Definizione di algoritmo

un algoritmo è un **insieme di istruzioni ordinate**, non ambigue ed effettivamente eseguibili, seguendo le quali per ogni **input** appropriato si ottiene un **output** in un numero finito di passi

Un algoritmo deve dunque possedere alcune caratteristiche:

L’input

deve essere finito e non ambiguo, e deve essere stabilito a priori il modo con cui i dati in ingresso vengono rappresentati.

Ad esempio, in un algoritmo per calcolare il prodotto di due numeri, i due numeri devono essere finiti, e rappresentati in una base (es. binaria o esadecimale) decisa a priori

Insieme di istruzioni ordinate, non ambigue ed effettivamente eseguibili

Le istruzioni devono cioè appartenere a un insieme finito e prefissato di operazioni di base che sappiamo eseguire.

Ad esempio, gli algoritmi per preparare ricette dovranno essere composti da una combinazione delle operazioni di base (versa, sposta, togli, mescola, aspetta…)

La codifica

L’algoritmo deve essere descritto in modo preciso e non ambiguo

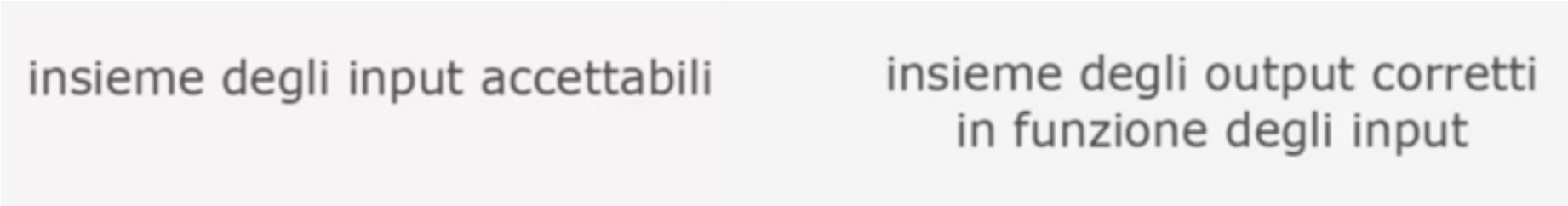
Ad esempio, gli algoritmi per computer saranno codificati con uno dei tanti linguaggi di programmazione disponibili

La terminazione

Per ogni input appropriato, l'esecuzione dell'algoritmo deve terminare in un numero finito di passi, cioè dopo aver eseguito un numero finito, (ma in generale non uguale per tutti gli ingressi) di istruzioni elementari.

Ad esempio, preparare una pasta al pomodoro richiederà un numero ben preciso di passi, meno di quelli necessari per preparare la pasta al forno

Più formalmente, un algoritmo è la soluzione ad un certo problema computazionale (ad esempio trovare il prodotto di due numeri), ed è ben definito quando siano ben definiti:



Un algoritmo risolve un problema algoritmico se produce un output corretto per ogni input ammissibile

Dato un certo tipo di problema algoritmico, cioè che può essere risolto con un algoritmo, esistono di solito algoritmi diversi che risolvono il problema.

Ad esempio, esistono diversi algoritmi di ordinamento (di solito di numeri, o di nomi) e diversi algoritmi per cercare il percorso più breve tra due o più punti (ad esempio città) su una mappa

Gli algoritmi per risolvere un certo problema non sono tutti uguali, alcuni sono più efficienti di altri, cioè richiedono meno lavoro (ossia meno tempo)

Ad esempio, alcuni algoritmi richiedono un tempo proporzionale a n^2 per ordinare n numeri, mentre altri algoritmi richiedono un tempo proporzionale solo a $n*\log_2 n$. Se n è molto grande, la differenza può essere notevole.

Ad esempio, se $n = 1.000$, allora $1.000^2 = 1.000.000$ mentre $1.000*\log_2 1.000 < 10.000$

Anche i problemi non sono tutti uguali:

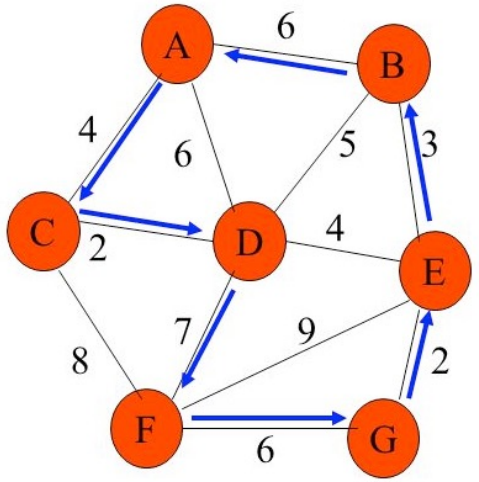
alcuni problemi - ad esempio ordinare *n* elementi - hanno soluzioni algoritmiche efficienti, si dice allora che sono **trattabili** (formalmente, si risolvono in un numero di passi non superiore a un polinomio in *n*)

Altri problemi - ad esempio la torre di Hanoi con *n* piatti - non hanno soluzioni efficienti, e si dice allora che sono **intrattabili** (formalmente, si risolvono in un numero di passi proporzionale ad una funzione esponenziale in *n)*

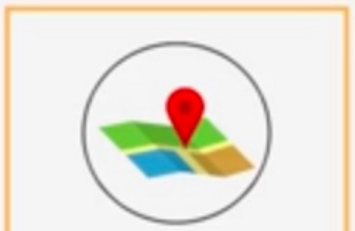
https://tinyurl.com/36k4tecu

Di altri problemi ancora, **detti NP-completi**, si sa che esistono soluzioni inefficienti, e non si sa se esistono soluzioni efficienti, ma si sospetta che non ne esistano

Ciclo hamiltoniano (Tour)



E' un ciclo che visita TUTTI i nodi UNA SOLA volta



CICLO DI HAMILTON

problemi NP-completi

tempo esponenziale

problemi trattabili?

Infine, alcuni problemi sono addirittura **insolubili**: possono esistere soluzioni algoritmiche per alcune istanze del problema, ma esisterà sempre almeno una istanza che nessun algoritmo saprà trattare (ad esempio, il problema di dire se due programmi sono equivalenti)

Il piccolo e il grande: algoritmi di ordinamento

I computer hanno spesso bisogno di organizzare un insieme di dati in una lista o sequenza ordinata

In questo modo, è più facile trovare uno dei dati della lista quando ne abbiamo bisogno

Possiamo molto facilmente sapere quali sono il primo e l'ultimo elemento della lista

Individuiamo velocemente eventuali elementi ripetuti, perché saranno vicini nella lista

Per questo in informatica si studiano a fondo gli **algoritmi di ordinamento**: algoritmi che ricevono in input una lista disordinata di elementi restituiscono in output una lista fatta dagli stessi elementi, ma ordinati secondo il criterio scelto

Ad esempio, 4 5 9 3 1 diventerà 1 2 4 5 9 mentre: *Rossi, Bianchini, Basile, Marchi, Conte* diventerà:

Basile, Bianchini, Conte, Marchi, Rossi

Esistono diversi algoritmi di ordinamento. Come già osservato, si distinguono tra loro principalmente rispetto alla quantità di lavoro necessaria per ordinare un insieme di elementi.

Ma come potremmo misurare la quantità di lavoro necessaria a ordinare un insieme di elementi?

