RICORSIONE

La ricorsione è una tecnica per risolvere problemi complessi riducendoli a problemi più semplici dello stesso tipo.

Per risolvere un problema ricorsivamente occorre individuare una dimensione del problema tale che:

1. per il valore più basso della dimensione la soluzione può essere espressa direttamente (cioè senza ricorrere alla ricorsione)

2. è possibile risolvere il problema per la dimensione generica supponendo di saper risolvere il problema per dimensioni inferiori.

La soluzione del problema per le dimensioni inferiori viene ottenuta attraverso le chiamate ricorsive, cioè richiamando la stessa funzione che si sta definendo

3. un metodo (o funzione) ricorsivo contiene un parametro che identifica la dimensione del problema su cui il metodo deve lavorare

Esempi di dimensioni:

- per un problema sugli interi il valore n dell'intero
- per un problema sugli interi il numero delle cifre dell'intero
- per un problema sugli array la lunghezza dell'array
- per un problema sulle liste la lunghezza della lista
- per un problema sugli alberi il numero di elementi dell'albero

Esempio: fattoriale

$n! = 1 \times 2 \times ... \times (n-1) \times n$

Per convenzione: $0! = 1$

fact(n): metodo ricorsivo per calcolare n!

dimensione del problema: n

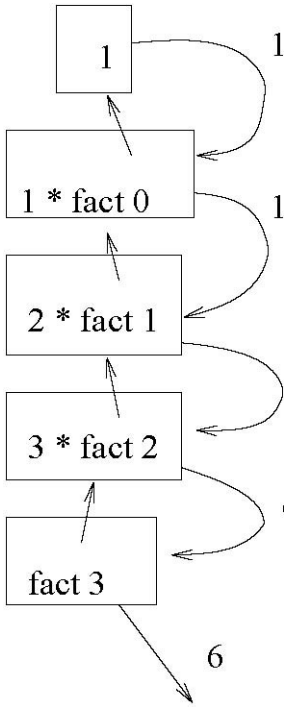
1. Caso base: fact(0) = 1
2. Vogliamo calcolare fact(n), per n> 0, supponendo di saper calcolare fact(n − 1)
3. Per ottenere fact(n), moltiplicare fact(n − 1) per n

$$\text{fact}(n) = \begin{cases} 1 \text{ se } n = 0 \\ n*\text{fact}(n-1) \text{ se } n>0 \end{cases}$$

Il fattoriale è “definito in termini di se stesso”, ma per un caso “più facile”.

```
static long fact(int n){
        if (n == 0)
                return 0;
        else
                return n*fact (n-1);
                                }
```

Processo di calcolo del valore di fact (3):



n chiamate ricorsive, numero di operazioni costante per ogni chiamata, quindi:
complessità temporale lineare rispetto ad n

n chiamate ricorsive attive contemporaneamente, quindi:
complessità spaziale lineare rispetto ad n

Che succede se fact è applicata a un numero negativo?

Fattoriale O(n)

Fibonacci O(n) - $O(2^n)$

Elevamento a potenza O(n) - O(log n)

Ricerca binaria O(log n)

==========

Quicksort O(n log n)

Torri di Hanoi $O(2^n)$

Definizioni ricorsive di funzioni

per calcolare F (n):

se n e' un caso base, riporta la soluzione per il caso n

altrimenti: risolvi i problemi più semplici

F (n_1),F (n_2), ...F (n_k) (* chiamate ricorsive *)

combina le soluzioni ottenute e riporta

combina(F (n_1),F (n_2), ...F (n_k))

```
valore funzioneRicorsiva(tipo dimensione){
    if (dimensione == valorePiuBasso)
        return valore;
    else{
        temp = funzioneRicorsiva (dimensione/qualcosa);
        return  (temp op qualcosa);
        }
```

Un processo ricorsivo termina se le chiamate ricorsive si avvicinano ai casi di base:
dopo un numero finito di chiamate ricorsive si arriva a casi base.

