

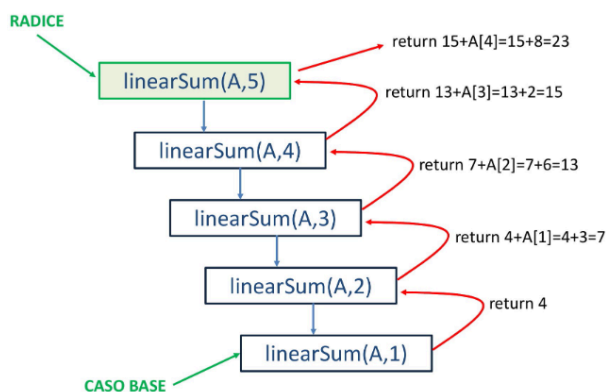
Lezione_06_DeA

2.5.6.2. Esecuzione di un algoritmo ricorsivo

All'esecuzione di un algoritmo ricorsivo su una data istanza è associato un *albero della ricorsione* (o *recursion trace*) tale che:

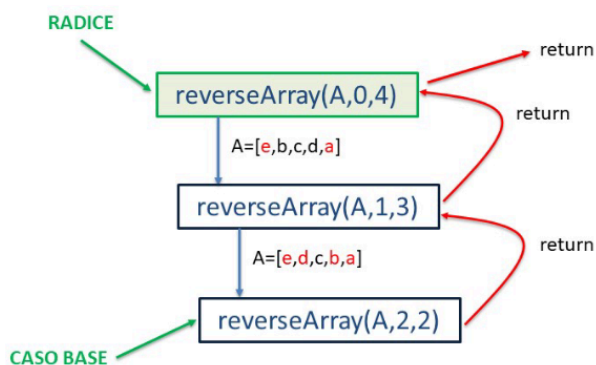
- Ogni nodo corrispondente a un'invocazione ricorsiva distinta fatta durante l'esecuzione dell'algoritmo;
- La *radice* dell'albero corrisponde alla prima invocazione, i *figli* di un nodo x sono associati alle invocazioni ricorsive fatte direttamente dall'invocazione corrispondente a x ;
- Le *foglie* dell'albero rappresentano i *casi base*.

L'albero della ricorsione per `linearSum(A,5)`, con $A = [4,3,6,2,8]$ risulta:



Avvengono un numero costante di operazioni per chiamata, quindi il numero totale sarà proporzionale al numero di chiamate. La complessità è allora $\Theta(n)$ (giustificazione a seguire).

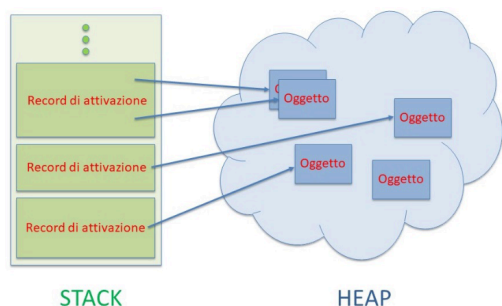
L'albero della ricorsione per `reverseArray(A,0,4)`, con $A = [a,b,c,d,e]$ sarà invece:



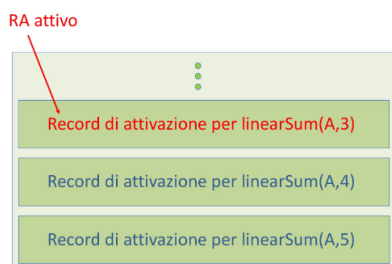
Alla fine dell'ultima invocazione si ha $A = [e,d,c,b,a]$

Nell'esecuzione di un programma (in Java) entrano di solito in gioco due spazi di memoria:

- **Stack**: spazio destinato a memorizzare variabili locali ai metodi e riferimenti a oggetti.
 - Per ogni invocazione di un metodo viene inserito un **record di attivazione** (abbreviato in RA; in inglese si usa il termine **activation record** o **activation frame**) contenente variabili e riferimenti a oggetti relativi a quell'invocazione.
 - Un RA viene eliminato quando l'invocazione del metodo corrispondente finisce l'esecuzione.
 - I RA sono inseriti/eliminati con politica LIFO (Last In First Out); in ogni istante possono essere acceduti i dati relativi all'ultimo RA inserito, ma non quelli di altri RA.
- **Heap**: spazio destinato a memorizzare gli oggetti.