Gli algoritmi operano di solito usando come passo base il confronto fra loro di due degli elementi da ordinare, in modo da capire quale viene prima e quale viene dopo.

Quindi possiamo stimare l'efficienza di un algoritmo di ordinamento dal numero di confronti necessari a produrre una lista ordinata a partire da una disordinata. Ad esempio:

4 2 1 3	confronta e scambia 2 e 4
2 4 1 3	confronta e scambia 4 e 1
2 1 4 3	confronta e scambia 4 e 3 e ricomincia
2 1 3 4	confronta e scambia 2 e 1
1 2 3 4	confronta: già ordinati
1 2 3 4	confronta: già ordinati, abbiamo finito!

Quindi possiamo stimare l'efficienza di un algoritmo di ordinamento dal numero di confronti necessari a produrre una lista ordinata a partire da una disordinata.

Se gli elementi da ordinare sono pochi, il numero di confronti è comunque basso, ma se dobbiamo ordinare migliaia o milioni di elementi, allora diversi algoritmi hanno prestazioni fra loro molto differenti.

Per ordinare 100.000 elementi, un algoritmo efficiente come **quicksort** è 2000 volte più veloce di un algoritmo di base come **selection sort** (vedremo entrambi più avanti)

Ma se gli elementi sono 1.000.000, allora quicksort diventa circa 20.000 volte più veloce di selection sort

Per questi ordini di grandezza, se quicksort ci mette pochi secondi a ordinare una lista di elementi, selection sort può metterci ore!

Problema: come ordinare un insieme di pesi?

(Attività che può essere anche proposta a bambini a partire dall'età di 8-9 anni)

Supponiamo di avere 3 oggetti identici fra loro ma di peso diverso. Abbiamo anche una bilancia a due braccia che ci permette di confrontare a coppie i due oggetti, ma non di sapere l'effettivo peso di ciascuno.

Come possiamo mettere i tre oggetti in ordine crescente rispetto al loro peso?

Problema: come ordinare un insieme di pesi?

Se gli oggetti sono A, B e C allora:

1) confronto A con B, e tengo il più leggero (es. B)
2) confronto B con C e tengo il più leggero (es. B)
3) metto da parte B, che è il più leggero dei tre
4) confronto A e C, e così
   trovo il secondo più leggero
   (ad esempio C)
5) i pesi ordinati sono B,C,A

Selection sort: come funziona?

Selection sort usa questo principio: dati n elementi, *seleziona* l'elemento più leggero e lo mette da parte, questo sarà il primo elemento della lista ordinata.

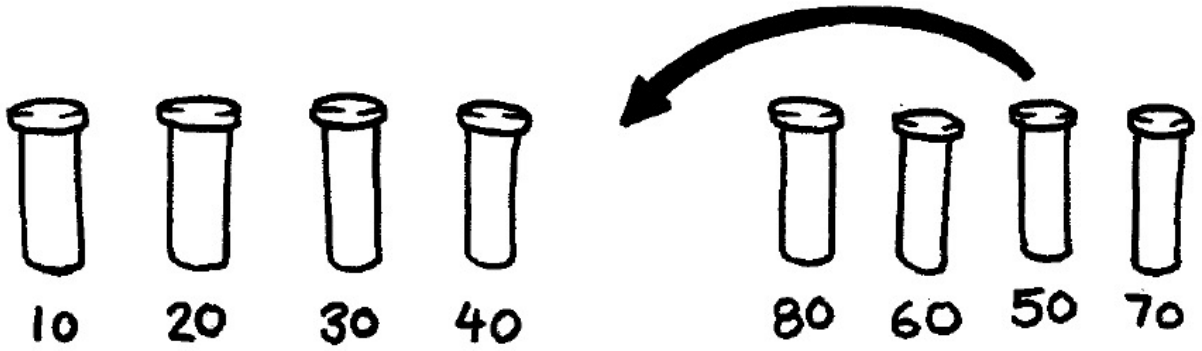
Poi ripete l'operazione con gli n-1 elementi rimasti: l'oggetto più leggero *selezionato* viene messo da parte come secondo elemento della lista ordinata.

Poi ripete l'operazione con gli n-2 elementi rimasti, ecc. ecc.

Selection sort: quanti confronti?

All'inizio, per trovare l'oggetto più leggero, ogni oggetto va confrontato con ogni altro: n-1 confronti

Al secondo passaggio faremo n-2 confronti, a terzo passaggi ne faremo n-3, e così via.



Selection sort: algoritmo inefficiente

In pratica faremo quindi (n-1)+(n-2)+(n-3)+…+1 confronti, pari a n*(n-1)/2 confronti.

Dunque, Selection sort richiede un numero di confronti quadratico rispetto al numero di elementi da ordinare. Selection sort è una algoritmo di ordinamento semplice ma inefficiente.

Selection sort: esempio

Ordiniamo questi nomi secondo il selection sort:

| Cavolo Pomodoro Zucchina Albicocca Pesca Lattuga Basilico

Albicocca | Cavolo Pomodoro Zucchina Pesca Lattuga Basilico

Albicocca Basilico | Cavolo Pomodoro Zucchina Pesca Lattuga

Albicocca Basilico Cavolo | Pomodoro Zucchina Pesca Lattuga

Albicocca Basilico Cavolo Lattuga | Pomodoro Zucchina Pesca

Albicocca Basilico Cavolo Lattuga Pesca | Pomodoro Zucchina

Albicocca Basilico Cavolo Lattuga Pesca Pomodoro | Zucchina

Albicocca Basilico Cavolo Lattuga Pesca Pomodoro Zucchina |

Selection sort: osservazione

Il computer non può ”osservare” la porzione di lista disordinata e a colpo d’occhio capire che *Lattuga* è il prossimo elemento da selezionare.

Albicocca Basilico Cavolo | Pomodoro Zucchina Pesca Lattuga

Deve invece: confrontare *Pomodoro* e *Zucchina* e “tenere” *Pomodoro*; confrontare *Pomodoro* e *Pesca* e “tenere” *Pesca*; confrontare *Pesca* e *Lattuga* e “tenere” *Lattuga*. Dato che siamo a fine lista, *Lattuga* è il prossimo elemento da selezionare.

Insertion sort: come funziona?

Insertion sort toglie il primo elemento della lista da ordinare e lo *inserisce* in una lista vuota, che quindi risulta banalmente ordinata.

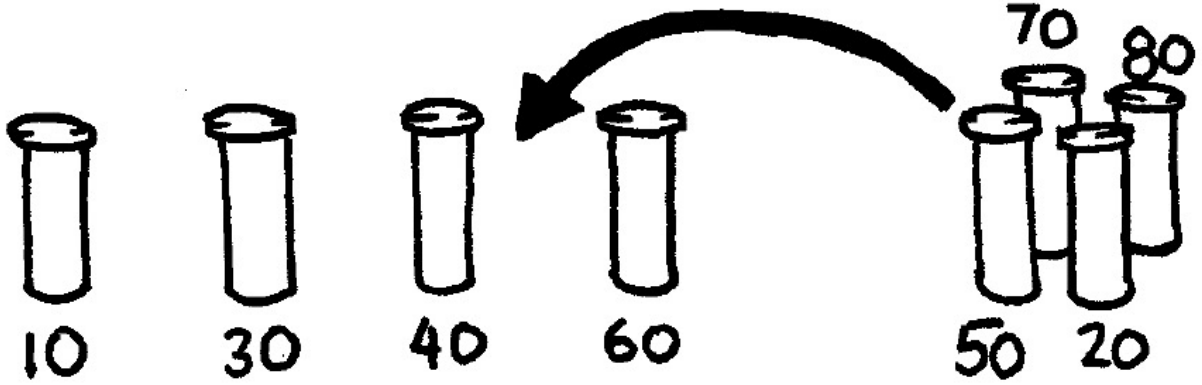
Poi Insertion sort prende il primo elemento della lista da ordinare rimasta e lo *inserisce* nella giusta posizione nella lista ordinata.

