Corso di TECNOLOGIE INFORMATICHE A.S. 2014/2015

Modulo: Algoritmi e Strutture Dati

Il modulo *Algoritmi e strutture dati* è rivolto agli studenti del primo anno delle scuole superiori all'interno del corso di Tecnologie informatiche. Ha l'obiettivo di presentare le strutture dati e gli algoritmi di base a diversi livelli di astrazione, nonché consentire l'acquisizione delle principali metodologie di progettazione e analisi degli algoritmi.

OBIETTIVI FORMATIVI

Obiettivo specifici del modulo sono:

- Definire formalmente la nozione di algoritmo e di modello di calcolo.
- Caratterizzare i dati da elaborare, organizzandoli e strutturandoli nel modo più opportuno al fine di agevolarne l'uso da parte degli algoritmi.
- Progettare algoritmi corretti (che risolvono cioè sempre e solo il problema a cui si è interessati) ed efficienti (cioè che lo risolvono il più velocemente possibile o usano il minor spazio di memoria possibile), attraverso l'esame di diversi paradigmi.
- Studiare le limitazioni inerenti ai problemi da risolvere, in particolare a quelli la cui soluzione richiede l'esame di tutte le possibilità.

La familiarizzazione con le tecniche algoritmiche permetterà l'implementazione concreta di algoritmi e strutture dati in un linguaggio di programmazione come, ad esempio, il linguaggio C.

PREREQUISITI E METODOLOGIE

Il corso previsto per le classi prime richiede conoscenze di base relative alle tecnologie informatiche, conoscenze generali su software di base e hardware, argomenti che, normalmente, vengono affrontati durante la prima metà dell'anno scolastico.

Si terranno principalmente lezioni frontali con esercitazioni singole e di gruppo per lo sviluppo di codice in laboratorio.

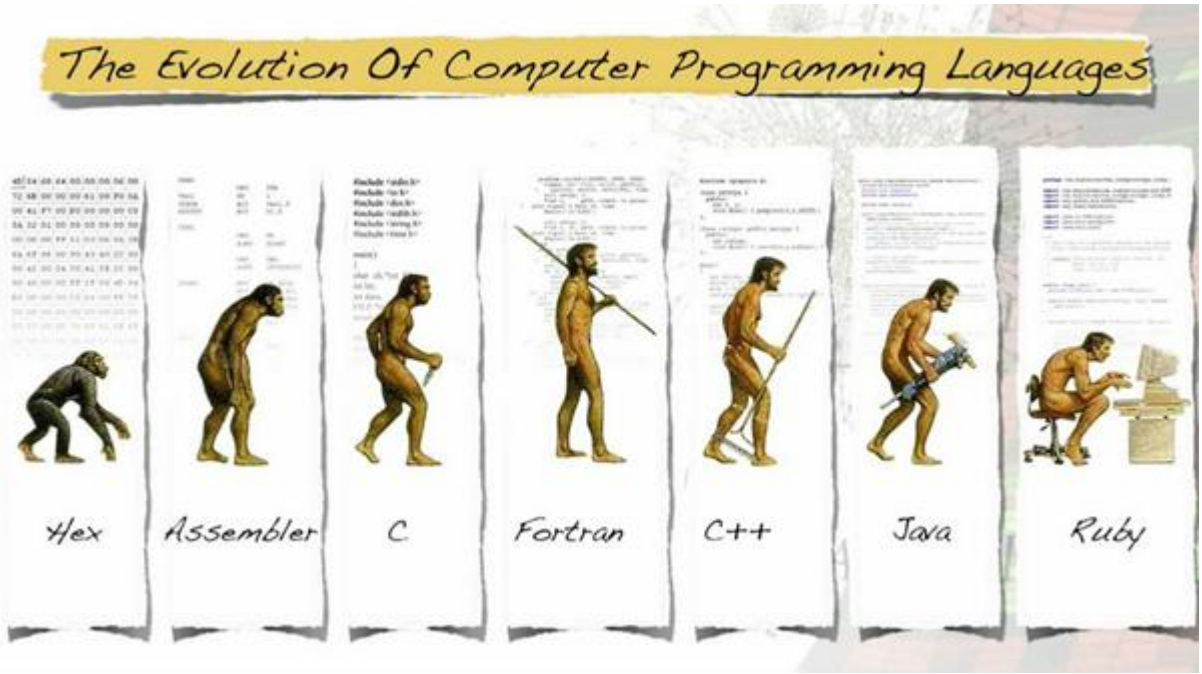
CONTENUTI

Introduzione agli algoritmi e alle strutture dati:

1. Algoritmi e loro tipologie
2. Proprietà degli algoritmi
3. Rappresentazione degli algoritmi
4. Classificazione degli algoritmi
5. Complessità di un algoritmo rispetto all'uso di risorse
6. Strutture dati e loro tipologie
7. Classi di problemi
8. Un esempio: trova la moneta falsa

MODALITÀ DI VERIFICA DELL'APPRENDIMENTO

Esercitazioni scritte e al computer di verifica, sviluppo di progetti.



INTRODUZIONE AGLI ALGORITMI E ALLE STRUTTURE DATI

1. Algoritmi e loro tipologie

Il termine algoritmo è ispirato dal nome del matematico e astronomo arabo del VII° secolo *Abu Ja Mohammed Ibn Musa Al-Khowarizmi* che, oltre ad essere l'inventore dello zero e dell'algebra, ha la paternità di quello che è considerato uno dei primi algoritmi della storia: il metodo per sommare due numeri incolonnandoli e procedendo cifra per cifra da destra verso sinistra tenendo conto dei riporti.

