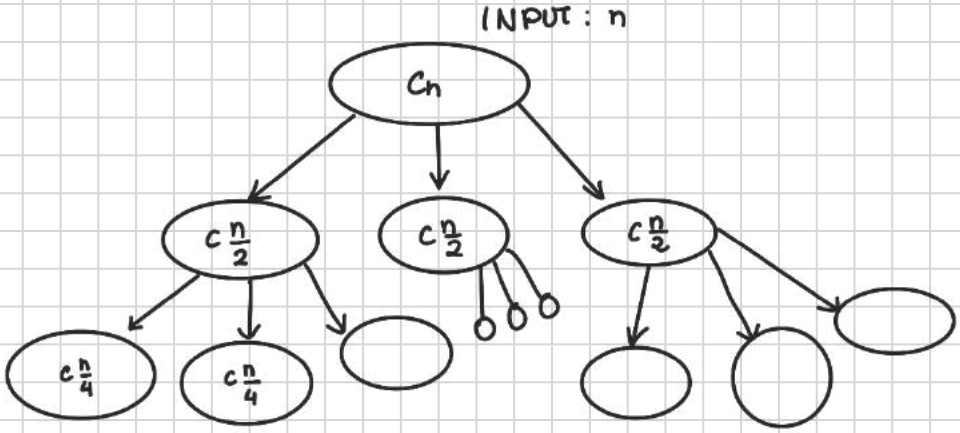
$$\exists\, c>0, \quad T(n) \le \begin{cases} 2T\left(\frac{n}{2}\right)+c_n & \forall n>2 \\ T(n) \le c & n \in \{0,1,2\} \end{cases}$$

R2

$$\exists\, c>0, \quad T_q(n) \le \begin{cases} qT\left(\frac{n}{2}\right)+cn & \forall n>2 \\ c & \text{OTHERWISE} \end{cases}$$

$q \ge 3$





It appears that in level i we have 3^i many calls and the runtime per call at level i is $c \cdot \frac{n}{2^i}$.

The proper cost at level i is $L_i \le \frac{q}{i} \cdot c\frac{n}{2^i} = \left(\frac{q}{2}\right)^i \cdot c \cdot n$

The total runtime

$$\sum_{i=0}^{\log n} L_i \le \sum_{i=0}^{\log n} \left(\left(\frac{q}{2}\right)^i \cdot cn\right)$$

Recall that $\sum_{i=0}^{t} \alpha^i = \alpha^0 + \alpha^1 + \alpha^2 + \dots + \alpha^t = \frac{\alpha^{t+1}-1}{\alpha-1}$

In our case, $\alpha = \frac{q}{2}$ and $t = \log_2 n$ thus,

$$\sum_{i=0}^{\log_2 n} L_i \leq cn \sum_{i=0}^{\log_2 n} \left(\frac{q}{2}\right)^i = cn \frac{\left(\frac{q}{2}\right)^{\log_2 n + 1} - 1}{\frac{q}{2} - 1} =$$

$$= 2cn \frac{\left(\frac{q}{2}\right)^{\log_2 n + 1} - 1}{q - 2}$$

$$= \frac{2cn}{q-2} \left(\frac{q}{2}\right)^{\log_2 n} \frac{q}{2}$$

$(c^x)^y = n^{xy} = (c^y)^x$

$$= \frac{2cn}{q-2} \left(2^{\log_2 \frac{q}{2}}\right)^{\log_2 n} \frac{q}{2}$$

$$= \frac{2cn}{q-2} \left(2^{\log_2 n}\right)^{\log_2 \frac{q}{2}} \frac{q}{2}$$

$$= \frac{2cn}{q-2} n^{\log_2 \frac{q}{2}} \frac{q}{2}$$

$$= \frac{2c}{q-2} n^{1 + (\log_2(q) - 1)} \frac{q}{2}$$

$$= \frac{qc}{q-2} n^{\log_2 q} = O(n^{\log_2 q})$$

C: The solution to R_q is the $T_q(n) \leq O(n^{\log_2 q})$

DIVIDE-ET-IMPERA / DIVIDE-AND-CONQUER ALGORITHMS

Rq $(q \geq 2)$ (RECURRENCIES)

$$T_q(n) \leq \begin{cases} q \cdot T(\frac{n}{2}) + c \cdot n & \forall n \geq 3 \\ c & n \in \{0,1,2\} \end{cases}$$

(q: parameter)

THM: $T_2(n) \leq O(n \log n)$

THM: $T_q(n) \leq O(n^{\log_2 q})$ $\forall$ INTEGER $q \geq 3$ ← different behaviour then when $q < 3$

$(T_1(n) \leq O(n))$

Thus, $T_4(n) \leq O(n^{\log_2 4}) = O(n^2)$

$T_3(n) \leq O(n^{\log_2 3}) = O(n^{1.5849\ldots})$ ← strange because we have been looking at combinatorial runtimes

$\log_2 3 = 1.5849$

INTEGER ADDITION

$n=4$

1100 +
0111 =
10011

Summing up two binary (decimal) numbers of "n" digits each takes $O(n)$ time.

INTEGER MULTIPLICATION

1100 ·
0111
1100
1100 ·
1100 ·
0000

Multiplying two binary (decimal) numbers of n digits each (using the primary school algo) takes $O(n^2)$ time.

HOW TO IMPROVE MULTIPLICATION?

Let us assume that our goal is to multiply two n digits numbers, x and y.

We would like to compute $M = x \cdot y$

To implement a divide-et-impera approach we will split x and y into two halves each.

$X = 100110$	$X_1 = 100$	$X_0 = 110$
$Y = 011010$	$Y_1 = 011$	$Y_0 = 010$

$x_1 \cdot 2^{n/2} = 100 \cdot 1000 = 100000$

$x_1 2^{n/2} + x_0 = 100000 + 110 = 100110$

Then, $x = x_1 \cdot 2^{n/2} + x_0$ and $Y = Y_1 \cdot 2^{n/2} + y_0$

$(a+b)(c+d)$

$ac + ad + bc + bd$

$$x \cdot y = (x_1 \cdot 2^{n/2} + x_0) \cdot (y_1 \cdot 2^{n/2} + y_0) =$$
$$= x_1 y_1 2^{n/2} 2^{n/2} + x_1 y_0 2^{n/2} + x_0 y_1 2^{n/2} + x_0 y_0$$
$$= x_1 y_1 2^{n} + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0$$

Thus, we have split the problem of multiplying two binary numbers of n-digits each into the problems of

1. Computing 4 products of binary numbers of $n/2$ digits each;
2. Performing 3 sums of n-digits binary numbers;
3. Performing 2 multiplications with powers-of-2

Now, 1 and 2 can be easily performed in $O(n)$ time. What about 3?

$$T_4(n) \leq 4T\left(\frac{n}{2}\right) + cn$$

If $q = 4$, we have that $T_4(n) \leq O(n^{\log_2 4}) = O(n^2)$

We didn't gain anything so far! =

If we want to do better than n^2, we need to reduce the number of subproblems to $q \leq 3$.

We aim to compute:

$$M = X_1 Y_1 \, 2^n + (X_1 Y_1 + X_0 Y_0) \, 2^{n/2} + X_0 Y_0$$

$$M_2 = (X_1 + X_0)(Y_1 + Y_0) = X_1 Y_1 + X_1 Y_0 + X_0 Y_1 + X_0 Y_0$$

$$M_1 = X_1 Y_1$$

$$M_0 = X_0 Y_0$$

↳ we are performing ONE SINGLE multiplication. We can see it gives 4 mult. later but it's still a single one.

$$M_2 - M_1 - M_0 = (X_1 + X_0)(Y_1 + Y_0) - X_1 Y_1 - X_0 Y_0 =$$

$$= \cancel{X_1 Y_1} + X_1 Y_0 + X_0 Y_1 + \cancel{X_0 Y_0} - \cancel{X_1 Y_1} - \cancel{X_0 Y_0} =$$

$$= X_1 Y_0 + X_0 Y_1$$

$$M = X_1 Y_1 \, 2^n + (X_1 Y_0 + X_0 Y_1) \, 2^{n/2} + X_0 Y_0 = M_1 \, 2^n + (M_2 + M_1 - M_0) 2^{n/2} + M_0$$

Thus, we can get the product of two n-digits numbers by taking 3 products of two $\frac{n}{2}$-digits numbers.

With $q = 3$, I get:

$$T_3(n) \leq O(n^{\log_2 3}) = O(n^{1.5849\ldots})$$

This improves significantly the primary school algo.