L'operazione viene ripetuta fino a che la lista iniziale è vuota e quella ordinata contiene tutti gli elementi

Insertion sort: quanti confronti?

Ogni elemento spostato dalla lista iniziale a quella ordinata va inserito nel punto giusto.

Se l'elemento va inserito in prima posizione, ci basta un confronto, ma se l'elemento va inserito in ultima posizione, andrà confrontato con tutti gli n elementi già presenti.



Insertion sort: algoritmo inefficiente

Nel caso medio dunque, il numero di confronti da fare per inserire un elemento nel punto giusto sarà n/2

Dovendo ripetere l'operazione per tutti gli elementi della lista iniziale, che man mano si allunga, il numero di confronti sarà proporzionale a $n^2/4$: anche insertion sort è un algoritmo inefficiente.

Una animazione di Insertion sort:

https://tinyurl.com/ymsmn65x

Insertion sort: un esempio

Ordiniamo questi nomi secondo il selection sort:

| Cavolo Pomodoro Zucchina Albicocca Pesca Lattuga Basilico

Cavolo | Pomodoro Zucchina Albicocca Pesca Lattuga Basilico

Cavolo Pomodoro | Zucchina Albicocca Pesca Lattuga Basilico

Cavolo Pomodoro Zucchina | Albicocca Pesca Lattuga Basilico

Albicocca Cavolo Pomodoro Zucchina | Pesca Lattuga Basilico

Albicocca Cavolo Pesca Pomodoro Zucchina | Lattuga Basilico

Albicocca Cavolo Lattuga Pesca Pomodoro Zucchina | Basilico

Albicocca Basilico Cavolo Lattuga Pesca Pomodoro Zucchina |

Bubble sort: Come funziona?

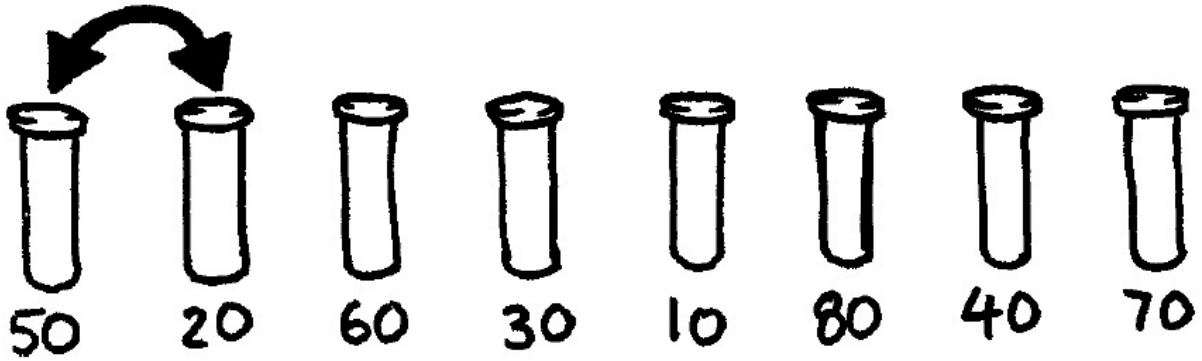
Bubble sort si chiama così perché fa *affiorare* come bolle gli elementi più leggeri verso la parte sinistra della lista iniziale, scorrendola più volte fino a che tutti gli elementi si trovano nella giusta posizione.

1) Scorri la lista da sinistra a destra: per ogni coppia di elementi, se sono nell'ordine sbagliato scambiali.

2) Ripeti più volte il punto 1. Quando non fai più nessuno scambio, hai ottenuto una lista ordinata.

Bubble sort: quanti confronti?

Si può dimostrare che anche bubble sort è un algoritmo inefficiente, in quanto richiede un numero di confronti proporzionale a $n^2/2$ confronti



Una animazione di bubble sort:

https://tinyurl.com/ymsmn65x

Bubble sort: un esempio

Cavolo Pomodoro Zucchina Albicocca Pesca Lattuga Basilico

Cavolo Pomodoro Zucchina Albicocca Pesca Lattuga Basilico

Cavolo Pomodoro Zucchina Albicocca Pesca Lattuga Basilico

Cavolo Pomodoro Albicocca Zucchina Pesca Lattuga Basilico

Cavolo Pomodoro Albicocca Pesca Zucchina Lattuga Basilico

Cavolo Pomodoro Albicocca Pesca Lattuga Zucchina Basilico

Cavolo Pomodoro Albicocca Pesca Lattuga Basilico Zucchina

Cavolo Pomodoro Albicocca Pesca Lattuga Basilico Zucchina

Cavolo Albicocca Pomodoro Pesca Lattuga Basilico Zucchina

Cavolo Albicocca Pesca Pomodoro Lattuga Basilico Zucchina

Cavolo Albicocca Pesca Lattuga Pomodoro Basilico Zucchina

Cavolo Albicocca Pesca Lattuga Basilico Pomodoro Zucchina

Quicksort: un algoritmo avanzato

Quicksort appartiene alla categoria degli algoritmi di ordinamento avanzati, e ordina una lista di n elementi in un tempo proporzionale a $n*\log_2 n$.

E' un algoritmo più sofisticato e complesso degli algoritmi di ordinamento di base che abbiamo visto, ma quando dobbiamo ordinare migliaia di elementi, quicksort è quasi sempre la scelta migliore

Quicksort: come funziona?

1) Scegli un elemento X a caso (detto “pivot”) nella lista da ordinare

2) metti gli elementi della lista più piccoli di X a sinistra di X (chiamiamo L_s questa sottolista)

3) Metti gli elementi della lista più grandi di X a destra di X (chiamiamo L_d questa sottolista)

