

Ricerca Seq (Ricerca L, eletta x)

```

for each y ∈ L do
    if (y = x) then return TRUE
return FALSE

```

$T_{best}(n) = O(1) = \text{Costante}$ ← TEMPO MIGLIORE (NON DIPENDE DALLA DIM. DEL DATO IN INPUT)

$T_{worst}(n) = C \cdot n = \text{costante} \cdot n = O(n)$

$T_{medio}(n) = C \cdot \frac{n}{2} = O(n)$ → bisognerebbe conoscere la distribuzione

Ci poniamo sempre nel caso peggiore, con T_{worst} .

$T_{worst}(n) = \max \text{ istanze} I \dim(n) \quad T(I)$

$T_{best}(n) = \min \text{ istanze} I \dim(n)$

$T_{avg} = \sum_{\substack{\text{istanza} \\ \text{di dim } n}} \varphi(I) \cdot T(I)$

↳ Probabilità di istanza

ALGORITMO

1) Operazione logico-aritmetica / assegnazione $\equiv O(1)$ [costante]

2) if (+test) then Body₁ else Body₂

Tempo di esecuzione di un if = $T(\text{test}) + \max \{T(B_1) + T(B_2)\}$

↳ IP peggiore tra i tre di B₁ e B₂

3) for i = 1 to n

Body

$$\sum_{i=1}^n T_i \longrightarrow \text{tempo corpo}$$

4) while (test) do

Body

5) repeat

Body
until (Test)

$$\sum_{i=\emptyset}^m T_i + \sum_{i=\emptyset}^{n-1} T_i$$

6) funz (x)

7) seq di istc. I₁
I₂
I₃

Fattoriale (n)

```

ris = 1
for i = 1 to n do
    ris = ris * i
return ris

```

$$T(n) = C_1 + \sum_{i=1}^n C_2 = C_1 + C_2 n = \Theta(n)$$

PER CASA

Scrivere un algoritmo in pseudocodice che prenda la lista $\boxed{3 \ 5 \ 4 \ 6 \ 3 \ 2 \ 9}$ e determini il massimo locale (maggiorre di precedente e successivo) LOCAL MAX

Svolgimento

LocalMax (lista L)

```

L = 3, 5, 4, 6, 3, 2, 9
while (L != 'STOP')
    if (L > L-1 & L < L+1)
        max = L;
    else
        if (L-1 > L)
            max = L-1;
        else
            max = L+1;
    return max;

```

ALGORITMI RICORSIVI

RICORRENZA

$$T(n) = \begin{cases} 1 & n=1 \\ C + T\left(\frac{n}{2}\right) & n \geq 2 \end{cases}$$

METODO DELL'ITERAZIONE

$$T(n) = C \cdot T\left(\frac{n}{2}\right)$$

$$= C + C + T\left(\frac{n}{2^2}\right) = 2C + T\left(\frac{n}{2^2}\right) = 2C + \left[C + T\left(\frac{n}{2^3}\right)\right] = 3C + T\left(\frac{n}{2^3}\right)$$

$$C \cdot \log_{2} n + 1 = \Theta(\log n)$$

$$\hookrightarrow kC + T\left(\frac{n}{2^k}\right) \rightarrow k = \log_2 n$$

$$T(n) = \begin{cases} 9T\left(\frac{n}{3}\right) + n & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

$$= 9 \left[9T\left(\frac{n}{3^2}\right) + \frac{n}{3} \right] = 9^2 T\left(\frac{n}{3^2}\right) + (3n + n)$$

$$= 9^3 \cdot T\left(\frac{n}{3^3}\right) + (3^2 n + 3n + n) = 9^3 \cdot T\left(\frac{n}{3^3}\right) + (3^2 n + 3^1 n + 3^0 n)$$

$$T(n) = q^k T\left(\frac{n}{3^k}\right) + n \sum_{i=0}^{k-1} 3^i$$

progressione geometrica

$$= q^k T\left(\frac{n}{3^k}\right) + n \frac{3^{k-1}}{2}$$

Impongo $\frac{n}{3^k} = 1$, $k = \log_3 n$

$$\begin{aligned} T(n) &= q^{\log_3 n} + n \frac{3^{\log_3 n - 1}}{2} = \\ &= 3^{\log_3 n} + n \frac{n-1}{2} = 3^{\log_3 n} + \frac{n(n-1)}{2} = n^2 + \frac{n(n-1)}{2} = \Theta(n^2) \end{aligned}$$

9.10.13

ABBIANO VISTO

- 1) Metodo delle iterazioni
- 2) Metodo di sostituzione

ESEMPIO

$$T(n) = \begin{cases} 1 & \text{se } n=1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + n & \text{se } n>1 \end{cases}$$

$$T(n) = O(n)? \quad \exists c > 0 \text{ tale che } T(n) \leq c \cdot n$$

$$\begin{aligned} T(n) &= T\left(\lfloor \frac{n}{2} \rfloor\right) + n \\ &\leq c \left\lfloor \frac{n}{2} \right\rfloor + n \leq c \cdot \frac{n}{2} + n = \left(\frac{c}{2} + 1\right) n \stackrel{?}{\leq} c \cdot n \end{aligned}$$

$$\text{dove } \frac{c}{2} + 1 \leq c \rightarrow c \geq 2$$

DIVIDE ET IMPERA

$$T(n) = \underbrace{T_D(n)}_{\text{DIVIDE}} + \sum_{i=1}^a T(h_i) + \underbrace{T_C(n)}_{\text{COMBINE}}$$

$$T(n) = \begin{cases} a T\left(\frac{n}{b}\right) + f(n) & n > 1 \\ 1 & n = 1 \end{cases}$$