RECURSIVE-MULTIPLY (x, y):

LET $x = x_1 2^{n/2} + x_0$ and $Y = Y_1 2^{n/2} + Y_0$	$O(n)$
COMPUTE $S_x = X_1 + X_0$ and $S_Y = Y_1 + Y_0$	$O(n)$
M_2 = RECURSIVE-MULTIPLY (S_x, S_Y)	$T_3(\frac{n}{2})$
M_1 = RECURSIVE-MULTIPLY (X_1, Y_1)	$T_3(\frac{n}{2})$
M_0 = RECURSIVE-MULTIPLY (X_0, Y_0)	$T_3(\frac{n}{2})$
RETURN $M_1 \, 2^n + (M_2 - M_1 - M_0)$	$O(n)$

$$T_3(n) \leq 3T_3(\tfrac{n}{2}) + O(n) \implies T_3(n) \leq O(n^{\log_2 3})$$

The best known algorithm for multiplying two n-digits numbers takes time $O(n \log n \cdot 2^{O(\log^* n)})$

n of logs to take to make result < 2

$\log^* n \leq 10$ IF $n \leq 2^{2^{2^{2^{2^{2^{2^{2^{2^2}}}}}}}}$ (10 times)

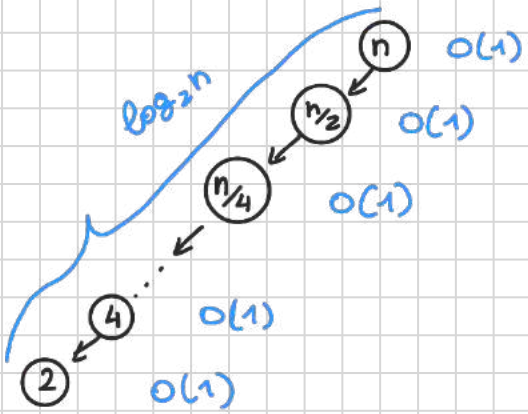
2^{80}

$((2^2)^2)^2 = 2^{2^3} = 2^8 = 256$

$< 2^{273}$ atoms in the universe

$\log^* 2^{273} < 5$

$$S(n) \leq \begin{cases} S(n/2) + c & \forall n \geq 3 \\ c & n \in \{0,1,2\} \end{cases}$$



TOTAL RUNTIME

$S(n) \leq O(\log n)$

THM: $S(n) \leq c \log_2 n, \quad \forall n \geq 2$

P: $S(0) = S(1) = S(2) = c$ (BASE CASE HOLDS)

$$S(n) \leq S\left(\frac{n}{2}\right) + c \leq c \log_2 \left(\frac{n}{2}\right) + c$$

$$= c(\log_2 n - \log_2 2) + c$$

$$= c(\log_2 n - 1) + c$$

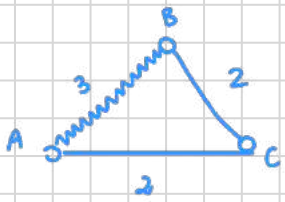
$$= c \log_2 n - c + c = c \log_2 n \quad \blacksquare$$

EX: Let x be an unimodal array of size n, that is, $\exists\, i \in \{0, 1, \ldots, n-1\}$, such that $x[i]$ is sorted increasingly, and $x[i]$ is sorted decreasingly

$x = [2, 3, 10, 100, 94, 20, 3, 1]$

Find the largest value in X as fast as you can.

[2, 3, 100, 90, 80, 70, 3]



Let V be a n-dimensional array.

We say that $0 \leq i \leq j \leq n-1$ form an inversion

IF V[i] > V[j]

1,2
1,4
3,4

```
c=0
FOR i=0,...,n-2
  FOR j=i+1,...,n-1
    IF V[i] > V[j]:
      c+=1
```

} O(1)

RETURN c

Algorithms

DATE: 26/4/22

(LOCAL SEARCH ALGORITHMS)

- GREEDY ALGORITHMS
- DIVIDE - ET- IMPERA "

WEIGHTED INTERNAL SCHEDULING

We are given a set of intervals:

$$I = \{ i_1, \dots, i_n \} = \{ (s_1, f_1), (s_2, f_2), \dots, (s_n, f_n) \}$$

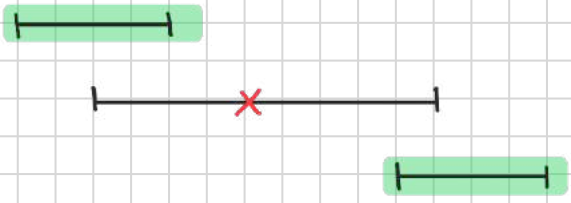
where s_j is the starting time of interval i_j, and f_j is the ending time of interval i_j, as well as a weighting function w that assigns a weight/value to each interval where $w(i_j)$ is the weight/value of i_j.

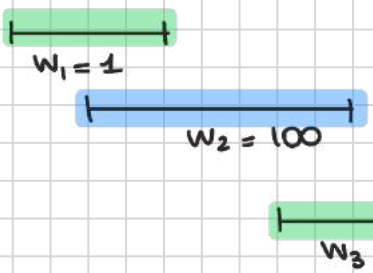
GOAL: Select a subset $S \subseteq I$ of non-overlapping intervals of maximum value, that is, that maximises $\sum_{i_j \in S} w(i_j)$

Intervals scheduling is a special case of weighted interval scheduling (just set $w(i_j) = 1 \quad \forall i_j \in I$).

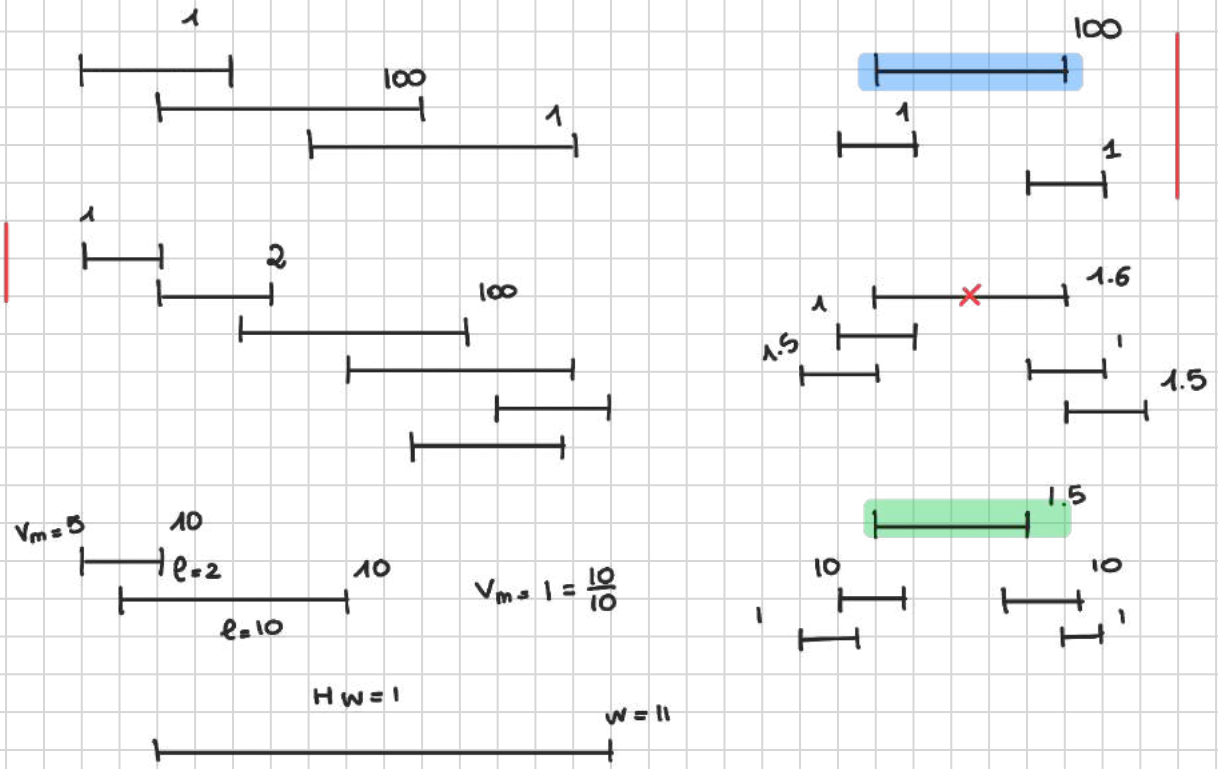
(Weighted interval sch. is a generalization of int. sch.)

We solved Interval scheduling with the "earliest-finishing-time" greedy algorithm.





with weights, the greedy algorithm can do a very bad job.



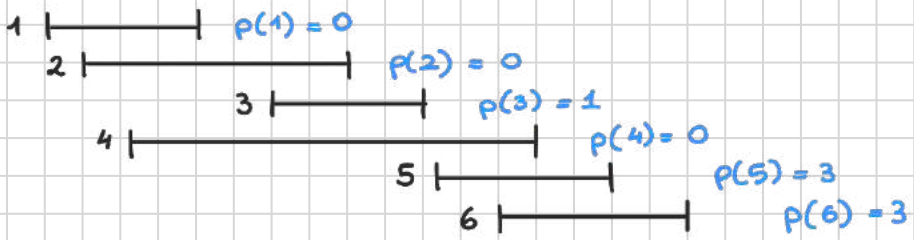
DINAMYC PROGRAMMING

An algorithmic technique that keeps track of all solutions at once, in "little" time.
(Greedy algorithms, instead, consider one partial solution at a time).

First of all, let us assume that the intervals are sorted by finishing time : $f_1 \leq f_2 \leq \dots \leq f_n$.

DEF: INTERVAL i "comes before" interval j IFF $f_i \leq f_j$.

DEF: Let $p(j)$, for an interval j, be the largest index $i < j$ S.T. intervals i and j are disjoint (compatible), or, if no such i exists, let $p(j) = 0$.



L: The generic interval i is disjoint (compatible) with each of the intervals $1, 2, \ldots, p(i)$.