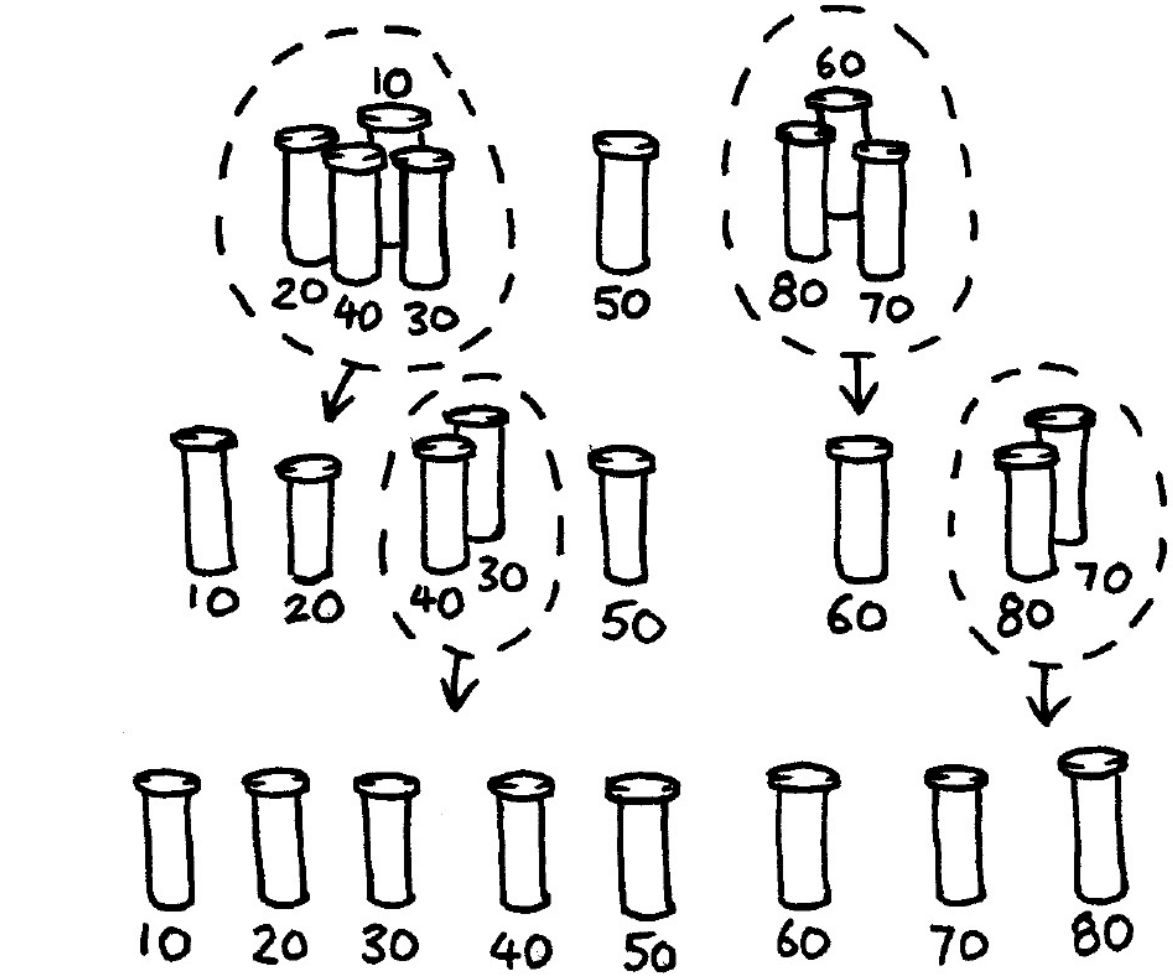
4) Ora ripeti la procedura dal punto 1) su L_s e su L_d

5) Quando ogni sottolista è formata da un solo elemento hai finito e la lista iniziale è ordinata

Quicksort: come funziona?

Quicksort è un classico algoritmo ricorsivo, in cui cioè la regola di base: *scegli un pivot X e costruisci due sottoliste L_s e L_d rispetto a X*, viene riapplicata ricorsivamente a ogni sottolista fino a che non c'è più nulla da fare e possiamo fermarci.

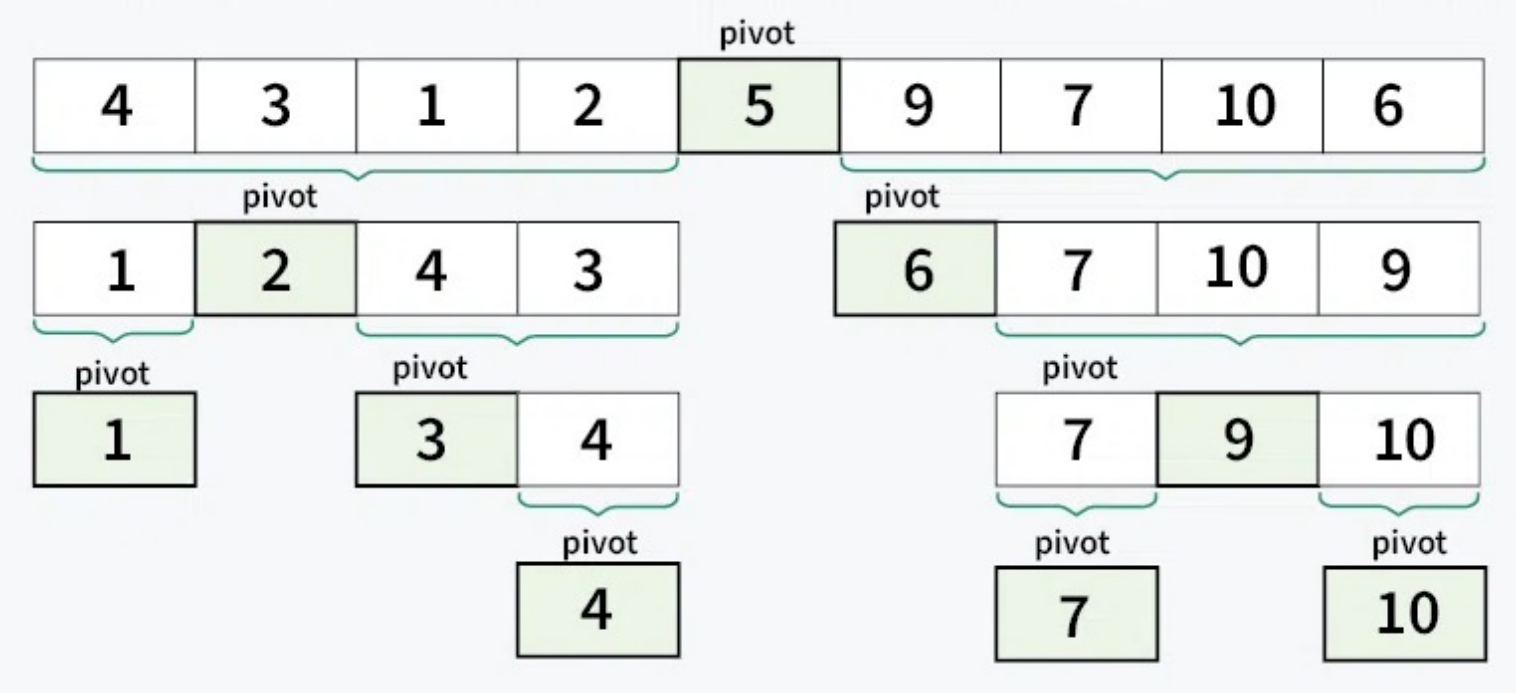
Quicksort usa una classica strategia *divide et impera*



Quicksort: un esempio

Notate che essendo il pivot scelto a caso, può anche succedere che tutti gli elementi si accumulino alla sua destra o alla sua sinistra: il pivot è allora l'elemento più grande o più piccolo della lista. Data la lista:

9 1 4 7 10 6 5 3 2. Pivot scelto = 5



Merge sort: un altro algoritmo avanzato e ricorsivo

Anche il merge sort (merge = *fusione*) è ricorsivo e lavora in un tempo proporzionale a $n*\log_2 n$, (ma ha bisogno di spazio aggiuntivo) e suddivide la lista di partenza L da ordinare in sottoliste sempre più piccole.