Un algoritmo è una sequenza finita di passi interpretabili da un esecutore. Un'altra fra le numerose definizioni possibili può essere la seguente: «metodo di elaborazione da applicare a certi dati iniziali per ottenere dei dati finali o risultati». In ogni caso è importante che, in partenza, il problema da risolvere sia «ben posto» e cioè che:

- sia chiaro l'obiettivo da raggiungere;
- i dati di partenza siano noti e sufficienti;
- il problema sia risolubile da parte di chi lo affronta.

Se manca una di queste condizioni l'algoritmo non può essere individuato.

Sebbene un algoritmo sia composto da un numero finito di passi, la sua esecuzione potrebbe richiedere un tempo non necessariamente finito. Si deve anche notare come sia importante, quando si affronta un problema, non confondere il risultato (ciò che si vuole ottenere) con la soluzione (il metodo che conduce al risultato elaborando i dati di partenza).

L'individuazione dell'algoritmo risolutivo di un problema è solo il primo passo di un procedimento in più fasi che conduce alla soluzione del problema stesso in modo automatico con l'utilizzo del sistema di elaborazione. Il passo successivo consiste nella trasformazione dell'algoritmo in un programma scritto usando un linguaggio di programmazione; si può affermare che l'algoritmo e il relativo programma sono due descrizioni, fatte con mezzi diversi, dello stesso metodo di soluzione.

L'esecutore di un algoritmo non deve necessariamente essere un computer, ma può essere un dispositivo meccanico, un circuito elettronico o un sistema biologico. Quand'anche l'esecutore fosse un computer, un algoritmo non deve necessariamente essere espresso in un linguaggio di programmazione, in quanto esso si trova comunque ad un livello di astrazione più alto rispetto ad ogni programma che lo implementa. Un algoritmo non deve necessariamente essere espresso attraverso un supporto linguistico. A volte può essere più comodo rappresentare un algoritmo graficamente con un albero di decisione.

2. Proprietà degli algoritmi

Distinguiamo l'aspetto della risoluzione del problema da quello dell'esecuzione del relativo algoritmo; distinguiamo cioè tra 'risolutore' (l'uomo, che individua il metodo di soluzione del problema) ed 'esecutore' (la macchina che deve eseguire tale metodo, descritto tramite un algoritmo). L'algoritmo è solo la 'descrizione delle operazioni' da svolgere; è quindi un'entità statica costituita da istruzioni, anch'esse statiche. Quando l'algoritmo viene eseguito si ottiene un 'flusso d'esecuzione' che invece è ovviamente dinamico ed è costituito da 'passi' (ogni passo è l'esecuzione di un'istruzione). A tale proposito può essere utile pensare alla differenza esistente tra una ricetta di una torta (descrizione delle operazioni da svolgere) e la sua «esecuzione» per la realizzazione della torta.

Nel contesto informatico l'esecutore, cioè il sistema di elaborazione, è una macchina non intelligente, capace solo di eseguire istruzioni molto elementari (anche se in modo rapidissimo), ma senza alcuna capacità critica o di ragionamento autonomo.

Un algoritmo deve allora avere le seguenti proprietà:

- '**generalità**', della quale abbiamo accennato in precedenza: non è un algoritmo il metodo che mi permette di stabilire se 19 è un numero primo, lo è invece il metodo che permette di decidere se un qualsiasi numero naturale è primo;
- '**comprensibilità**': l'algoritmo deve essere espresso in una forma comprensibile all'esecutore; questo non vuol dire che gli algoritmi debbano essere espressi nel linguaggio proprio della macchina, cioè quello binario, e neanche che si debbano subito utilizzare i linguaggi di programmazione, ma significa che si deve rispettare fin da subito un formalismo abbastanza rigido nella loro stesura;

- '**eseguibilità**': l'algoritmo deve anche essere eseguibile dalla macchina e quindi non deve prevedere operazioni ad essa sconosciute (l'insieme delle istruzioni eseguibili da un computer ha 'cardinalità finita') o che non possano essere eseguite in un tempo finito (le istruzioni devono avere 'complessità finita');
- '**finitezza**': l'algoritmo deve avere un numero finito di istruzioni;
- '**riproducibilità**': esecuzioni successive dell'algoritmo sugli stessi dati di input devono dare lo stesso risultato;
- '**non ambiguità**': non possono esserci istruzioni vaghe, che prevedano un comportamento probabilistico o la scelta casuale del prossimo passo da eseguire; la macchina non è infatti in grado di prendere decisioni complesse con ragionamento autonomo e quindi il metodo di soluzione deve essere 'completo' (devono essere previste tutte le possibilità che possano verificarsi durante l'esecuzione) e 'deterministico' (in caso contrario verrebbe meno anche la caratteristica della riproducibilità);
- '**discretezza**': l'esecuzione deve avvenire per passi discreti; i dati coinvolti nei calcoli assumono valori che cambiano tra un passo e il successivo e non assumono altri valori «intermedi»;
- '**non limitatezza dell'input**': non deve esserci un limite (almeno in linea teorica) alla lunghezza dei dati di input (e quindi anche alla capacità di memoria dell'esecutore);
- '**non limitatezza dei passi d'esecuzione**': devono essere possibili (almeno in linea teorica) esecuzioni costituite da un numero infinito di passi.