P: i is disjoint from $p(i)$ intervals are sorted increasingly by their finishing time.
i being compatible with $p(i) < i$ then means that $s_i > f_{p(i)}$. Also, $f_{p(i)} \geq f_{p(i)-2} \geq \cdots \geq f_1$. Thus, $s_i > f_j$ for each $j \in \{1, 2, \ldots, p(i)\}$. ■

Suppose that O_i is an optimal solution of the problem restricted to the first i intervals ($\{(s_1, f_1), (s_2, f_2), \ldots, (s_i, f_i)\}$)

Let OPT_i be the value of O_i

For each $j \geq 1$:

- either interval j is in O_j, $j \in O_j$, then

$$OPT_j = w_j + OPT_{p(j)}.$$

- or $j \notin O_j$, then

$$OPT_j = OPT_{j-1}$$

OBS: ① Interval j is in an optimal solution O_j if

$$w_j + OPT_{p(j)} \geq OPT_{j-1}.$$

② Interval j is <u>not</u> in an optimal solution O_j if

$$OPT_{j-1} \geq w_j + OPT_{p(j)}.$$

Dynamic Programming, in general, expresses the value of optimal solutions in terms of the value of optimal solutions to <u>smaller</u> problems.

```
DEF   Compute - OPT (j) :        // compute the value of OPT_j for j=0,1,..,n
      IF   j==0 :
           RETURN 0
```

ELSE:

RETURN $\max(w_j + \text{COMPUTE-OPT}(p(j)), \text{COMPUTE-OPT}(j-1))$

L: COMPUTE-OPT(j) returns OPT_j for $j = 0, 1, \ldots, n$

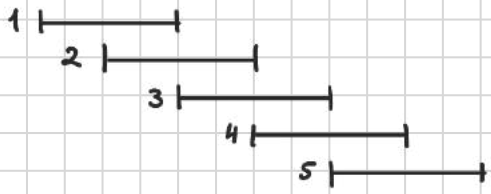
P: BASE CASE: $j = 0$ and $OPT(0) = 0 = \text{COMPUTE-OPT}(0)$. ✓

IND. STEP: Suppose that $\text{COMPUTE-OPT}(i) = OPT(i)$ $\forall\, i \leq j$.

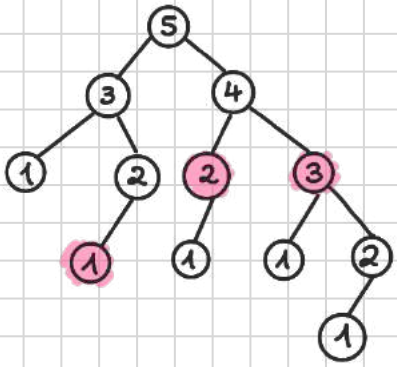
Then, by OBS. $OPT(j+1) = \max(w_{j+1} + OPT(p(j+1)), OPT(j))$.

Then, $\text{COMPUTE-OPT}(j+1) = OPT(j+1)$. ■

The issue with the algorithm is that it takes exponential time



COMPUTE-OPT(5)



Then, COMPUTE-OPT(n) takes time $\geq 2^{n/2}$

M-COMPUTE-OPT(j): MEMOIZATION

GLOBAL M

IF j A KEY OF M:

RETURN M[j]

```
ELSE:
    IF j == 0:
        M[j] = 0
    ELSE:
        M[j] = max (w_j + M-COMPUTE-OPT(p(j)), M-COMPUTE-OPT(j-1))
    RETURN M[j]
```

L: M-COMPUTE-OPT(j) RETURNS OPT_j = COMPUTE-OPT(j)

L: M-COMPUTE-OPT(j) TAKES O(j) TIME.

EXERCISE:

Write down a version of M-COMPUTE-COST that is not recursive.

Algorithms

DATE: 28/4/22

M-COMPUTE-OPT(j): MEMOIZATION

```
GLOBAL M
IF j A KEY OF M:
    RETURN M[j]
ELSE:
    IF j==0:
        M[j] = 0
    ELSE:
        M[j] = max(w_j + M-COMPUTE-OPT(p(j)), M-COMPUTE-OPT(j-1))
    RETURN M[j]
```

THIS ALGORITHM FILLS UP DICTIONARY M

OBS:

① Interval j is in an optimal solution O_j if

$$w_j + OPT_{p(j)} \geq OPT_{j-1}.$$

② Interval j is <u>not</u> in an optimal solution O_j if

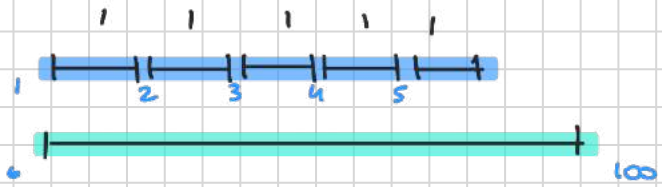
$$OPT_{j-1} \geq w_j + OPT_{p(j)}.$$

FIND SOLUTION(j): // Returns an optimal sequence of intervals among the first j

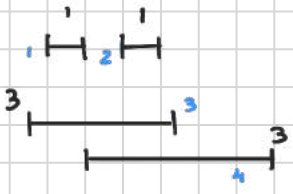
```
GLOBAL M
IF j==0:
    RETURN []
ELSE:    // should interval j be in O_j?
    IF w_j + M[p(j)] ≥ M[j-1]:
        RETURN FIND-SOLUTION(p(j)) + [j]      FIND-SOL(p(j))
    ELSE:    // M[j-1] > w_j + M[p(j)]          or
        RETURN FIND-SOLUTION(j-1)             FIND-SOL(j-1)
```

T: FIND-SOLUTION(n) takes O(n) time

But, to run FIND-SOLUTION(n) we first need to fill up M (that is, to call M-COMPUTE-OPT(n))



M[0] = 0
M[1] = 1
M[2] = 2
M[3] = 3
M[4] = 4
M[5] = 5
M[6] = 100



M[0] = 0
M[1] = 1
M[2] = 2
M[3] = 3
M[4] = max(M[3], W[4] + M[1]) =
max(3, 3+1) = 4

WIS(I,W): // ITERATIVE ALGO FOR FILLING UP M/OPT$_1$, ... OPT$_n$

// $I = [(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)]$, with $f_1 \leq f_2 \leq \ldots \leq f_n$

// W[i] is the weight of interval i

LET P[i] BE THE LARGEST $j < i$ S.T. $f_j < s_i$ // $p(i) = P[i]$

M = [None] * (n+1) O(n)

FOR $i = 0, 1, \ldots, n$ (n+1 iteration)

IF $i == 0$

M[i] = 0

ELSE:

M[i] = max (w[i] + M[P[i]], M[i-1]) } O(1)

RETURN M // Return M[n]

EX: PROVE THAT ONE CAN COMPUTE ALL OF P[1], P[2], ..., P[n] IN O(n) TIME (PROVIDED THAT THE INTERVALS ARE SORTED)

```
DEF WIS-SOL(I, W):
    LET P[i] BE THE LARGEST j < i S.T. f_j < s_i
    M = WIS(I, W)
    O = [ ]
    i = n
    WHILE i > 0 :
      IF M[i] == W[i] + M[P[i]]:
          O.append(i)
          i = P[i]
      ELSE:        // M[i] = M[i-1]
          i = i-1
    RETURN O
```

THE ARRAY M (THE "TABLE") IS THE KEY TO THE ALGORITHM

As we said, dynamic programming solves a problem by means of the solutions to (some of) it's subproblems.

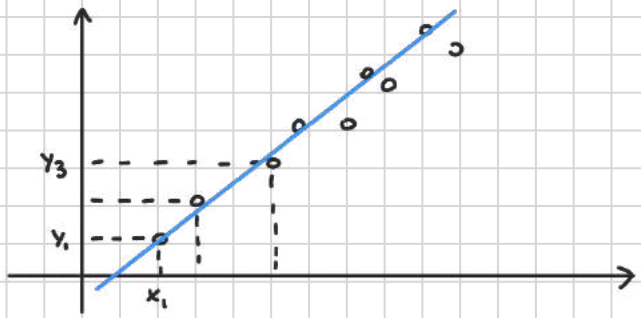
For this approach to go through, the following properties are very useful:

1. There is only a polynomial number of subproblems to consider;
2. The solution to the problem can be computed efficiently given the solutions to the subproblems;
3. Subproblems have to have some "ordering" (e.g., from smallest to largest), and there must exists a recurrence that gives us the optimal value of a problem using the optimal values of the problems preceding it in the ordering.

(THINK OF THESE AS "GUIDELINES", NOT RULES)

IN WIS, we had [2] subproblems per problem

LEAST SQUARES FIT



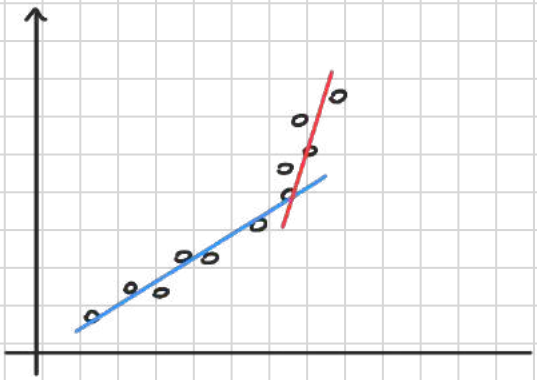
Let the set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ be given, with $x_1 < x_2 < x_3 < \ldots < x_n$

Given a line L, $y = ax + b$, we say that the error of L with respect to P is the sum of the squared distances.

$$\text{ERROR}(L, P) = \sum_{i=1}^{n} (y_i - (ax_i + b))^2$$

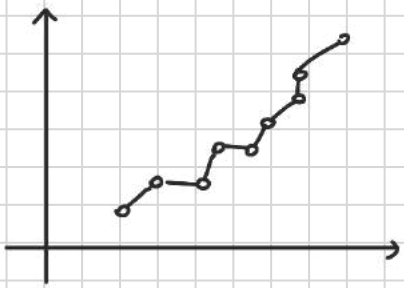
The line having the smallest error can be shown to be $y = ax + b$, with

$$a = \frac{n \sum_{i=1}^{n} (x_i y_i) - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}, \quad b = \frac{\sum_{i=1}^{n} y_i - a \sum_{i=1}^{n} x_i}{n}$$



Essentially, here, we would like to have two different lines (because they "look right").