Supponiamo di eseguire `linearSum(A,5)` e consideriamo l'invocazione ricorsiva `linearSu, (A,3)`. Quando viene creato il RA per tali invocazione ricorsiva, lo stato della Stack è il seguente:



Ricorda

Un algoritmo ricorsivo è un algoritmo che invoca sé stesso su istanze più piccole.

Un **algoritmo iterativo** è un algoritmo non-ricorsivo.

Osservazione

Il termine **iterativo** è usato per evidenziare che (tranne per casi banali) **un algoritmo non-ricorsivo fa uso di cicli per poter elaborare istanze di qualsiasi taglia**.

Mentre i cicli sono essenziali in un algoritmo iterativo, un

algoritmo ricorsivo può non avere cicli (come `linearSum` e `reverseArray`), ma può anche averli (come `MergeSort`).

2.5.7. Complessità di algoritmi ricorsivi

La *complessità* di un algoritmo ricorsivo A può essere *stimata tramite l'Albero della Ricorsione*.

Consideriamo l'albero associato all'esecuzione di A su un'istanza i di taglia n :

- A ogni nodo è attribuito un *costo* pari al numero di operazioni eseguite dall'invocazione corrispondente a quel nodo, *escluse quelle fatte dalle invocazioni ricorsive al suo interno*;
- Il numero totale di operazioni eseguite da A per risolvere i si ottiene sommando i costi associati a tutti i nodi.

Per ottenere un *upper bound* a $t_A(n)$ si ricava una stima per eccesso del numero totale di operazioni che valga per tutte le istanze i di taglia n , mentre per ottenere un *lower bound* a $t_A(n)$ si trova un'istanza particolare o si fa una stima inferiore che vada bene per tutte le istanze.

2.5.7.1. Esempi

2.5.7.1.1. Complessità di `linearSum(A,n)`

Algoritmo `linearSum(A,n)`

Input: array A , intero $n \geq 1$.

Output: $\sum_{i=0}^{n-1} A[i]$.

```
if (n = 1) then{
    return A[0];
}
else{
    return linearSum(A, n-1) + A[n-1];
}
```

L'albero della ricorsione associato a una generica istanza di taglia n :

- n nodi associati a `linearSum(A,j)`, $j = n-1, \dots, 1$;
- Costo associato a ciascun nodo: $\Theta(1)$ operazioni (esclude quello delle chiamate ricorsive)-

\Rightarrow Ogni istanza di taglia n richiede $\Theta(n)$ operazioni.

$\Rightarrow t_{\text{linearSum}}(n) \in \Theta(n)$ è la complessità al caso pessimo.

2.5.7.1.2. Complessità di `reverseArray(A, i, j)`

Algoritmo `reverseArray(A, i, j)`

Input: array A , indici $i, j \geq 0$.

Output: array A con gli elementi in $A[i \div j]$ ribaltati.

```
if(i < j) then{
    swap(A[i], A[j]);
    reverseArray(A, i+1, j-1);
}
return
```

L'albero della ricorsione associato a una generica istanza di taglia n ($n = j - i + 1$):

- $\lfloor \frac{n}{2} \rfloor + 1$ nodi associati;
- Costo associato a ciascun nodo: $\Theta(1)$ operazioni (esclude quello delle chiamate ricorsive).

\Rightarrow Ogni istanza di taglia n richiede $\Theta(n)$ operazioni.

$\Rightarrow t_{\text{reverseArray}}(n) \in \Theta(n)$ è la complessità al caso pessimo.