Le ultime due proprietà possono sembrare irrealistiche nelle applicazioni pratiche, ma sono necessarie per i seguenti motivi:

- se ci fosse un limite alla lunghezza dei dati di input verrebbe meno la proprietà della generalità: ad esempio non potremmo più concepire un algoritmo per decidere se un qualsiasi numero naturale è primo oppure no;
- esistono funzioni che si dimostra essere computabili a patto di ammettere un numero infinito di passi nell'algoritmo di calcolo.

Si deve comunque notare che queste grandi potenzialità teoriche degli algoritmi (input di lunghezza infinita e esecuzioni con infiniti passi) non sono sufficienti ad assicurare che tutte le funzioni siano computabili o, detto in altri termini, che tutti i problemi prevedano un algoritmo risolutivo.

Esiste ad esempio il famoso problema della «terminazione degli algoritmi»: si può dimostrare che «non esiste un algoritmo che permetta di stabilire se un qualsiasi altro algoritmo abbia termine».

3. Rappresentazione degli algoritmi

Per quanto detto in precedenza appare del tutto impraticabile l'idea di affidare la descrizione di un algoritmo all'uso del linguaggio naturale (nel nostro caso la lingua italiana) con il rischio di introdurre ambiguità dovute alla presenza di sinonimi, omonimie, modi diversi di intendere la stessa frase. Si pensi ad esempio alle possibili diverse interpretazioni che si possono dare alle seguenti frasi: «succede al Sabato», «lavoro solo da due anni», «mi piace molto la pesca».

Le interpretazioni e le sfumature delle frasi espresse in linguaggio naturale, spesso dipendenti dal contesto, possono essere colte solo dalla mente umana che è un «esecutore» molto più sofisticato, versatile ed elastico di qualsiasi macchina. Per descrivere gli algoritmi si deve, quindi, ricorrere a 'linguaggi formali' creati appositamente (e quindi artificiali); in queste dispense prenderemo in considerazione due esempi:

- un linguaggio 'lineare' basato sul testo, spesso denominato 'linguaggio di progetto';
- un linguaggio 'grafico' basato su simboli chiamato 'diagramma di flusso' (*flow chart*) o 'diagramma a blocchi'.

Notiamo che, qualsiasi sia il metodo utilizzato per la rappresentazione dell'algoritmo, vale sempre la regola che le operazioni da esso descritte vengono eseguite una alla volta nell'ordine in cui sono scritte (a meno che non intervengano costrutti di programmazione particolari che alterano il normale flusso di esecuzione).

4. Classificazione degli algoritmi

In base al numero di passi che si possono eseguire contemporaneamente possiamo avere algoritmi:

- **sequenziali**: eseguono un solo passo alla volta.
- **paralleli**: possono eseguire più passi per volta, avvalendosi di un numero prefissato di esecutori.

In base al modo in cui risolvono le scelte gli algoritmi possono essere considerati:

- **deterministici**: ad ogni punto di scelta, intraprendono una sola via determinata in base ad un criterio prefissato (p.e. considerando il valore di un'espressione aritmetico-logica).
- **probabilistici**: ad ogni punto di scelta, intraprendono una sola via determinata a caso (p.e. lanciando una moneta o un dado).
- **non deterministici**: ad ogni punto di scelta, esplorano tutte le vie contemporaneamente (necessitano di un numero di esecutori generalmente non passabile a priori).

In questo modulo assumeremo che l'esecutore sia un computer e ci limiteremo a considerare solo gli algoritmi sequenziali e deterministici. Useremo inoltre il linguaggio ANSI C per formalizzare tali algoritmi.

5. Complessità di un algoritmo rispetto all'uso di risorse

Dato un problema, possono esistere più algoritmi che sono corretti rispetto ad esso. Questi algoritmi possono essere confrontati rispetto alla loro complessità o efficienza computazionale, cioè rispetto alla quantità di uso che essi fanno delle seguenti risorse durante la loro esecuzione:

- Tempo di calcolo.
- Spazio di memoria.
- Banda trasmissiva.

Poiché tramite un'opportuna gerarchia di memorie è possibile avere a disposizione una capacità di memoria praticamente illimitata e inoltre gli algoritmi che considereremo non richiedono lo scambio di dati tra computer, in questo modulo ci concentreremo sulla complessità temporale degli algoritmi.

Un motivo più profondo per concentrare l'attenzione sulla complessità temporale è che lo spazio e la banda occupati in un certo momento possono essere riusati in futuro (risorse recuperabili), mentre il passare del tempo è irreversibile (risorsa non recuperabile).

Dato un problema, è sempre possibile trovare un algoritmo che lo risolve in maniera efficiente?

6. Strutture dati e loro tipologie

C'è uno stretto legame pure tra algoritmi e strutture dati, in quanto le strutture dati costituiscono gli ingredienti di base degli algoritmi.

Anche l'efficienza di un algoritmo dipende in maniera critica dal modo in cui sono organizzati i dati su cui esso deve operare.

Una struttura dati è un insieme di dati logicamente correlati e opportunamente memorizzati, per i quali sono definiti degli operatori di costruzione, selezione e manipolazione. Le varie strutture dati sono riconducibili a combinazioni di strutture dati appartenenti alle quattro classi fondamentali: array, liste, alberi e grafi.