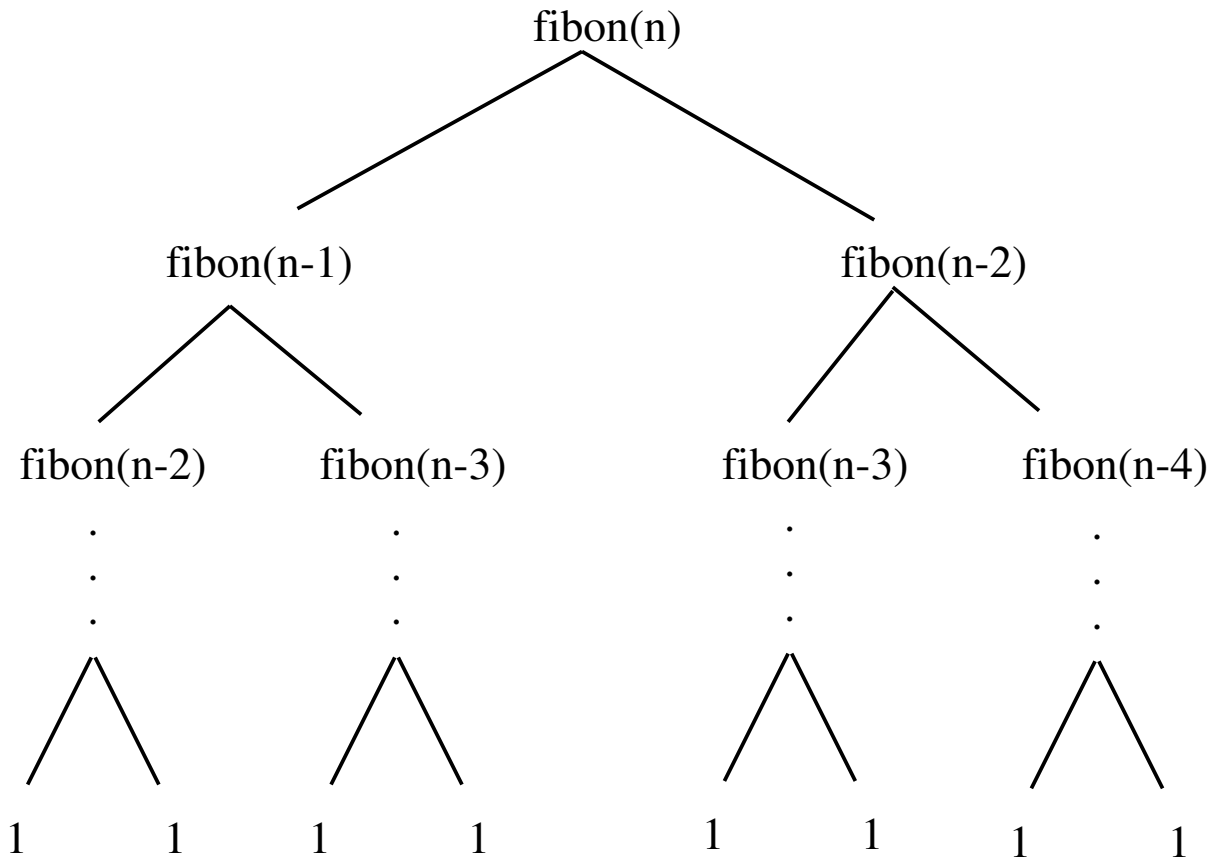
Numeri di Fibonacci

$$\text{fibon}(n) = \begin{cases} 1 \text{ se } n = 1,2 \\ \text{fibon}(n-1) + \text{fibon}(n-2) \text{ se } n>2 \end{cases}$$

```
static long fibon (int n){
      if (n <= 2)
            return 1;
      else return fibon(n-1) + fibon(n-2);
}
```

La complessità temporale della funzione è $O(2^n)$.
La complessità spaziale della funzione è O(n): ci sono al massimo n chamate attive contemporaneamente.

Infatti la funzione fibon(n) richiamerà al suo interno fibon(n-1) e fibon(n-2);
fibon(n-1) richiamerà al suo interno fibon(n-2) e fibon(n-3). Entrambe le funzioni fibon(n-2) richiameranno al loro interno fibon(n-3) e fibon(n-4) e così via:



Una **stima** dall'alto è: $1+2+4+8+\ldots+ 2^{n-1} = 2^n -1$

Osserviamo che il valore della funzione fibon(i) viene ricalcolato più volte se $i < n-1$ e questo è fonte di inefficienza.
Questo algoritmo ricorsivo è risultato semplice da scrivere, ma presenta una complessità esponenziale.