2.5.7.2. Calcolo efficiente di potenze

Dato $x \in \mathbb{R}$ e $n \geq 0$ intero, calcolare $p(x, n) = x^n$.

È fondamentale osservare che

$$p(x, n) = \begin{cases} 1 & n = 0 \\ x \cdot p(x, \frac{n-1}{2})^2 & n > 0 \text{ dispari} \\ p(x, \frac{n}{2})^2 & n > 0 \text{ pari} \end{cases}$$

Algoritmo `power(x, n)`

Input: $x \in \mathbb{R}$ e $n \geq 0$.

Output: $p(x, n)$.

```
if(n = 0) then{
    return 1;
}
if(n è dispari) then{
    y <- power(x, (n-1)/2);
    return x*y*y;
}
else{
```

```

    y <- power(x, n/2);
    return y*y;
}

```

2.5.7.2.1. Complessità di `power(x,n)`

Supponiamo $n \geq 1$ (consideriamo n come taglia dell'istanza):

- Alla i -esima chiamata ricorsiva l'algoritmo viene invocato per un esponente $n_i \leq \frac{n}{2^i}$. Di conseguenza l'albero della ricorsione avrà $O(\log n)$ nodi;
- Ogni invocazione di `power` esegue $\Theta(1)$ operazioni, esclusa l'eventuale chiamata ricorsiva. Quindi il costo associato a ciascun nodo è $\Theta(1)$.
 $\implies t_{\text{power}}(n) \in O(\log n)$.

2.5.7.2.2. Applicazione di `power` al calcolo di $F(n)$

Il seguente algoritmo efficiente sfrutta la formula $F(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ e l'algoritmo `power` visto in precedenza:

Algoritmo `powerFib(n)`

Input: intero $n \geq 0$.

Output: $F(n)$.

```

ψ <- ((1 + sqrt{5})/2);
ψ^ <- ((1 - sqrt{5})/2);
return (power(ψ,n) - power(ψ^, n))/sqrt{5};

```

Da quanto provato per `power`, otteniamo che la complessità di `powerFib` è $O(\log n)$.

2.5.8. Correttezza di algoritmi ricorsivi

Per provare la correttezza (o una qualsiasi proprietà) di un algoritmo ricorsivo A si ricorre all'induzione.

Sia n la taglia dell'istanza:

1. Si dimostra la correttezza per i casi base $n \in [n_0, n_0 + k]$;
2. Supponendo che A risolva correttamente tutte le istanze di taglia $m \in [n_0, n]$, per un qualche $n \geq n_0 + k$, si dimostra che esso risolve correttamente tutte le istanze di taglia $n+1$.

2.5.8.1. Correttezza di `linearSum(A,n)` per $n \geq 1$

Caso base ($n=1$): correttezza banale.

Passo induttivo: Fisso $n \geq 1$ arbitrario.

Ipotesi induttiva: `linearSum(A,j)` sia corretto $\forall A, \forall 1 \leq j \leq n$.

Considero `linearSum(A,n+1)` con A array di $\geq n+1$ elementi:

`linearSum(A,n+1)` restituisce $\text{linearSum}(A,n) + A[n] = \sum_{i=0}^{n-1} A[i] + A[n] = \sum_{i=0}^n A[i]$
per ipotesi induttiva.

\Rightarrow Il valore restituito è corretto.

Riepilogo sulle nozioni di base

- Nozioni di: problema computazionale, algoritmo (che risolve un problema computazionale), taglia dell'istanza, struttura dati;
- Specifica di un algoritmo tramite pseudocodice;
- Complessità al caso peggio di un algoritmo:
 - Definizione;
 - Analisi asintotica espressa tramite ordini di grandezza ($O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$);
- Tecniche di dimostrazione: esempio, controesempio, per assurdo, induzione;
- Invarianti e loro uso per provare la correttezza di cicli;
- Algoritmi ricorsivi e loro analisi:
 - Complessità: tramite l'albero della ricorsione o tramite guess e induzione;
 - Correttezza: tramite induzione.