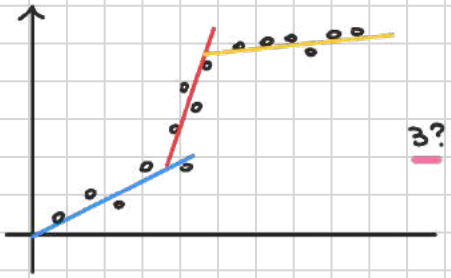
Can we, say, let the algorithm decide the optimal number of lines?



No!

For otherwise the algo might overfit (it would choose n-1 lines)

CAN WE THEN ENFORCE SAME NUMBER OF LINES?



We need to formulate the problem so that

1. the fit is good
2. the lines are few.

We have $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ with $x_1 < x_2 < \ldots < x_n$

Let $p_i = (x_i, y_i)$ For $i = 1 \ldots n$

First, we aim to partition P into "segments"
- a segment is just a continous set of points, e.g., $\{p_i, p_{i+1}, \ldots, p_n\}$ For $i \leq j$.

For each segment, we compute the optimal line as above (the line inducing the smallest error of the points of the segment).

The penalty of a partition, is the sum of the following quantities:

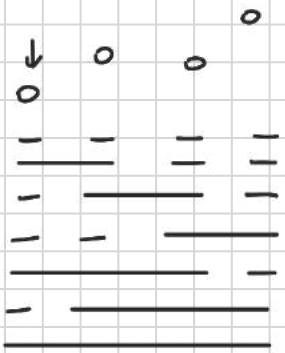
1. c times the number of segments of our partition,
2. For each segment, the error of the segment's approximating line

The goal is to find a partition of minimum penalty.

By changing "C" we can either:

- get solutions with few lines / segments ($C \to \infty$);
- get small total error ($C \to 0$)

There are exponentially many partitions.

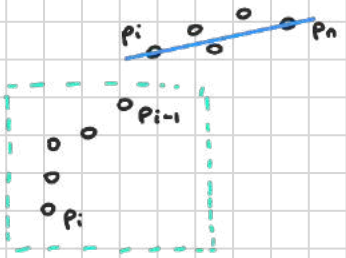


Dynamic Programming

OBSERVATION:

In the optimal solution point p_n (the last point) belongs to some segment $p_i, p_{i+1}, \dots, p_n$.

If we know what the segment is (i.e., if we know i), we can compute the cost of the segment $p_i, \dots, p_n$ (by using the above formula) and add this COST to the optimal cost for points $p_1, p_2, \dots, p_{i-1}$.



LET $OPT(i)$ BE THE OPTIMAL COST FOR POINTS $p_1, \dots, p_i$.

LET $OPT(0) = 0$

LET e_{ij} be THE MINIMUM ERROR A LINE CAN MAKE ON THE SEGMENT $p_i, p_{i+1}, \dots, p_j$. (e_{ij} CAN BE COMPUTED BY THE PREVIOUS FORMULA).

Our observation entails the following:

L: If the last segment of the optimal solution is $p_i, \dots, p_n$, then $OPT(n) = e_{in} + C + OPT(i-1)$

Thus,

T: $OPT(j) = \min_{1 \le i \le j} \left(e_{ij} + C + OPT(i-1) \right)$, AND

The segment $p_i, \dots, p_j$ is used in an optimal solution IFF $OPT(j) = e_{i,j} + c + OPT(i-1)$.

SEGMENTED - LEAST- SQUARES (P):

$M[0] \leftarrow 0$

FOR $i = 1 \dots n$:

FOR $j = i \dots n$:

COMPUTE THE ERROR $e_{i,j}$ FOR THE SEGMENT $p_i, p_{i+1}, \dots, p_j$

FOR $j = 1 \dots n$:

$$M[j] = \min_{1 \le i \le j} (e_{i,j} + c + M[i-1])$$

RETURN $M[n]$

This returns the minimum cost.

Ex: Write an algorithm for finding the actual segmentation.

Ex: Try to make the algorithms as fast as possible.

Algorithms DATE: 3/5/22

DYNAMIC PROGRAMMING

2 - SUBPROBLEMS

Weighted Interval scheduling

n - SUBPROBLEMS

"Least square fit"
SEGMENTATION

SEGMENTATION

We are given a sequence $p_1, \dots, p_n$ "POINTS" and we aim to segment it (select $I = i_1 < i_2 < \dots < i_\kappa = n+1$, FOR SOME $\kappa \geq 1$; cut the sequence $(p_i, \dots, p_{i_2-1})$, $(p_{i_2} \dots p_{i_3-1}), \dots, (p_{i_{n-1}}, \dots p_{i_\kappa-1})$).

FOR $i \leq j$, the cost of the segment $(p_i, p_{i+1}, \dots, p_j)$ is c_{ij}. (In least-square-fit, $c_{ij} = a_{ij} + c$)

What is the min-cost segmentation (the cost of a segmentation is the sum of the costs of its segments)?

Let us define $OPT(j)$ to be the value of the minimum cost segmentation, for the input $p_1, \dots, p_j$

L: $OPT(j) = \min_{1 \leq i \leq j} (c_{ij} + OPT(i-1))$; the segment $(p_i, \dots, p_j)$ is in an optimal solution for the input $p_1 \dots p_j$ IFF

$$OPT(j) = c_{ij} + OPT(i-1)$$

P: If the sequence is $p_1 \dots p_j$ the element p_j should be the right endpoint of a segment, of the last segment.

Where does the last segment begin? It will begin at some point p_i with $1 \leq i \leq j$

The cost of the sequence $p_1 \dots p_j$ if the last segment begins at i is equal to

$$c_{ij} + OPT(i-1),$$

Since i have to pay for the segment $(p_i, \dots, p_j)$, and for the best segmentation of the sequence $p_1, \dots, p_{i-1}$. Thus, the minimum cost for the $p_1 \dots p_j$ is:

$$OPT(j) = \min_{1 \le i \le j} \left(c_{ij} + OPT(i-1) \right)$$

And, the segment $(p_k, \dots, p_j)$ is part of an opt. solution for the sequence $p_1 \dots p_j$ IFF

$$c_{kj} + OPT(k-1) = \min_{1 \le i \le j} \left(c_{ij} + OPT(i-1) \right) \blacksquare$$

DYNAMIC PROGRAMMING

2 - SUBPROBLEMS

Weighted Interval scheduling

n - SUBPROBLEMS

"Least square fit" } look at prefixes

SEGMENTATION

"ADDING A VARIABLE"

Consider the following "scheduling" problem:

- there are "n" jobs, the ith of which takes time w_i seconds (of our CPU)

  Our CPU is available for W seconds.

  If we schedule the generic job we are paid 1€ per second it runs.

  Which subset S of jobs should we schedule to maximize our gain?

  The set S of jobs is "feasible" (is a valid sol) IFF $\sum_{i \in S} w_i \le W$

  Thus, we are asking: what is the set of jobs S satisfying $\sum_{i \in S} w_i \le W$, that maximizes $\sum_{i \in S} w_i$?

$w_1 = 3 \quad w_2 = 3 \quad w_3 = 5 \qquad W = 5$

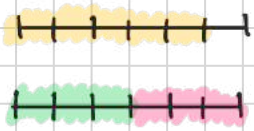
INVALID

VALID (value = 3)

OPTIMAL AND VALID (VALUE = 5)

HOW TO SOLVE IT?

$w_1 = 3$ $w_2 = 3$ $w_3 = 5$ $W = 6$



In dynamic programming, we want to split a problem into some number of subproblems... which subproblems to use here?

For WIS/segm. we considered prefixes of the instance as subproblems.

Let us try to do the same.

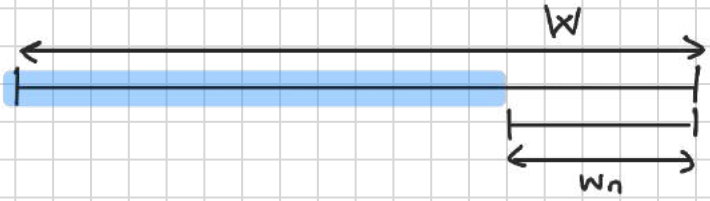
Let $OPT(i)$ be the optimal gain you can achieve by using only jobs $1, 2, \ldots, i$. Let O be an optimal solution.

Then,

L: IF $n \notin O$, then $OPT(n) = OPT(n-1)$

Ok but what if $n \in O$?

IF $n \in O$, then the remaining jobs will only have $W - w_n$ seconds available.



