1 Analisi ammortizzata

Normalmente si vuole analizzare la complessità di strutture dati che evolvono nel tempo ed esse sono soggette a una serie di operazioni (operazioni dipendenti dalle strutture).
L'analisi ammortizzata è una tecnica di complessità che valuta il tempo per eseguire, **nel caso pessimo**, una sequenza di operazioni su una struttura dati.
In alcuni casi, le strutture dati possono avere operazioni più o meno costose, a seconda dello stato della struttura dati stessa e quindi alcune operazioni possono essere estremamente poco costose, mentre altre possono essere molto costose.

Importante differenza

- **Analisi caso medio**: Probabilistica, su singola operazione;
- **Analisi ammortizzata**: Deterministica, su operazioni multiple, caso pessimo.

Se calcolo la media, non la devo calcolare su una singola operazione presa in sé e per sé, ma su un insieme di operazioni che si considerano come una sequenza tipica di operazioni che vengono fatte su una struttura dati. A quel punto si esegue un'analisi di caso pessimo su operazioni multiple. Quest'analisi è spesso deterministica, quindi non probabilistica, perché si va a considerare la sequenza di operazioni che è la più costosa possibile tra tutte le sequenze di operazioni.

1.1 Metodi per eseguire l'analisi

Esistono 3 metodi per poter eseguire l'analisi ammortizzata:

- **Metodo dell'aggregazione**: Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel **caso pessimo**. Questa è una tecnica derivata dalla matematica;
- **Metodo degli accantonamenti**: Alle operazioni vengono assegnati dei **costi ammortizzati** (da cui deriva il termine "ammortizzata" per l'analisi) che possono essere maggiori/minori del loro costo effettivo. Questa è una tecnica che deriva dall'economia;
- **Metodo del potenziale**: Lo stato del sistema viene descritto come una **funzione di potenziale**. Questa tecnica deriva dalla fisica.

1.2 Esempio: Contatore binario

Si ha un contatore binario di k bit in un vettore A di booleani 0/1. Il bit meno significativo sta nella posizione $A[0]$ e il bit più significativo è alla posizione $A[k-1]$.
Il valore del contatore è: $x = \sum_{i=0}^{k-1} A[i]2^i$. Si ha un'operazione detta *increment()* che incrementa il contatore di 1: L'operazione "*increment*" va a cercare il primo bit con valore 0, se non lo trova, ovvero se i bit hanno tutti valore 1, li trasforma in un valore 0 e passa al bit successivo, dal meno significativo verso il più significativo.
Quando arriva al primo bit vero, quindi quando la condizione del while non è rispettata, al bit i-esimo viene assegnato il valore 1.

```
increment(int[]A, int k)
1: int i=0
2: while i<k and A[i]==1 do
3:     A[i]=0
4:     i=i+1
5: end while
6: if i<k then
7:     A[i]=1
8: end if
```

x	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

1.3 Metodo dell'aggregazione

Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel caso pessimo. Poi si calcola il costo ammortizzato $T(n)/n$ come media su n operazioni.
Allora: come sequenza si devono considerare tutte le possibili sequenze di operazioni e nel caso pessimo si considera la peggiore in assoluto.
In questo caso (esempio del contatore), c'è un solo tipo di operazione, ovvero l'incremento. n operazioni sono quindi n operazioni di incremento e l'**aggregazione** è la sommatoria delle varie complessità individuali.
Nell'esempio il caso pessimo è $O(k)$ perché l'indice superiore è k, stabilito dal **while**.
Nel caso pessimo n operazioni costano: $k = \lceil log(n+1) \rceil$, cioè il numero di bit necessari per rappresentare n, una chiamata *increment()* costa $O(k)$, n operazioni costano $T(n) = O(nk)$ e il costo di un'operazione è $T(n)/n = O(k) = O(logn)$.
Per fare n operazioni, il costo ammortizzato calcolato in questo modo delle varie operazioni è pari a $log\, n$.
Si può osservare che il tempo necessario ad eseguire l'intera sequenza è proporzionale al numero di bit che vengono modificati.

x	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	#bit
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5
#bit	0	0	0	1	2	4	8	16	

Quindi, quello che si va a fare è provare a calcolare il numero di bit che verranno cambiati, che è la dimensione del ciclo "**while**", si prova a sommarli tutti e si guarda cosa salta fuori utilizzando i metodi dell'analisi matematica.
Si può notare che:

- $A[0]$ viene modificato ad ogni incremento: $(\lfloor n/2^0 \rfloor)$
- $A[1]$ viene modificato ogni 2 incrementi: $(\lfloor n/2^1 \rfloor)$
- $A[2]$ viene modificato ogni 4 incrementi: $(\lfloor n/2^2 \rfloor)$
  Arrivando quindi a:
- $A[i]$ viene modificato ogni 4 incrementi: $(\lfloor n/2^i \rfloor)$

Analisi ammortizzata

- **Costo totale**: $T(n) = \sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{+\infty} (\frac{1}{2})^i = 2n$. Il 2 rappresenta la sommatoria. Quindi "n" operazioni su un contatore binario costano $2 * n$
- **Costo ammortizzato**: Se divido $2n$ per n si ottiene il valore 2 come limite superiore e quindi: $T(n)/n \leq 2n/n = 2 = O(1)$

1.4 Metodo degli accantonamenti

Quello che si va a fare con questo metodo è assegnare un costo ammortizzato potenzialmente distinto ad ognuna delle operazioni possibili.
In questo caso (sempre il contatore binario) esiste un'unica operazione, quindi si assegna un costo all'unica operazione possibile.
Il costo ammortizzato può essere diverso dal costo effettivo, ovvero alcune operazione, quelle che coinvolgono tanti bit, sono più costose e quindi il costo ammortizzato di queste operazioni viene "pagato" dal **credito** che la struttura dati ha accumulato eseguendo operazioni costose.
Le operazioni meno costose vengono caricate di un costo aggiuntivo detto **credito** e il costo ammortizzato per le operazioni meno costose è dato da: **costo ammortizzato** $=$ **costo effettivo** $+$ **credito prodotto**.
Il costo ammortizzato per le operazioni più costose è dato dal costo effettivo meno il credito consumato: **costo ammortizzato** $=$ **costo effettivo** $-$ **credito consumato**.
Nel determinare qual è il costo che si vuole dare alle operazioni, si ha un obiettivo, ovvero dimostrare che la somma dei costi ammortizzati "a_i" è un limite superiore alla somma dei costi

effettivi "c_i":

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$$

Il segno "$\leq$" indica il limite superiore definito da a_i.
Si vuole dimostrare che questa relazione è sempre valida per come si sono creati i valori ammortizzati e anche che il valore ottenuto da "$\sum_{i=1}^{n} a_i$" è poco costoso. Si deve anche assumere che esista sempre un credito dopo una serie di operazione, quindi l'operazione t-esima su n possibili operazioni. Il credito è espresso da questa formula:

$$\sum_{i=1}^{t} a_i - \sum_{i=1}^{t} c_i \geq 0$$

Ovvero la sommatoria dei costi ammortizzati meno la sommatoria dei costi reali.
Tale differenza deve sempre essere maggiore o uguale a zero.
Il costo effettivo dell'operazione di `increment()` è d, dove d è il numero di bit che cambiano valore, però, per ogni riga, d continua a variare.
L'idea è che $\sum_{i=1}^{t} c_i$ sarà pari a d per ognuna delle operazioni. Invece si deve trovare un modo più semplice per fare questa operazione, quindi ciò che si fa, è cercare di dire che questo costo $\sum_{i=1}^{t} c_i$ è minore o uguale di $\sum_{i=1}^{t} a_i$ e si decide che il costo ammortizzato è uguale a 2, quindi:

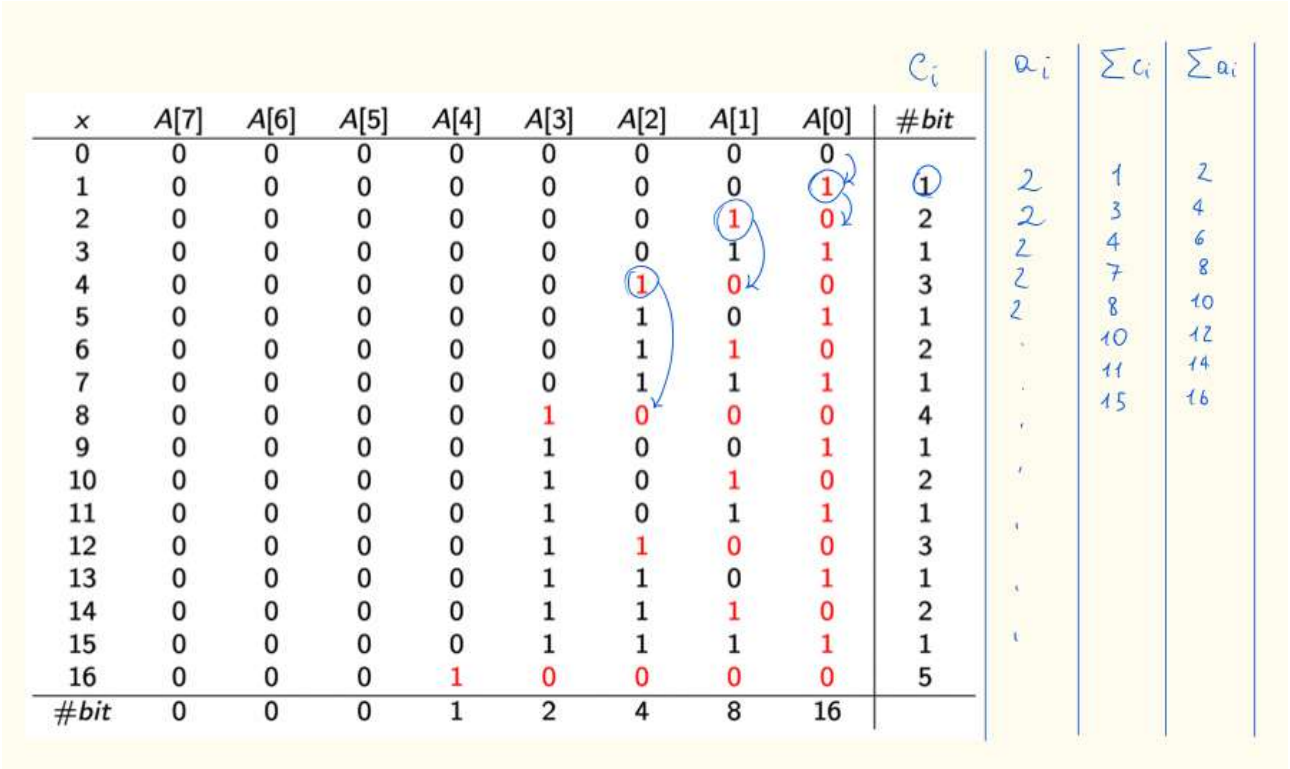
$$\sum_{i=1}^{t} c_i \leq \sum_{i=1}^{t} a_i = \sum_{i=1}^{t} 2 = 2t$$

Ma come si è stabilito?

Si sa che all'inizio si hanno tutti i bit a 0. Tutte le volte che si cambia un bit e lo si trasforma a 1 si ha un costo effettivo, ma, ad un certo punto nel futuro, quel bit a 1 tornerà a 0, quindi si paga subito quest'altro costo, pari a 1, per il futuro cambio del bit da 1 a 0.
Il costo ammortizzato dell'operazione `increment()` sarà quindi $2 = 1 + 1$:

- 1 per il cambio di un bit da 0 a 1 (**costo effettivo**)
- 1 per il futuro cambio dello stesso bit da 1 a 0.

Lo si paga subito, anche se questo cambio potrebbe non essere fatto in futuro Perché quel bit potrebbe rimanere a 1 sino al termine delle n operazioni, però questo è un limite superiore.
La somma di t operazioni sarà limitata superiormente da $2 * t$, quindi, su n operazioni, si avrà un costo, che è $O(n)$ (che sarebbe $2n$), e dividendo per n si ottiene $O(1)$.
Ricapitolando, il costo deriva dai cambi di bit, perché i bit che non vengono toccati, perché sono troppo avanti nella somma, non interessano. Quando si cambia un bit da 0 a 1, in futuro lo si potrebbe dover ri-cambiare a 0, quando arriverà il suo turno nei vari `increment` e lo si paga subito.
La somma dei costi ammortizzati è sempre più alta della somma dei costi effettivi e deve essere così secondo la relazione.