È possibile in questo caso trovare per questo problema un algoritmo iterativo di complessità lineare:

```
static long fibon2 (int n){
        fn = 1;
        fnm1 = 1;
        for (i = 3; i <= n; i++){
                temp = fn;
                fn = fn+fnm1;
                fnm1 = temp;
        }
        return fn;
}
```

Che risultato si ottiene richiamando prima fibon2(70) e poi fibon(70)?

Elevamento a potenza

```
static double power (double base, long esponente){
        double potenza = 1;
       for (long i = 0; i < esponente; i++)
              potenza *= base;
       return potenza;
}
```

Ciclo di esponente passi, per ogni iterazione un numero costante di operazioni ⇒ complessità temporale O(esponente)

```
static double powerRic (double base, long esponente){
       if (esponente==0)
              return 1;
       else
              return base* powerRic (base, esponente-1);
}
```

Numero di chiamate ricorsive pari al valore di esponente, per ogni chiamata un numero costante di operazioni ⇒ complessità temporale O(esponente)

Massimo numero di chiamate ricorsive attive contemporaneamente pari al valore di esponente ⇒ complessità spaziale O(esponente)

Cosa succede richiamando:

```
potenza(1.0000000000000001,3000000000L));
```

si osservi come raddoppiando (o triplicando) l'esponente il tempo di esecuzione raddoppia (o triplica), così come ci aspettavamo visto che la complessità è esponenziale

Cosa succede richiamando:

potenzaRic(1.000000000000001, 30000L));

si ha StackOverflowError: viene prodotto un numero lineare di chiamate ricorsive contemporanee e queste esauriscono lo spazio dello stack. Se ciò non accade, si aumenti il valore dell'esponente.

Si può migliorare l'algoritmo?

$$\text{power}(b,e) = \begin{cases} 1 & \text{se } e = 0 \\ (b^2)^{n/2} & \text{se } e \text{ pari} \\ b*(b^2)^{n/2} & \text{se } e \text{ dispari} \end{cases}$$

```
static double potenzaRic(double base, long esponente){
        if (esponente == 0 )
                return 1;
        else if (esponente % 2 == 0 )
                return potenzaRic(base*base,esponente/2);
        else return base*potenzaRic(base*base,esponente/2);
}
```

Numero di chiamate ricorsive pari al numero di volte che è possibile dividere l'esponente per 2, per ogni chiamata un numero costante di operazioni $\Rightarrow$ complessità temporale O(log(esponente))

Massimo numero di chiamate ricorsive attive contemporaneamente pari al $\log_2$(esponente) $\Rightarrow$ complessità spaziale O(log(esponente))

Cosa succede richiamando:

potenzaRic(1.0000000000000001,3000000000L));
si osservi che la computazione diventa praticabile e la differenza nel tempo di esecuzione col metodo “potenza” realizzato prima.
Come varia il tempo di esecuzione decuplicando o centuplicando l’esponente?

ESERCIZIO: si scriva una versione iterativa del metodo “potenzaRic”

Ricerca binaria

Si tratta di ricercare un elemento all'interno di un vettore ordinato.

Supponiamo di aver definito la classe Vettore che contiene al suo interno la proprietà dati (un array di interi).
Se l'array *dati* è ordinato la ricerca di un elemento può essere fatta utilizzando l'algoritmo di ricerca binaria.

Consideriamo tutto il vettore:
prendiamo l'elemento di mezzo. Se questo è l'elemento che cerchiamo la ricerca è finita. Se l'elemento che cerchiamo è più grande dell'elemento centrale del vettore proseguiamo la ricerca nella metà superiore, altrimenti proseguiamo nella metà inferiore. Ci fermiamo quando troviamo l'elemento cercato o quando la porzione di vettore da esaminare diventa vuota: in questo caso vuol dire che l'elemento cercato non è presente nel vettore.

Complessità dell'algoritmo: O(log n)

```
public class Vettore {

  int [] dati;

   public int ricercaBinRic (int x){
     return ricercaBinRic(x,0,dati.length-1);
     }
```

```
private int ricercaBinRic (int x, int inizio, int fine){
      if (inizio > fine)
         return -1;
      else
      {
         int centro = (inizio+fine)/2;
         if (dati[centro]==x)
            return centro;
         else
            if (dati[centro] > x)
               return ricercaBinRic(x,inizio,centro-1);
            else
               return ricercaBinRic(x,centro+1,fine);
      }
  }


}
```

Ora vedremo due algoritmi scrivibili facilmente in modo ricorsivo. Essi sono realizzabili in modo iterativo, ma simulando “a mano” il lavoro che la ricorsione compie automaticamente.