1) Per prima cosa, L viene suddivisa in due sottoliste L_a e L_b di dimensione uguale (a meno di un elemento se il numero n di elementi di L è dispari)

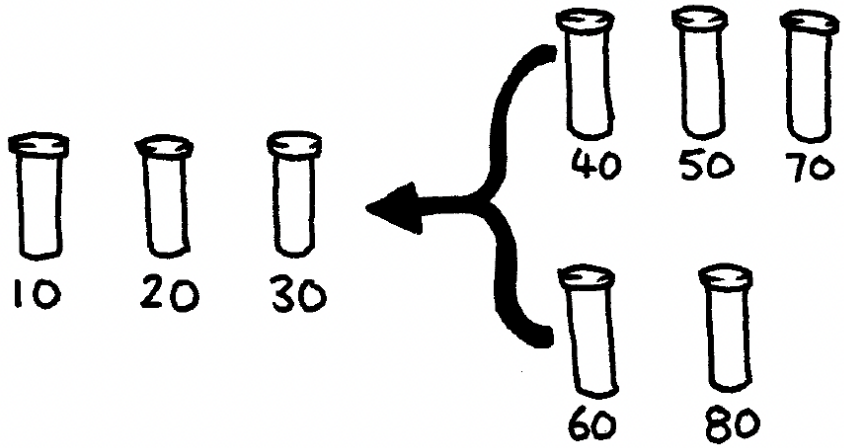
2) L_a e L_b vengono ordinate e poi fuse insieme in una unica lista ordinata rispettando l'ordinamento relativo degli elementi nelle due liste.

Merge sort: un altro algoritmo avanzato e ricorsivo

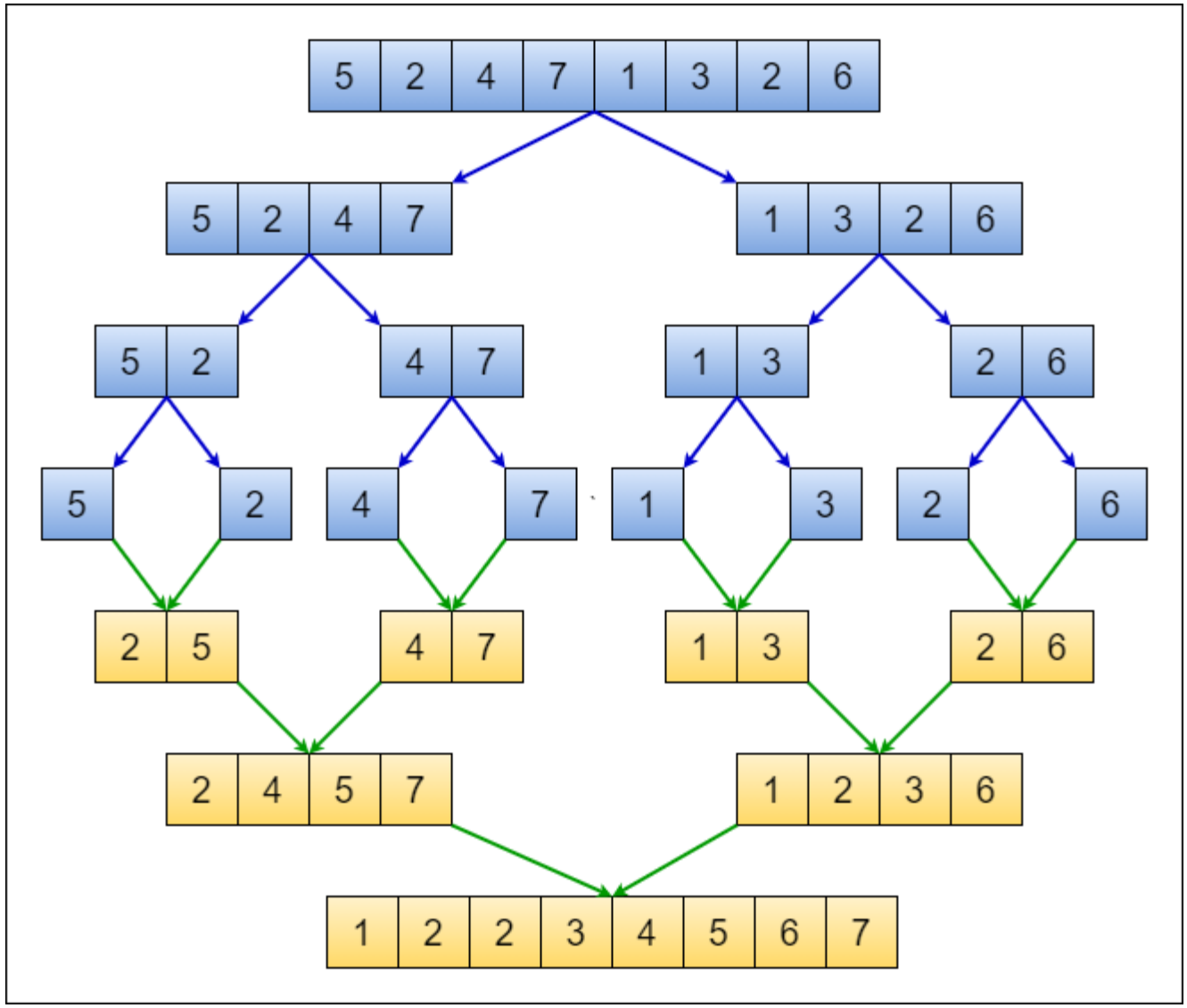
3) Però… come si fa a ordinare L_a e L_b? Semplice, usiamo merge sort su L_a e su L_b!

4) Infatti, quando una sottolista contiene un solo elemento, sarà automaticamente ordinata, e potrà essere fusa con l'altra sottolista, fino a ricostruire la lista iniziale, ma con tutti gli elementi ordinati.

5) Per fondere due liste ordinate basta scegliere ripetutamente l'elemento più piccolo di entrambe