\star

$a \geq 1$
 $b > 1$
 $d = \log_b a$

TEOREMA MASTER

Sia $T(n)$ definita come in \star , con $a \geq 1$, $b > 1$, $f(n)$ asintoticamente non negativa

1) $T(n) = \Theta(n^a)$, se $f(n) = O(n^{a-\epsilon})$, per $\epsilon > 0$

2) $T(n) = \Theta(n^a \log n)$, se $f(n) = \Theta(n^a)$

3) $T(n) = \Theta(f(n))$, se $\begin{cases} 3.1 & f(n) = \Omega(n^{a+\epsilon}), \epsilon > 0 \\ 3.2 & \exists c < \epsilon \text{ c.c. } af\left(\frac{n}{b}\right) \leq cf(n) \text{ per } n \text{ suff. grande} \end{cases}$

ESEMPIO 1

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2$$

Siamo nelle condizioni di usare master?

$$\begin{aligned} a &= 3 \\ b &= 2 \\ d &= \log_2 3 \\ f(n) &= n^2 \end{aligned}$$

Si

Ora devo vedere se è $\Theta - \Theta - \Omega$

$$n^2 = \Omega(n^{\underline{\log_2 3 + \varepsilon}}) \quad \varepsilon > \emptyset \quad \rightarrow \text{verificata in 3.1}$$

$$d < \varepsilon \leq 2 - \log_2 3 \quad \text{rende vero} \quad \heartsuit$$

$$\log_2 3 + \varepsilon \leq 2$$

$$n^2 = \Omega(n^{\underline{\log_2 3 + \varepsilon}})$$

$$\exists c < 1 \quad t.c. \quad 3\left(\frac{n}{2}\right)^2 \leq c \cdot n^2 \quad ? \quad \rightarrow \text{verificata in 3.2}$$

$$3\left(\frac{n}{2}\right)^2 \leq c \cdot n^2$$

Quindi $T(n) = \Theta(f(n))$

$$\hookrightarrow \frac{3}{4}y^2 \leq cy^2 \quad \rightarrow \frac{3}{4} \leq c < 1$$

ESEMPIO 2

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

$$\begin{aligned} a &= 4 \\ b &= 2 \\ d &= \log_2 4 = 2 \\ f(n) &= n^2 \end{aligned}$$

$$n^d = n^2 \quad \rightarrow \text{caso 2} \quad \rightarrow T(n) = \Theta(n^2 \log n)$$

ESEMPIO 3

$$T(n) = T\left(\frac{n}{2}\right) + 2^n$$

$$\begin{aligned} a &= 1 \\ b &= 2 \\ d &= \log_2 1 = \emptyset \\ f(n) &= 2^n \end{aligned}$$

devo confrontare 2^n con n^d

$$\varepsilon > \emptyset. \quad f(n) = 2^n = \Omega(n^\varepsilon)$$

$$2^n = \Omega(n) \text{ se scelgo } \varepsilon = 1$$

$$\exists c < 1 \quad t.c. \quad af\left(\frac{n}{b}\right) \leq c \cdot f(n) \quad ?$$

\hookrightarrow ho che $\Omega(n^{0+1}) = \Omega(n)$
[in def. dice $\Omega(n^{d+\varepsilon})$]

$$\frac{n}{2^2} \leq c \cdot 2^n = c \cdot 2^{\frac{n}{2}} \cdot 2^{\frac{n}{2}}$$

$$\hookrightarrow 1 \leq c \cdot 2^{\frac{n}{2}}$$

$$\hookrightarrow \frac{1}{2^{\frac{n}{2}}} \leq c < 1$$

concludo che è lo caso 3

ESEMPIO

$$T(n) = 2^n T\left(\frac{n}{2}\right) + n^2$$

$a = 2^n$
 $b = 2$
 $d = \log_2 2^n = n$

→ NO TEOREMA MASTER PERCHÉ A NON È COSTANTE!

ESEMPIO

$$T(n) = 16 T\left(\frac{n}{4}\right) + n$$

$a = 16$
 $b = 4$
 $d = \log_4 16 = 2$

$\varepsilon > \phi$ t.c. $n = O(n^{2-\varepsilon})$ se prendo $\varepsilon = 1$ ottengo: $n = O(n^1) \rightarrow n = O(n)$

primo caso, $T(n) = \Theta(n^2)$

ESEMPIO

$$T(n) = \frac{1}{2} T\left(\frac{n}{2}\right) + \frac{1}{2}$$

$a = \frac{1}{2} \quad \rightsquigarrow a < 1 \text{ quindi } \underline{\text{NO MASTER}}$
 $b = 2$
 $d = \log_2 \frac{1}{2} = -1$

ESEMPIO

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log n$$

$a = 2$
 $b = 2$
 $d = \log_2 2 = 1$

Dimostra che non puoi usare master perché non si verifica nessuna delle opzioni.