Ordinamento di un vettore: algoritmo quicksort

Possiamo realizzare un algoritmo per l'ordinamento di un vettore utilizzando la seguente idea

Se il vettore contiene un solo elemento allora //caso terminale
    il vettore è già ordinato
altrimenti

1. scegliamo un elemento del vettore (per esempio il primo); sia esso X; permutiamo gli elementi del vettore in modo da portare nella parte iniziale del vettore tutti gli elementi più piccoli di X e nella parte finale del vettore tutti gli elementi più grandi di X;
2. ordiniamo la parte iniziale del vettore (che ora contiene gli elementi più piccoli di X)
3. ordiniamo la parte finale del vettore (che ora contiene gli elementi più grandi di X)

Il punto 2. ed il punto 3. sono lo stesso problema dell'ordinamento, ma applicati ad un vettore più piccolo:
sono risolvibili tramite una chiamata ricorsiva.

Vediamo un esempio di applicazione dell'algoritmo:

supponiamo di voler ordinare il vettore:
[42,69,24,18,26,66,69,84,19,43,3,82,64,27,11,40,7,31,33,59]

1.
Fissiamo un elemento per esempio il primo che è 42
permutiamo gli elementi del vettore in modo da portare nella parte iniziale del vettore tutti gli elementi più piccoli di 42 e nella parte finale del vettore tutti gli elementi più grandi di 42;
[33,31,24,18,26,7,40,11,19,27,3,**42**,64,82,43,84,69,66,69,59]
2. e 3. A questo punto basta ordinare separatamente i due sottovettori:
[33,31,24,18,26,7,40,11,19,27,3]
[64,82,43,84,69,66,69,59]
Essendo lo stesso problema dell'ordinamento, ma applicato a vettori più piccoli, è risolvibile tramite due chiamate ricorsive.

1.

Metodo partition
Dato un vettore (di interi tanto per fissare le idee) di cui si considera la parte che va dalla posizione *inizio* alla posizione *fine*, fissa un elemento detto pivot (perno) permuta gli elementi in modo da portare tutti gli elementi più piccoli del perno alla sua sinistra e gli elementi più grandi del perno alla sua destra.
Restituisce la posizione finale del perno

```
int partition (int v[], int inizio, int fine) {
int pivot = v[inizio];
do
{
	while (inizio < fine && v[fine] >= pivot)
		fine--;
		if (inizio < fine)
		{
			v[inizio]=v[fine];
			while (inizio < fine && v[inizio] <= pivot)
				inizio++;

			if (inizio < fine)
				v[fine] = v[inizio];
		}

} while (inizio<fine);
v[inizio] = pivot;
return inizio;
}
```

2. e 3.

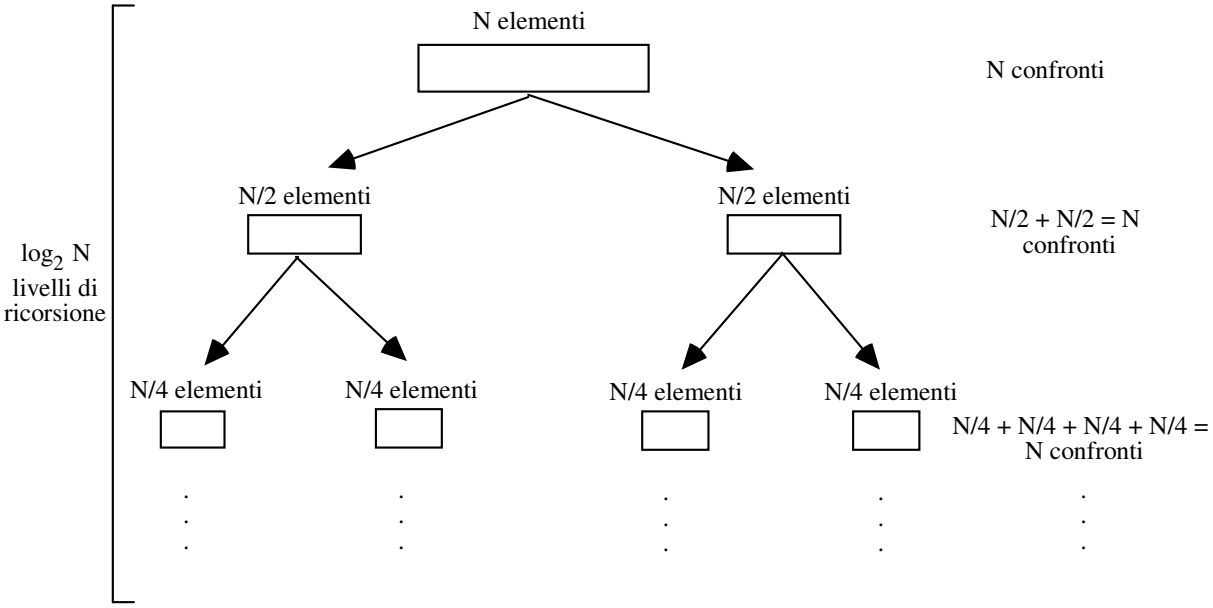
L’algoritmo di ordinamento è dunque:

```
private void quicks (int v[], int inf, int sup) {
if (inf < sup)
        {
        int mid = partition(v, inf, sup);       //O(sup-inf)
        quicks(v,inf,mid-1);
        quicks(v,mid+1,sup);
        }

}

public void quicks (int v[]) {
        quicks (v, 0, v.length-1);

}
```