The amount of time available strictly depends on the length of the job we schedule.

In particular, we cannot obtain $OPT(n)$ in terms of just $OPT(1), OPT(2), \ldots, OPT(n-1)$.

We need to solve the following subproblems?

- Given V and i, what is the optimal value if I have time V and the jobs $1, 2, \ldots, i$.

("Adding a variable" ↳ we came up with the idea of using V)

Assume that W is an integer and, also, that $w_1, \dots, w_n$ are also integers.

Then,

$$OPT(i, W) = \max_{\substack{S \subseteq \{1,\dots,i\} \\ (\sum_{j \in S} w_j \leq W)}} \sum_{j \in S} w_j$$

Using these "enlarged" set of subproblems, we can solve the original problem:

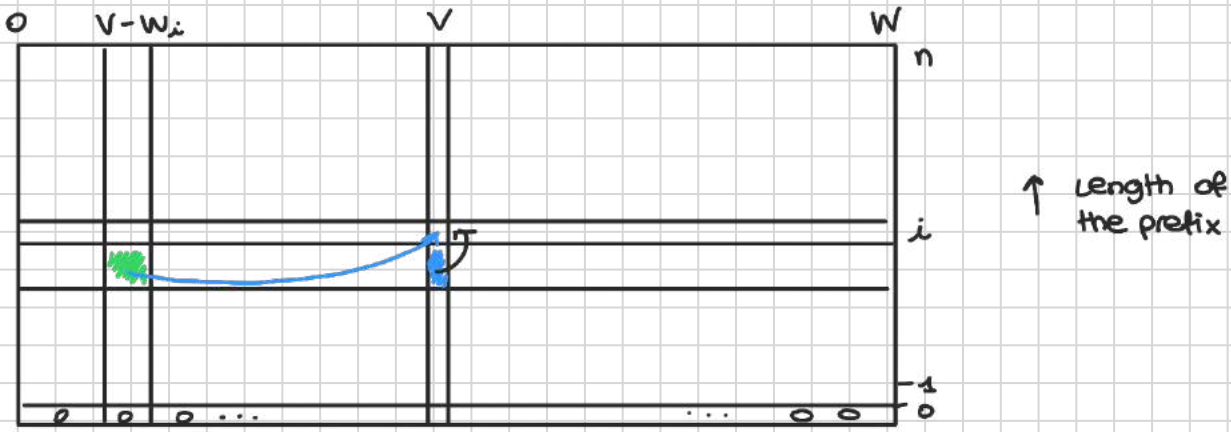
- If $n \notin O$, then $OPT(n, W) = OPT(n-1, W)$
- If $n \in O$, then $OPT(n, W) = w_n + OPT(n-1, W-w_n)$,

(IF $w_n > W$, then $OPT(n, W) = OPT(n-1, W)$)

T1: If $W < w_i$ then $OPT(i, W) = OPT(i-1, W)$

Otherwise;

$$\underbrace{OPT(i, W)}_{\text{the answer}} = \max(OPT(i-1, W), w_i + OPT(i-1, W-w_i)).$$



SUBSET-SUM (n, W):

INITIALIZE THE ARRAY $M[0 \dots n][0 \dots W]$ $O(n \cdot W)$

LET $M[0][V] = 0$ $\forall\ V = 0, 1, \dots, W$ $O(W)$

$O(nW)$ {

FOR $i = 1 \dots n$

FOR $w = 0 \dots W$

use the recurrence of T1 to set the value of $M[i][W]$

}

RETURN $M[n, W]$

$$W = \overbrace{100\ldots0}^{\text{BINARY } n} = 2^{n-1}$$

Algorithms

DATE: 5/5/22

KNAPSACK

- we have n jobs, the ith of which takes w_i seconds to finish, and has value v_i.

  Our CPU is available for W seconds.

  A set S of jobs is feasible if:

$$\sum_{i \in S} w_i \leq W$$

  What is a set S of jobs satisfying $\sum_{i \in S} w_i \leq W$ and having maximum total value?

We assume that $W, w_1, \ldots, w_n$ are integers.

$$OPT(i, W) = \max_{S \subseteq \{1,2,\ldots,i\}} \sum_{j \in S} v_j$$

$$\sum_{j \in S} w_j \leq W$$

w_j = "length" of job j
v_j = "payment" you get for running job j

T: If $O_{i,w}$ is an optimal solution (a set of jobs) for the jobs $\{1, \ldots, i\}$ and with w available seconds. Then,

- If $i \notin O_{i,w}$, then $OPT(i,w) = OPT(i-1, w)$, and
- If $i \in O_{i,w}$, then $OPT(i,w) = v_i + OPT(i-1, w-w_i)$

Moreover, if $w_i > w$, then $i \notin O_{i,w}$, thus $OPT(i,w) = OPT(i-1, w)$

KNAPSACK (n, W)

INITIALIZE THE ARRAY $M[0 \ldots n][0 \ldots W]$ $O(nW)$

LET $M[0][W] = 0$ $\forall\, 0 \leq w \leq W$ $O(W)$

FOR $i = 1 \ldots n$

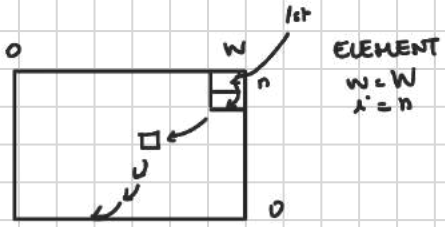
  FOR $w = 0 \ldots W$

$O(n \cdot W)$

```
        IF (w_i > W) or (M[i-1][W] > v_i + M[i-1][W-w_i])
            M[i][W] = M[i-1][W]
        ELSE:
            M[i][W] = v_i + M[i-1][W-w_i]
RETURN M
```

We could run M = KNAPSACK(n, W), and then conclude that M[n][W] is the value of the best solution

FINDSOL (n, W)

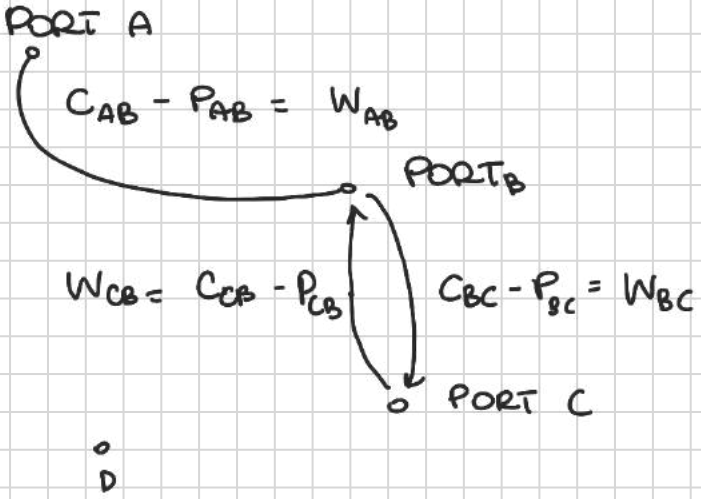



```
M = KNAPSACK(n, W)        O(n·W)
S = [ ]
w = W
i = n
WHILE i > 0:                                   O(n)
    IF M[i][w] == M[i-1][w]
        i -= 1   // throw away i
    ELSE:
        S.APPEND(i)   // schedule i
        w -= w_i
        i -= 1
RETURN S        // Σ_{j∈S} v_j = M[n][W] and Σ_{j∈S} w_j ≤ W
```

```
IF M[i][W] ≠ M[i-1][W]:
    S.APPEND(i)
    W -= w_i
i -= 1
```

SHORTEST PATHS ON GRAPHS WITH "NEGATIVE AND/OR POSITIVE" WEIGHTS.

cruise ship / taxi drive

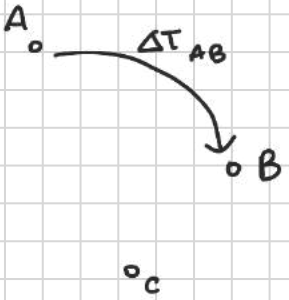


$W_{AB} > 0$ $C_{AB} > P_{AB}$

$W_{AB} = 0$ $C_{AB} = P_{AB}$

$W_{AB} < 0$ $C_{AB} < P_{AB}$

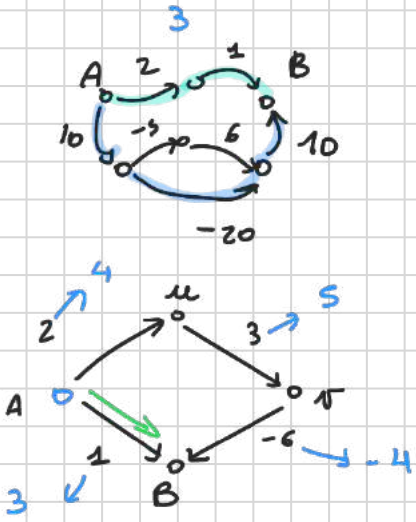
CHEMICAL REACTIONS



$V = \{A, B, C\}$ chemical compounds

HOW to go from s to t with the smallest increase in temperature

---



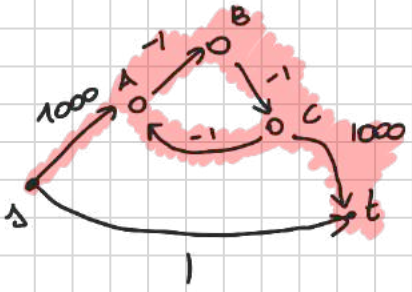
A TO B

DIJKSTRA DOESN'T WORK HERE

A	0
B	1
u	2

ADD $-\min_e W_e$ to each edge

(so that edge weights become non-neg.)