x	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	#bit	a_i	$\sum c_i$	$\sum a_i$
0	0	0	0	0	0	0	0	0				
1	0	0	0	0	0	0	0	1	1	2	1	2
2	0	0	0	0	0	0	1	0	2	2	3	4
3	0	0	0	0	0	0	1	1	1	2	4	6
4	0	0	0	0	0	1	0	0	3	2	7	8
5	0	0	0	0	0	1	0	1	1	2	8	10
6	0	0	0	0	0	1	1	0	2	2	10	12
7	0	0	0	0	0	1	1	1	1	2	11	14
8	0	0	0	0	1	0	0	0	4	2	15	16
9	0	0	0	0	1	0	0	1	1			
10	0	0	0	0	1	0	1	0	2			
11	0	0	0	0	1	0	1	1	1			
12	0	0	0	0	1	1	0	0	3			
13	0	0	0	0	1	1	0	1	1			
14	0	0	0	0	1	1	1	0	2			
15	0	0	0	0	1	1	1	1	1			
16	0	0	0	1	0	0	0	0	5			
#bit	0	0	0	1	2	4	8	16				

I costi ammortizzati per ogni riga sono uguali a 2 e la sommatoria dei costi ammortizzati $\sum a_i$ è maggiore della sommatoria dei costi effettivi $\sum c_i$.
Alla fine si otterrà:

x	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	c_i	a_i	$\sum c_i$	$\sum a_i$
1	0	0	0	0	0	0	0	1	1	2	1	2
2	0	0	0	0	0	0	1	0	2	2	3	4
3	0	0	0	0	0	0	1	1	1	2	4	6
4	0	0	0	0	0	1	0	0	3	2	7	8
5	0	0	0	0	0	1	0	1	1	2	8	10
6	0	0	0	0	0	1	1	0	2	2	10	12
7	0	0	0	0	0	1	1	1	1	2	11	14
8	0	0	0	0	1	0	0	0	4	2	15	16
9	0	0	0	0	1	0	0	1	1	2	16	18
10	0	0	0	0	1	0	1	0	2	2	18	20
11	0	0	0	0	1	0	1	1	1	2	19	22
12	0	0	0	0	1	1	0	0	3	2	22	24
13	0	0	0	0	1	1	0	1	1	2	23	26
14	0	0	0	0	1	1	1	0	2	2	25	28
15	0	0	0	0	1	1	1	1	1	2	26	30
16	0	0	0	1	0	0	0	0	5	2	31	32

1.5 Metodo del potenziale

Si ha una struttura dati con uno stato, una serie di bit a 0 e una serie di bit a 1. Si crea una **funzione di potenziale** ϕ che associa ad uno stato D della struttura dati la "*quantità di lavoro*" $\phi(D)$ che è stato contabilizzato nell'analisi ammortizzata, ma non ancora eseguito. Significa che, in altre parole, $\phi(D)$ rappresenta la quantità di energia potenziale immagazzinata in quello stato e il costo ammortizzato dipende dal costo effettivo a cui viene associata una variazione di potenziale, che può essere negativa, se sta consumando energia immagazzinata in quello stato, o positiva, se sta invece accumulando energia immagazzinata in quello stato. Quello che si va a fare è calcolare il costo ammortizzato per definizione come:

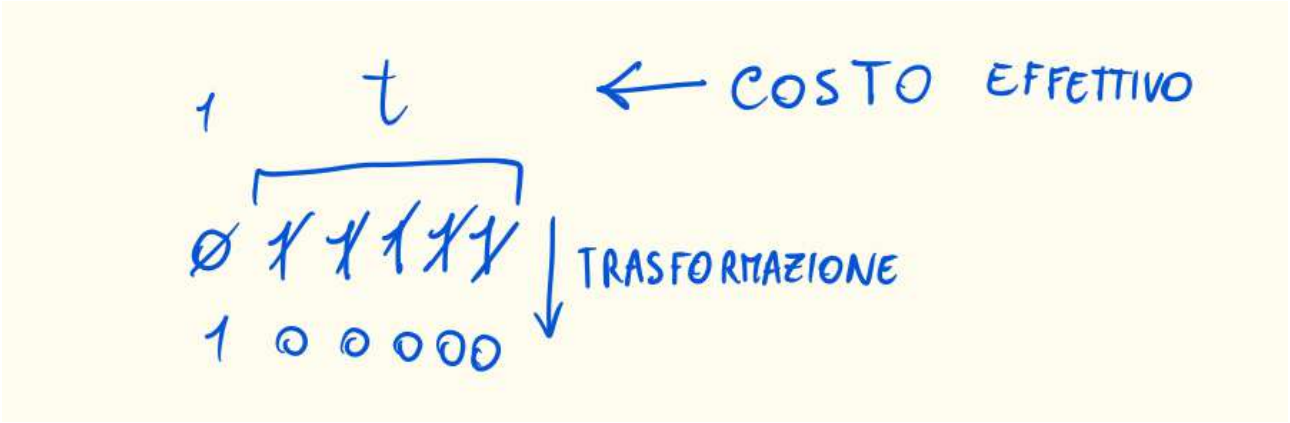
Costo ammortizzato

Costo ammortizzato = **Costo effettivo** + **Variazione di potenziale**

$$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

Dove D_i è lo stato associato alla i-esima operazione.

In questo caso, utilizzando i concetti precedenti, si può scegliere come funzione di potenziale ϕ il numero di bit 1 presenti nel contatore, perché quelli sono i bit che che in qualche modo si sono trasformati da 0 a 1 e che avranno un costo per essere ritrasformati al valore 0.
Tutte le volte che si chiama un `increment`, si indica con t il numero di bit a 1 incontrati a partire dal meno significativo, prima di incontrare uno 0, e si considera come costo effettivo $1 + t$:



La variazione di potenziale è $1 - t$ perché si sono tolti tutti i bit a 1 facenti parte di t e si è trasformato lo 0 a 1.

$$
\begin{aligned}
A &= \sum_{i=1}^{n} a_i \\
&= \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1})) \\
&= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\phi(D_i) - \phi(D_{i-1})) \\
&= C + \phi(D_1) - \phi(D_0) + \phi(D_2) - \phi(D_1) + \cdots + \phi(D_n) - \phi(D_{n-1}) \\
&= C + \phi(D_n) - \phi(D_0)
\end{aligned}
$$

Il costo ammortizzato si calcola come una sequenza di n operazione, quindi si vanno a sommare i costi ammortizzati, poi si sostituisce con la formula del costo ammortizzato, si separano le sommatorie del costo effettivo, che è quella che si conosce, è il costo effettivo delle operazioni, e la sommatoria di $(\phi(D_i) - \phi(D_i - 1))$. Alla fine si ottiene $C + \phi(D_n) - \phi(D_0)$.
Il costo è dato di n operazioni è dato dalla differenza di potenziale a partire dallo stato iniziale. Se la variazione di potenziale $\phi(D_n) - \phi(D_0)$ è non negativa, il costo ammortizzato A è un limite superiore al costo reale.
La formula del costo ammortizzato per il contatore binario nel metodo del potenziale è: $1 + t + 1 - t = 2$, dove $1 + t$ è il costo effettivo e $1 - t$ è la variazione del potenziale.

1.6 Analisi ammortizzata e strutture dati: Array VS Liste

Le sequenze lineari possono essere realizzate con:

- **Array**: Sequenze ad accesso diretto in cui, dato j, si accede direttamente all'elemento a_j. Il costo di ciascun accesso è costante ed è tutto memorizzato in locazioni di memoria consecutive.
- **Liste**: Sequenze ad accesso sequenziale in cui, dato j, si accede ad $a_0, a_1, \ldots, a_j$ attraversando la sequenza a partire dall'inizio (o dalla fine). Si ha un costo $O(p)$ per raggiungere a_{j+p} partendo a_j. Gli elementi della lista sono memorizzati in locazioni di memoria non necessariamente contigue.

Vantaggio delle liste

Nessun limite alla dimensione massima della sequenza

1.6.1 Array dinamici

Alcuni linguaggi di programmazione prevedono array che possono essere ridimensionati, detti dinamici.
Ci sono delle operazioni che sono necessarie per ridimensionare un array A di dimensione n:

- Creare un nuovo array B;
- Copiare gli elementi di A in B;
- Deallocare A dalla memoria;
- Ridenominare B come A.