1) $n \log n = O(n^{1-\varepsilon}) \quad \varepsilon > \phi ?$

$c > \phi$ per n sufficientemente grande $n \log n \leq c \cdot n^{1-\varepsilon} = \frac{n}{n^\varepsilon} \rightarrow n^\varepsilon \log n \leq c$

2) $n \log n = \Theta(n) \rightarrow n \log n \in \Theta(n)$

$n \log n = O(n) ?$

$\lim_{n \rightarrow \infty} \frac{n \log n}{n} = +\infty$ Quindi non è mai $O(n)$, quindi non vale

3) $n \log n = \Omega(n^{1+\varepsilon})$ per $\varepsilon > \phi$

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^{1+\varepsilon}} = \lim_{n \rightarrow \infty} \frac{n \log n}{n^{1+\varepsilon}} = \infty$$

↓ prevalse

Quindi $n \log n = o(n^{1+\varepsilon})$. o -piccolo e Ω -grande sono disgiunti. Quindi non vale.

ESERCIZI PER CASA

① $T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$

② $T(n) = 3T\left(\frac{n}{3}\right) + \sqrt{n}$

③ $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

RISOLVO ESERCIZI PER CASA

$$\textcircled{1} \quad T(n) = 2T\left(\frac{n}{4}\right) + n^{0.5}$$

$$a = 2 \quad d = \log_4 2 = \frac{1}{2}$$

$$b = 4 \quad f(n) = n^{0.5}$$

$$\textcircled{2} \quad T(n) = 3T\left(\frac{n}{3}\right) + \sqrt{n}$$

$$a = 3 \quad d = \log_3 3 = 1$$

$$b = 3 \quad f(n) = \sqrt{n}$$

$$\textcircled{3} \quad T(n) = 64T\left(\frac{n}{8}\right) + n^2 \log n$$

$$a = 64 \quad d = \log_8 64 = 2$$

$$b = 8 \quad f(n) = n^2 \log n$$

C_D non posso applicare il teorema master.

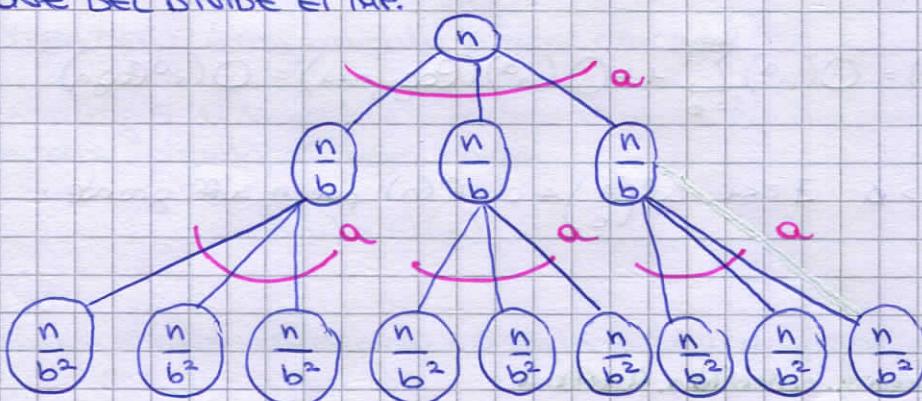
SPIEGAZIONE DEL DIVIDE ET IMP.

$$i = \emptyset$$

$$i = 1$$

$$i = 2$$

$$10.10.13$$



1) a livello i la dimensione dei sottoproblemi è $\frac{n}{b^i}$, $i = \emptyset, 1, \dots$

2) il contributo di un nodo a livello i sul tempo di esecuzione è $f\left(\frac{n}{b^i}\right)$

3) il numero massimo di livelli dell'albero è $\log_b n$

4) a livello i ci sono a^i nodi

$$a^i = n^{\frac{\log_b n}{\log_b a}}$$

5) il numero di foglie è n^d ($d = \log_b a$)

$$V(n) = \sum_{i=0}^{\log_b n} a^i$$

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

DIMOSTRAZIONE DEL TEOREMA MASTER

$$1) f(n) = O(n^{d-\varepsilon}), \varepsilon > 0 \Rightarrow T(n) = O(n^d)$$

dimostra.

$$\begin{aligned} a_i f\left(\frac{n}{b^i}\right) &= a_i O\left(\left(\frac{n}{b^i}\right)^{d-\varepsilon}\right) \\ &= O\left(a_i \left(\frac{n}{b^i}\right)^{d-\varepsilon}\right) \\ &= O\left(n^{d-\varepsilon} \cdot \frac{a_i}{(b^i)^{\varepsilon} (b^i)^{-\varepsilon}}\right) = O\left(n^{d-\varepsilon} \cdot \frac{a_i \cdot (b^\varepsilon)^i}{a_i}\right) = O\left(n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) \end{aligned}$$