Classificazione delle strutture dati basata sulla loro occupazione di memoria:

- **Strutture dati statiche**: la quantità di memoria di cui esse necessitano è determinabile a priori (array).
- **Strutture dati dinamiche**: la quantità di memoria di cui esse necessitano varia a tempo d'esecuzione e può essere diversa da esecuzione a esecuzione (liste, alberi, grafi).

7. Classi di problemi

7.1 Problemi decidibili e indecidibili

Purtroppo non è sempre possibile trovare un algoritmo che risolve un problema dato. Di conseguenza i problemi si dividono in decidibili e indecidibili a seconda che possano essere risolti oppure no.

- Un problema è detto **decidibile** se esiste un algoritmo che produce la corrispondente soluzione in tempo finito per ogni istanza dei dati di ingresso del problema.
- Un problema è detto **indecidibile** se non esiste nessun algoritmo che produce la corrispondente soluzione in tempo finito per ogni istanza dei dati di ingresso del problema. In tal caso, sarà possibile calcolare la soluzione in tempo finito solo per alcune delle istanze dei dati di ingresso del problema, mentre per tutte le altre istanze non μe possibile decidere quali siano le corrispondenti soluzioni.

7.2 Problemi trattabili e intrattabili

I problemi decidibili vengono poi classificati in base alla loro trattabilità, cioè alla possibilità di risolverli in maniera efficiente. Al fine di confrontare i problemi decidibili in modo equo rispetto alla loro trattabilità, conviene portarli tutti nella loro forma più semplice di problemi di decisione.

Un problema di decisione decidibile è intrinsecamente intrattabile se non è risolvibile in tempo polinomiale nemmeno da un algoritmo non deterministico.

Per i problemi di decisione decidibili che non sono intrinsecamente intrattabili si usa adottare la seguente classificazione:

- **P** è l'insieme dei problemi di decisione risolvibili in tempo polinomiale da un algoritmo deterministico.
- **NP** è l'insieme dei problemi di decisione risolvibili in tempo polinomiale da un algoritmo non deterministico. Equivalentemente, esso può essere definito come l'insieme dei problemi di decisione tali che la correttezza di una soluzione corrispondente ad un'istanza dei dati di ingresso può essere verificata in tempo polinomiale da un algoritmo deterministico.

8. UN ESEMPIO: TROVA LA MONETA FALSA

Immaginiamo di avere n monete tutte identiche d'aspetto; una delle monete è falsa e pesa leggermente più delle altre. Abbiamo a disposizione una bilancia a due piatti.

Il nostro obiettivo è individuare la moneta falsa facendo poche pesate.

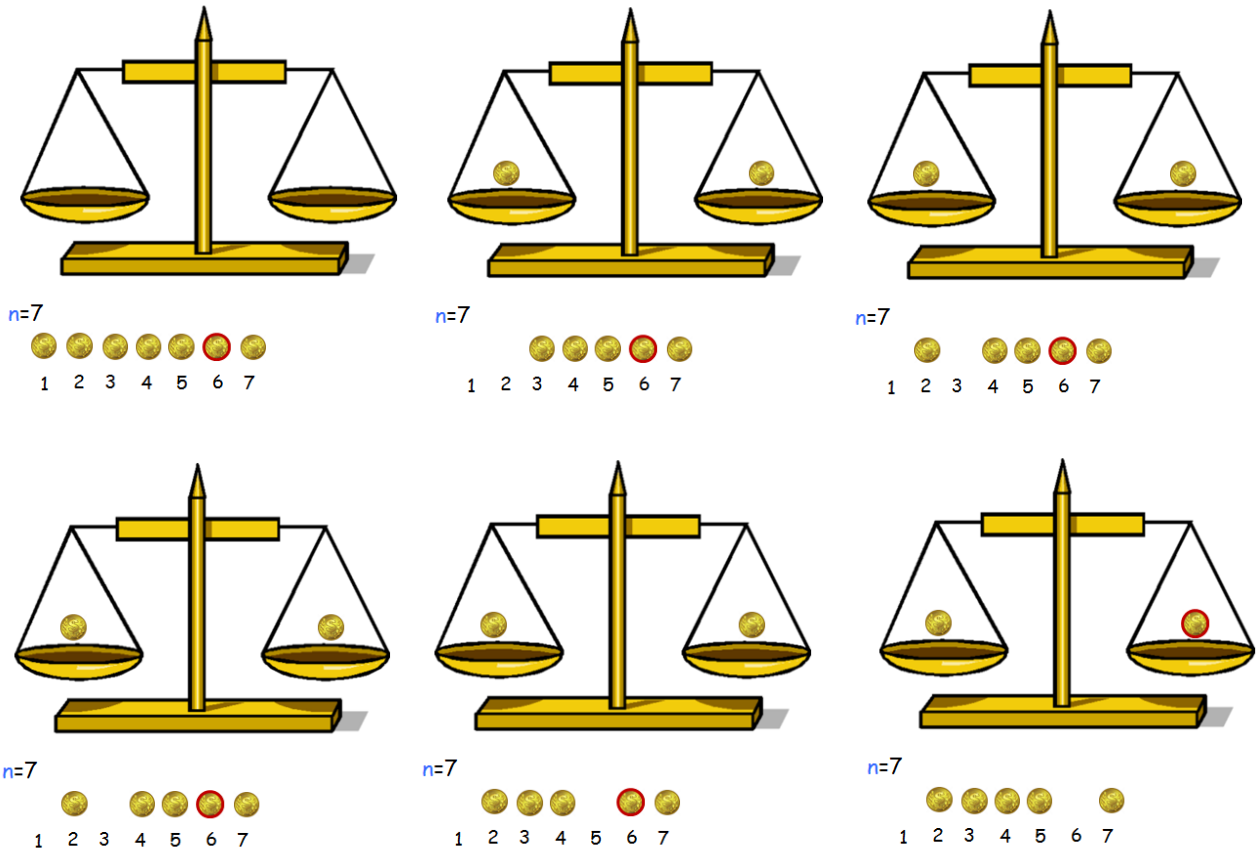
Riformuliamo il problema usando i concetti fondamentali degli algoritmi.