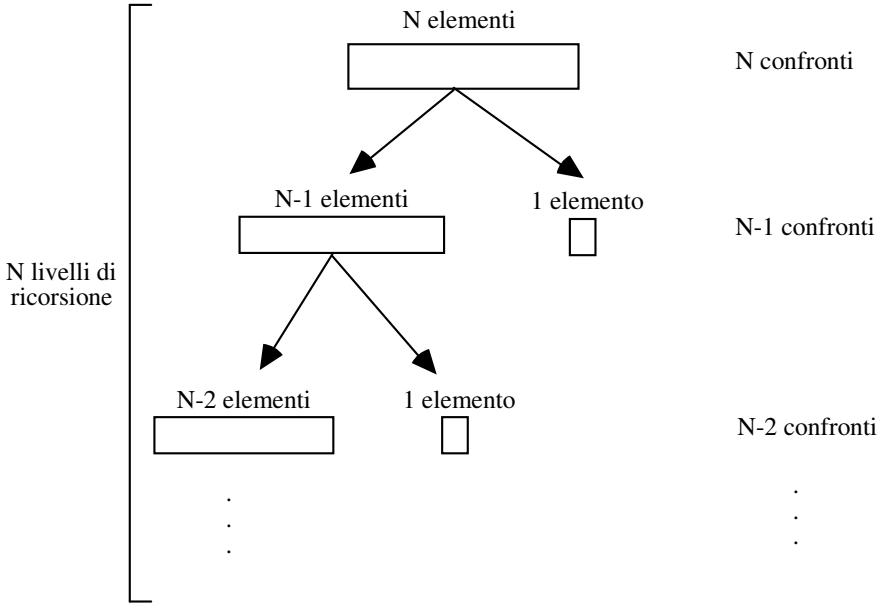
Complessità

Per valutare la complessità di quicksort considereremo il numero di confronti effettuati. All'interno della procedura partition ogni elemento viene confrontato con l'elemento X che fa da perno: vengono quindi effettuati fine-inizio+1 confronti. Per quanto riguarda il numero di chiamate ricorsive vediamo che il caso migliore è quello in cui ad ogni passo l'insieme viene diviso in due sottoinsiemi di dimensioni uguali. In questo caso sono sufficienti $\log_2 N$ livelli di ricorsione. Il caso più sfavorevole è quando X è sempre l'elemento più grande o quello più piccolo dell'insieme: in questo caso l'insieme viene diviso in un insieme di un elemento ed in un insieme di n-m elementi. In questo caso saranno necessari N

livelli di ricorsione. Nel caso più favorevole la complessità risulta dunque di ordine O(Nlog2 N):



Nel caso più sfavorevole la complessità è di ordine O(N^2):



Perché N+(N-1)+(N-2)+… +3+2+1= (N+1)*N/2

È possibile dimostrare che nel caso medio la complessità è di ordine O(Nlog N) con coefficiente minore di 2. Quicksort risulta dunque mediamente meno complesso dell'algoritmo di ordinamento di scambio.