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} O\left(n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) = O\left(\sum_{i=0}^{\log_b n} n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) = O\left(n^{(d-\varepsilon)} \sum_{i=0}^{\log_b n} (b^\varepsilon)^i\right) = \\ &= O\left(n^{d-\varepsilon} \cdot \frac{(b^\varepsilon)^{\log_b n + 1 - 1}}{b^{\varepsilon-1}}\right) = O\left(n^{d-\varepsilon} \cdot \frac{(b^\varepsilon)^{\log_b n} \cdot b^{\varepsilon-1}}{b^{\varepsilon-1}}\right) \end{aligned}$$

$$T(n) \geq n^d \Rightarrow T(n) = \Omega(n^d)$$

$$2) T(n) = \Theta(n^d) \Rightarrow T(n) = \Theta(n^d \log n)$$

dimostra.

$$a_i f\left(\frac{n}{b^i}\right) = \Theta\left(a_i \left(\frac{n}{b^i}\right)^d\right) = \Theta\left(n^d \frac{a_i}{(b^i)^d}\right) = \Theta\left(n^d \frac{a_i}{(b^d)^i}\right) = \Theta(n^d)$$

$$T(n) = \sum_{i=0}^{\log_b n} \Theta(n^d) = \Theta(n^d) \sum_{i=0}^{\log_b n} 1 = \Theta(n^d)(\log_b n + 1) = \Theta(n^d \log n)$$

$$3) f(n) = \Omega(n^{d+\varepsilon}), \varepsilon > 0 \quad \exists c < 1 \quad a_i f\left(\frac{n}{b^i}\right) \leq c \quad f(n) \text{ per } n \text{ suff. grande}$$

$$\Rightarrow T(n) = \Theta(f(n))$$

dimostra. Parte a.

Tempo di esecuz. carica tabella questo

$$a_i f\left(\frac{n}{b^i}\right) \leq c \cdot f(n) < f(n)$$

$$\forall i : a_i f\left(\frac{n}{b^i}\right) \leq c f(n)$$

induzione. suppongo vero fino a(i-1)

$$a_i f\left(\frac{n}{b^i}\right) = a_i \cdot a^{i-1} \cdot f\left(\frac{n}{b^{i-1}}\right) \leq a_i \cdot c^{i-1} f\left(\frac{n}{b^{i-1}}\right) = c^{i-1} a_i f\left(\frac{n}{b^i}\right) \leq c^{i-1} \cdot c f(n) = c^i f(n)$$

$$T(n) \leq \sum_{i=0}^{\log_b n} c^i f(n) = f(n) \sum_{i=0}^{\log_b n} c^i \leq f(n) \sum_{i=0}^{\infty} c^i = f(n) \frac{1}{1-c} = T(n) = \Theta(f(n))$$

sono tutti > 0

Parte b. $T(n) = \Omega(f(n))$
 $T(n) \geq f(n)$

ESEMPIO DI COLLEZIONE DI OGGETTI: dizionario (a ciascun elemento è associata una chiave di identificazione presa in un dominio totalmente ordinato)

Operazioni sulla collezione "Dizionario":

- (1) inserimento di elemento e di chiave a esso associata
- (2) cancellazione di un elemento data una chiave
- (3) ricerca dell'elemento associato a una data chiave

Tipo di dato: modello matematico che consiste in una coppia che ha un insieme di valori e quello che può fare

Tipo di dato "Dizionario": → Punteggiata

dati: un insieme S di coppie costituite dall'elemento che voglio memorizzare e dalla sua chiave (elmt, key)

Operazioni: insert (dizionario S , dim, chiave k)

delete (dizionario S , chiave k)

search (dizionario S , chiave k) → elmt

Post: aggiunge a S una coppia (elmt, key)

Pre: k è presente in S

Post: cancella da S le coppie con chiave k

Post: Se la chiave k è presente in S , restituisce l'elemento (e, k) ad essa associato, altrimenti restituisce null.

CHE COS'È UNA STRUTTURA DATI? È una particolare organizzazione delle informazioni che permette di supportare in modo efficiente le operazioni di un tipo di dato.

↳ Punteggiata

1^a STRUTTURA DATI: Array ordinato in senso crescente

DATI: Un array S di dimensione n contenente record con due campi (info, key) e ordinato in senso crescente rispetto al campo chiave

$$S(n) = \Theta(n)$$

pre: $S[p..r]$

post: restituisce l'indice dell'elemento in S che ha chiave k se esiste, -1 altrimenti.

