# 20. Testing II

**IS 2024-2025**


**Laura Semini, Jacopo Soldani**
Corso di Laurea in Informatica
Dipartimento di Informatica, Università of Pisa

# NELLA PUNTATA PRECEDENTE

Bisogna poter **ripetere** una sessione di **test** in **condizioni immutate**

- ambiente definito (hardware, condizioni, …)
- casi di prova definiti (input predefiniti e comportamenti/output attesi)
- procedure definite

Un **caso di prova** è una tripla $\langle input, output\_atteso, ambiente \rangle$

**Black-box** vs **white-box**

- I criteri **black-box** si basano solo solo sulla specifica del metodo
- I criteri **white-box** si basano sul codice che implementa il metodo

.. ma anche **test mutazionale** e **oracolo**

# CRITERI WHITE-BOX (O "A SCATOLA APERTA")

Individuano casi di input che si basano sulla **struttura del codice**

→ Anche detti **criteri strutturali**

I criteri strutturali aiutano ad **aggiungere altri test** oltre a quelli generati con criteri funzionali

rispondendo alla domanda

«Quali altri casi devo aggiungere per far emergere malfunzionamenti
che non sono apparsi con il testing fatto con casi di prova
basati su criteri black-box?»

Nota: Per abuso di linguaggio si parla di white-box e black-box testing, ma la sola progettazione dei casi di test è white-box o black box, non il testing effettivo

# FLUSSO DI CONTROLLO

Un programma non è testato adeguatamente se alcuni **elementi non vengono esercitati** dai test

I criteri **strutturali** di progettazione di casi di test (aka **control flow testing**)

- sono definiti per classi particolari di elementi (**comandi**, **branch**, **condizioni** o **cammini**) e
- richiedono che i **test** esercitino **tutti quegli elementi** del programma

**Come possiamo farlo?**

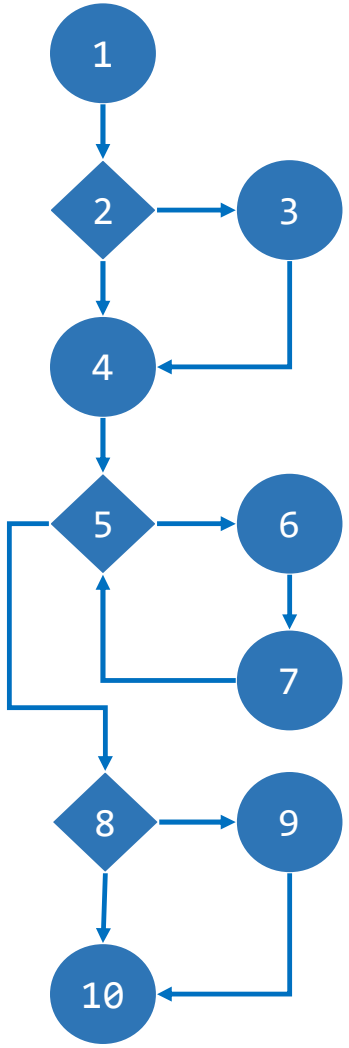
# GRAFO DI FLUSSO

Definisce la **struttura del codice**[1]

- Identifica le **parti** e come sono **collegate** tra loro
- È ottenuto **a partire dal codice**

Esempio: **`double eleva(int x, int y)`** // restituisce $x^y$

```
double eleva(int x, int y) {
  pow = y                  // 1
  if (y<0)                 // 2
    pow = -pow;            // 3
  z = 1.0;                 // 4
  while (pow!=0) {         // 5
    z = z*x;               // 6
    pow = pow-1            // 7
  }
  if (y<0)                 // 8
    z = 1.0 / z;           // 9
  return z;                // 10
}
```



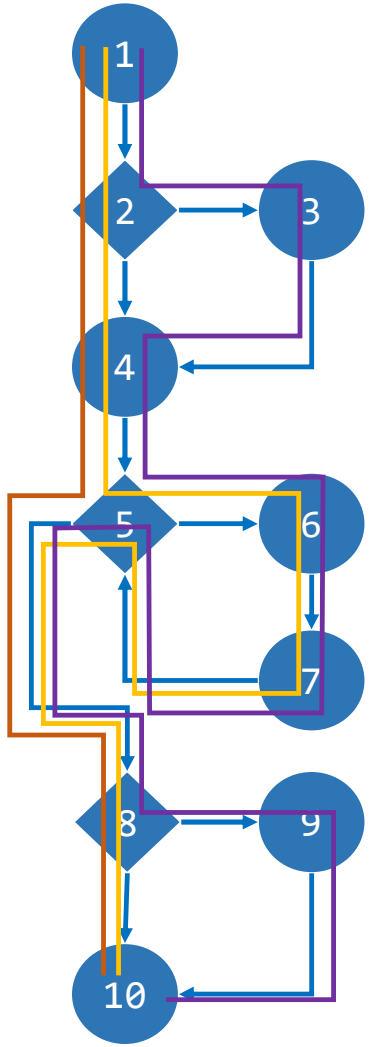

1. I diagrammi di flusso (aka. **flow chart**) sono un linguaggio di modellazione grafico per rappresentare algoritmi «in senso lato»

# CRITERIO DI COPERTURA DEI COMANDI

Si identificano valori di input che esercitino **tutti i comandi**

Esempio:

```
double eleva(int x, int y) {
  pow = y                    // 1
  if (y<0)                   // 2
    pow = -pow;              // 3
  z = 1.0;                   // 4
  while (pow!=0) {           // 5
    z = z*x;                 // 6
    pow = pow-1              // 7
  }
  if (y<0)                   // 8
    z = 1.0 / z;             // 9
  return z;                  // 10
}
```




- {(x=0,y=0)} // **non** esercita tutti i comandi
- {(x=0,y=0),(x=2,y=2)} // **non** esercita tutti i comandi
- {(x=0,y=0),(x=2,y=2),(x=2,y=-2)} // esercita **tutti** i comandi!