Problema: *individuare una moneta falsa fra n monete.*

- **Istanza**: n specifiche monete; quella falsa è una di queste; può essere la "prima", la "seconda", ecc.;
- **dimensione dell'istanza**: il valore n;
- **modello di calcolo**: bilancia a due piatti. Specifica quello che si può fare;
- **algoritmo**: strategia di pesatura. La descrizione deve essere "comprensibile" e "compatta". Deve descrivere la sequenza di operazioni sul modello di calcolo eseguite per una generica istanza;
- **correttezza dell'algoritmo**: la strategia di pesatura deve funzionare (individuare la moneta falsa) per una generica istanza, ovvero indipendentemente da quante monete sono, e se la moneta falsa è la "prima", la "seconda", ecc.;
- **complessità temporale** (**dell'algoritmo**): numero di pesate che esegue prima di individuare la moneta falsa. Dipende dalla dimensione dell'istanza e dall'istanza stessa;
- **complessità temporale nel caso peggiore**: numero massimo di pesate che esegue su una istanza di una certa dimensione. E' una delimitazione superiore a quanto mi "costa" risolvere una generica istanza. Espressa come funzione della dimensione dell'istanza;
- **efficienza** (**dell'algoritmo**): l'algoritmo deve fare poche pesate, deve essere cioè veloce. Ma veloce rispetto a che? quando si può dire che un algoritmo è veloce?

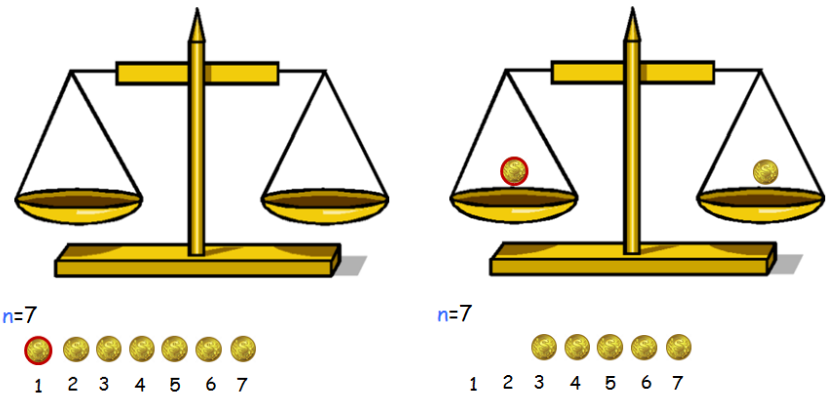
ALGORITMO 1: USO LA PRIMA MONETA E LA CONFRONTO CON LE ALTRE

Esempio 1: la moneta falsa è la 6



TROVATA !

Esempio 2: la moneta falsa è la 1



TROVATA !

Algoritmo1 ($X=\{x_1, x_2, \ldots, x_n\}$)

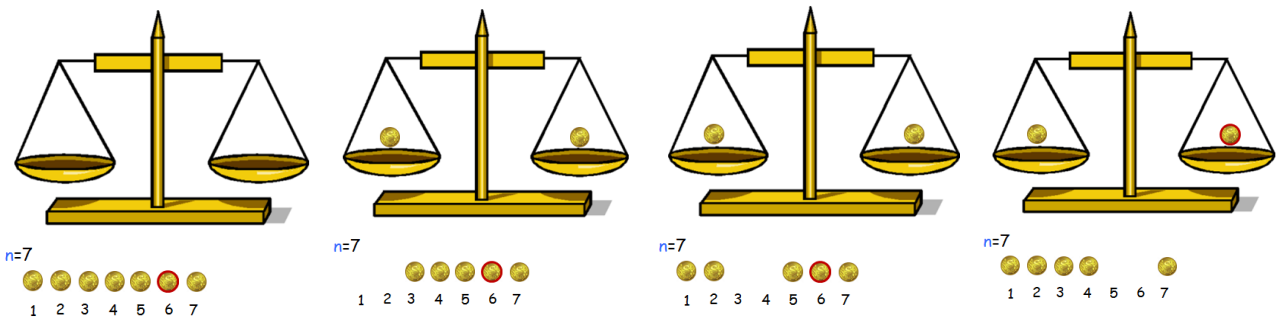
1. **for** i=2 **to** n **do**
2. **if** peso(x_1) > peso(x_i) **then return** x_1
3. **if** peso(x_1) < peso(x_i) **then return** x_i

Domande: È corretto? Sì – Numero di pesate? Dipende (al max n-2) – È efficiente? Boh!

Si può fare meglio?

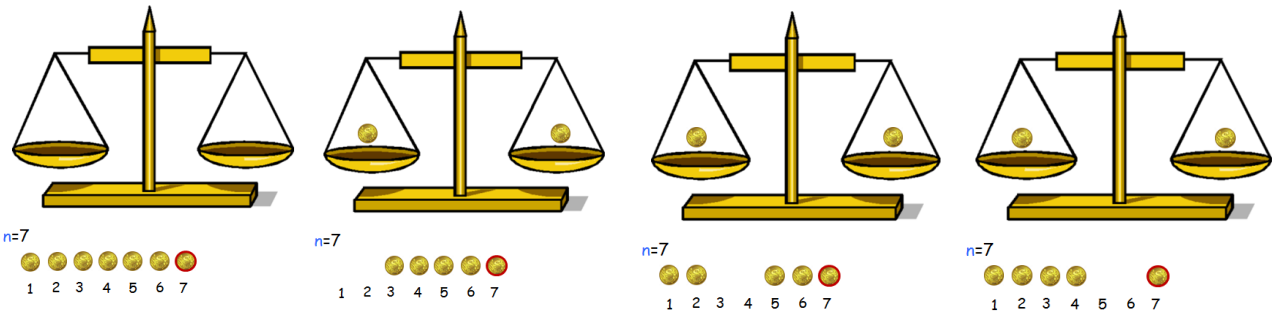
ALGORITMO 2: PESO LE MONETE A COPPIE

Esempio 1: la moneta falsa è la 6



TROVATA !

Esempio 2: la moneta falsa è la 7



TROVATA !

Algoritmo2 (X={x_1, x_2, …, x_n})

1. k=⌊n/2⌋
2. **for** i=1 **to** k **do**
3. **if** peso(x_{2i-1}) > peso(x_{2i}) **then return** x_{2i-1}
4. **if** peso(x_{2i-1}) < peso(x_{2i}) **then return** x_{2i}
5. *//ancora non ho trovato la moneta falsa; n è dispari //e manca una moneta* **return** x_n

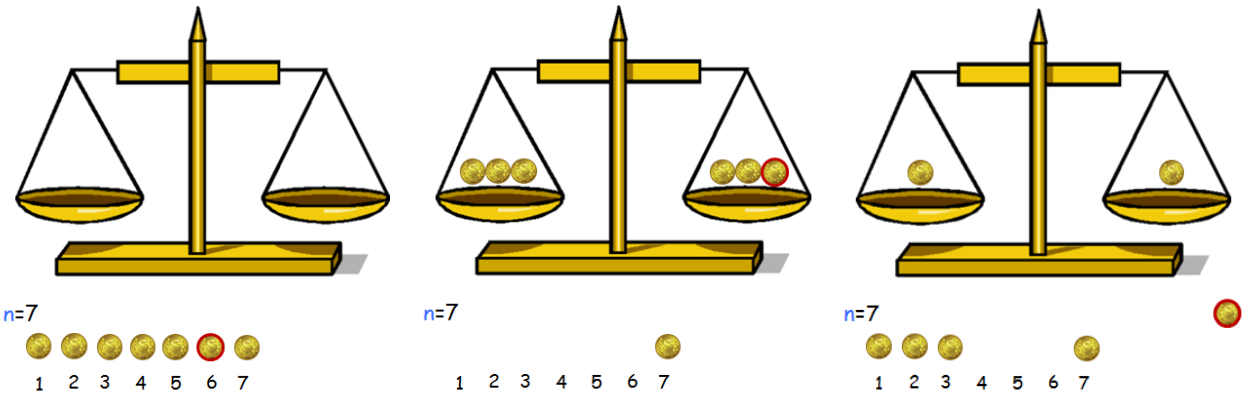
Domande: È corretto? Sì – Numero di pesate? Dipende (al max n/2) – È efficiente? Boh!

Sicuramente è meglio dell'Algoritmo1.

Si può fare meglio?

ALGORITMO 3: PESO LE MONETE DIVIDENDOLE OGNI VOLTA IN DUE GRUPPI

Esempio: la moneta falsa è la 6



TROVATA !

Algoritmo3(X)

1. **if** (|X|=1) **then** return unica moneta in X
2. dividi X in due gruppi X_1 e X_2 di (uguale) dimensione $k = \lfloor |X|/2 \rfloor$ e se |X| è dispari una ulteriore moneta y
3. **if** peso(X_1) = peso(X_2) **then return** y
4. **if** peso(X_1) > peso(X_2) **then return** Alg3(X_1)
   else return Alg3(X_2)

Domande: È corretto? Sì – Numero di pesate? Dipende (al max $\log_2$ n) – È efficiente? Boh!

Sicuramente è meglio dell'Algoritmo2.

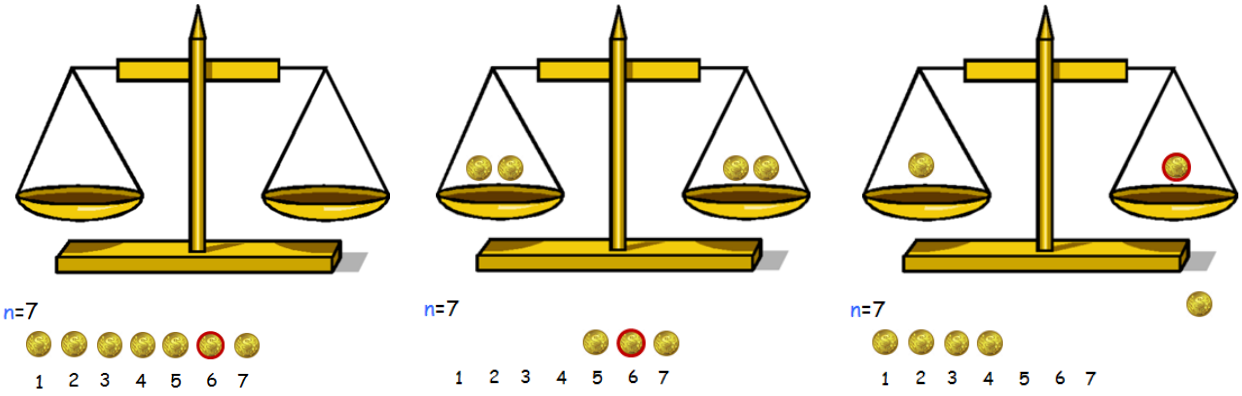
Tabella riassuntiva di confronto tra gli algoritmi

n	10	100	1.000	10.000	100.000
Algoritmo1	9 m	~ 1h, 39 m	~16 h	~7 gg	~69 gg
Algoritmo2	5 m	~ 50 min	~8 h	~3,5 gg	~35 gg
Algoritmo3	3 m	6 m	9 m	13 m	16 m

Posso fare meglio dell'Algoritmo 3?

ALGORITMO 4: DIVIDO IN TRE GRUPPI INVECE DI DUE

Esempio: la moneta falsa è la 6



TROVATA !

Algoritmo4 (X)

1. **if** (|X|=1) **then** return unica moneta in X
2. dividi X in tre gruppi X_1, X_2, X_3 di dimensione bilanciata
   siano X_1 e X_2 i gruppi che hanno la stessa dimensione (ci sono sempre)
3. **if** peso(X_1) = peso(X_2) **then return** Alg4(X_3)
4. **if** peso(X_1) > peso(X_2) **then return** Alg4(X_1)
   else return Alg4(X_2)

Domande: È corretto? Sì – Numero di pesate? Dipende (al max $\log_3 n$) – È efficiente? Boh!

Sicuramente è meglio dell'Algoritmo3.

Tabella riassuntiva di confronto tra i quattro algoritmi

n	10	100	1.000	10.000	100.000
Algoritmo1	9 m	~ 1h, 39 m	~16 h	~7 gg	~69 gg
Algoritmo2	5 m	~ 50 min	~8 h	~3,5 gg	~35 gg
Algoritmo3	3 m	6 m	9 m	13 m	16 m
Algoritmo4	3 m	5 m	7 m	9 m	11 m

Esercitazioni pratiche di Tecnologie Informatiche

Modulo: Algoritmi

DALL'ALGORITMO AL PROGRAMMA

Obiettivo principale: spiegare i concetti di *Algoritmo*, *Programma*, *Pseudocodice* e *Diagrammi di flusso* e presentare esempi, di difficoltà crescenti, di algoritmo sviluppati in diagrammi di flusso e, successivamente, far provare alla classe ad eseguire lo stesso procedimento per un problemi simili e/o di livello superiore.

Gli studenti impareranno:

- a far esperienze pratiche creando algoritmi che descrivono casi reali;
- a trasformare un problema in un programma;
- a riconoscere, mediante confronto con altre soluzioni, la creazione di più "efficiente" per la soluzioni di problemi;

Contenuti

- Algoritmo
- Programma
- Pseudocodice
- Diagramma di flusso

Attività

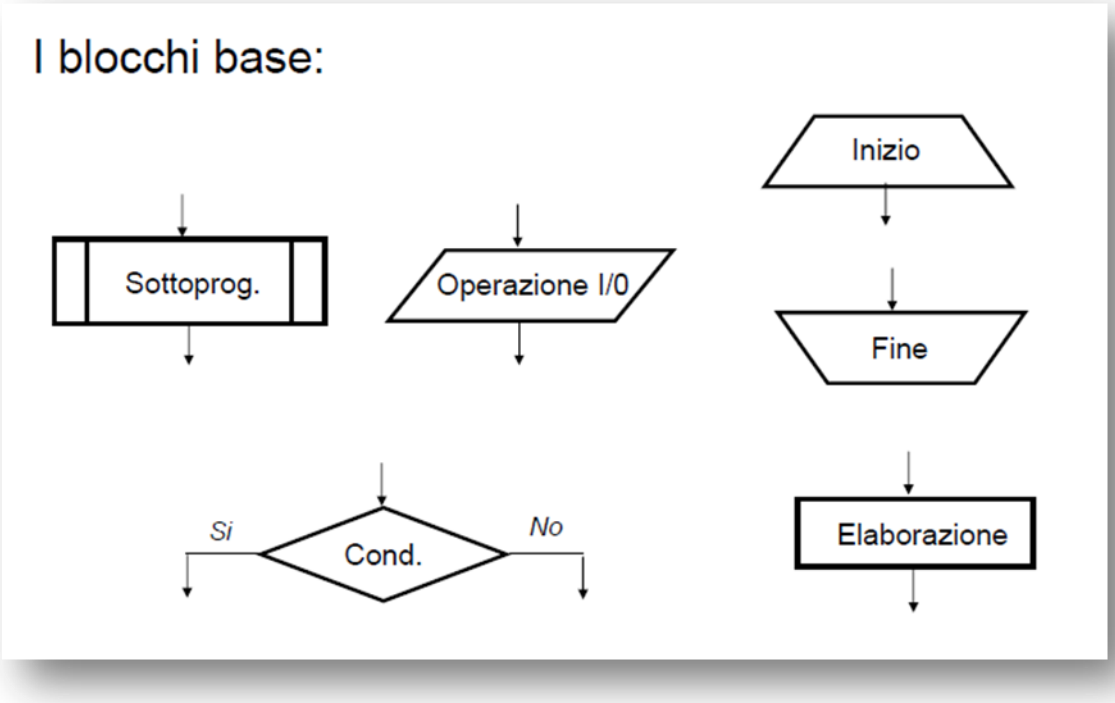
Esercitazioni singole e di gruppo mediante utilizzo di strumenti informatici.

Durata dell'esercitazione

- 1,5 - 2 ore

Introduzione

Il termine **Algoritmo** definisce una sequenza di azioni non ambigue che trasforma i dati iniziali nel risultato finale utilizzando un insieme di azioni elementari che possono essere eseguite da un opportuno esecutore. Il **Programma** specifica di un algoritmo utilizzando un linguaggio non ambiguo e direttamente comprensibile dal computer. Lo **Pseudocodice** è un linguaggio artificiale e informale, che aiuta i programmatori a sviluppare gli algoritmi. È simile all'italiano di tutti i giorni, non è eseguito sui computer ma aiuta il programmatore a "riflettere" sul programma, prima che provi a scriverlo. Inoltre, è facilmente convertibile in un corrispondente programma.
I **diagrammi di flusso** sono grafici che permettono di esprimere un algoritmo in modo preciso ed intuitivo. Si costruiscono a partire da un certo numero di 'blocchi base' che rappresentano le operazioni elementari ed i costrutti di controllo.

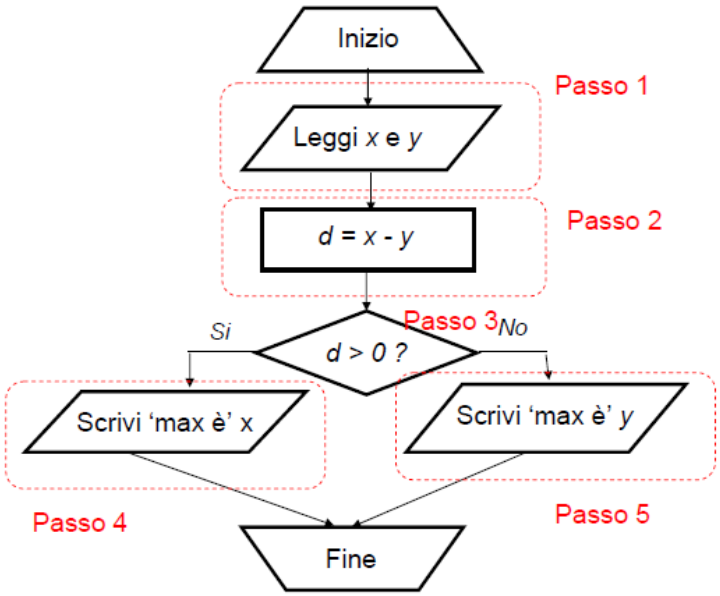


I **blocchi base** vengono collegati tramite 'frecce' che collegano un'azione alla successiva all'interno dell'algoritmo.

Realizziamo il diagramma di flusso del seguente algoritmo:
Trovare il maggiore fra 2 numeri interi x e y

Algoritmo max

1. Leggi i valori di *x* e *y* dall'esterno
2. Calcola la differenza *d* fra *x e y (d=x-y)*
3. Se d è maggiore di 0 allora esegui il passo 4 altrimenti esegui il passo 5
4. Stampa 'il massimo è …' seguito dal valore di x e termina
5. Stampa 'il massimo è …' seguito dal valore di y e termina



I diagrammi di flusso sono un primo passo verso la formalizzazione di un algoritmo in modo non ambiguo. Per ottenere una codifica interpretabile direttamente dalla macchina dobbiamo però specificare molti più dettagli: bisogna trasformare tutte le 'frasi' in variabili e modifiche su di esse.

Programma: max in C

```
main() /* calcola max */
{
int x, y, d;
scanf ("%d %d", &x, &y) ;
d = x - y ;
if (d > 0)
printf ("il max è %d", &x) ;
else
printf ("il max è %d", &y) ;
return ;
}
```

Comando: *if-else*

La forma generale dell'istruzione *if-else* è la seguente:

if (espressione)
istruzione;
else
istruzione;

dove **istruzione** può essere una singola istruzione, un blocco di istruzioni o l'istruzione nulla. La clausola else è opzionale.

- Se espressione fornisce un risultato vero, viene eseguita l'istruzione o il blocco relativo alla parte *if*;
- Se espressione fornisce un risultato falso, verrà eseguita, se esiste, l'istruzione o il blocco *else*.

Compito:

1. Realizzare un diagramma di flusso e
2. scrivere un programma che legge due numeri e stampa il maggiore

Programma1 in C:

```
#include <stdio.h>
int main ( ) {
int x, y, max;
printf ("Digita due numeri: ");
scanf ("%d%d", &x, &y);
if (x>y)
max = x;
else
max = y;
printf ("%d\n", max);
}
```

Programma2 in C:

Scrivere un programma come il precedente che non usa la variabile max

```
#include <stdio.h>
int main ( )
{
int x, y;
printf ("Digita due numeri: ");
scanf ("%d%d", &x, &y);
if (x>y)
printf ("%d\n", x);
else
printf ("%d\n", y);
}
```