↳ Ricorsivo

Search_index (dizionario S , chiave k , inoltre p , int r) → int

if ($p > r$)

return -1

→ vettore vuoto, quindi non è presente

med = $(p+r)/2$

→ per trovare l'elemento centrale del vettore

if [med], Key > K

return Search_index ($S, k, p, med-1$)

else

return Search_index ($S, k, med+1, r$)

Search (dizionario S , chiave k)

i = Search_index ($S, k, 1, S.length$)

if $i == -1$ return NULL

else return $S[i].info$

COMPLESSITÀ DI SEARCH_INDEX:

$$T(n) = \begin{cases} \Theta(1) & n \neq \emptyset \\ T(\frac{n}{2}) + \Theta(1) & n > \emptyset \end{cases}$$

$$n^{\log_2 a} = n^{\log_2 1} = n^0 = 1 \quad \text{→ TEOREMA MASTER}$$

$f(n) = 1$ Stesso ordine di grandezza, caso 2 teorema master

$$\text{Complessità } \Theta(n^{\log_2 1} \log n) = \Theta(1 \cdot \log n) = \Theta(\log n)$$

ESERCIZIO 2 - OPERAZIONE 1

insert(dizionario S, elem e, key k)

reallocate(S, S.length+1) $\Theta(n)$ Ha costo n

i = 1

while i < S.length and S[i].key < k

do i = i + 1

for j = S.length downto i + 1

do S[j] = S[j - 1]

S[i].info = e

S[i].key = k

viene eseguito i volte

viene eseguito n-i volte

$\Theta(n)$ complessità lineare

ESERCIZIO 3 - OPERAZIONE 2

delete(dizionario S, chiave k)

i = Search_Index(S, k, 1, S.length) $\Theta(\log(n))$

for j = i to S.length - 1

S[j] = S[j + 1]

reallocate(S, S.length - 1) $\Theta(n)$

$n-i-1$ → al caso peggiore $\Theta(n)$

LA SEARCH È QUELLA CHE FUNZIONA MEGLIO, PERCIÒ QUESTO SARÀ UTILE QUANDO CI SONO MOLTE SEARCH.

Tecnica raddoppio - dimensionamento array dimensione h dove memorizza n elementi soddisfa la seguente invariante

$$h \leq 4n$$

Inizialmente, quando $n=0$, poniamo $h=1$. Ogni volta che n supera h , l'array viene riallocato raddoppiando la dimensione

$$h = 2h$$

Cognitivamente n scende di $h/2$, l'array viene riallocato dimenticando la dimensione.

$$h \leftarrow h/2$$

$$S(h) = \Theta(h)$$

n inserimenti costano

$$n = h \rightarrow 2h$$

tempo ammortizzato dato dal tempo totale richiesto dall'algoritmo nel caso pessimo per tutte le K-operazioni su istanze di dimensione n

Esercizio. Realizzare le tipi del dizionario utilizzando vettore ordinato + tecnica del rdimensionamento.

I implementazione, lista doppia

Atti:
 Una collezione L di n record contenenti (info, key, next, prev) dove next e prev sono puntatori al successivo e al precedente record della collezione.
 L.head punta al primo elemento della lista.
 L.head = NULL è la lista vuota

Operazioni:

insert (dizionario, elem e, chiave k)

Creare un nuovo record

p.info = e

p.key = k

p.next = L.head

if L.head ≠ NULL

L.head.prev = p

L.head = p

p.prev = NULL

delete (dizionario L, chiave k)

X = L.head

while X.key ≠ k

X = X.next

if X.next ≠ NULL

X.next.prev = X.prev

if X.prev ≠ NULL

X.prev.next = X.next

else

L.head = X.next

Rimuovi X

assegnamento = costante

} $\Theta(n)$ nel caso pessimo

```

search (dizionario L, chiave k)
    x = L.head
    while x ≠ NULL AND x.key ≠ k
        x = x.next
    if x ≠ NULL
        return x.info
    else
        return NULL

```

TECNICHE PER RAPPRESENTARE COLLEZIONI

1) Basata su strutture indirizzate (array)

Ipotesi: in un array di dimensione n gli indici possono essere compresi tra $0 \dots n-1$ (in C) oppure tra $1 \dots n$ (pseudo codice)

È possibile accedere in lettura e scrittura ad una qualsiasi cella in tempo costante

PROPRIETÀ BASILARE ARRAY

- (1) (forte) gli indici delle celle di un array sono numeri consecutivi
- (2) (debole) non è possibile aggiungere nuove celle ad un array

2) Basata su strutture collegate

PROPRIETÀ BASILARI

- (1) (forte) è sempre possibile aggiungere e togliere record da una struttura collegata
- (2) (debole) gli indirizzi dei record di una struttura collegata non sono necessariamente consecutivi (fondamentali per buone prestazioni)

ESERCIZI SUL CALCOLO DELLA COMPLESSITÀ

MyAlgorithm (int n) → int

```

int a, i, j
if (n > 1) then
    a = 0
    for i = 1 to n-1
        for j = 1 to n-i
            a = a + (i+j) * (j+n)
        end for
    end for
    for i = 1 to 16
        a = a + MyAlgorithm (n/4)
    end for
    return a
else
    return n-1
end if

```

} costo: $\Theta(n^2)$

$(n-1)(n-1) \rightarrow$ numero di volte che viene ripetuto

—> si ripete 16 volte
 $\rightarrow T(n/4) \rightarrow T(n/4)$

DETERMINARE IL COSTO COMPUTAZIONALE $T(n)$ DELL'ALGORITMO IN FUNZIONE DEL PARAMETRO $n \geq 0$

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 16T(n/4) + c \cdot n^2 & n > 1 \end{cases}$$

$\Theta(n^2)$ \rightarrow Teorema master

$$n^{\log_4 16} = n^{\log_4 16} = n^2$$

$$f(n) = c \cdot n^2 \rightarrow \text{caso 2, complessità } \Theta(n^2 \log n)$$