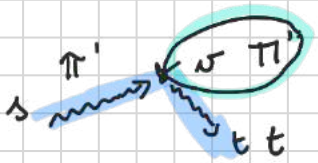
we assume that the graph has no negative cycles

L1: IF G is a graph with no negative cycles, then ∃ exists a shortest path from s to t that is simple (it has no node repetition), and that contains $\leq n-1$ edges.

P: Let π be a s-t shortest path with minimal number of edges.

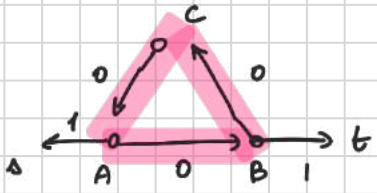
suppose that $\exists\, v$ s.t. $\pi = \pi' v \pi'' v \pi'''$

Since G contains no negative cycles, it must be that $v\pi''$ is a cycle having a positive total edge weight.



Consider, now, the path $\tilde{\pi} = \pi' v \pi''$. This path has fewer edges than π.
Moreover the total weight of $\tilde{\pi}$ is no larger than the total weight of π. Since π has more edges than $\tilde{\pi}$, and since the total weight of $\tilde{\pi}$ is not larger than that of π, π cannot be a s-t shortest path with a minimal numbers of edges. CONTRADICTION.

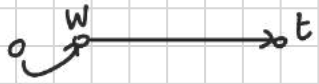
Then no node is repeated in a shortest path with a minimal number of edges — since we have n nodes, such a path has at most $n-1$ edges ■



$W(s\ AB\ t) = 2$
$W(s\ ABCAB\ t) = 2$

Let $OPT(i, v)$ be the minimum cost/weight of a path from v to t, having **at most** i edges.

Let π be a path of i edges from v to t, having minimum cost $OPT(i, v)$



π has at most i edges

- If π has at most $i-1$ edges, then $OPT(i,v) = OPT(i-1,v)$
- If π has exactly i edges, then $\exists w$ such that π starts from v, moves to w, and then progresses optimally to t using at most $i-1$ edges. That is,

$$OPT(i,v) = c_{vw} + OPT(i-1, w)$$

This DP is called "the BELLMAN-FORD" algorithm.

$$OPT(0, t) = 0$$
$$OPT(0, v) = \infty \quad \forall v \neq t$$

L2: If $i > 0$, $OPT(i, v) = \min\left(OPT(i-1, v), \min_{\substack{w \\ (v,w) \in E}} \left(c_{vw} + OPT(i-1, w)\right)\right)$

BELLMAN-FORD (G, s, t):

```
INITIALIZE M[0 ... n-1][v ∈ V(G)]
M[0][t] = 0
M[0][v] = +∞   ∀ v ∈ V - {t}
FOR i = 1 .... n-1:
    FOR v ∈ V(G):
        M[i][v] = M[i-1][v]
        FOR w ST. (v,w) ∈ E(G):
            γ = c_{v,w} + M[i-1][w]
            IF γ < M[i][v]:
                M[i][v] = γ
RETURN M   (or M[n-1][s])
```

RECURRENCE OF L2

THM: Bellman-Ford returns the shortest distance from s to t if the graph contains no no negative cycles.

P: Apply L1 and L2.

Algorithms

DATE: 10/5/22

HOW TO DEAL WITH "HARD" PROBLEMS, THAT IS, PROBLEMS THAT MIGHT NOT ADMIT POLYTIME ALGORITHM?

Consider the following algorithms we studied:

- we proved that the shortest path problem on graphs with positive weights can be solved in $O((n+n)\log n)$ with DIJKSTRA ALGORITHM;
- we proved that sorting an array can be done in $O(n \log n)$ through HEAPSORT or MERGESORT.
- we proved that interval scheduling can be solved in $O(n \log n)$
- ...

Each of the above problems can be solved efficiently, in polytime.

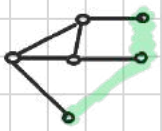
We also solved some generalizations of the above probs:

- Shortest path on general weighted graphs can be solved in $O(n^3)$;
- Weighted int. scheduling can also be solved in $O(n \log n)$.
- ...

There are many possible generalizations of the above problems:

- INDEPENDENT SET

  If $G(V,E)$ is a graph, then $S \subseteq V$ is an independent set of the graph iff there are NO edges between the nodes of S.



THE I.S. PROBLEM:

Given $G(V,E)$, find a largest independent set.

INTERVAL SCHEDULING

That is, if we could solve independence set, we could also solve interval scheduling.

Thus, independence set is more general than interval scheduling.

- VERTEX COVER

A vertex cover of an und. graph $G(V,E)$ is a set $S \subseteq V$ such that each edge $e \in E$ is incident on at least one node of S.

THE VC PROBLEM:

- Given $G(V,E)$ find a smallest VC.

As we will see, these two problems are "hard" (we believe they cannot be solved in polytime).

In other words, here, we are looking for negative answers to the question "does there exists an efficient algorithm?"

Negative answers are useful for a number of reasons:

1. A negative answer can stop you from wasting time;
2. A " " can give you a reason to simplify your problem, or to use heuristics.

The most important tool CS has for indicating that no efficient algorithms exist is the "reduction".

Reductions are based on the following idea:

- Suppose that there exists an **oracle** (a black-box, or a function) that, magically, in polynomial time solves instances of problem X;
- Suppose that you devise a **reduction** (an algorithm) that, with polynomially many calls to the **oracle** for X, plus some additional polynomial time computation, solves generic instances of problem Y.

Then, using your reduction together with the oracle for X, you can efficiently solve Y.

DEF: For a pair of problems X and Y as above, we write $Y \leq_p X$ if the above reduction exists.

We read "$Y \leq_p X$" as "Y is polynomial-type reducible to X", or "X is at least as hard as Y (w.r.t. polynomial time".

L1: If $Y \leq_p X$, and X can be solved in polynomial time, then Y = = = = = =

P: Each call to the X-oracle can be answered in polynomial time (according to L1). That is, $\exists\, c \geq 0$ s.t. for an instance I of problem X the call to "oracle(I)" can be answered in time $O(|I|^c)$.

(If $n = |I|$, then $O(|I|^c) = O(n^c)$).

The reduction (which exists since $Y \leq_p X$) makes at most $O(n^{c'})$ calls to the oracle, for some $c' \geq 0$.

The reduction might also make some other computation, for a time of $O(n^{c''})$, for $c'' \geq 0$.

Then everything can be implemented with a runtime of

$$O(n^{c'}) \cdot O(n^c) + O(n^{c''}) = O(n^{c'+c} + n^{c''}) = O(n^{\max(c'+c,\, c'')}) \quad \blacksquare$$

L2: If $Y \leq_p X$, and Y cannot be solved in polynomial time, then X " " " " " time.

P: Apply L1.

Let us use this idea on independent set and Vertex Cover.

Up until now, we have no idea how hard they are. We will show that they share the same "hardness".

L: Independent Set $\leq_p$ Vertex Cover

L: Vertex Cover $\leq_p$ Independent Set

T: Let $G(V,E)$ be an undirected graph. Then, $S \subseteq V$ is an Independent Set IFF $V - S$ is a Vertex Cover.

P: First, assume that S is and Ind. Set.

$\nexists \{u,v\} \in E$ s.t. $\{u,v\} \subseteq S$.

That is, $\forall \{u,v\} \in E$, either $u \notin S$, or $v \notin S$ or $u, v \notin S$.

If $w \in V$ is s.t. $w \notin S$ then $w \in V-S$.

Thus, $\forall \{u,v\} \in E$, either $u \in V-S$, or $v \in V-S$, or $u, v \in V-S$

Thus, $V-S$ is a vertex cover.

Now, assume that $V-S$ is a vertex cover.

Then, if $u, v \notin V-S$, then $\{u,v\} \notin E$.
And, $u, v \in V-S$ is equivalent to $u, v \in S$.

That is, for any two $u, v \in S$, $\{u,v\} \in E$ - then S is an Ind. Set ■

L: Independent Set $\leq_p$ Vertex Cover

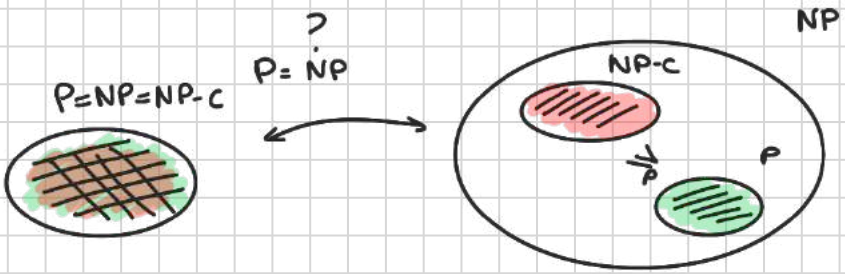
P: Suppose we have an oracle for vertex cover. Then, to determine whether $G(V,E)$ has an independent set of k nodes, I can ask the oracle whether $G(V,E)$ has a vertex cover of $|V| - k$ nodes. ■

L: Vertex Cover $\leq_p$ Independent Set

P: Omitted (same as above.)

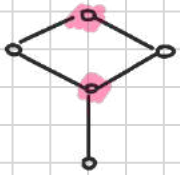
$\min VC \leq K \iff \max I.S. \geq n-k$

Vertex Cover and Independence Set are equivalent. (w.r.t. polynomial time)



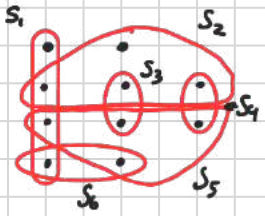
REDUCTION TO A MORE GENERAL CASE

Vertex Cover is a special case of a more general "covering" problems.



SET COVER: Given a ground set $U = \{1, 2, \dots, n\}$, and a collection of its subsets $S_1, S_2, \dots, S_m \subseteq U$ and an integer $k \geq 1$,

does there exists a subcollection of k subsets (that is, of k S_i's), whose union equals U?



Do there exist $k=3$ subsets that cover U? Yes (S_1, S_2, S_5)

Intuitively, Vertex Cover looks easier than set cover

L: Vertex Cover $\leq_p$ Set Cover

P: (We assume to have an oracle for Set Cover)

Let $G(V,E)$ is an instance of set cover.
In VC, we aim to cover each edge $e \in E$.
So, let us set $U = E$ in the Set Cover Instance.
In VC, when we pick a vertex $v \in V$, we cover all the ends that are incident on v.

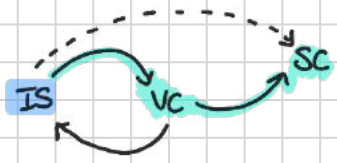
For each $v \in V$, we create a subset S_v as follows:

$$S_v = \{e \mid e \in E \text{ and } v \in V \text{ is incident on } e\}$$

CLAIM: U can be covered with k subsets IFF $G(V,E)$ can be $=$ $=$ k nodes.

P: Suppose that $S_{v_1}, S_{v_2}, \dots, S_{v_k}$ is a set cover, that is, $\bigcup_{i=1}^{k} S_{v_i} = U = E$.

Then, by our construction, $\{v_1, v_2, \dots, v_k\}$ is a vertex cover for $G(V,E)$.
Conversely, if $(v_1, \dots, v_k)$ is a Vertex Cover, then $S_{v_1}, S_{v_2}, \dots, S_{v_k}$ is a set cover. ■



$f(n) = n^c$
$g(n) = n^d$
$h(n) = g(f(n)) = (f(n))^d$
$= (n^c)^d = n^{cd}$

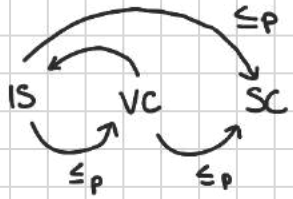
TRANSITIVITY OF REDUCTIONS

L: If $Z \leq_p Y$ and $Y \leq_p X$, then $Z \leq_p X$.

P: We aim to solve an instance of Z, using an oracle for X.

First, we could run the algorithm/reduction for solving instances of Z, using an oracle for Y.
Second, we can implement an algorithm for solving instances of Y, using an oracle for X.

Thus, I have produced an algorithm that solves instances of Z, using an oracle of X. ■



SATISFIABILITY (SAT)

Suppose we are given a set X of n boolean variables, $x_1, x_2 \dots x_n$. (a boolean variable can only have 2 values: True & False)

A term over X is either a variable x_i or the negation $\overline{x_i}$ of x_i ($x_i \equiv$ NOT x_i).

A clause is a disjunction (a "or") of distinct terms.

$t_1 \vee t_2 \vee t_3 = t_1$ OR t_2 OR t_3

(INCLUSIVE) OR
$\wedge \rightarrow$ AND

(t_i can be either a variable x_j or its negation $\overline{x_j}$)

A **formula** is a conjunction (an "AND") of clauses.

$$(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_2 \vee \bar{x}_3)$$

SAT: Can I find a truth assignment to the variables that makes the formula true?

3-SAT: Given a SAT instance where each clause has exactly 3 literals, is it satisfiable?

L: $3SAT \leq_p SAT$

P: Trivial (SAT is a generalization of 3-SAT)

L: $SAT \leq_p 3SAT$

P: Omitted

SAT and 3SAT are two algebraic problems - so it is not surprising that they can be reduced to one another.

L: $3SAT \leq_p IS$

REDUCTION THROUGH GADGETS

P: As usual, we assume to have an oracle for IS. We will be using the IS oracle to solve the input instance of the 3-SAT problem.

The 3-SAT instance is composed of a set X of variables, an of a set of clauses $\{C_1, \dots, C_k\}$ with $|C_i| = 3 \quad \forall i = 1, \dots, k$.

$$X = \{x_1, x_2, x_3, x_4\}$$
$$C_1 = \{x_1, \bar{x}_2, x_3\}$$
$$C_2 = \{\bar{x}_4, x_2, x_1\}$$

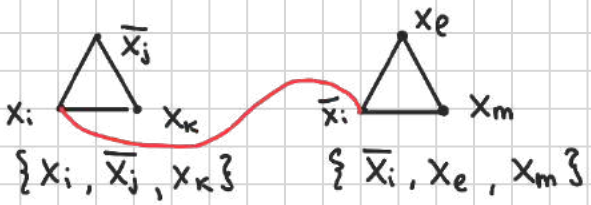
The key to our proof is to look at 3-SAT in a different way:

- We said that 3SAT can be solved by assigning a T/F value to each variable in order to have at least one true literal per clause;
- But, what if we select <u>one</u> term per clause, and choose the value of that term so to make it* true (for instance, we could choose the literal $\bar{x}_3$, which can be made true by setting $x_3 =$ FALSE), with

* and the clause

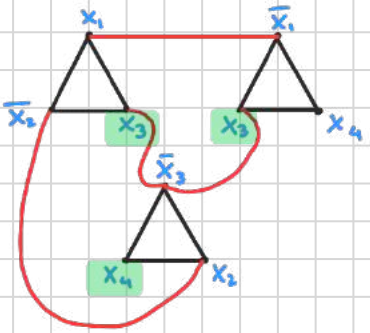
the goal of never selecting both x_i and $\bar{x}_i$, for $i = 1 \dots n$.

We will construct for a reduction on the second view.



- We will construct a graph $G(V, E)$ (starting from the formula) with $3k$ nodes (where k is the number of clauses in the formula):
  - for each $i = 1 \dots k$, we create 3 nodes v_i, v_{i2}, v_{i3} we "label" node v_{ij} with the jth term of the ith clause.
    The nodes v_i, v_{i2}, v_{i3} will form a triangle.
    The graph will also contain other edges:
    - if v_{ij} and $v_{i'j'}$ are two nodes with opposites labels (e.g. "x_k" and "$\bar{x}_k$") we add an edge between v_{ij} and $v_{i'j'}$.

This concludes the description of the reduction:



$C_1 = \{x_1, \bar{x}_2, x_3\}$

$C_2 = \{\bar{x}_1, x_3, x_4\}$

$C_3 = \{\bar{x}_3, x_4, x_2\}$

We will prove that the formula is satisfiable IFF the graph has an ind. set of k nodes.

- Formula SAT $\Rightarrow$ $\exists$ ind. set of size k

  Suppose to have a satisfying assignment τ.
  τ will guarantee that each clause has at least one true term.
  Since τ assigns a single value to each variable, it cannot make both x_i and $\bar{x}_i$ true, for any $i = 1 \dots n$. Thus, no red edge will see both its endpoints's labels set to true.
  Thus, an ind. set of size k can be obtained by picking one node (with "true" label) per triangle.

- $\exists$ independence set of size $K \Rightarrow$ formula SAT.

  As we said, if S is an ind. set of size $\geq K$, then (I) $|S| = K$ (the graph is composed of K disjoint triangles, and we can pick at most 1 node per triangle), and (II) S contains exactly 1 node per triangle.

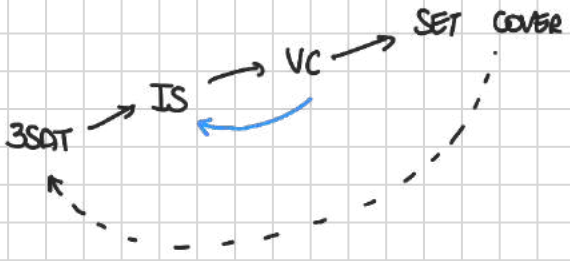
  We create a satisfying assignment as follows:

  - for each variable x_i s.t. the ind. set contains no nodes labeled "x_i" or "$\bar{x}_i$", we set x_i arbitrarily to TRUE or FALSE;
  - for each other variable x_i, either the ind. set does not contain labels "x_i" (I set x_i = FALSE), or it contains labels "$\bar{x}_i$" (I set x_i = TRUE).

  This satisfies each clause. ■

Algorithms

DATE: 17/05/22

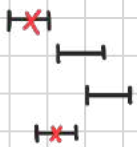


What is P?

DEF: P is the class of decision problems X such that $\exists_{\delta \geq 0}$ and an algorithm A s.t. for each input I to problem X having n bits, algorithm A determines in time $O(n^{\delta})$ (in polytime) if I is a "YES" - instance, or a "no" - instance, of problem X.

A decision problem X could be:

- does there exist an interval scheduling with $\geq n/2$ intervals?



"CERTIFICATES"

Sometimes, it is hard to solve a problem, but it is easy to check if a solution is valid:

- 3-SAT is an example
  (we don't know how to find a solution; but, checking whether a solution - a truth assignment - is valid is an easy task)

  $(x_1 \vee \bar{x}_2 \vee x_3)$

- VC is another example

  (we can easily check whether k vertices form a VC)

These are called "YES-CERTIFICATES"

So let us define the concept of certifier

DEF: Let $\gamma \geq 0$, then an algorithm B, that takes as input two strings I (the instance) and C (the certificate) of, respectively, $n = |I|$ bits and $|C| \leq O(n^\gamma)$ bits is an efficient certifier for problem X if

1. the runtime of B is $O(n^\gamma)$,
2. I is a yes-instance of X IFF $\exists$ certificate C s.t. $B(I,C) = TRUE$,
3. I is a no-instance of X IFF $\forall \quad = \quad C, \; B(I,C) =$ FALSE

DEF: NP is the class of problems for which an efficient certifier exists.

L: $P \subseteq NP$

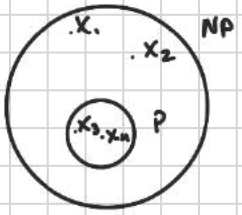
P: Pick arbitrary a problem X in P. Then, $\exists$ algorithm A that solves X in polytime. We aim to show that $x \in NP$. To do so, we need to give an (efficient) certifier B for X. Let us just set B = A (throw away the certificate and just solve the instance).

Non-deterministic Machine
LET $x = [0] * poly(n)$
$x =$ GUESS ( )

$x = [0,0,0..]$ $x = [1,0,0..]$ $x = [0,1,0..]$

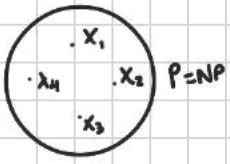
Q: is $P = NP$?

IF $P \neq NP$



?

IF $P = NP$



DEF: If $x \in NP$, and $\forall Y \in NP : Y \leq_p X$, then X is "NP-complete".

US URSS
COOK - LEVIN THEOREM: 3-SAT is NP-Complete.

($\forall y \in NP : Y \leq_p$ 3-SAT)

COR: P = NP IFF $\exists x \in P$ s.t. X is NP-Complete

(EXP TIME - COMPLETE)

COR: VC is NP-Complete
" IS " "
" SC " "

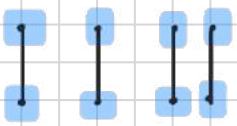
APPROXIMATION ALGORITHMS

APPROX - VC (G(V,E)) :

- $S \leftarrow \emptyset$
- WHILE $E \neq \emptyset$: ($n/2$ ITERATIONS)
  - PICK ANY EDGE $\{u, v\} \in E$ $O(1)$
  - $S \leftarrow S \cup \{u, v\}$ $O(1)$
  - REMOVE FROM E ALL THE EDGES COVERED BY u, OR BY v, OR BY BOTH u AND v. $(\deg(u) + \deg(v))$
- RETURN S

$\sum_u \deg(u) \leq 2|E|$

$O(m+n)$



T: APPROX-VC returns a VC that is not larger than twice the smallest VC. (APPROX-VC is a 2-APPROX)

APPROXIMATION ALGORITHMS

$O(n^{\downarrow}\underline{W}^{\downarrow})$ $\quad W = 2^n$

APPROX - VC (G(V,E))

$S \leftarrow \varnothing$

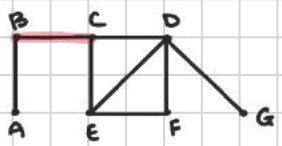
WHILE $E \neq \varnothing$:

PICK ANY EDGE $e = \{u, v\} \in E$

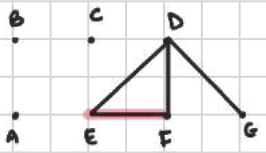
$S \leftarrow S \cup \{u, v\}$

REMOVE EACH EDGE $e' \in E$ S.T. e' is INCIDENT on u OR v

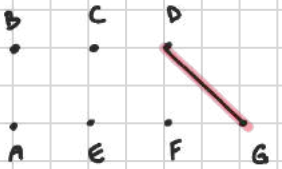
RETURN S



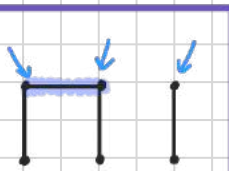
$S = \{B, C\}$



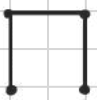
$S = \{B, C, E, F\}$



$S = \{B, C, E, F, D, G\}$ $\quad |S| = 6$

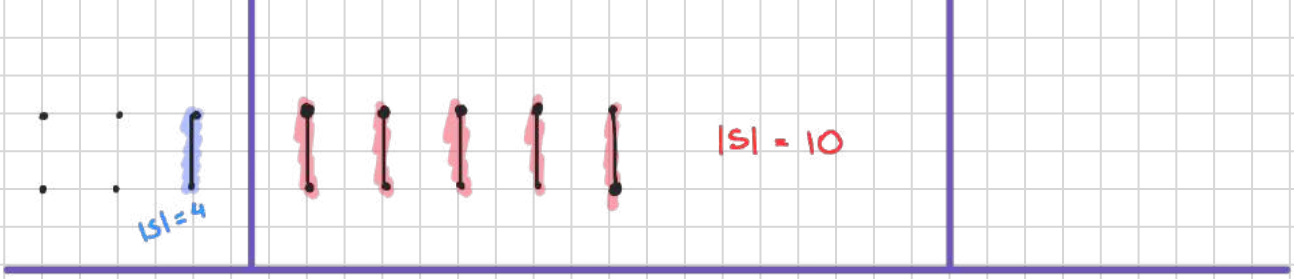


min VC = 5



|S| = 2

minVC = 2



THM: APPROX-VC returns a set of nodes that:

1. is a vertex cover, and
2. that has a cardinality that is not larger than twice the cardinality on a smallest VC.

Moreover, 3. the algorithm runs in polytime.

P: The algorithm iterates for as long as there are edges in the graph.
Whenever the algorithm picks an edge $\{u,v\}$, it adds both "u" and "v" to the partial solution S.
In doing so, it removes from the graph all the edges incident on "u" or "v".

Thus, the final solution is a vertex cover.
(① is proved)

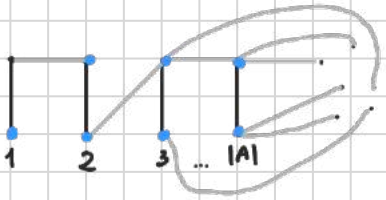
Let $A \subseteq E$ is the set of edges picked by the algorithm.

Observe that if $\{u,v\}, \{x,y\} \in A$, with $\{u,v\} \neq \{x,y\}$, then $\{x,y\} \cap \{u,v\} = \emptyset$.

(Indeed, if, W.L.O.G., $\{u,v\}$ is picked before $\{x,y\}$ then all the edges incident on u, or v, are removed from the graph — Thus, when $\{x,y\}$ is picked, it cannot be that $x=u$, or $x=v$, or $y=u$, or $y=v$.).

Then, the solution returned by the algorithm has size $2|A|$: The algorithm adds to "S" 2 new vertices for each edge of A.

Recall that $A \subseteq E$. Let us consider the graph $G(V, A)$.

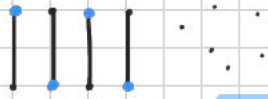


black edges are those in A

grey edges are those in E-A

Since $A \subseteq E$, if $T \subseteq V$ is a vertex cover for $G(V,E)$, then T must be a = = = $G(V,A)$ as well.

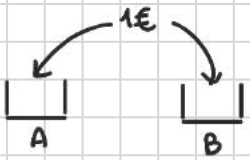
Now, in $G(V, A)$ no two edges share an end-point.
Therefore each vertex cover T for $G(V, A)$ must contain at least one node per edge — that is, at least $|A|$ nodes.

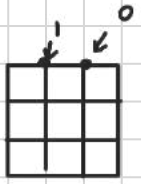


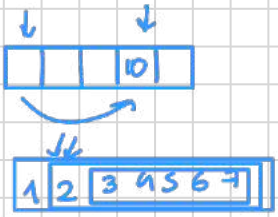
Thus, the minimum VC for $G(V, E)$ must contain at least $|A|$ nodes.
Since our algorithm returns a solution with $2|A|$ nodes, (2) is proved. ■

It is NP-HARD to approximate Independent Set to $n^{0.99}$.

RANDOMIZED ALGORITHM







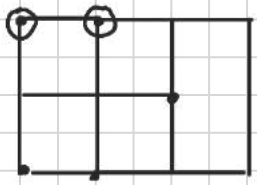

```
ALG A ()
    OPEN BOX A
```

0 €

```
ALG B ()
    OPEN BOX B
```

```
ALG' ()
    FLIP A COIN
    IF THE COIN COMES UP HEADS:
        OPEN BOX A
    ELSE:
        OPEN BOX B
```

$$E[x] = \frac{1}{2}(0€) + \frac{1}{2}(1€) = 0{,}5\ €$$



$\Rightarrow$ $\sqcup\ \sqcup\ \sqcup\ \sqcup$

$B_0\ B_1\ B_2\ B_3$

$\{B_0, B_1, B_2, B_3\} = \{1, 10, 100, 1000\}$

$= \{1, 1, 1, 1\}$