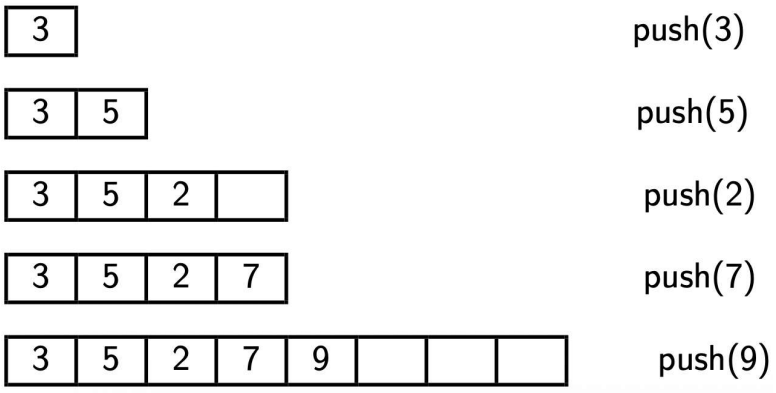
Il costo del ridimensionamento è $O(n)$. Il ridimensionamento è necessario quando si vuole aggiungere l'elemento $n+1-esimo$ all'array di dimensione n e ogni aggiunta di un elemento $n+1-esimo$ ad una lista di dimensione n ha costo $O(1)$. In termini di memoria allocata si ha un vantaggio, ma in termini computazionali è un costo oneroso.

1.6.2 Array dinamici - Espansione

Il problema di un array dinamico è che non è noto a priori quanti elementi entreranno nella sequenza e quindi la capacità iniziale disponibile potrebbe non rivelarsi sufficiente. La soluzione p allocare un vettore di capacità maggiore, si copia il contenuto del vecchio vettore nel nuovo e si rilascia il vecchio vettore. Il caso peggiore è quello in cui si devono fare continui inserimenti nel vettore.
L'espansione può avvenire in più modi, qui vengono considerati il **raddoppiamento** e l'**incremento costante**.

1.6.3 Analisi ammortizzata - Raddoppiamento del vettore

Il ridimensionamento comporta un costo elevato, perché implica la costruzione di un nuovo array con dimensione aumentata e la successiva copia degli elementi. Per evitare di eseguire questo procedimento ogni volta che si aggiunge un nuovo elemento, gli array dinamici si ridimensionano raddoppiando le dimensioni.



Il costo effettivo di un'operazione di inserimento c_i è i se $i-1$ è una potenza di 2 mentre è 1 in tutti gli altri casi.

$$c_i = \begin{cases} i & \exists k \in Z_0^+ : i = 2^k + 1 \\ 1 & altrimenti \end{cases}$$

n	costo
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Assunzioni:

- Dimensione iniziale dell'array: 1
- Costo di scrittura di un elemento: 1

Il costo effettivo di n operazioni di inserimento in questo caso risulta essere:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} c_i \\ &= n + \sum_{i=1}^{\lfloor log\, n \rfloor} 2^j \\ &= n + 2^{\lfloor log\, n \rfloor + 1} - 1 \\ &\leq n + 2^{log\, n + 1} - 1 \\ &= n + 2n - 1 = O(n) \end{aligned}$$

E si ottiene il costo ammortizzato di un'operazione di inserimento in questo modo:

$$T(n)/n = \frac{O(n)}{n} = O(1)$$

1.6.4 Analisi ammortizzata - Incremento del vettore

Costo effettivo di un'operazione di inserimento:

$$c_i = \begin{cases} i & (i \, mod \, d) = 1 \\ 1 & altrimenti \end{cases}$$

n	costo
1	1
2	1
3	1
4	1
5	$1 + d = 5$
6	1
7	1
8	1
9	$1 + 2d = 9$
10	1
11	1
12	1
13	$1 + 3d = 13$
14	1
15	1
16	1
17	$1 + 4d = 17$

Assunzioni:

- Dimensione iniziale dell'array: d
- Incremento: d
- Costo di scrittura di un elemento: 1

Nell'esempio d=4

Il costo effettivo di n operazioni di inserimento in questo caso risulta essere:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} c_i \\ &= n + \sum_{j=1}^{\lfloor n/d \rfloor} d * j \\ &= n + d \sum_{j=1}^{\lfloor n/d \rfloor} *j \\ &= n + d * \frac{(\lfloor n/d \rfloor + 1)\lfloor n/d \rfloor}{2} \\ &\leq n + \frac{(n/d + 1)n}{2} = O(n^2) \end{aligned}$$

E si ottiene il costo ammortizzato di un'operazione di inserimento in questo modo:

$$T(n)/n = \frac{O(n^2)}{n} = O(n)$$