Per quanto riguarda la complessità spaziale, il numero di chiamate attive contemporaneamente è di ordine O(N) nel caso peggiore (che, per come abbiamo scritto la partition, si presenta ad esempio se l'insieme è già ordinato), mentre è O(log N) nel caso migliore.

Se il linguaggio ottimizza la ricorsione di coda (cosa che l'attuale versione di Java non fa) la complessità spaziale diventa O(log N) se si ha cura di effettuare prima la chiamata ricorsiva sul sottovettore di ampiezza minore..
Possiamo applicare questa idea in una implementazione non ricorsiva di quicksort.

Implementazione non ricorsiva di quicksort

Definiamo una struttura a pila in cui immagazzinare le parti del vettore ancora da ordinare: in questo modo simuliamo il meccanismo di attivazione e de attivazione degli ambienti effettuato automaticamente dalla ricorsione:

```
public class Elemento {
   int inizio;
   int fine;
   Elemento next;

   public Elemento(int inizio, int fine, Elemento next) {
      this.inizio = inizio;
      this.fine = fine;
      this.next = next;
   }
}
```

La pila è una struttura dati gestita con politica LIFO (Last In First Out): l'elemento estratto è l'ultimo che è stato inserito. La

proprietà lunghezzamassima misura il massimo numero di elementi che la pila ha contenuto durante l'elaborazione.

```
public class Pila {
    Elemento testa = null;
    int lunghezza = 0;
    int lunghezzamassima = 0;

    void inserisci (int inizio, int fine){
        testa = new Elemento(inizio, fine, testa);
        lunghezza++;
        if (lunghezzamassima < lunghezza )
            lunghezzamassima = lunghezza;
    }

        Elemento estrai (){
            if (testa == null)
                return null;
            else
            {
                Elemento e = testa;
                testa = testa .next;
                lunghezza --;
                return e;
            }
    }

}
```

Il metodo partition è esattamente lo stesso visto prima. L'implementazione non ricorsiva di quicksort diventa:

1) inserisco nella pila l'intervallo [0, lunghezza vettore]
2) while (l'elemento *e* che estraggo dalla pila non è nullo){
eseguo la partition sull'intervallo rappresentato da *e*
inserisco nella pila i 2 sottointervalli prodotti dalla partition
(per ottimizzare la dimensione della pila inseriamo per ultimo l'intervallo più piccolo, così che sia elaborato per primo)
}

```
void quicksort(int v[]){
    int inizio, fine, mid;
    Pila p= new Pila();
    Elemento e;
    p.inserisci(0, v.length-1);

    while ((e= p.estrai()) != null ){
       inizio = e.inizio;
       fine = e.fine;
       mid = partition (v, inizio, fine);
       if ((mid-1 -inizio ) < (fine - (mid+1))){
          if (mid+1 < fine ) p.inserisci(mid+1, fine);
          if (inizio < mid-1) p.inserisci(inizio, mid-1);
       }
       else
        {
          if (inizio < mid-1) p.inserisci(inizio, mid-1);
          if (mid+1 < fine ) p.inserisci(mid+1, fine);
       }
    }
    System.out.println("lunghezza massima pila: "+p.lunghezzamassima);

  }
```

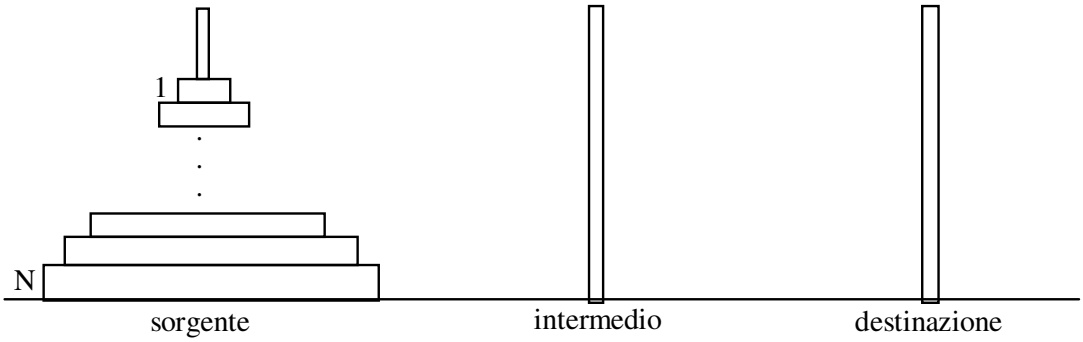
L'algoritmo Quicksort è un esempio particolare di un modo generale di impostare un algoritmo, detto *divide et impera*. Esso consiste nel dividere i dati, risolvere ricorsivamente il problema sui sottinsiemi ed ottenere la soluzione globale come combinazione delle soluzioni parziali. Tale tecnica può portare vantaggi in termini di complessità, purché la dimensioni dei sottoinsiemi dei dati non siano troppo differenti fra loro. Per esempio Quicksort presenta una complessità di $O(N\log_2 N)$ quando ogni volta l'insieme da ordinare viene diviso in due parti uguali, mentre presenta una complessità di $O(N^2)$ quando ogni volta l'insieme viene diviso in una parte formata da un solo elemento ed in una parte formata dagli altri elementi.