ESERCIZIO

```

A (int n)
S = φ
for i=1 to n
    do S = S + B(i)
return S
B (int m)
S = φ
for j=1 to n
    do S = S + j
return S
  
```

$$\begin{aligned}
 T_A(n) &= \sum_{i=1}^n c \cdot i = c \sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2) \\
 O(n) &\rightarrow S = \sum_{k=0}^n k = \underline{\underline{\Theta(n^2)}} \\
 \rightarrow B(m) &= m \text{ funzione identità} \\
 \Theta(n) &
 \end{aligned}$$

ESERCIZIO

```

Fun (A, n)
if n<3 return 4
t = Fun (A, n/2) → T(n/2)
if t > A[n]
    t = t + Fun (A, n/2) ↗ caso peggiore, va tenuto conto
for i=1 to n
    t = t + A[i] + Proc(n)
return t
  
```

$$T_{\text{fun}}(n) = T(n/2) + T(n/2) + \Theta(n\sqrt{n})$$

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} n\sqrt{n}$$

Nell'ipotesi che $\text{Proc}(m) = \Theta(\sqrt{m})$, determinare la complessità asintotica (al caso peggiore) della funzione $\text{Fun}(A, n)$ al crescere di $n \in \mathbb{N}$

24.10.13

Si vuole e' ordine di grandezza della complessità della seg. funzione al crescere di $n \in \mathbb{N}$

```

foo (int n)
if n≤5
    return 7
else
    if n≥75
        for i=1 to  $\lfloor n/2 \rfloor$ 
            k =  $\lfloor n/2 \rfloor$ 
        return k * foo(k) + foo( $n/2$ )
  
```

$$T(n) = 2T(n/2) + n/2$$

$$(\Rightarrow T(n) = T(n/2) + T(n/2) + n/2)$$

$$(\Rightarrow T(n) = 2T(n/4) + \Theta(n))$$

$$n^{\log_b a} = n^{\log_b 2} = n^{1/2} = \sqrt{n}$$

$$f(n) = n$$

Siamo nel caso 3?

$$f(n) = \Omega(n^{\log_b a + \epsilon}) \quad \text{con } \epsilon > 0$$

$$= \Omega(n^{1/2 + \epsilon}) \quad \text{fisso } \epsilon = 1/2$$

Verifico la condizione di regolarità: $a f(n/b) \leq c f(n)$ con $c < 1$

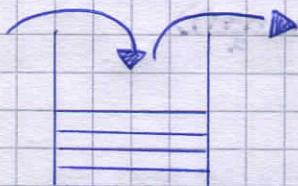
$$2 \frac{n}{\sqrt{n}} \leq c n \rightarrow c \leq \frac{1}{2}$$

pongo $c = \frac{1}{2}$ → condiz. di regolarità soddisf.

La complessità è $\Theta(n)$

Tipi di dati: Pila

Una pila è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi solo da un estremo (il top) della pila



Tipi di dato:

Dati: una sequenza S di n elementi

operazioni: initstack() \rightarrow stack

Post: restituisce una stack vuota

op: stack-empty (stack s) \rightarrow bool

Post: restituisce true se s è vuota, false altrimenti

op: push (stack s, elem e)

Post: aggiunge e come ultimo elemento di s

op: pop (stack s) \rightarrow elem

Pre: s è non vuota

Post: toglie da s l'ultimo elemento e lo restituisce

op: top (stack s) \rightarrow elem

Pre: s è non vuota

Post: restituisce l'ultimo elemento di s

PILA BASATA SUGLI ARRAY

Dati: uno stack s è un vettore di dimensione n $s[1 \dots n]$ e' manteniamo un attributo s.top che è l'indice dell'ultimo elemento inserito.

Lo stack è vuoto se s.top = \emptyset

```

init_stack()
  S = allocare (n)
  S.top =  $\emptyset$ 
  return S

stack-empty (stack s)
  return s.top ==  $\emptyset$ 

push (stack s, elem e)
  S.top = S.top + 1
  S[S.top] = e

pop (stack s)
  S.top = S.top - 1
  return S[S.top + 1]

top (stack s)
  return S[S.top]

```

QUESTE FUNZIONI
HANNO COMPLESSITÀ
COSTANTE

Tipo di dato: Coda

Una coda è una sequenza di elementi di un certo tipo in cui è possibile aggiungere elementi ad un estremo (la coda) e togliere elementi dall'altro (la testa) FIFO

Tipi di dato:

Dati: una sequenza Q di n elementi

op: initqueue () \rightarrow Queue

Post: restituisce una coda vuota

op: queue-empty (Queue q) \rightarrow bool

Post: restituisce true se q è vuota, false altrimenti

op: enqueue (Queue q, elem e)

Post: aggiunge e come ultimo elemento di q

op: dequeue (Queue q) \rightarrow elem

Pre: q non è vuota

Post: toglie da q il primo elemento e lo restituisce

op: first (Queue q) -> elem

Pre: q è non vuota

Post: restituisce il primo elemento in q

RAPPRESENTAZIONE VETTORE CIRCOLARE



Q è un vettore di dim n con 2 attributi

- (1) Q.head è l'indice della posizione da cui estrarre un elemento
- (2) Q.tail da cui inserire un elemento

Il successore dell'ultimo elemento (n) è la prima posizione

$$Q.\text{head} = Q.\text{tail} = 2$$

In ogni istante, gli elementi della coda si trovano nel segmento $Q.\text{head}, Q.\text{head}+1, \dots, Q.\text{tail}-1$

In ogni istante, si garantisce che la coda abbia capacità massima di $n-1$ elementi.

coda vuota $\Rightarrow Q.\text{head} == Q.\text{tail}$

coda piena $\Rightarrow Q.\text{head} = (Q.\text{tail} \bmod n) + 1$

enqueue()

Q = allocate(n)

Q.tail = 1

Q.head = -1

return Q

COMPLESSITÀ COSTANTE ($\Theta(1)$)

queue_empty (Queue Q)

x = Q[Q.head]

if Q.head = Q.tail

Q.head = -1

else

Q.head = Q.head + 1

return x

first (Queue Q)

return Q[Q.head]

REALIZZARE UNA CODA Q UTILIZZANDO DUE FILE p1 e p2

enqueue (Queue Q, elem e)

push (Q.p1, e)

dequeue (Queue Q)

if stack_empty (q.p2)

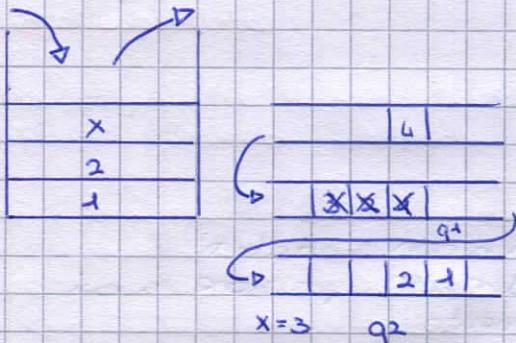
while not stack_empty . p2

x = pop (q.p2)

push (q.p2, x)

return pop (q.p2)

IDEA



N.B! Tempo ammortizzato:
diviso per il n° di operazioni.

ESERCIZIO DELL'ALTRA VOLTA

push (Stack s, Element x)

enqueue (s.q1, x)

pop (Stack s)

x = dequeue (s.q1)

while not queue-empty (s.q1) }

enqueue (s.q2, x)

x = dequeue (s.q1)

} $\Theta(n)$ nel caso peggiore

↳ risulta molto costosa perché per estrarre
devo ribaltare tutto

scambio (s.q1, s.q2)

:

return x

→ Scambia i contenuti delle
due pile usando i puntatori

INSERISCI I NUMERI DA 1 A 5 IN UNA PILA E POI STAMPALLA

```
*include <stdio.h>
*include <stdlib.h>

/* dichiara le type pile */
typedef struct nodo{
    int info;
    struct nodo *next;
} nodo;
typedef nodo * List;

struct stack {
    List contents;
    int size;
}

typedef struct stack * stack;

/* definisce le funzioni */
Stack initstack (){
    stack s;
    s = (stack) malloc (sizeof (struct stack));
    s->contents = NULL;
    s->size = 0;
    return s;
}

int stack_empty (stack s){ ... }

void push (stack s, int elem){ ... }

int pop (stack s){ ... }

int top (stack s){ ... }

int size (stack s){ ... }
```



```

int main () {
    stack s;
    int i;
    s = initstack();
    for (i=1, i <= 5, i++)
        push (s, i);
    while (!stackempty(s))
        printf ("%d\n", pop(s));
    return EXIT_SUCCESS;
}

```

PROGRAMMA CUENTE

Aggiungi `#include "stack.h"` così da includere l'interfaccia

File `usePila.c`

SVANTAGGI:

- (1) Se ho un errore devo ricompilare
- (2) push / pop legate e non esportabili
- (3) accesso elementi che non sono gestito
- (4) per utilizzare un array al posto di una lista devo cambiare il programma

SOLUZIONE: uso più file

PROGRAMMA CUENTE: usa le tipi di dato

IMPLEMENTAZIONE: definizione tipo di dato + realizzazione operazioni sul tipo di dato

INTERFACCIA: [rappresenta i prototipi che possiamo usare] insieme dei prototipi delle funzioni che operano sul tipo di dato. `stack.h`

`stack.h` FILE

```

typedef struct stack* stack;           → ben definita perché è un indirizzo
stack initstack();
int stack_empty (stack s);
void push (stack s, int elem);
int pop (stack s);
int top (stack s);
int size (stack s);

```

} Esempio di dichiarazioni in cui ho un nome, stack, che non so cosa sia. Quindi ASSUNGO LA DEFINIZIONE DEL TIPO STACK (CHE È NEL FILE stack.h)

Implementazione - Imp. P. lista.c

`#include "stack.h"`



item.h

```

typedef struct item * Item;
Item oggi();
void stampa (Item x);
int compare (Item x, Item y)
/* ImpItemInt.c */
#include <stdlib.h>
#include <stdio.h>
#include "item.h"
struct item {
    int info;
};
/* usePila.c */
#include "item.h"
#include "stack.h"

```

`gcc -c modolo.c` → restituisce file oggetto.o
`gcc -c impPilaLista.c`] 2 file oggetto
`gcc -c usePila.c`

↳ gcc -c impPilaLista.c
usePila.c

`gcc impPilaLista.o usePila.o -o output`

↳ crea un eseguibile output

Una lista è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi in posizioni arbitrarie.

dati: una sequenza ℓ di elementi

op: $\text{createList}(\ell) \rightarrow \text{list}$

Post: restituisce una lista vuota

op: $\text{isEmpty}(\ell) \rightarrow \text{Bool}$

Post: restituisce true se ℓ è vuota, false altrimenti

op: $\text{search}(\ell, k) \rightarrow \text{Node}$

Post: restituisce il primo elemento con chiave k , se esiste, altrimenti null.

op: $\text{insert}(\ell, k, x) \rightarrow \text{list}$

Post: inserisce un nodo x (la chiave key è già stata impostata) nella lista

op: $\text{delete}(\ell, k) \rightarrow \text{list}$

Pre: $x \in \ell$

Post: cancella il nodo x dalla lista ℓ

op: $\text{size}(\ell) \rightarrow \text{int}$

Post: restituisce il numero di elementi della lista ℓ

Realizzo la lista con strutture collegate

1) Lista semplice



Un nodo della lista ha i seguenti attributi:

$x.\text{key}$ contiene la chiave (l'info che vogliamo memorizzare)

$x.\text{next}$ punta al successore di x nella nostra lista

$L.\text{head}$ denota la testa della lista

$L.\text{head} = \text{null}$ lista vuota

2) Lista doppia



N.B. elemento = nodo

$x.\text{key}$

$x.\text{next}$ punta al successore

$x.\text{prev}$ punta al predecessore di x nella lista

$L.\text{head}$

Una lista può anche avere:

1) un puntatore all'ultimo elemento



2) può essere circolare



3) può avere un nodo sentinella per meglio gestire i casi limite (inserimento in testa e in coda)



I dati memorizzati possono

- essere ordinati oppure non ordinati

- ammettere duplicati o tutti i valori delle chiavi sono distinti

ESERCIZIO : Ho ora questa semplice e voglio determinare se è circolare oppure no

int circolare (List e)

```
    primo = e;
    if (e == NULL || e->next == NULL)
        return 0;
    fineLista = 0;
    e = e->next;
    while & fineLista AND primo != e
        if e->next == NULL
            fineLista = 1
        else
            e = e->next;
    return primo == e;
```

ESERCIZIO : Voglio scorrere una sola volta una lista e dividerla a metà

L^d uso due puntatori, uno a velocità snella e uno a velocità doppia, ovvero uno che scorre ogni elemento e uno che ne salta unoogni volta

typedef struct node{

```
    int key;
    struct node *next;
```

} node;

typedef Node *list;

void split (list e, list *e1, list *e2) {

```
    list primo, secondo;
```

```
    primo = e;
```

```
    secondo = NULL;
```

```
    if (!e || !e->next)
```

```
        *e1 = e;
```

```
        *e2 = segundo;
```

```
    else {
```

```
        secondo = primo->next;
```

```
        while (secondo != NULL){
```

```
            if (secondo->next == NULL)
```

```
                secondo = secondo->next;
```

```
            else
```

```
                primo = primo->next;
```

```
                secondo = secondo->next->next;
```

```
}
```

```
*e2 = primo->next;
```

```
primo->next = NULL;  $\rightarrow$  così DIVIDO LE 2 LISTE
```

```
*e1 = e;
```

COMPLESSITÀ: $O(n)$



}

INVARIANTE = asserzione vera prima, dopo e ad ogni iterazione del ciclo.

Per dimostrare che è invariante, dimostro che:

(1) Inizializzazione: è vera prima della prima iterazione del ciclo

(2) Conservazione: se è vera prima di un'iterazione del ciclo, rimane vera prima della successiva
Inv \wedge Guardia \rightarrow Inv [dopo l'esec. del corpo del ciclo]

(3) Condizione: quando il ciclo termina l'invariante fornisce un'altra proprietà che il rapporto è corretto

Inv \wedge Guardia \rightarrow asserzione finale

Liste doppie

search(list L, item k)

x = L.head

while x ≠ NULL AND x.key ≠ k

x = x.next

return x

funzione di terminazione: funzione a valori naturali che decrese strettamente ad ogni iterazione dell'ut

$f(n) = \# \text{ elementi della lista}$
non ancora visitati

Inv = gli elementi da L.head a x non compreso sono diversi da k

INIZIAZZAZIONE

Inv = gli elementi compresi tra x e x escluso hanno valori diversi da k

Inv [L.head]

non ci sono elementi \rightarrow proprietà vacuamente vera

CONSERVAZIONE

Inv \wedge x ≠ NULL \wedge x.key ≠ k

Inv [x.next / x]

Ipotesi

(1) Inv = gli ee da L.head] \star

(2) x ≠ NULL \wedge x.key ≠ k

Dico dimostrare

Inv [x.next / x] = \star