Merge sort: esempio



quicksort vs mergesort: https://tinyurl.com/4e5xjmyu

La città fangosa: minimal spanning tree

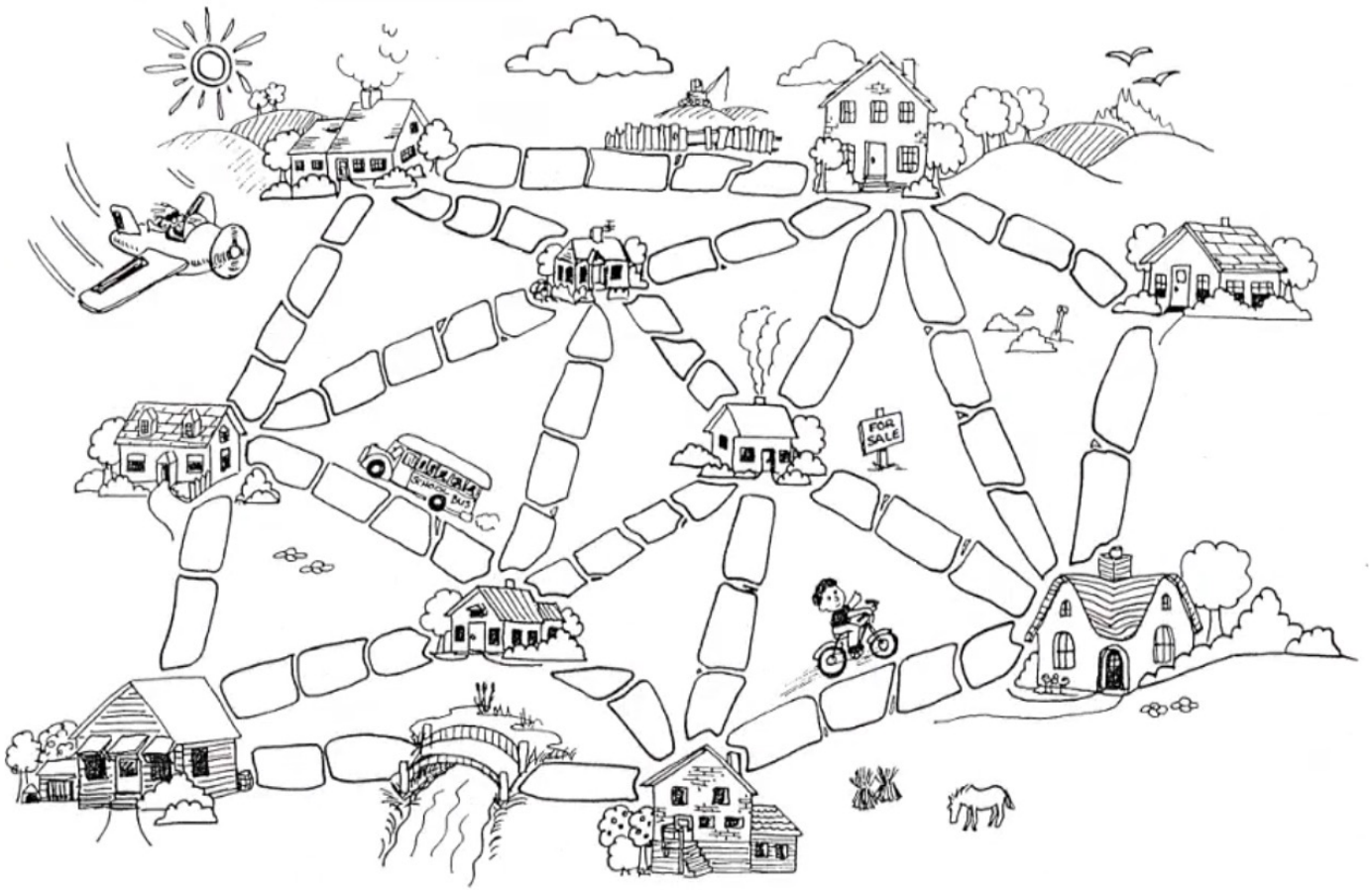
(Attività che può essere anche proposta a bambini a partire dall'età di 8-9 anni)

In una città fangosa, i cittadini sono scontenti, vorrebbero che i sentieri di collegamento fra le loro case fossero ricoperti da lastre di pietra.

Il sindaco vuole accontentare i suoi concittadini, ma i soldi scarseggiano. Come soddisfare la richiesta spendendo il minimo possible?

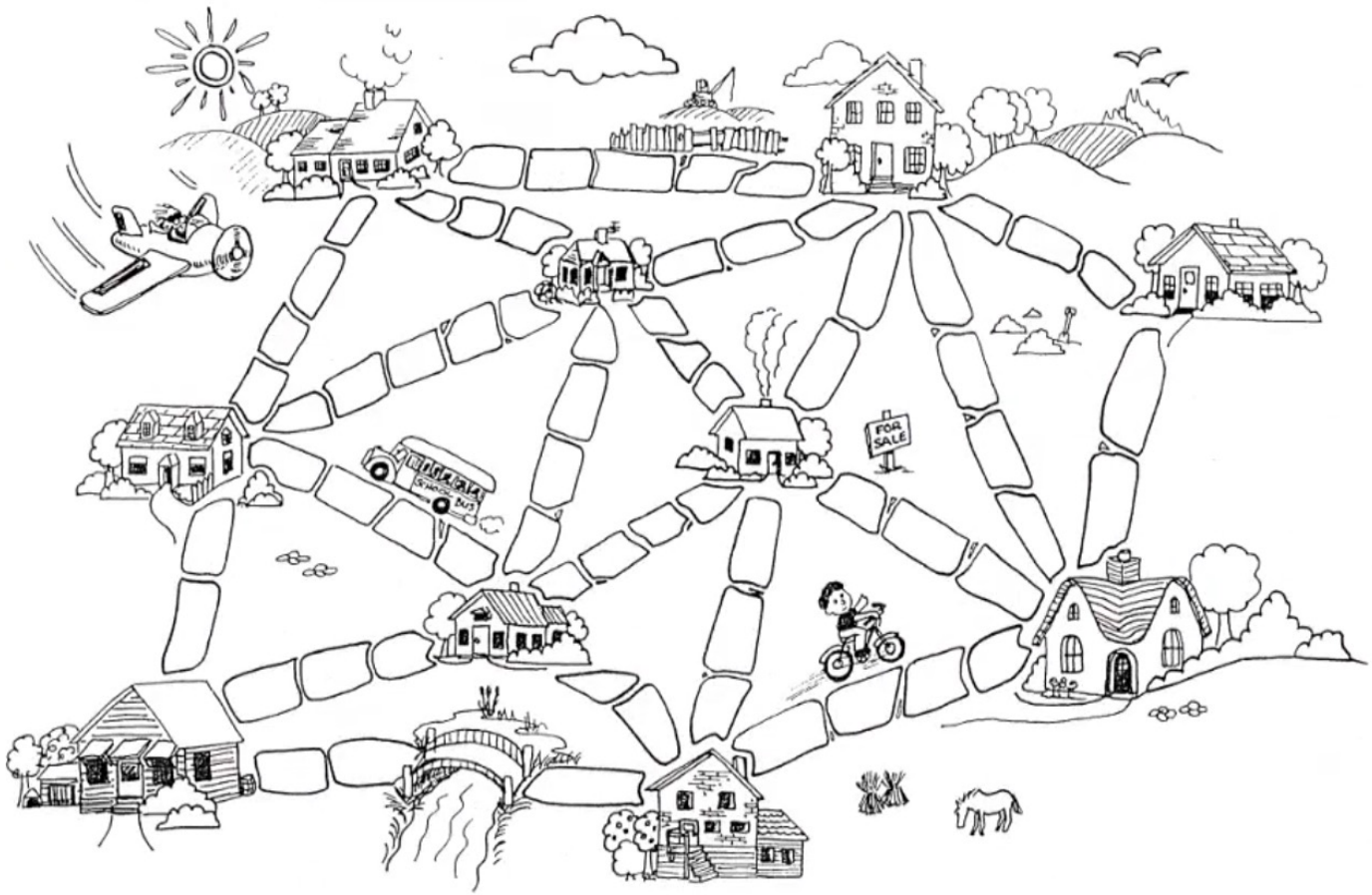
La città fangosa: minimal spanning tree

In altre parole, come permettere a ogni abitante di raggiungere la casa di ogni altro abitante passando solo per sentieri lastricati e spendendo meno possibile per lastricare i percorsi?



La città fangosa: minimal spanning tree

Notate: ogni sentiero può essere lastricato usando da certo numero di lastre di pietra: maggiore è la lunghezza del sentiero (numero di lastre necessarie), maggiore sarà la spesa per lastricare quel sentiero.

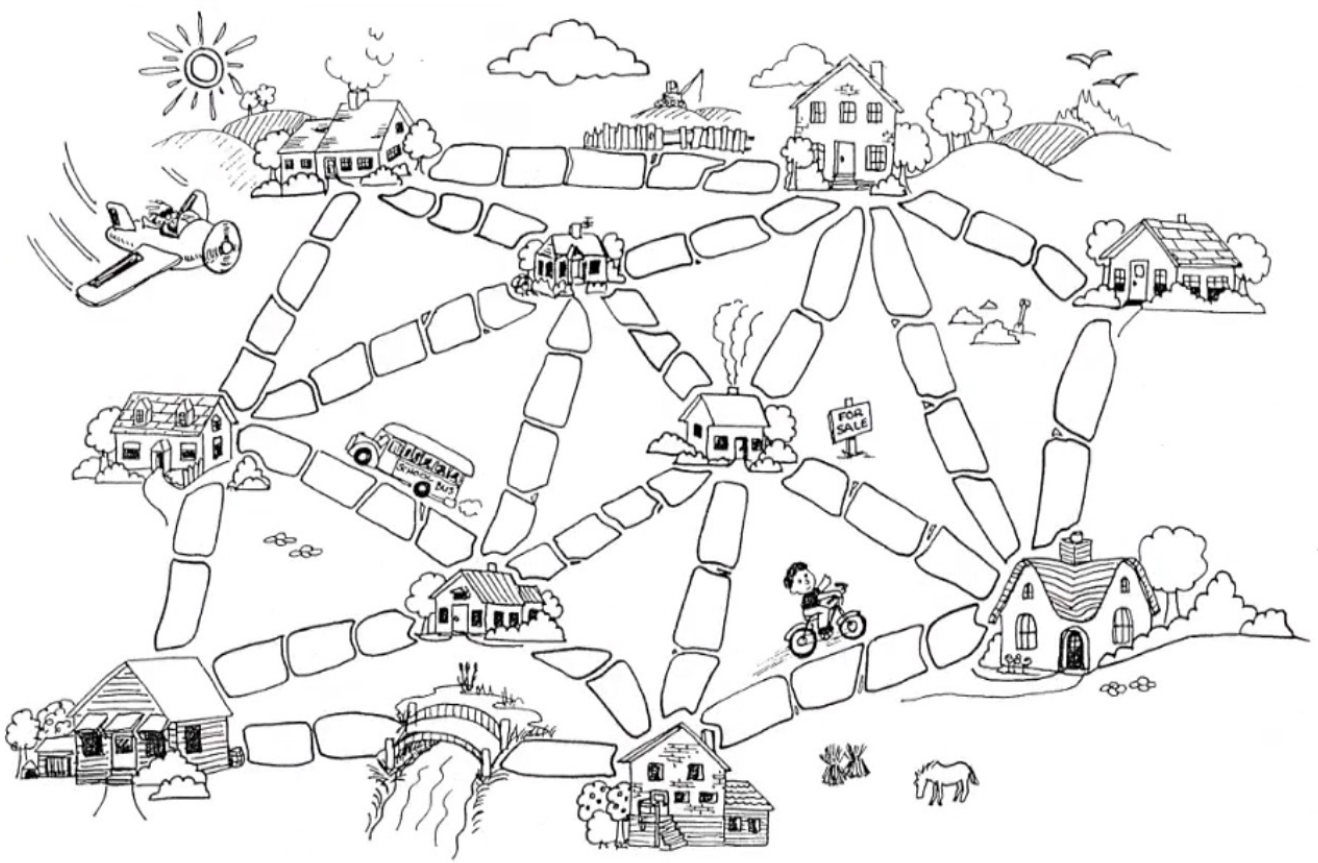


La città fangosa: minimal spanning tree

Dunque, per risparmiare, il sindaco non si preoccupa di lastricare ogni sentiero, ma solo far si che ogni casa sia collegata a ogni altra casa da un percorso lastricato, non necessariamente il più corto possible.

Quindi, non tutti i sentieri verranno lastricati.

Come decidere quali sentieri lastricare?

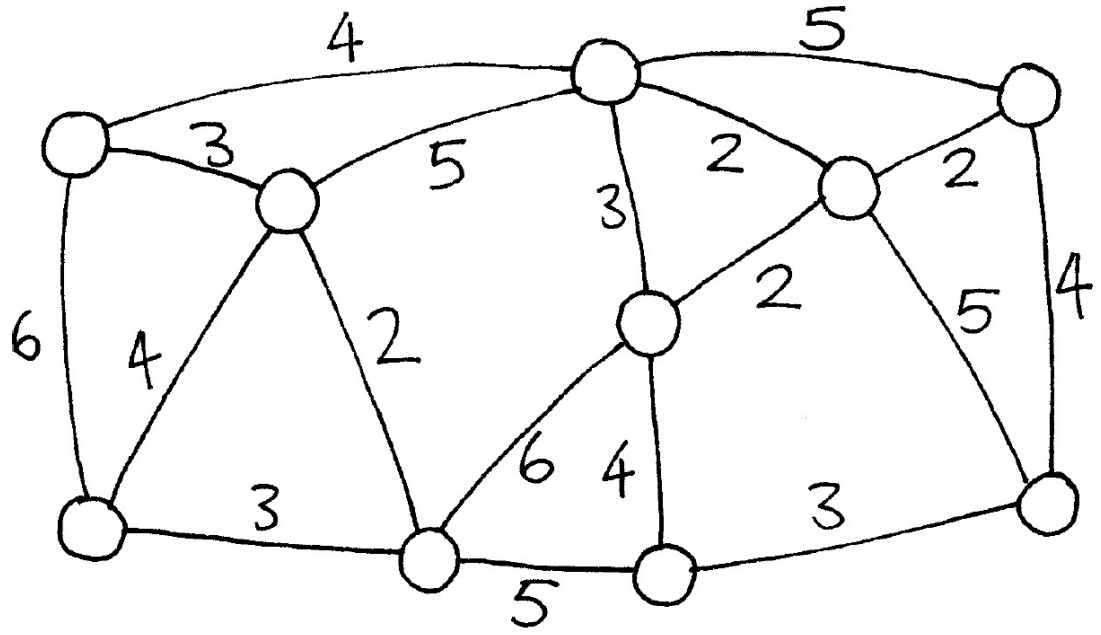


La città fangosa: minimal spanning tree

In informatica, la città fangosa è un esempio di *grafo*, in cui ci sono dei nodi (le case), e degli archi (i sentieri) di peso/costo diverso tra i nodi.

Il problema del *minimal spanning tree* consiste nel trovare un insieme di archi che colleghi a peso/costo minimo tutti i nodi

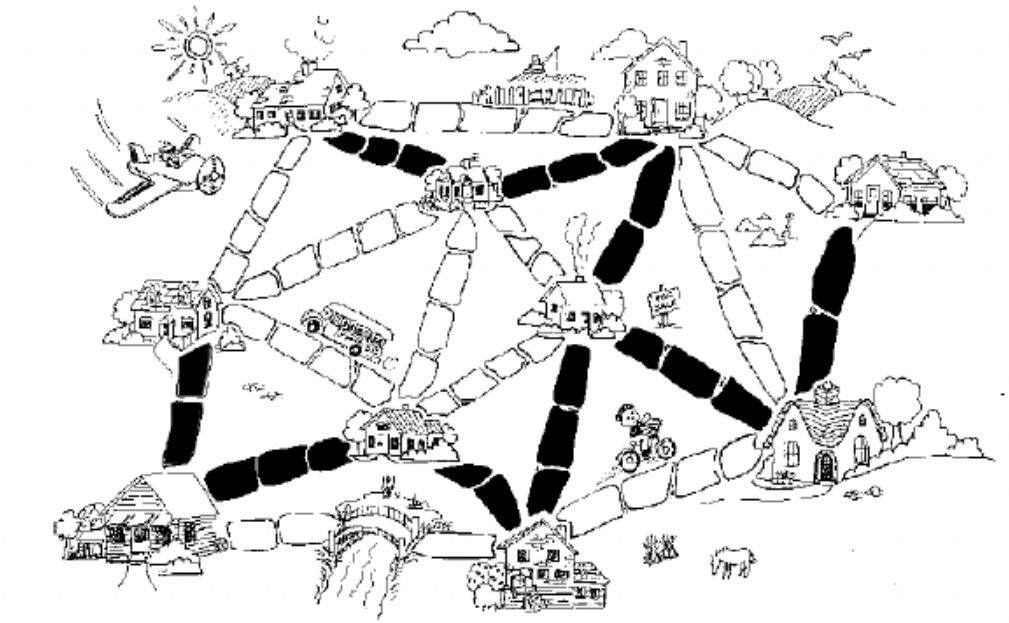
Occhio! Questo grafo non riproduce la città fangosa…

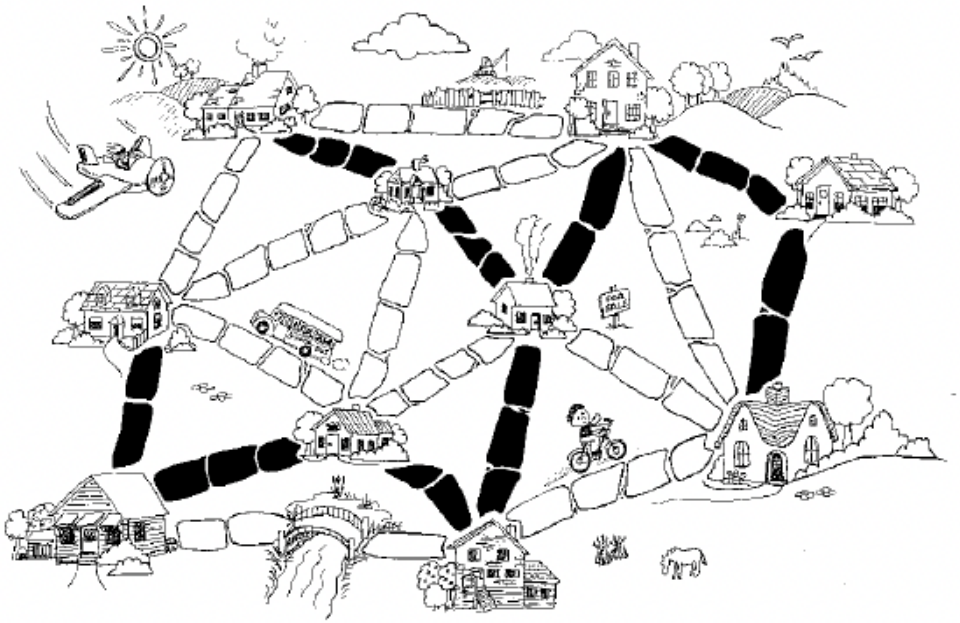


La città fangosa: l'algoritmo di Kruskal

Trova **il minimal spanning tree di un grafo pesato:** seleziona gli archi più leggeri in ordine crescente e li aggiunge all'albero, evitando di formare cicli.

Ecco due possibili soluzioni per il problema della città fangosa: sono equivalenti?





La città fangosa: minimal spanning tree

Molti problemi pratici hanno una formulazione simile a quella del minimal spanning tree.

Ad esempio, i collegamenti elettrici fra case, oppure il miglior modo di progettare le strade di una città, o anche in che modo connettere delle città con collegamenti aerei.

Inoltre, molti altri problemi simili possono essere ricondotti a problemi sui grafi, ad esempio il ***problema del commesso viaggiatore***: trovare il minimo percorso da seguire per visitare un insieme di città una ed una sola volta e ritornare alla città di partenza.

minimal spanning tree online

Nella sezione Algoritmica lezione 1- approfondimenti trovate alcuni ”giochi algoritmici online” che sono tipici problemi sui grafi:

Road Repairs:

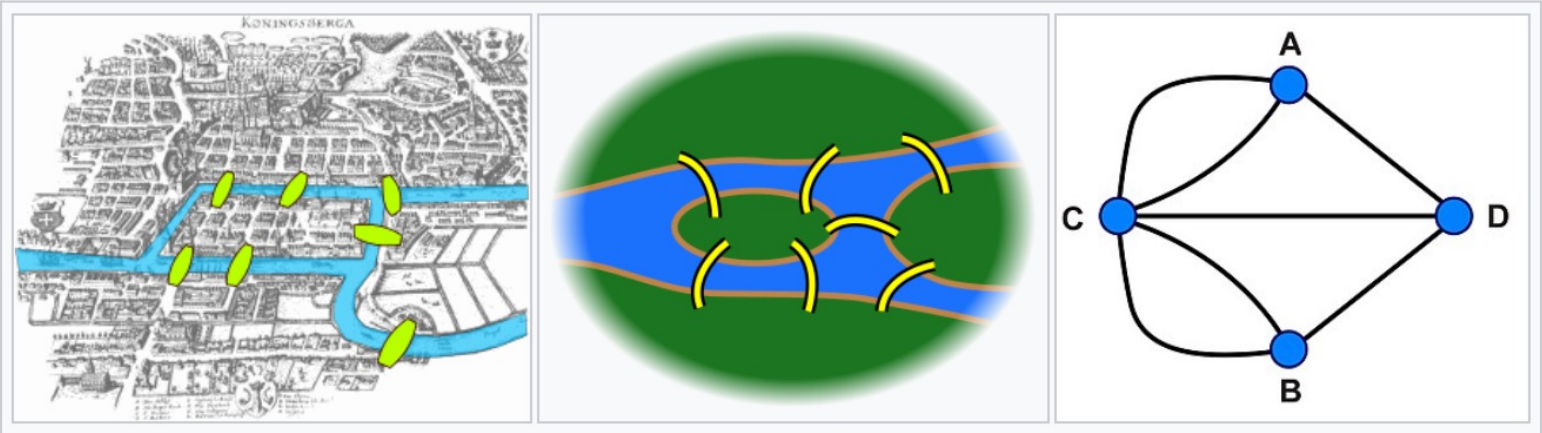
https://discrete-math-puzzles.github.io/puzzles/road-repair/index.html

Bridges of Königsberg:

https://discrete-math-puzzles.github.io/puzzles/bridges/index.html

Bridges of Königsberg

È uno dei più famosi problemi sui grafi, e si può riformulare con il grafo a destra nella figura



Il problema allora consiste nel passare una e una sola volta per ciascuno dei 7 archi, toccando tutti i 4 nodi

Nel 1736 il matematico Eulero dimostrò che il problema non ha una soluzione

Bridges of Königsberg

Se invece di 7 ponti ne dobbiamo attraversare solo 6, allora il problema ha una soluzione