In generale un algoritmo di tipo "divide et impera" avrà la seguente struttura:

```
output  divimp(input S){

if (la dimensione di S è minore di k)
        risolvi direttamente il problema
else
        {
        dividi S nei sottoinsiemi S1 ... Sh;
        .
        .
        .
        T1 = divimp(S1);
        .
        .
        .
        Th = divimp(Sh);
        calcola il risultato come combinazione di T1 ... Th
        }
}
```

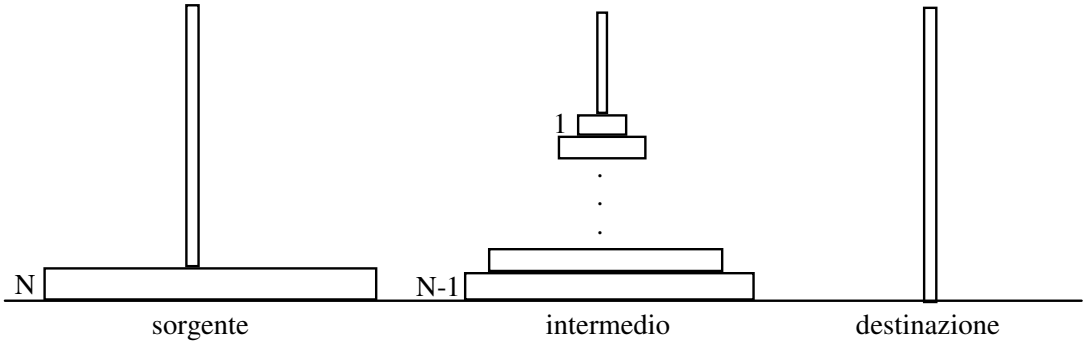
Il problema delle torri di Hanoi

Si consideri la seguente situazione: siano dati tre pioli sul primo dei quali, detto sorgente, siano infilati N dischi di diametro differente, ordinati dal più grande (che si trova in basso), al più piccolo. Gli altri due pioli non hanno nessun disco infilato e si dicono sorgente e destinazione.

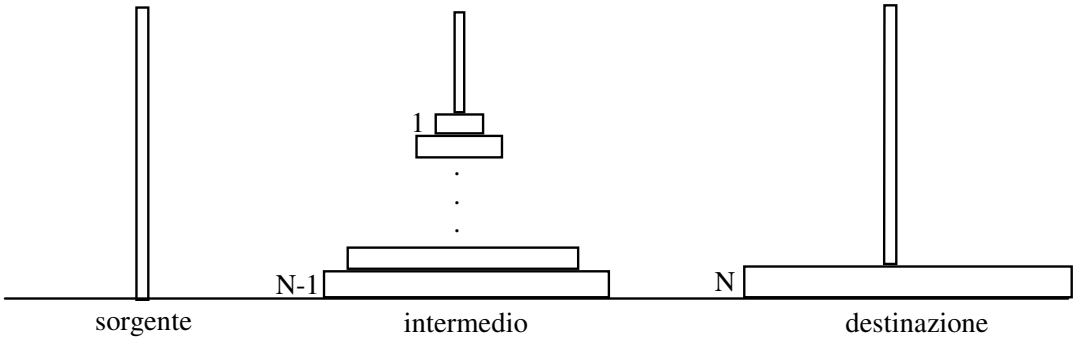


Il problema delle torri di Hanoi può così enunciato: trasferire gli N dischi dalla posizione sorgente alla posizione destinazione, utilizzando la posizione intermedia, muovendo un solo disco alla volta e non sovrapponendo mai un disco di dimensione maggiore ad un disco di dimensione minore.