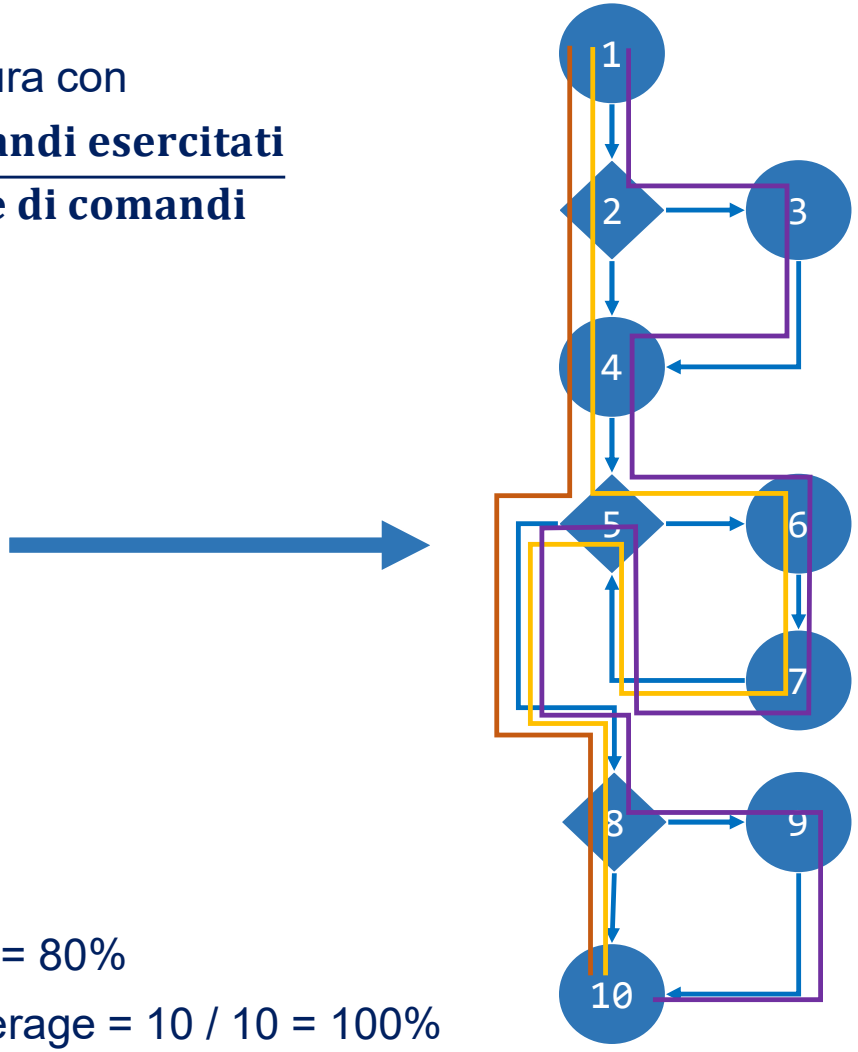
# CRITERIO DI COPERTURA DEI COMANDI (CONT.)

L'adeguatezza della copertura dei comandi si misura con

$$\frac{\textbf{numero di comandi esercitati}}{\textbf{numero totale di comandi}}$$

Esempio:

```
double eleva(int x, int y) {
  pow = y                    // 1
  if (y<0)                   // 2
    pow = -pow;              // 3
  z = 1.0;                   // 4
  while (pow!=0) {           // 5
    z = z*x;                 // 6
    pow = pow-1              // 7
  }
  if (y<0)                   // 8
    z = 1.0 / z;             // 9
  return z;                  // 10
}
```



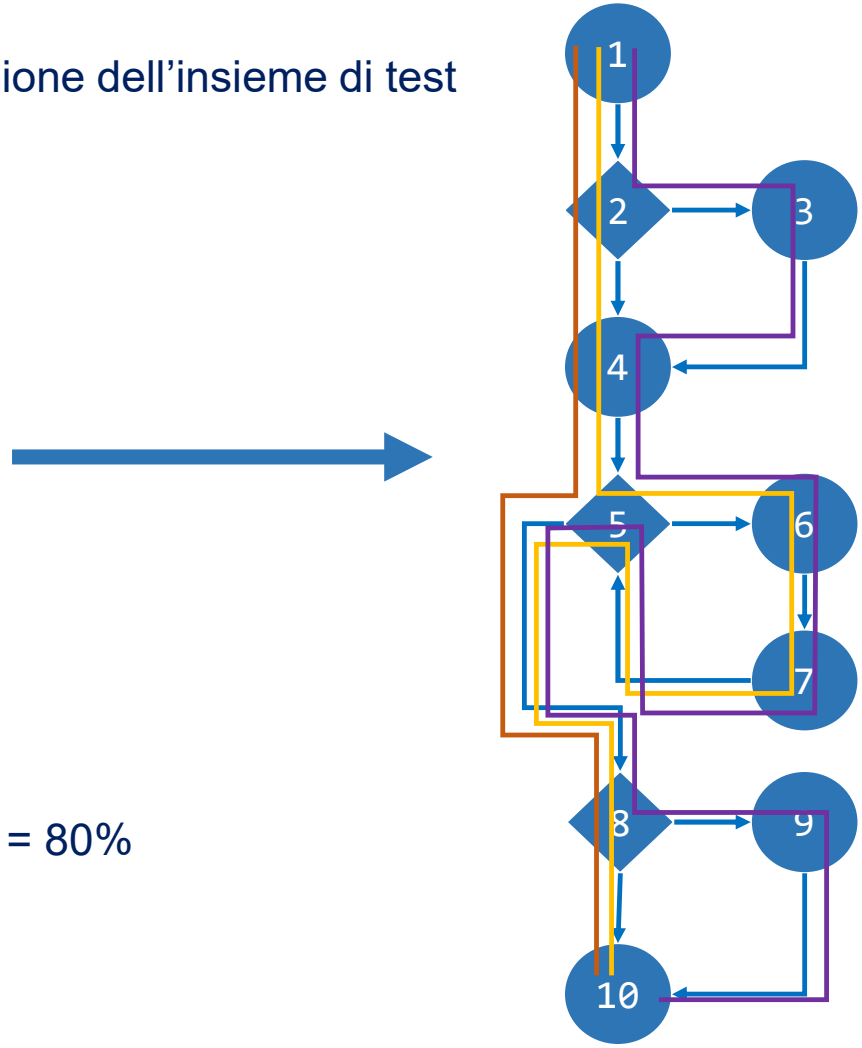

- {(x=0,y=0)} // coverage = 6 / 10 = 60%
- {(x=0,y=0),(x=2,y=2)} // coverage = 8 / 10 = 80%
- {(x=0,y=0),(x=2,y=2),(x=2,y=-2)} // coverage = 10 / 10 = 100%

# CRITERIO DI COPERTURA DEI COMANDI (CONT.)

La **coverage** non è monotona rispetto alla dimensione dell'insieme di test

Esempio:

```
double eleva(int x, int y) {
  pow = y                    // 1
  if (y<0)                   // 2
    pow = -pow;              // 3
  z = 1.0;                   // 4
  while (pow!=0) {           // 5
    z = z*x;                 // 6
    pow = pow-1              // 7
  }
  if (y<0)                   // 8
    z = 1.0 / z;             // 9
  return z;                  // 10
}
```




- {(x=0,y=0),(x=2,y=2)} // coverage = 8 / 10 = 80%
- {(x=2,y=-2)} // coverage = 10 / 10 = 100%

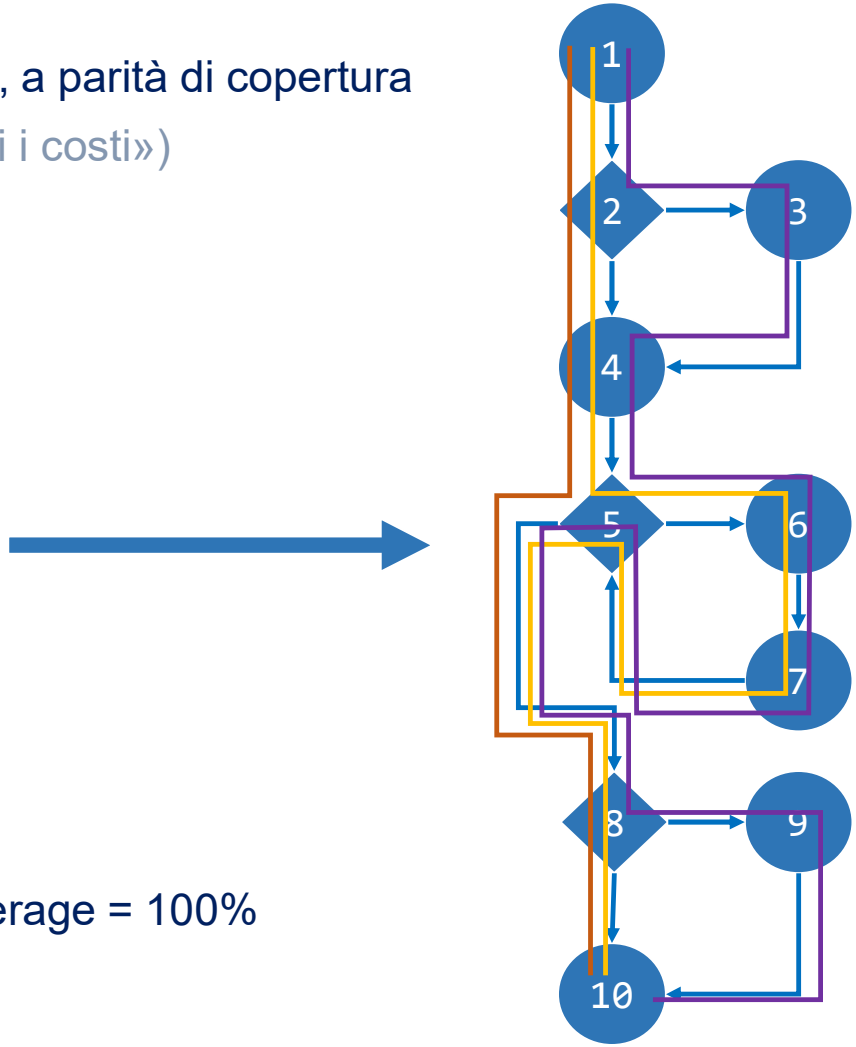
# CRITERIO DI COPERTURA DEI COMANDI (CONT.)

Conviene cercare di **minimizzare** il numero di test, a parità di copertura

(ma non sempre conviene trovare il minimo «a tutti i costi»)

Esempio:

```
double eleva(int x, int y) {
  pow = y                    // 1
  if (y<0)                   // 2
    pow = -pow;              // 3
  z = 1.0;                   // 4
  while (pow!=0) {           // 5
    z = z*x;                 // 6
    pow = pow-1              // 7
  }
  if (y<0)                   // 8
    z = 1.0 / z;             // 9
  return z;                  // 10
}
```



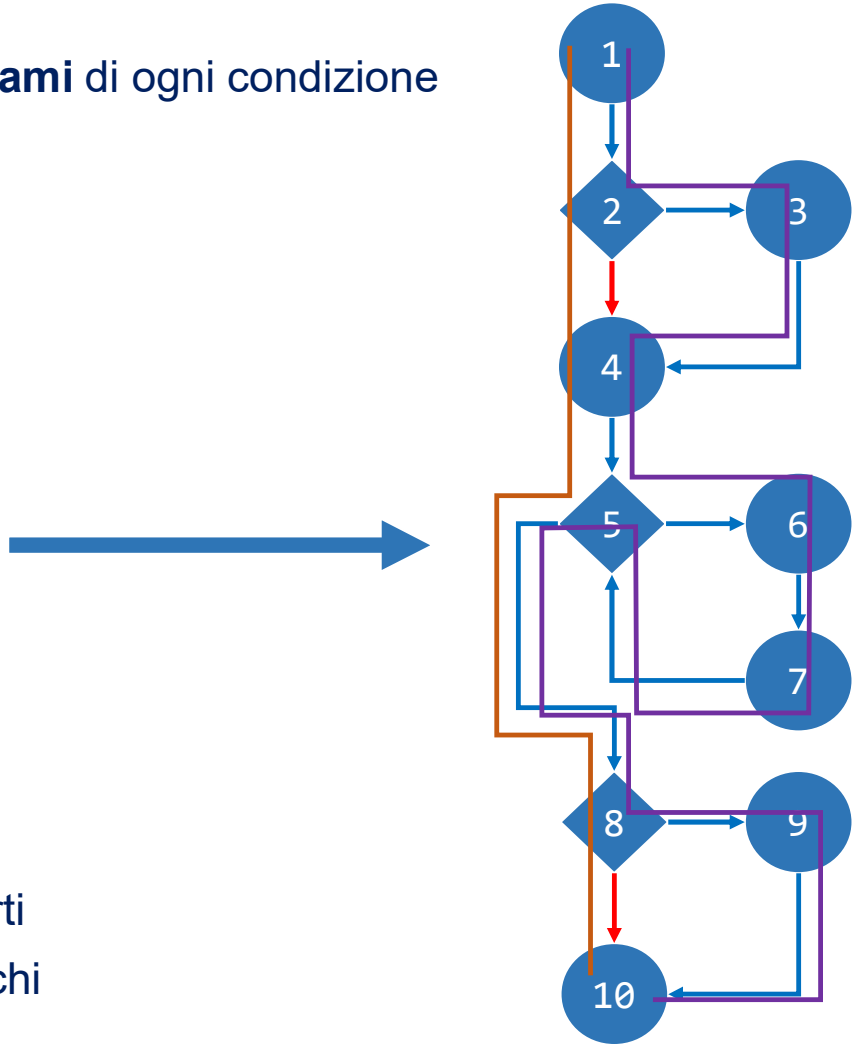

- {(x=0,y=0),(x=2,y=2),(x=2,y=-2)} // coverage = 100%
- {(x=2,y=-2)} // coverage = 100%

# CRITERIO DI COPERTURA DELLE DECISIONI

Si identificano valori di input che esercitino **tutti i rami** di ogni condizione

Esempio:

```
double eleva(int x, int y) {
  pow = y                    // 1
  if (y<0)                   // 2
    pow = -pow;              // 3
  z = 1.0;                   // 4
  while (pow!=0) {           // 5
    z = z*x;                 // 6
    pow = pow-1              // 7
  }
  if (y<0)                   // 8
    z = 1.0 / z;             // 9
  return z;                  // 10
}
```




- {(x=2,y=-2)} // gli archi rossi non sono coperti
- {(x=0,y=0),(x=2,y=-2)} // copre **tutti** gli archi

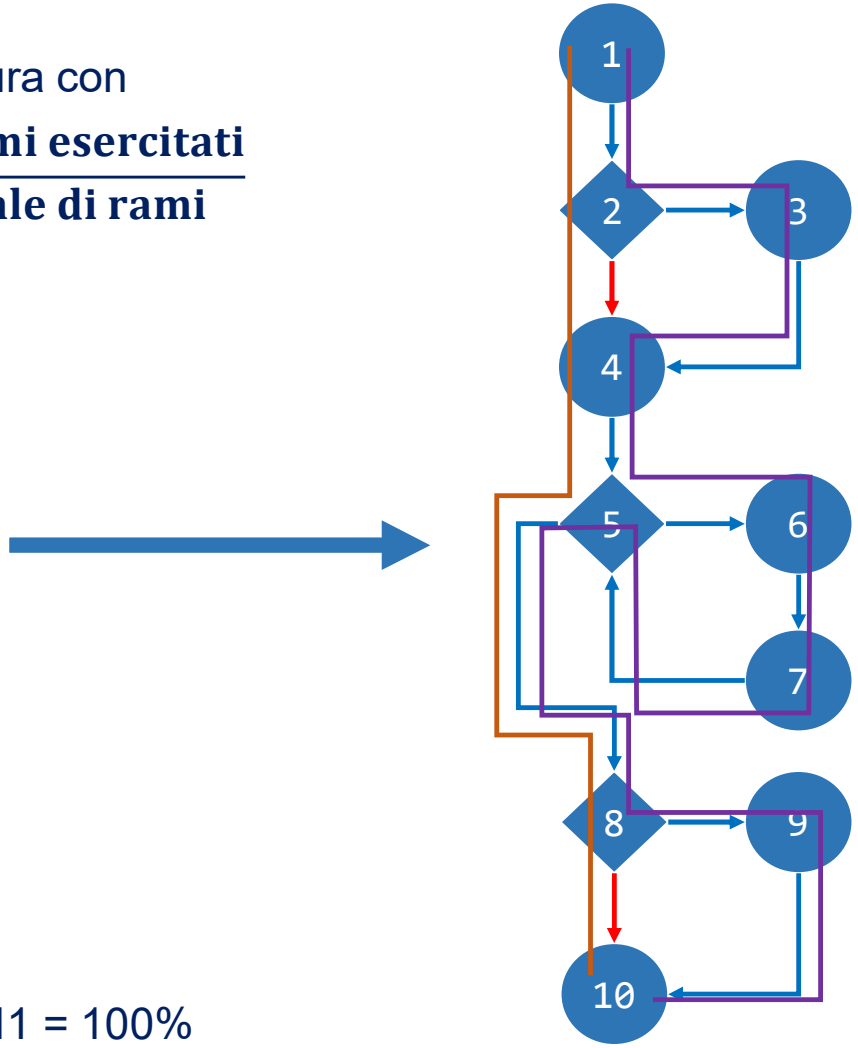
# CRITERIO DI COPERTURA DELLE DECISIONI (CONT.)

L'adeguatezza della copertura dei comandi si misura con

$$\frac{\textbf{numero di rami esercitati}}{\textbf{numero totale di rami}}$$

Esempio:

```
double eleva(int x, int y) {
  pow = y                    // 1
  if (y<0)                   // 2
    pow = -pow;              // 3
  z = 1.0;                   // 4
  while (pow!=0) {           // 5
    z = z*x;                 // 6
    pow = pow-1              // 7
  }
  if (y<0)                   // 8
    z = 1.0 / z;             // 9
  return z;                  // 10
}
```



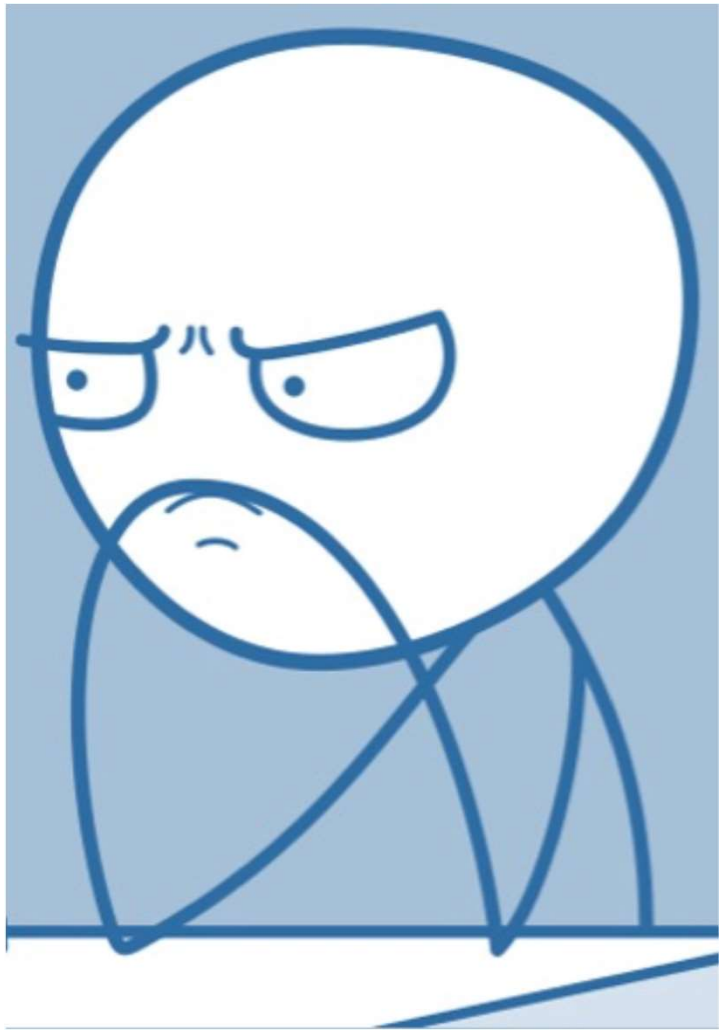

- {(x=2,y=-2)} // coverage = 9 / 11 ≈ 81.8%
- {(x=0,y=0),(x=2,y=-2)} // coverage = 11 / 11 = 100%

# CRITERIO DI COPERTURA DELLE DECISIONI (CONT.)

**FAQ: Come funziona nel caso di condizioni composte?**

```
if (x>1 || y==0) {comando1}
else {comando2}
```

- `{(x=0,y=0),(x=0, y=1)}`
  - piena copertura delle decisioni
  - **non** esercita tutti i valori di verità della prima condizione
- `{(x=2,y=2),(x=0,y=0)}`
  - **non** copre tutte le decisioni
  - esercita tutti i valori di verità delle due condizioni
- `{(x=2,y=0),(x=0,y=1)}`
  - piena copertura delle decisioni
  - esercita tutti i valori di verità delle due condizioni



# CRITERIO DI COPERTURA DELLE DECISIONI (CONT.)

Utile garantire la copertura delle **condizioni semplici**

> Un insieme di test `T` per un programma `P` copre **tutte le condizioni semplici** di `P` se, per ogni condizione semplice `CS` in `P`, `T` contiene un test in cui `CS` vale `true` e un test in cui `CS` vale `false`

La copertura delle condizioni semplici si misura come

$$\frac{\textbf{numero di valori assunti dalle condizioni semplici}}{2 * \textbf{numero di condizioni semplici}}$$

# CRITERIO DI COPERTURA DELLE DECISIONI (CONT.)

La copertura delle condizioni semplici si misura come

$$\frac{\text{numero di valori assunti dalle condizioni semplici}}{2 * \text{numero di condizioni semplici}}$$

Esempio:

```
if (x>1 || y==0) {comando1}
else {comando2}
```

- {(x=0,y=0),(x=0, y=1)} // copertura = 3 / 4 = 75%
- {(x=2,y=2),(x=0,y=0)} // copertura = 4 / 4 = 100%
- {(x=2,y=0),(x=0,y=1)} // copertura = 4 / 4 = 100%

# CRITERIO DI COPERTURA DELLE DECISIONI (CONT.)

**FAQ: Come coprire tutte le possibili combinazioni?**

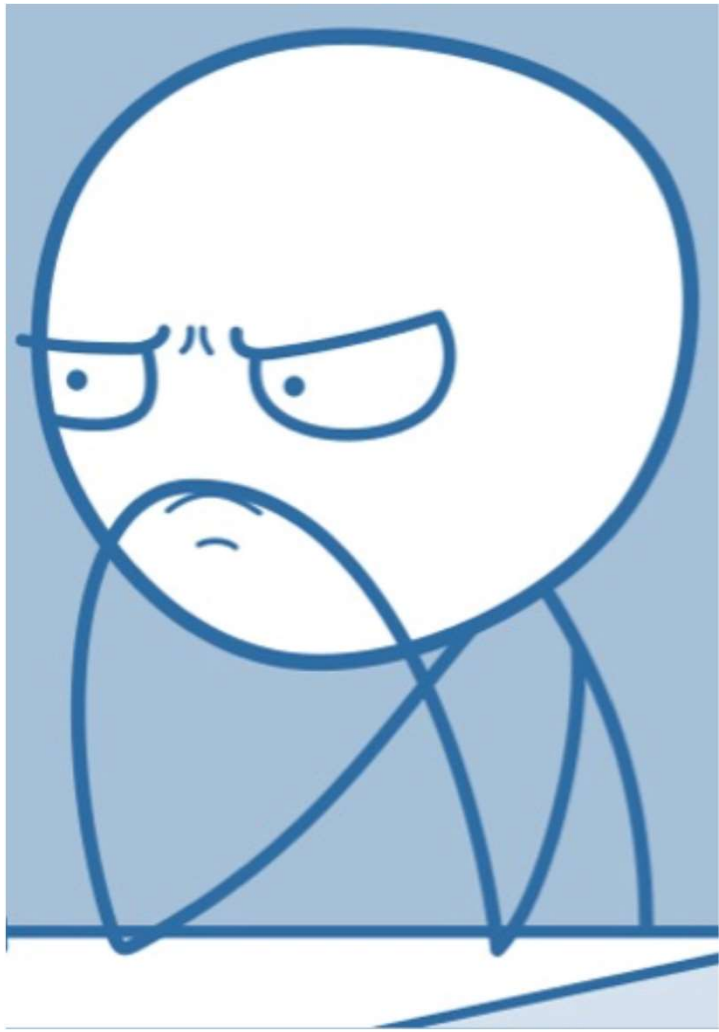
$n$ condizioni semplici ⇒ $2^n$ possibili combinazioni da testare

```
if (x>1 && y==0 && z>3) {comando1}
else {comando2}
```

$2^3$ casi possibili

Grazie alla semantica Java di «&&» ci si può ridurre a 4 casi:

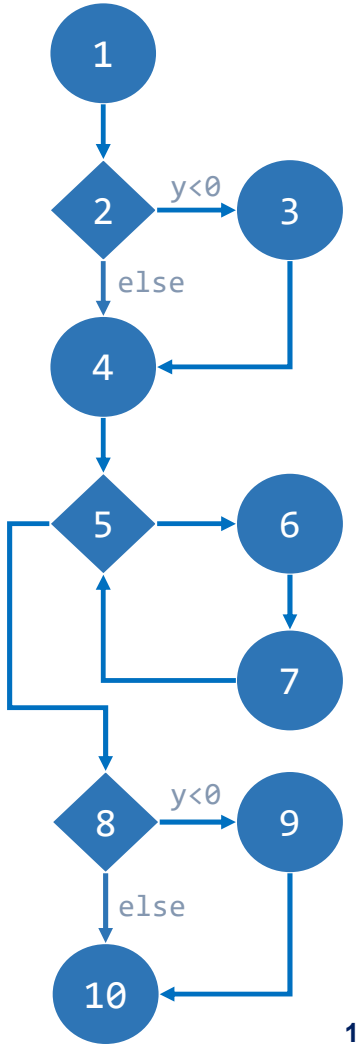
- `true && true && true`
- `true && true && false`
- `true && false && ·`
- `false && · && ·`



# CRITERIO DI COPERTURA DEI CAMMINI

Si identificano valori di input per esercitare **tutti i cammini possibili**[1]

- Cresce in modo **esponenziale** con il numero di decisioni
- In presenza di **cicli**
  - il numero di cammini possibili è potenzialmente **infinito** e
  - si impostano casi di test che esercitino il ciclo
    - zero volte
    - esattamente una volta
    - più di una volta




1. Alcuni cammini sono impossibili, p.e. 1 → 2 → 3 → 4 → 5 → 8 → 10

Fino a qui 28/11

# ALTRI CRITERI DI TEST

- **Test mutazionale** (e fault-based testing)

- **Oracolo** (e individuazione degli output attesi)

# FAULT-BASED TESTING

**Idea** di base

- **Ipotizzare** dei **difetti** potenziali nel codice sotto test
- Creare/valutare una **test suite** sulla base della sua **capacità** di **rilevare i difetti** ipotizzati

La più nota tecnica di fault based testing è il **test mutazionale**

→ si iniettano difetti modificando il codice

# TEST MUTAZIONALE

Metodo di **test strutturale** per **valutare/migliorare l'adeguatezza** delle suite di test e **stimare il numero di difetti** nei sistemi sotto test

**Precondizione**:

- aver esercitato un programma P su una batteria di test T e
- aver verificato che P sia corretto rispetto a T

Si vuole fare una **verifica più profonda** sulla correttezza di P

- si introducono dei **difetti** (aka. **mutazioni**) in P, ottenendo un programma **mutante** P'
- si **ripetono i test** in T sul programma **mutante** P'
- si verifica se i test manifestino malfunzionamenti
    - se T **non rileva** i difetti introdotti, allora la batteria di test **non era** (abbastanza) **buona**
    - se T **rileva** i difetti introdotti, abbiamo una **maggior fiducia** sella batteria di test

# TEST MUTAZIONALE (CONT.)

### FAQ: Che cos'è una mutazione?

Una **mutazione** è un **piccolo cambiamento** in un programma

→ ad esempio, si cambia "`i < 0`" in "`i <= 0`"

### FAQ: Esempi di mutazioni?

- `crp`: sostituzione (replacement) di costante per costante (ad esempio, da "`x<5`" a "`x <12`")
- `ror`: sostituzione dell'operatore relazionale (ad esempio, da "`x<=5`" a "`x<5`")
- `vie`: eliminazione dell'inizializzazione di una variabile (ad esempio, da "`int x=5`" a "`int x`")
- `lrc`: sostituzione di un operatore logico (ad esempio, da "`&`" a "`|`")
- `abs`: inserimento di un valore assoluto (ad esempio, da "`x`" a "`abs(x)`")

# ALTRI ESEMPI DI MUTAZIONI, IN C

| ID | Operator | Description | Constraint |
|---|---|---|---|
| *Operand Modifications* | | | |
| crp | constant for constant replacement | replace constant $C1$ with constant $C2$ | $C1 \neq C2$ |
| scr | scalar for constant replacement | replace constant $C$ with scalar variable $X$ | $C \neq X$ |
| acr | array for constant replacement | replace constant $C$ with array reference $A[I]$ | $C \neq A[I]$ |
| scr | struct for constant replacement | replace constant $C$ with struct field $S$ | $C \neq S$ |
| svr | scalar variable replacement | replace scalar variable $X$ with a scalar variable $Y$ | $X \neq Y$ |
| csr | constant for scalar variable replacement | replace scalar variable $X$ with a constant $C$ | $X \neq C$ |
| asr | array for scalar variable replacement | replace scalar variable $X$ with an array reference $A[I]$ | $X \neq A[I]$ |
| ssr | struct for scalar replacement | replace scalar variable $X$ with struct field $S$ | $X \neq S$ |
| vie | scalar variable initialization elimination | remove initialization of a scalar variable | |
| car | constant for array replacement | replace array reference $A[I]$ with constant $C$ | $A[I] \neq C$ |
| sar | scalar for array replacement | replace array reference $A[I]$ with scalar variable $X$ | $A[I] \neq X$ |
| cnr | comparable array replacement | replace array reference with a comparable array reference | |
| sar | struct for array reference replacement | replace array reference $A[I]$ with a struct field $S$ | $A[I] \neq S$ |
| *Expression Modifications* | | | |
| abs | absolute value insertion | replace $e$ by abs($e$) | $e < 0$ |
| aor | arithmetic operator replacement | replace arithmetic operator $\psi$ with arithmetic operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| lcr | logical connector replacement | replace logical connector $\psi$ with logical connector $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| ror | relational operator replacement | replace relational operator $\psi$ with relational operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| uoi | unary operator insertion | insert unary operator | |
| cpr | constant for predicate replacement | replace predicate with a constant value | |
| *Statement Modifications* | | | |
| sdl | statement deletion | delete a statement | |
| sca | switch case replacement | replace the label of one case with another | |
| ses | end block shift | move } one statement earlier and later | |

# TEST MUTAZIONALE (CONT.)

**FAQ: Perché far «mutare» i programmi?**

L'ipotesi è che i difetti reali siano piccole variazioni sintattiche del programma corretto

→ i **mutanti** forniscono modelli ragionevoli di programmi con difetti

**FAQ: Quando un mutante è «valido»?**

Un mutante è **valido** se è **sintatticamente corretto**; altrimenti, è da considerarsi non valido

**FAQ: Quando un mutante è «utile»?**

Un mutante è **utile** se è valido e **non è facile distinguerlo** dal programma originale

- solo un (piccolo) sottoinsieme dei casi di test lo distingue dal programma originale

**FAQ: È facile trovare mutazioni che producano mutanti validi e utili?**

No, e dipende dal linguaggio

NB: un mutante è **inutile** se è **ucciso** da **quasi tutti i casi di test**

# TEST MUTAZIONALE (CONT.)

Dato un programma P e una suite di test T

- Applichiamo delle mutazioni a P per ottenere una **sequenza $P_1$, $P_2$, ... $P_n$ di mutanti** di P (**ogni mutante** $P_i$ deriva dall'applicazione di una **singola operazione di mutazione** a P)
- Si esegue la suite di test T su ciascuno dei mutanti
- T **uccide il mutante** $P_i$ se rileva un malfunzionamento
  - ovvero se fallisce almeno in un caso di test $t \in T$
  - si dice anche che il caso di test t ha ucciso il mutante
- Il mutante $P_i$ **sopravvive** a T se non cambia l'esito dei test rispetto al programma originale P

L'**efficacia** di un **test mutazionale** si misura come

$$\frac{\textbf{numero di mutanti uccisi}}{\textbf{numero totale di mutanti}}$$

(ovvero se T uccide $k$ mutanti su $n$, l'efficacia è $k/n$)

L'efficacia di un test mutazionale (aka Mutation Score) è una qualità della test suite e non dei mutanti e quantifica la capacità della suite di test di uccidere i mutanti (e quindi identificare possibili difetti)

# TEST MUTAZIONALE (CONT.)

- Si applica in congiunzione con **altri criteri di test**
  (prevede l'esistenza di un insieme di test già realizzati)

- Può essere quasi completamente **automatizzato**

- L'**idea** di base è che
  - test che trovano **semplici difetti** allora trovano anche **difetti complessi**
  - una test suite che **uccide i mutanti** è capace anche di **trovare difetti reali**

# VALUTARE LA QUALITÀ DI UNA BATTERIA DI TEST

Esempio: una funzione `foo(x,y)` restituisce x+y se x<=y e x*y altrimenti

```
\\ originale
int foo(int x, int y){
  if(x <= y) return x+y;
  else return x*y;
}
```

```
\\ mutante (un solo difetto iniettato)
int foo(int x, int y){
  if(x < y) return x+y;
  else return x*y;
}
```

Consideriamo la batteria di test: {<(0,0),0>,<(2,3),5>,<(4,3),12>}

- Rispetta criteri funzionali (classi di equivalenza e frontiera)
- Rispetta criteri strutturali (copertura dei comandi e copertura delle decisioni)

**Problema**: il **mutante sopravvive** ⇒ la batteria è **poco adeguata** e va **riprogettata**

# VALUTARE IL NUMERO DI DIFETTI NEL SISTEMA

Vogliamo contare i pesci in un lago

- Mettiamo $n$ pesci meccanici nel lago che contiene un numero imprecisato di pesci
- Osserviamo $m$ pesci
- Vediamo che (tra gli $m$ pesci osservati) $n'$ sono pesci meccanici

Il numero $tot$ di pesci nel lago è stimabile con la proporzione

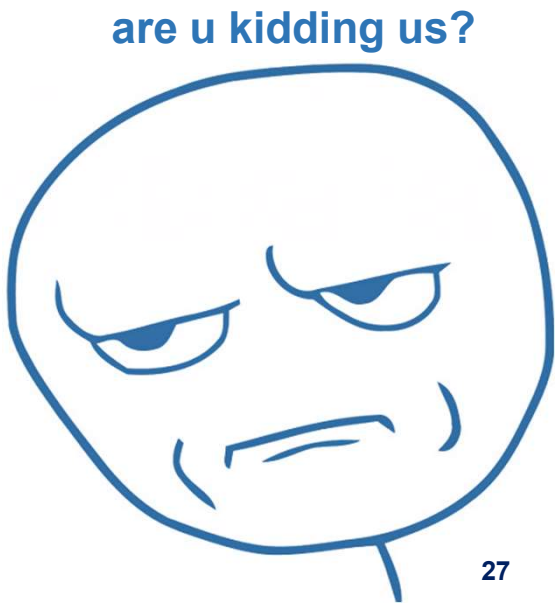
$$n : tot = n' : m$$

ovvero

$$tot = \frac{n * m}{n'}$$

E quindi?

- Pesci meccanici come **mutanti** dei pesci originali
- Possiamo fare la stessa cosa con un programma



# VALUTARE IL NUMERO DI DIFETTI (CONT.)

Esempio: una funzione `foo(x,y)` restituisce x+y se x<=y e x*y altrimenti

```
\\ originale
int foo(int x, int y){
  if(x < y) return x+y;
  else return x*y;
}
```

```
\\ mutante (un solo difetto iniettato)
int foo(int x, int y){
  if(x < y) return 3;
  else return x*y;
}
```

$\leftarrow n = 1$

Consideriamo una batteria di test: {<(0,0),0>,<(2,3),5>,<(4,3),12>}

- No difetti con il programma originale
- $m = 2$ difetti con il programma mutante (di cui $n' = 1$ iniettati)

Il numero totale di difetti presenti (incluso quello iniettato) è stimabile con

$$tot = \frac{n * m}{n'} = \frac{1 * 2}{1} = 2$$

Stimiamo di avere **un difetto** nella versione originale che **non** abbiamo ancora **trovato**

# QUANDO SOPRAVVIVE UN MUTANTE?

1) Un **mutante** può essere **equivalente** al programma originale

- Passare da "`x<=0`" a "`x<0 || x==0`" non ha cambiato l'output
- La mutazione non è un vero difetto

NB: Determinare se un mutante è equivalente al programma originale

- può essere facile o difficile, ma
- nel peggiore dei casi è indecidibile

2) La **suite di test** potrebbe essere **inadeguata**

- Il mutante poteva (e doveva) essere ucciso, ma non lo è stato
- Debolezza nella suite di test

# ESEMPI

```
\\ originale
int foo(int x, int y) {
  if(x < y) return x+y;
  else return x*y;
}
```

```
\\ non valido
int foo(int x, int y) {
  if(x < "a") return x+y;
  else return x*y;
}
```

```
\\ inutile
int foo(int x, int y) {
  if(x < y) return x*y;
  else return x*y;
}
```

```
\\ equivalente
int foo(int x, int y) {
  if(x < y) return x+y+1-1;
  else return x*y;
}
```

# ESERCIZIO

```
public void insertionSort(int[] a, int min, int max) {
  for(int i=min+1; i<max; i++) {
    temp=a[i];
    j=i-1;
    while(j>=min && a[j]>temp) {
      a[j+1]=a[j];
      j--;
    }
    a[j+1]=temp;
  }
}
```

Batteria di test (due test case):

- `int[] a = {5,9,0,2,7,3};`
  `insertionSort(a,0,6);`
- `int[] a2 = {3,1,0,2,7,3};`
  `insertionSort(a2,2,4);`

Fornire due mutazioni M1 e M2 tali che:

1. Il mutante ottenuto applicando M1 viene ucciso da TS // M1: `while(j>=min && a[j]<temp)`
2. Il mutante ottenuto applicando M2 NON è ucciso da TS // M2: `while(j>min && a[j]>temp)`

# TEST MUTAZIONALE: DISCUSSIONE

Strategia adottata per obiettivi diversi

- scoperta di malfunzionamenti ipotizzati (intervenire sul codice può essere più conveniente rispetto alla generazione di casi di test ad hoc)
- valutare l'efficacia di una batteria di test (controllando se "si accorge" delle modifiche introdotte sul programma originale)
- cercare indicazioni circa la localizzazione dei difetti (la cui esistenza è stata denunciata dai test eseguiti sul programma originale)

Uso limitato dal «costo»

- numero di mutanti che possono essere definiti
- costo della realizzazione dei singoli mutanti
- tempo e dalle risorse necessarie a (ri)eseguire i test sui mutanti e confrontare i risultati

# ALTRI CRITERI DI TEST

- **Test mutazionale** (e fault-based testing)

- **Oracolo** (e individuazione degli output attesi)

# ORACOLO

Produrre automaticamente 10000 casi di input è **inutile** se l'output atteso va calcolato «a mano»

Un **oracolo** è uno strumento o metodologia in grado di **generare i risultati attesi** di un test case

Un buon oracolo

- fornisce un **punto di riferimento** per valutare la correttezza del sistema durante il testing
- è cruciale per garantire la **qualità del software** e la corretta **individuazione** di eventuali **difetti**

# COME TROVARE IL RISULTATO ATTESO

1) Risultati **ricavati dalle specifiche**
   - specifiche formali
   - specifiche eseguibili

2) **Inversione** delle **funzioni**
   - quando la funzione **inversa** è «più **facile**» o **disponibile** fra le funzionalità
   - quando possiamo **partire dall'output** e trovare l'input (ad esempio, per un algoritmo di ordinamento, partire da un array ordinato e rimescolarlo per ottenere l'input)
   - limitazioni per difetti di approssimazione

3) **Versioni precedenti** dello stesso codice
   - se disponibili (ad esempio, per funzionalità non modificate)
   - prove di «non regressione»

# COME TROVARE IL RISULTATO ATTESO (CONT.)

4) **Versioni multiple** indipendenti
- programmi preesistenti «back-to-back»
- sviluppate ad-hoc (ad esempio, poco efficienti ma corrette)

5) **Semplificazione** dei dati di **input**
- provare le funzionalità su **dati semplici**
- **risultati noti** o **calcolabili** con altri mezzi
- **ipotesi** di **comportamento costante**

6) **Semplificazione** dei **risultati**
- accontentarsi di **risultati plausibili**
- tramite **vincoli** fra ingressi e uscite
- tramite **invarianti** sulle uscite

# RICAPITOLANDO

Diversi criteri per la costruzione di test

- **Black-box** vs **white-box** testing
- Test **mutazionale**
- **Oracolo**

# TEST DI SISTEMA

**Facility test** // il più intuitivo

- Test delle funzionalità
- Controlla che le funzionalità stabilite nei requisiti siano state realizzate correttamente

**Security test**

- Test della sicurezza
- Controlla l'efficacia dei meccanismi di sicurezza del sistema (ad esempio, provando ad accedere funzionalità o dati che non dovrebbero essere accessibili)

**Usability test**

- Test di usabilità
- Controlla la «facilità d'uso» del prodotto da parte dell'utente finale (prodotto + documentazione + livello di competenza dell'utenza + caratteristiche operative dell'ambiente d'uso del prodotto)

# TEST DI SISTEMA (CONT.)

**Performance test**

- Test delle **prestazioni** // critico, ad esempio, per i sistemi in tempo reale
- Controlla l'efficienza di un sistema rispetto ai **tempi di elaborazione** e di **risposta** (ad esempio, sottoponendo il sistema a diversi livelli di carico)

**Load test** (o volume test)

- Test di **carico**
- Controlla l'efficienza del sistema anche con il **carico di lavoro massimo** previsto dai **requisiti**
- Consente di individuare **malfunzionamenti** che non si presentano in condizioni normali (ad esempio, difetti nella gestione della memoria, buffer overflows)

NB: Performance test e load test si possono fare con le **stesse tecniche** e gli **stessi strumenti**, ma hanno **obiettivi** molto **differenti**:

- **Performance** test → valutare le **prestazioni** a **vari livelli di carico** (non limite)
- **Load** test → valutare il **comportamento** del sistema con il **carico limite**

# TEST DI SISTEMA (CONT.)

Performance test e load test si possono fare con le **stesse tecniche** e gli **stessi strumenti**, ma hanno **obiettivi** molto **differenti**:

- **Performance** test → valutare le **prestazioni** a **vari livelli di carico** (non limite)
- **Load** test → valutare il **comportamento** del sistema con il **carico limite**

considerazioni simili per

**Stress test**

- Test con **esplicito superamento** dei **limiti operativi** previsti dai requisiti (ad esempio, carichi di lavoro superiori a quelli previsti dai requisiti o condizioni operative eccezionali, come riduzione delle risorse di memoria/calcolo disponibili)
- Controlla la capacità di «**recovery**» (recupero) del sistema dopo un fallimento

# TEST DI SISTEMA (CONT.)

**Storage use test**

- Test di utilizzo della **memoria persistente**
- Controlla l'efficienza di utilizzo della memoria persistente durante il funzionamento (ad esempio, per determinare «quanto serve» per poter installare il sistema)

**Configuration test**

- Test delle **configurazioni previste**
- Controlla che il sistema funzioni nelle configurazioni previste (ad esempio, per diverse infrastrutture – in termini di sistemi operativi, dispositivi hardware installati, o requisiti funzionali)

**Compatibility test**

- Test della **compatibilità**
- Controlla la compatibilità del sistema con altri sistemi software (ad esempio, versioni precedenti dello stesso sistema, sistemi funzionalmente equivalenti che deve rimpiazzare, o altri sistemi software con cui deve interagire)


Q
A

# RIFERIMENTI

**Contenuti**

- **Capitolo 20** di "Software Engineering" (G. C. Kung, 2023)

**Approfondimenti**

- **Capitoli 12, 16 e 17** di "Software Testing and Analysis: Process, Principles and Techniques" (M. Pezzè, M. Young, 2008)