Tale problema ha una naturale soluzione ricorsiva:
se N=1 basta collocare il disco nel piolo destinazione.
Se invece N > 1 basta collocare i primi N-1 dischi nel piolo intermedio, utilizzando il piolo destinazione.



A questo punto si può portare il disco N nel piolo destinazione.



Infine si portano gli N-1 dischi del piolo intermedio nel piolo destinazione, utilizzando il piolo sorgente.

L'algoritmo risulta dunque essere:

if (N==1)
    1. trasferisci il disco N da S in D;
else
    {
    2. trasferisci uno alla volta N-1 dischi da S in I utilizzando D e mantenendo l'ordinamento;
    3. trasferisci il disco N da S in D;
    4. trasferisci uno alla volta N-1 dischi da I in D utilizzando S e mantenendo l'ordinamento;
    }

1. e 3. posono essere eseguiti direttamente
2. e 4. sono equivalenti al problema iniziale avendone però diminuito la dimensione (cioè il numero di dischi), e la funzione dei tre pioli S, I e D.

```
private void hanoi (int N, String S, String I, String D)
{
if (N == 1)
        trasferisci (1,S,D);
else
        {
        hanoi (N-1, S,D,I);
        trasferisci (N,S,D);
        hanoi (N-1, I,S,D)
        }
}
```

Il metodo trasferisci può, nel nostro caso, limitarsi ad indicare quale disco viene mosso, il piolo di partenza ed il piolo di arrivo:

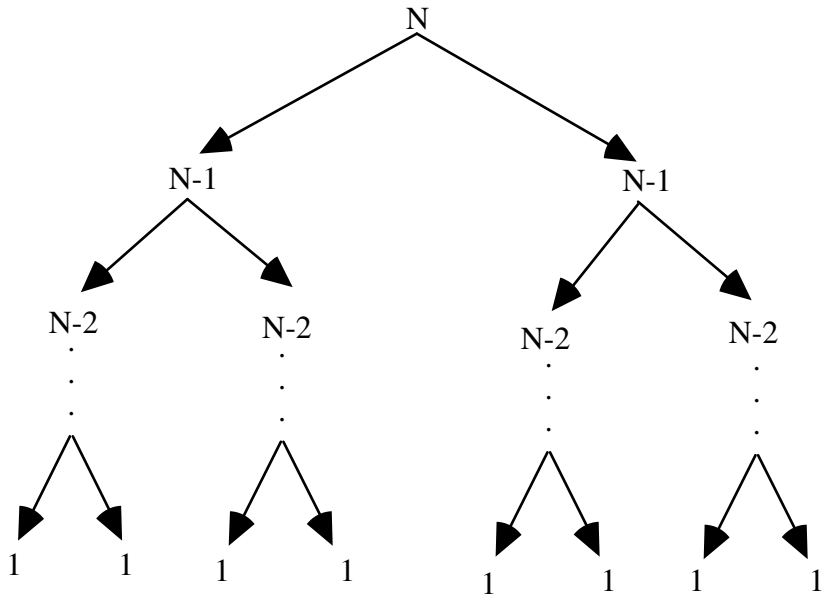
```
private void trasferisci (int N, String S, String D)
{
System.out.println("Sposta il disco "+N + " da "+S+" a "+D);
}
```

Il metodo principale:

```
public void hanoi (int N)
{
hanoi (N, "sorgente", "intermedio", "destinazione");
}
```

Complessità

Per valutare l'ordine di complessità basta contare il numero di chiamate ricorsive, osservando che per ogni chiamata viene effettuato un solo trasferimento. Ogni procedura richiama due procedure sullo stesso numero di nodi meno uno. La procedura hanoi(N) richiamerà ricorsivamente due volte hanoi(N-1). Ciascuna delle hanoi(N-1) richiamerà due volte hanoi(N-2) e così via:



In tutto ci saranno quindi state $2^0+2^1+2^2+...+2^{N-1} = 2^N - 1$ chiamate ricorsive. La complessità risulta quindi $O(2^N)$

Al massimo ci sono N chiamate ricorsive attive contemporaneamente: