Progammazione e Algoritmica

Angelo Passarelli

April 26, 2022

Sommario

1 Algoritmi di Ordinamento

1.1 Analisi Complessità

	Complessità		
Algoritmo	Caso Migliore	Caso Peggiore	Caso Medio
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
QuickSort	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Counting Sort	$O(max\{n, k\})$	$O(max\{n, k\})$	$O(max\{n, k\})$
Radix Sort	$O(d(n + k))$	$O(d(n + k))$	$O(d(n + k))$

1.2 Insertion Sort

1.2.1 Descrizione

1. All'inizio dell'algoritmo l'insieme ordinato è vuoto (`[]`).
2. Il primo elemento dell'array (`A[0]`) risulta già ordinato rispetto alla sottosequenza che prendiamo in considerazione (infatti il `for` parte da `j = 1`).
3. L'elemento successivo (`key`) viene confrontato dall'elemento precedente a `j` fino all'ultima cella dell'array, solo fino a quando `key` risulta più piccolo dei sui elementi precedenti.
4. Nel caso in cui `key` deve occupare la cella già occupata da un altro elemento, occorre shiftare tutti gli elementi più grandi di `key` alla sua destra (riga `6`).
5. Si ripete dal punto `3.` per tutti gli `n` elementi.

1.2.2 Invariante di Ciclo

All'inizio, durante e alla fine del ciclo `for` la porzione dell'array `[0, j-1]` risulta ordinata.

```
1  function InsertionSort(A) {
2      for(let j = 1; j < A.length; j++){
3          let key = A[j];
4          i = j - 1;
5          while(i >= 0 && A[i] > key){
6              A[i + 1] = A[i];
7              i = i - 1;
8          }
9          A[i + 1] = key;
10     }
11 }
```

code/insertion.js

1.3 Selection Sort

1.3.1 Descrizione

1. Si cerca il minimo dell'array nella sottoporzione `[i, n-1]` con `i` che parte da `0`.
2. Alla fine della ricerca, l'elemento minimo viene posto all'inizio della sottoporzione.
3. Si ritorno al punto `1.` fino a `n-1`.

1.3.2 Invariante di Ciclo

All'inizio, durante e alla fine del primo ciclo `for` la porzione dell'array `[0, i]` risulta ordinata.

```
1 function SelectionSort(A) {
2     for(let i = 0; i < A.length - 1; i++){
3         let min = i;
4         for(let j = i + 1; j < A.length; j++){
5             if(A[j] < A[min]) min = j;
6         }
7         Swap(A[i], A[min]);
8     }
9 }
```

code/selection.js

1.4 Merge Sort

1.4.1 Descrizione

1. L'array viene prima diviso in 2 parti ricorsivamente fino a quando la sottoporzione da dividere non raggiunge dimensione 1.
2. Successivamente, a partire dall'ultima scomposizione (quindi all'inizio avremo tutte le celle di lunghezza 1 che per definizione sono già ordinate) viene effettuata la procedura di `Merge()` che prende due porzioni di array già ordinate e le fonde in un unico array in modo da mantenere l'ordinamento.

1.4.2 Descrizione `Merge()`

1. All'inizio vengono prima calcolate le dimensioni delle 2 sottosequenze.
2. Successivamente vengono creati due array di appoggio dove saranno copiati i valori delle 2 sottoporzioni.
3. Nell'ultima cella dei due array di appoggio viene posto un valore sentinella (in questo caso $+\infty$) in modo tale che quando avremo terminato di inserire nell'array principale i valori di una delle due sottosequenze, potremo continuare a copiare gli elementi dell'altra sottoporzione in modo corretto dato che verranno sempre confrontati con $+\infty$.

4. L'idea di base del `Merge()` si trova all'interno del ciclo `for`. Infatti i due array essendo già ordinati, per trovare il valore più piccolo della loro unione basterà confrontare i valori minimi corrispettivi che all'inizio si troveranno nella cella `0`.

5. Nel caso in cui il valore minimo si trovi in `L`, il suo valore sarà copiato in `A[k]` (con `k` che parte da `p`) e l'indice corrispondente a `L` sarà incrementato di 1, invece se il valore minimo è contenuto in `R`, l'indice incrementato sarà `j`.

6. Quindi alla fine di ogni ciclo `for` saranno sempre confrontati gli elementi più piccoli dei due array che non sono stati ancora copiati in `A`.

7. Il ciclo termina quando vengono copiati tutti gli `r` elementi. In questo modo preveniamo anche che non vengano copiati i valori sentinella.

1.4.3 Invariante di Ciclo

All'inizio di ogni iterazione del `for` il sottorray `A[p...k-1]` contiene i `k-p` elementi più piccoli di `L` e `R` già ordinati.
Inoltre `L[i]` e `R[j]` contengono i più piccoli elementi che non sono stati ancora copiati in `A`.

```
1  function MergeSort(A, p, r) {
2      if(p < r){
3          let q = (p + r) / 2;
4          MergeSort(A, p, q);
5          MergeSort(A, q + 1, r);
6          Merge(A, p, q, r);
7      }
8  }
9
10 function Merge(A, p, q, r) {
11     let n1 = q - p + 1;
12     let n2 = r - q;
13     let L = new Array(n1 + 1);
14     let R = new Array(n2 + 1);
15     for(let i = 0; i < n1; i++) L[i] = A[p + i - 1];
16     for(let j = 0; j < n2; j++) R[j] = A[q + j];
17     L[n1] = +Infinity;
18     R[n2] = +Infinity;
19     let i = 0;
20     let j = 0;
21     for(let k = p; k < r; k++){
22         if(L[i] <= R[j]){
23             A[k] = L[i];
24             i = i + 1;
25         }
26         else{
27             A[k] = R[j];
28             j = j + 1;
29         }
30     }
31 }
```

code/merge.js

1.5 QuickSort

1.5.1 Descrizione

1. Viene scelto un elemente chiamato `pivot` (nel codice `q`) e vengono spostati a sinistra tutti gli elementi di piccoli del `pivot` e a destra tutti gli elementi più grandi (funzione `Partiziona()`).
2. Successivamente viene richiamato il `QuickSort()` ricorsivamente sulla partizione a sinistra del `pivot` e su quella a destra.

1.5.2 Descrizione `Partiziona()`

1. Come `pivot` viene scelto `x` che rappresenta l'ultima cella della sottoporzione.
2. Il funzionamento si base su due indici `i` e `j`, in modo tale che alla fine tra `p` e `i` avremo gli elementi più piccoli di `x` e tra `i+1` e `j` avremo i più grandi.
3. In questo modo, nel ciclo `for` ogni volta che troviamo un elemento minore di `x` incrementiamo di 1 la dimensione della sottosequenza `[p, i]` e ci spostiamo l'elemento in considerazione nell'ultima posizione.
4. Al termine del `for` scambiamo il `pivot` con il primo degli elementi più grandi di esso e restituiamo la posizione del `pivot`.
5. In questo modo il l'elemento corrispondente al `pivot` si troverà nella posizione corretta per ottenere l'array ordinato.

1.5.3 Invariante di Ciclo

All'inizio di ogni iterazione del `for` preso qualsiasi indice `k` della sottoporzione:

- Se $\texttt{p} \leq \texttt{k} \leq \texttt{i}$, allora $\texttt{A[k]} \leq \texttt{x}$.
- Se $\texttt{i+1} \leq \texttt{k} \leq \texttt{j-1}$, allora $\texttt{A[k]} > \texttt{x}$.
- Se $\texttt{k} = \texttt{r}$, allora $\texttt{A[k]} = \texttt{x}$.

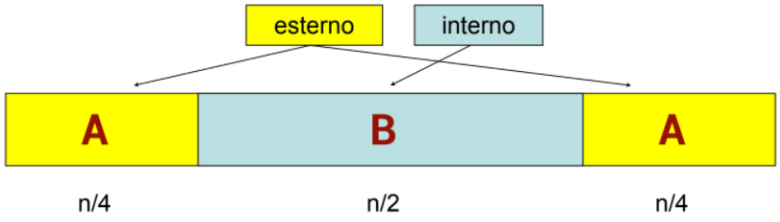
Gli indici tra `j` e `r-1` non ci interessano perchè sono ancora da confrontare.

1.5.4 Costo

Il costo del `QuickSort()` dipende dal bilanciamento delle 2 sottoporzioni.
Infatti se il `pivot` si trova sempre al centro della sottosequenza allora la complessità sarà $\Theta(n \log n)$ (caso migliore).
Invece il caso peggiore si verifica quando `Partiziona()` produce una sottosequenza lunga `n-1` e l'altra quindi lunga `0`.
In questo caso la complessità è uguale a $\sum_{i=1}^{n}(n-i) = \Theta(n^2)$.

1.5.5 Dimostrazione Costo al Caso Medio

- Per randomizzare la procedura devo fare in modo che il `pivot` venga scelto in modo casuale tra `p` ed `r`.
- Definiamo due eventi con la stessa probabilità che possano verificarsi: ovvero che il `pivot` finisca nella zona esterna della sequenza o che finisca nella zona interna.



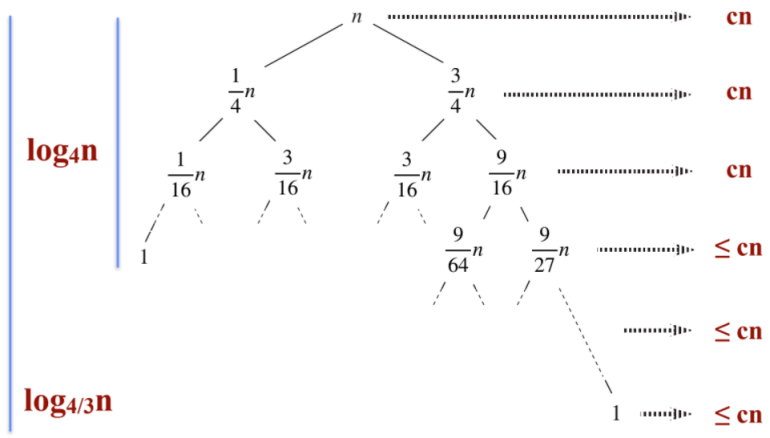
- Nel caso in cui il `pivot` finisca nella zona esterna, il caso peggiore si verifica quando esso si trova nella prima posizione, nel caso della sezione gialla di sinistra, o nell'ultima nel caso di quella di destra.
- Quindi in questo caso la complessità sarà: $T(n) = T(n-1) + O(n)$.
- Invece, nel caso in cui il `pivot` finisca nella zona interna, il caso peggiore si verifica quando esso si trova o all'inizio di `B` quindi in posizione $\frac{n}{4}$ o alla fine di `B`, in posizione $\frac{3}{4}n$.
- Quindi la complessità dell'algoritmo al caso peggiore sarà: $T(n) = T(n/4) + T(3/4\ n) + O(n)$.
- Adesso occorre combinare la situazione `A` con `B`, e dato che, come già detto, hanno la stessa probabilità di verificarsi:

$$\begin{aligned} T(n) &\leq A/2 + B/2 = \frac{1}{2}(A+B) \\ &= \frac{1}{2}[T(n-1) + T(n/4) + T(3/4\ n) + O(n)] \\ &\leq \frac{1}{2}[T(n) + T(n/4) + T(3/4\ n) + O(n)] \end{aligned}$$

- Adesso moltiplico per 2 a destra e a sinistra e porto il `T(n)` nel secondo membro nel primo:

$$2T(n) \leq T(n) + T(n/4) + T(3/4\ n) + O(n)$$
$$T(n) \leq T(n/4) + T(3/4\ n) + O(n)$$

- Ora per risolvere questa equazione utilizziamo l'albero di ricorrenza:



- Qui possiamo notare come l'altezza del ramo più a sinistra sarà $\log_4 n$ mentre quella del ramo più a destra è $\log_{\frac{4}{3}} n$.
- Inoltre possiamo dire che se facciamo la somma dei nodi su ogni livello, fino a quando ogni livello è completo, questa sarà sempre uguale a cn.
- Invece dal livello in cui l'albero inizia a diventare sbilenco, fino all'ultimo livello, possiamo limitare superiormente la somma sempre con cn.
- Quindi in conclusione possiamo dire che la complessità del `QuickSort()` al caso medio è $O(n \log n)$.

```
1  function QuickSort(A, p, r) {
2      if (p < r) {
3          q = Partiziona(A, p, r);
4          QuickSort(a, p, q - 1);
5          QuickSort(a, q + 1, r);
6      }
7  }
8
9  function Partiziona(A, p ,r) {
10     let x = A[r];
11     let i = p - 1;
12     for(let j = p; j <= r - 1; j++){
13         if(A[j] <= x){
```

```
14              i = i + 1;
15              Swap(A[i], A[j]);
16          }
17      }
18      Swap(A[i + 1], A[r]);
19      return i + 1;
20  }
```

code/quicksort.js

1.6 HeapSort

1.6.1 Descrizione

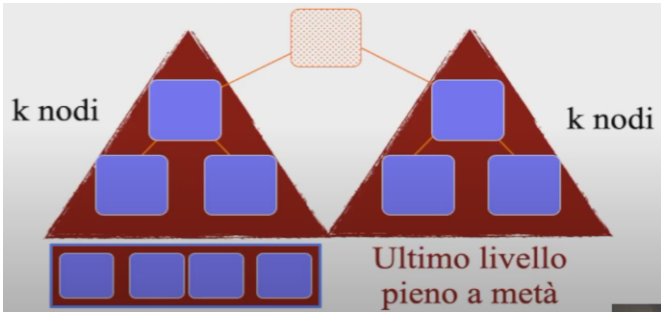
- L'`HeapSort()` si basa su una struttura dati chiamata Heap, ovvero un albero binario quasi completo, quindi dove tutti i livelli tranne l'ultimo sono completi e le foglie sull'ultimo livello vengono inserite da sinistra a destra.
- L'implementazione dell'Heap avviene tramite un array, in questo modo dato un nodo interno di indice `i`, il figlio a sinistra si trova in posizione $2 \cdot i$ e quello a destra in $(2 \cdot i) + 1$ (se l'array parte da 1, nel caso dovesse partire da 0 il figlio a sinistra si troverebbe in posizione $(2 \cdot i) + 1$, mentre quello a destra in $(2 \cdot i) + 2$).
- Invece il genitore di `i` si trova in posizione $\lfloor \frac{i-1}{2} \rfloor$.
- Nell'array la prima foglia si trova in posizione $\frac{n}{2}$.

1.6.2 Descrizione `Max_Heapify()`

- L'idea della procedura di `Max_Heapify()` è quella di avere un albero con radice `i` dove il sottoalbero sinistro e il sottoalbero destro sono già degli alberi di Max-Heap, in questo modo la funzione fà scorrere `i` nell'albero fino a quando non raggiunge la posizione corretta.
- All'inizio viene prima verificato se `A[i]` è minore del suo figlio sinistro o del suo figlio destro, nel caso in cui questo non si verifica vuol dire che l'albero è già un Max-Heap, altrimenti viene assegnato a `max` il valore di `l` o di `r` a seconda del caso.
- Quindi, a questo punto, se l'elemento maggiore è proprio `A[i]` la funzione termina, altrimeni vengono scambiati `A[i]` e `A[max]` e viene richiamata ricorsivamente la `Max_Heapify()` sul sottoalbero sinistro o destro (a seconda se il valore massimo si trova in `A[l]` o in `A[r]`).

Costo al Caso Peggiore

- Nel caso peggiore l'albero avrà l'ultimo livello tutto pieno solo a sinistra, questo perchè la differenza tra i nodi dei due sottoalberi è massima.



- Dato che nell'ultimo livello avrò `k + 1` nodi, i nodi totali dell'albero saranno $n = (2k + 1) + (k + 1) = 3k + 2$.
- Dato che il caso peggiore avviene quando si procede verso sinistra dove abbiamo $2k + 1$ nodi, occorre scrivere questo valore in funzione di `n`.
- Quindi dato che $n = 3k + 2$, allora $k = (n - 2)/3$.
- Adesso sostituisco `k` nel numero di nodi a sinistra e ottengo: $2(n-2)/3 + 1 = \frac{2}{3}n - \frac{1}{3} < \frac{2}{3}n$.
- Quindi dato che a sinistra c'è un numero di nodi inferiore a $\frac{2}{3}n$, l'equazione di ricorrenza al caso peggiore sarà:

$$T(n) = T(2/3n) + \Theta(1)$$

- Applicando il `Master Theorem` otteniamo $O(\log n)$.

1.6.3 Descrizione `Build_Max_Heap()`

- La procedura `Build_Max_Heap()` consente di trasformare un array `A` in un albero di Max-Heap.
- Dato che le foglie, se prese singolarmente, sono già degli alberi di Max-Heap, questa funzione chiama la `Max-Heapify` su tutti i nodi interni a partire dall'ultimo nodo non foglia fino alla radice dell'albero generale.

1.6.4 Invariante di Ciclo `Build_Max_Heap()`

All'inizio di ogni iterazione del `for`, ogni nodo $x \in \{i + 1, \ldots, n\}$ è radice un Max-Heap.
La proprietà viene sempre soddisfatta, anche prima della prima iterazione, perchè i nodi presi in considerazione sono tutte delle foglie che per definizione sono già dei Max-Heap.

Costo

- Quando chiamiamo la `Max-Heapify()`, il suo costo è proporzionale all'altezza del nodo su cui la invochiamo.
- Quindi per determinare il costo della `Build-Max-Heap()` occorre calcolare una somma pesata del valore dell'altezza su ogni nodo:

$$T(n) = \sum_{i=0}^{h} n_i \cdot h_i$$

- In questa sommatoria, `h` rappresenta l'altezza dell'albero e quindi il numero di livelli che ha, mentre n_i è il numero di nodi al livello `i`, i quali, dato che avranno la stessa altezza possiamo moltiplicare il valore dell'altezza al livello `i` (h_i) per il numero di nodi su quel livello.
- Ora dato che il numero di nodi su un dato livello `i` è uguale a 2^i e l'altezza di un nodo è uguale all'altezza dell'albero meno il livello a cui si trova il nodo ($h_i = h - i$), possiamo sostituire:

$$T(n) = \sum_{i=0}^{h} 2^i \cdot (h - i)$$

- Adesso scrivo 2^i come $\frac{1}{2^{-i}}$ e moltiplico numeratore e denominatore per 2^h:

$$T(n) = \sum_{i=0}^{h} \frac{h-i}{2^h \cdot 2^{-i}} 2^h = \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$

- Ora cambio variabile $k = h - i$ e porto 2^h fuori dalla sommatoria:

$$\begin{aligned} T(n) &= 2^h \sum_{i=0}^{h} \frac{k}{2^k} \\ &\leq n \sum_{i=0}^{\infty} \frac{k}{2^k} \\ &= n \cdot 2 \\ &= O(n) \end{aligned}$$

- Negli ultimi passaggi abbiamo posto un limite superiore della sommatoria facendola iterare fino ad infinito, quella essendo una sommatoria notevole, il suo valore è 2 e quindi $T(n) = O(n)$.

1.6.5 Descrizione `HeapSort()`

1. All'inizio l'array `A` viene trasformato in un albero di `Max-Heap`.
2. Successivamente dato che l'elemento più grande si trova nella radice viene posto alla fine dell'array.
3. La lunghezza dell'Heap (`n`) viene decrementata di 1 perchè nell'ultima cella abbiamo già l'elemento in posizione corretta.
4. Data che abbiamo inserito in `A[0]` un valore che non sappiamo essere più grande dei suoi sottoalberi, occorre richiamare la `Max_Heapify` di nuovo sull'array ma questa volta solo fino ad `n-1`.
5. Questa procedura viene eseguita iterativamente fino a quando `i` non raggiunge il valore di 1 (non fino a 0 dato che non ha senso ordinare una porzione di array con solo un elemento).

1.6.6 Invariante di Ciclo

All'inizio di ogni `for` la sottoporzione `A[0,...,i]` è un Max-Heap che contiene gli `i` elementi più piccoli di `A` e il sottoarray `A[i+1,...,n]` contiene gli `n-1` elementi più grandi di `A` ordinati.

```
1  function HeapSort(A) {
2      let n = A.length - 1;
3      Build_Max_Heap(A, n);
4      for(let i = A.length - 1; i <= 1; i--){
5          Swap(A[0], A[i]);
6          n = n - 1;
7          Max_Heapify(A, 0, n);
8      }
9  }
10
11 function Max_Heapify(A, i, n) {
12     let l = 2 * i;
13     let r = 2 * i + 1;
14     let max;
15     if (l <= n && A[l] > A[i]) {
16         max = l;
17     } else {
18         max = i;
19     }
20     if(r <= n && A[r] > A[max]){
21         max = r;
22     }
23     if(max != i){
24         Swap(A[i], A[max]);
25         Max_Heapify(A, max, n);
26     }
27 }
28
29 function Build_Max_Heap(A, n) {
30     for(let i = (n / 2); i <= 0; i--){
31         Max_Heapify(A, i, n);
```

```
    }
}
```

code/heapsort.js

1.7 Counting Sort

1.7.1 Descrizione

- L'idea è quella di ordinare un array `A` di `n` elementi dove ogni `A[i]` $\in \{0, 1, 2, \ldots, k\}$.
- Inizialmente viene istanziato un array `C` di lunghezza `k` dove in posizione `i` saranno contate il numero di occorrenze del numero `i` in `A`.
- Infine ogni numero `z` $\in \{0, \ldots, k\}$ viene inserito nell'array ordinato `B` per `v` volte.

```
function CountingSort(A, B, k) {
    let C = [];
    for(let i = 0; i <= k; i++) C[i] = 0;    //O(k)
    for(let j = 0; j < A.length; j++) C[A[j]] += 1; //O(n)
    let j = 0;
    for(let z = 0; z <= k; z++){    //O(n) -> il nr. di scritture
    su B e' per 'n' volte
        for(let v = 0; v < C[z]; v++){
            B[j] = z;
            j++;
        }
    }
}
```

code/counting.js

1.8 Radix Sort

1.8.1 Descrizione

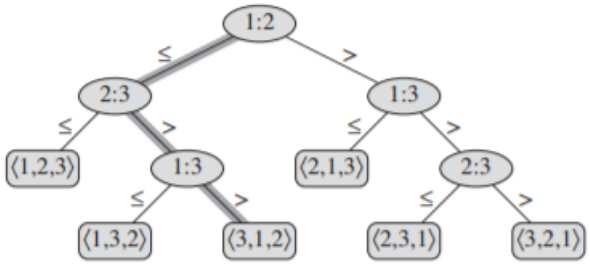
- L'idea si basa sull'ordinare i numeri decimali per cifre, partentendo dalla meno alla più significativa.
- In questo caso conviene utilizzare il `CountingSort()` come algoritmo di ordinamento stabile da che `k` $\in \{0, \ldots, 9\}$ (ogni cifra è sempre compresa tra 0 e 9).

```
function RadixSort(A, d) {
    for(let i = 0; i < d; i++){
        StableSort(A) //sulla cifra 'i', in questo caso usiamo il
    Counting Sort
    }
}
```

code/radix.js

1.9 Ordinamento per Confronti - Lower Bound

- Dati n elementi, il loro ordine corretto di trova in una delle loro $n!$ permutazioni.
- Gli ordinamenti per confronti possono essere visti come degli alberi di decisioni, ovvero alberi binari pieni dove ogni nodo contiene una coppia di numeri e a seconda di chi è più grande dell'altro si procede verso destra o verso sinistra.



- Nell'albero ogni foglia rappresenta una permutazione degli n elementi.
- La lunghezza del cammino dalla radice fino alla permutazione che rappresenta l'ordine corretto della sequenza, rappresenta il numero di confronti effettuati dall'algoritmo.
- Nel caso peggiore il numero di confronti, e quindi la lunghezza del cammino, è uguale all'altezza h dell'albero binario.
- Quindi considerando un albero di decisione di altezza h con l foglie.
- Poichè ciascuna delle $n!$ deve comparire in una foglia si ha che $n! \leq l$.
- Dato che un un albero di altezza h non ha più di 2^h foglie vale la seguente disuguaglianza: $n! \leq l \leq 2^h$.
- Applicando la funzione logaritmica a tutti i termini abbiamo che $\log n! \leq \log l \leq h$.
- Quindi abbiamo che $h \geq \log n!$.
- Per la formula di Stirling $h = \Omega(n \log n)$.

■

2 Dimostrazione Master Theorem

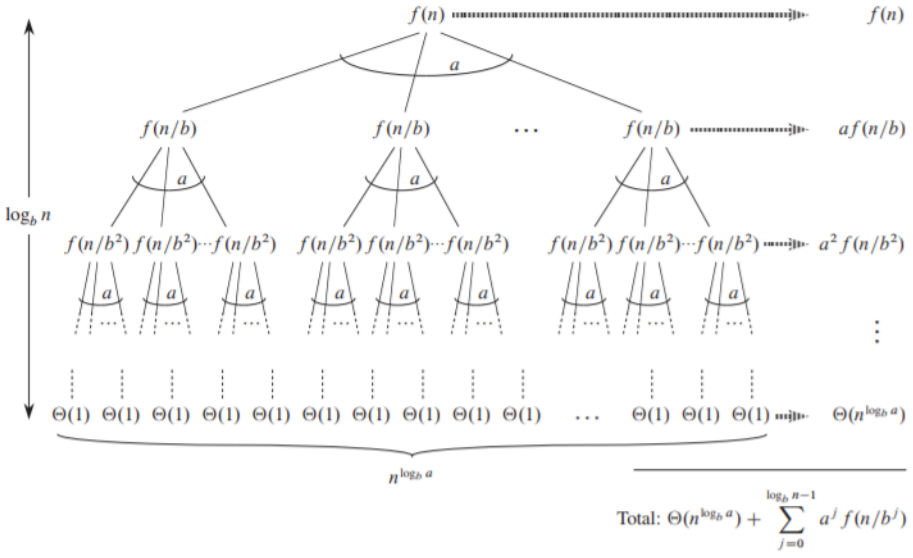
Ipotesi: n è una potenza esatta di b.

$$T(n) = aT(n/b) + f(n) \tag{1}$$

Data l'equazione di ricorrenza descritta in precedenza, possiamo dire che:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \tag{2}$$

Per dimostrare questa uguaglianza occorre utilizzare l'albero di ricorsione associato all'equazione presa in considerazione.



- Quindi la radice dell'albero ha costo $f(n)$, e ci sono ogni volta a chiamate ricorsive (a sottoproblemi, ognuno dal costo di $f(n/b)$).
- Generalizzando possiamo dire che ogni livello ha costo $a^j f(n/b^j)$.
- L'altezza dell'albero è ovviamente $\log_b n$, e il numero di foglie sarà uguale a $a^{\log_b n}$.
- Utilizzando le proprietà dei logaritmi abbiamo che $a^{\log_b n} = n^{\log_b a}$.
- Quindi se vogliamo calcolare il costo complessivo dell'albero, possiamo sommare il costo delle foglie che hanno tutte costo $\Theta(1)$. Quindi $\Theta(1) \cdot \Theta(n^{\log_b a}) = \Theta(n^{\log_b a})$.

- E a questo valore possiamo sommare il costo di ogni livello, a partire dalla radice ($j = 0$) fino al penultimo livello dell'albero ($\log_b n - 1$).

Adesso occorre dimostrare i 3 casi del Master Theorem.

2.1 Caso 1

Dimostrare che:

$$f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) = O(n^{\log_b a})$$

- Quindi se sappiamo che $f(n) = O(n^{\log_b a - \epsilon})$, allora $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$.
- $g(n)$ quindi diventa: $g(n) = O(\sum_{j=0}^{\log_b n - 1} a^j (n/b^j)^{\log_b a - \epsilon})$.
- Adesso posso tirare tutto ciò che non dipende da j fuori dalla sommatoria: $g(n) = \sum_{j=0}^{\log_b n - 1} a^j (\frac{n}{b^j})^{\log_b a - \epsilon} = n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} a^j (\frac{1}{b^j})^{\log_b a - \epsilon}$.
- Ora utilizzando le proprietà delle potenze possiamo portare j fuori dalle parentesi e portiamo dentro l'esponente che si trova fuori dalle parentesi e lo spezziamo, dividendo il logaritmo con ϵ: $g(n) = n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} a^j (\frac{b^\epsilon}{b^{\log_b a}})^j$.
- Adesso possiamo semplificare il denominatore all'interno della sommatoria: $g(n) = n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} a^j (\frac{b^{\epsilon \cdot j}}{a^j})$.
- A questo punto semplifichiamo a^j: $g(n) = n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j$.
- Per semplificare la sommatoria possiamo utilizzare la seguente serie geometrica: $\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$.
- Quindi: $g(n) = n^{\log_b a - \epsilon} (\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1})$.
- Ora di nuovo applicando le proprietà dei logaritmi abbiamo che: $g(n) = n^{\log_b a - \epsilon} (\frac{n^\epsilon - 1}{b^\epsilon - 1})$.
- Adesso dato che b ed ϵ sono costanti possiamo riscrivere l'equazione in questo modo: $g(n) = n^{\log_b a - \epsilon}\, O(n^\epsilon)$.

- Come ultimo passaggio possiamo portare tutto dentro l'O-grande ed eliminare ϵ: $g(n) = O(n^{\log_b a - \epsilon + \epsilon}) = O(n^{\log_b a})$.
- Quindi $g(n) = O(n^{\log_b a})$.

Adesso effettuando le opportune sostituzioni abbiamo che:

$$T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$$

2.2 Caso 2

Dimostrare che:

$$f(n) = \Theta(n^{\log_b a}) \Rightarrow g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) = \Theta(n^{\log_b a} \lg n)$$

- Se sappiamo che $f(n) = \Theta(n^{\log_b a})$, allora $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$.
- $g(n)$ quindi diventa: $g(n) = \Theta(\sum_{j=0}^{\log_b n - 1} a^j (\frac{n}{b^j})^{\log_b a})$.
- Ora portiamo fuori ciò che non dipende da j: $g(n) = n^{\log_b a} \sum_{j=0}^{\log_b n - 1} (\frac{a}{b^{\log_b a}})^j$.
- Adesso applicando le proprietà delle potenze abbiamo che: $g(n) = n^{\log_b a} \sum_{j=0}^{\log_b n - 1} 1$.
- Quindi calcolando la serie abbiamo che: $g(n) = \Theta(n^{\log_b a} \log_b n)$.

Adesso effettuando le opportune sostituzioni abbiamo che:

$$T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_b a} \lg n)$$

2.3 Caso 3

Dimostrare che:

$$af(n/b) \leq cf(n) \Rightarrow g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) = \Theta(f(n))$$

- Riscriviamo $f(n)$ dividendo la disequazione per a: $f(n/b) \leq (c/a) f(n)$.
- Successivamente iteriamo la disequazione per j volte: $f(n/b^j) \leq (c/a)^j f(n)$.
- Adesso moltiplichiamo per a^j: $a^j f(n/b^j) \leq c^j f(n)$.

- A questo punto possiamo effettuare il passaggio alla serie: $g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \leq \sum_{j=0}^{\log_b n-1} c^j f(n) + O(1)$.
- Possiamo riscrivere la disequazione in questo modo: $g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \leq \sum_{j=0}^{\infty} c^j f(n) + O(1)$.
- Adesso è possibile calcolare il valore della serie: $g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \leq f(n)(\frac{1}{1-c}) + O(1)$.
- Dato che c è una costante: $g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) = O(f(n))$.

Adesso effettuando le opportune sostituzioni abbiamo che:

$$T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$$

Questo perchè nelle ipotesi del Caso 3 abbiamo che $f(n) = \Omega(n^{\log_b a+\epsilon})$.

3 Algoritmi di Ricerca

3.1 Ricerca Lineare

```
1 function Ricerca_Lineare(A, k) {
2     for(let i = 0; i < A.lenngth; i++){
3         if(A[i] == k) return i;
4     }
5     return -1;
6 }
```

code/linear_search.js

3.2 Ricerca Binaria

```
1 function Ricerca_Binaria(A, k, sx, dx) {
2     if(sx > dx) return -1;
3     c = (sx + dx) / 2;
4     if(A[c] == k) return c;
5     if(k < A[c]) Ricerca_Binaria(A, k, sx, c - 1);
```

```
6       else Ricerca_Binaria(A, k, c + 1, dx);
7   }
```

code/binary_search.js

4 Tabelle Hash

4.1 Gestione delle Collisioni

	Operazione		
Gestione	Inserimento	Ricerca con successo	Ricerca senza successo
Liste di Trabocco	$O(1)$	$\Theta(1+\alpha)$	$\Theta(1+(1+\frac{\alpha}{2}-\frac{\alpha}{2n}))$
Open Hash	$T_{ottimo} = \Theta(1)$ $T_{pessimo} = \Theta(n) = \Theta(m)$ $T_{medio} = O(\frac{1}{1-\alpha})$	$O(\frac{1}{\alpha}\ln(\frac{1}{1-\alpha}))$	$O(\frac{1}{1-\alpha})$

4.2 Liste di Trabocco

Teorema 4.1 (Ricerca senza Successo - Caso Medio)**.** *In una tabella hash con concatenamento la ricerca senza successo richiede al caso medio* $\Theta(1+\alpha)$*, dove* $\alpha = \frac{n}{m}$*.*

Dimostrazione.

- Definiamo $\alpha = \frac{|S|}{dim\ T} = \frac{n}{m}$ come il fattore di carico.
- Data la chiave `k`, effettuo l'hashing `h(k)` e verifico se in testa alla lista `T[h(k)]` è presente la chiave cercata.
- Se in testa non è presente, occerrerà scorrere tutta la lista.
- Se gli `n` elementi sono distribuiti uniformemente sulle liste, ogni lista conterrà α elementi.
- Quindi $T_{medio}(n,m) = \Theta(1+\alpha)$, dove l'1 è dovuto al calcolo di `h(k)` e α al numero di ispezioni nella lista.
- Se α è costante allora: $T_{medio}(n,m) = \Theta(1)$.

■

Teorema 4.2 (Ricerca con Successo - Caso Medio)**.** *In una tabella hash con concatenamento la ricerca con successo richiede al caso medio* $\Theta(1+(1+\frac{\alpha}{2}-\frac{\alpha}{2n})) = \Theta(1+\alpha)$*, dove* $\alpha = \frac{n}{m}$*.*

Dimostrazione.

- Il numero di ispezioni per trovare `k` è dovuto dal numero di elementi che precedono `k` nella lista `T[h(k)]`. Essendoci l'inserimento in testa, questi sono gli elementi inseriti dopo `k`.
- Se `x` è l'i-esimo elemento, quelli inseriti dopo sono `n - i` elementi.
- Gli elementi, invece, che avranno lo stesso valore hash `h(k)` saranno in media $\lfloor \frac{n-i}{m} \rfloor$.
- Quindi gli elementi ispezionati durante la ricerca di `i` sono $1 + \frac{n-i}{m}$.
- A questo punto occorre fare una media su tutte le posizioni che può assumere `i` nella lista; quindi supponiamo che stiamo cercando uno degli elementi con la stessa probabilità $1/n$.
- Quindi il numero di ispezioni al caso medio è uguale a:

$$\frac{1}{n}\sum_{i=1}^{n}(1+\frac{n-i}{m}) = \frac{1}{n}\sum_{i=1}^{n}1 + \frac{1}{n}\sum_{i=1}^{n}\frac{n-i}{m} =$$

- La prima sommatoria è uguale a n e possiamo portare fuori dalla seconda sommatoria m:

$$= \frac{n}{n} + \frac{1}{n \cdot m}\sum_{i=1}^{n}(n-i) =$$

- Adesso possiamo notare che la sommatoria rimasta corrisponde alla somma dei primi n numeri interi (formula di Gauss), quindi:

$$= 1 + \frac{1}{n \cdot m} \cdot \frac{n(n-1)}{2} = 1 + \frac{n-1}{2m} = 1 + \frac{n}{2m} - \frac{1}{2m} =$$

- Dato che $\alpha = \frac{n}{m}$ e $m = \frac{n}{\alpha}$:

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

■

4.3 Open Hash

Ipotesi:

1. $\alpha = \frac{n}{m} < 1$.
2. Non sono previste cancellazioni.

3. Deve valere l'ipotesi di Hashing Uniforme (la sequenza di ispezione deve essere una permutazione generata con pari probabilità).

Teorema 4.3 (Ricerca senza Successo - Caso Medio)**.** *In una tabella hash a indirizzamento aperto, la ricerca senza successo effettua, al caso medio, un numero di accessi* $\leq \frac{1}{1-\alpha}$.

Dimostrazione.

- Poniamo `X` come il numero di accessi alla tabella per effettuare la nostra ricerca.
- Il valore medio di `X` è uguale alla somma di tutti i valori che può assumere, moltiplicato per la probabilità che `X` assuma quel valore:

$$\sum_{i=1}^{\infty} i \cdot Prob[x = i] = \sum_{i=1}^{\infty} Prob[x \geq i]$$

- Nel passaggio precedente la serie è stata semplificata, e a questo punto ci ritroviamo con la probabilità che `X` effettui almeno `i` accessi:
  - Se $i = 1$, $Prob[x \geq 1] = 1$.
  - Se $i = 2$, $Prob[x \geq 2] = \alpha$. Essenzialmente è la probabilità di trovare la prima cella già occupata da una chiave diversa ma con lo stesso `h(k)`.
  - Se $i = 3$, $Prob[x \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$. Quindi la probabilità di avere sia la prima che la seconda cella già occupata).
- Quindi generalizzando:

$$\sum_{i=1}^{m} Prob[x \geq i] = \sum_{i=1}^{m} \alpha^{i-1} = \sum_{i=0}^{m-1} \alpha^i \leq \sum_{i=1}^{\infty} \alpha^i =$$

- L'ultima sommatoria è una serie geometrica, quindi essendo $\alpha \leq 1$:

$$= \frac{1}{1-\alpha}$$

■

Teorema 4.4 (Ricerca con Successo - Caso Medio)**.** *In una tabella hash a indirizzamento aperto, la ricerca con successo effettua, al caso medio, un numero di accessi* $\leq \frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha})$.

Dimostrazione.

- Chiamiamo `k` la chiave che stiamo cercando; `k` è l'i-esimo elemento inserito nella tabella.
- Se chiamiamo α_i il fattore di carico nella tabella prima dell'inserimento di `k`:
$$\alpha_i = \frac{i}{m}$$
- Quindi il numero di accessi fatti per inserire `k` è uguale al numero di accessi per una ricerca senza successo:
$$\leq \frac{1}{1-\alpha_i} = \frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$$
- Quindi il valore medio del numero di accessi è:
$$\frac{1}{n}\sum_{i=0}^{n-1}\alpha_i = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} =$$
- Adesso per calcolare il valore della serie possiamo applicare il Criterio dell'Integrale:
$$= \frac{1}{\alpha}\sum_{i=0}^{n-1}\frac{1}{m-i} \leq \frac{1}{\alpha}\cdot\int_{m-n}^{m}\frac{1}{x}\,dx = \frac{1}{\alpha}\cdot\ln(\frac{m}{m-n}) =$$
- A questo punto occorre solo dividere il numeratore e il denominatore del logaritmo per m:
$$= \frac{1}{\alpha}\cdot\ln(\frac{1}{1-\alpha})$$

■

5 2-3 Alberi

Lemma 5.1. *Dato un 2-3 Albero alto h, con n nodi e con f foglie, vale che:*

$$2^{h+1} - 1 \leq n \leq (3^{h+1} - 1)/2$$

$$2^h \leq f \leq 3^h$$

Dimostrazione. Poniamo T come un albero alto $h+1$, e T' come l'albero alto h ottenuto eliminando da T tutte le foglie.
Adesso dimostriamo la seguente Ipotesi Induttiva:

$$2^{h+1} - 1 \leq n' \leq (3^{h+1} - 1)/2$$

$$2^h \leq f' \leq 3^h$$

Dato che ogni foglia in T' ha o 2 o 3 figli in T risulta che:

$$2 \cdot 2^h \leq f \leq 3 \cdot 3^h$$

$$2^{h+1} \leq f \leq 3^{h+1}$$

Come ultimo passo sappiamo che il numero di nodi in T è uguali al numero di nodi in T' più il numero di foglie in T, quindi:

$$2^{h+1} - 1 + 2^{h+1} \leq n' + f \leq (3^{h+1} - 1)/2 + 3^{h+1}$$

$$2^h(2+2) - 1 \leq n \leq (3^{h+1} - 1 + 2 \cdot 3^{h+1})/2$$

$$2^{h+2} - 1 \leq n \leq [3^h(3 + 2 \cdot 3) - 1]/2$$

$$2^{h+2} - 1 \leq n \leq (3^{h+2} - 1)/2$$

■

6 Programmazione Dinamica

6.1 LCS

Teorema 6.1 (Sottostruttura ottima delle LCS)**.** *Date due stringhe $X = x_1, \cdots, x_m$ e $Y = y_1, \cdots, y_n$ e una stringa $Z = z_1, \cdots, z_k$ tale che $Z = LCS(X, Y)$:*

1. *$x_m = y_m \Rightarrow z_k = x_m = y_n$ e Z_{k-1} è $LCS(X_{m-1}, Y_{n-1})$.*
2. *$x_m \neq y_m \Rightarrow z_k \neq x_m$ e Z è $LCS(X_{m-1}, Y_n)$.*
3. *$x_m \neq y_m \Rightarrow z_k \neq y_n$ e Z è $LCS(X_m, Y_{n-1})$.*

Dimostrazione.

1. Se per assurdo $z_k \neq x_m$ ma $x_m = y_n$ allora possiamo accodare x_m a Z per ottenere una sottosequenza comune di X e Y lunga $k+1$ contraddicendo l'ipotesi che Z sia una LCS di X e Y.
   Ora dimostriamo che $Z_{k-1} = LCS(x_{m-1}, y_{n-1})$ lunga $k-1$. Supponiamo che esista una stringa chiamata W che è una sottosequenza comune di X_{m-1} e Y_{n-1} di lunghezza maggiore di $k-1$. Allora accodando $x_m = y_n$ a W si otterrebbe una sottosequenza di lunghezza maggiore di k, ma questo contraddice l'ipotesi $Z = LCS(X, Y)$.

2. Se esistesse una stringa W sottosequenza comune di X_{m-1} e Y di lunghezza maggiore di k, allora W sarebbe anche una sottosequenza comune di X e Y (questo perchè $x_m \neq y_n$), ma questo contraddice l'ipotesi $Z = LCS(X, Y)$.

3. La dimostrazione è simmetrica al punto 2.

■

7 Grafi

7.1 Dimostrazione Calcolo Cammino Minimo BFS

Lemma 7.1. *Dato un grafo $G = (V, E)$ e una sorgente $s \in V$, per ogni arco $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u)+1$, questo perchè la distanza tra s e v è sicuramente minore o uguale a qualsiasi altro cammino da s a v; in questo caso un cammino che da s va in u e tramite l'arco (u, v) arriva in v.*

Lemma 7.2. *Al termine dell'algoritmo* `BFS(G, s)`*, per ogni vertice $v \in V$, $v.d \geq \delta(s, v)$.*

Dimostrazione. Poniamo la seguente Ipotesi Induttiva: $v.d \geq \delta(s, v)$.

1) **Caso Base**
   $s.d = 0 = \delta(s, s)$
   $v.d = \infty \geq \delta(s, v)$ per ogni $v \in V \backslash \{s\}$
   Queste uguaglianze ovviamente sono vere dopo la `Enqueue(Q, s)` nella `BFS(G, s)`.

2) **Passo Induttivo**
   Se v è un vertice bianco scoperto da u, dimostriamo che l'ipotesi Induttiva è sempre valida:

   $v.d = u.d + 1$ (Questo è l'assegnamento che esegue l'algoritmo).
   $\geq \delta(s, u) + 1$ (per Ipotesi Induttiva).
   $\geq \delta(s, v)$ (Per il Lemma 7.1).

   A questo punto questa disuguaglianza è sempre verificata dato che v.d non cambia più perchè il vertice diventa grigio.

■

Lemma 7.3. *Durante la BFS, se $Q = [v_1, v_2, \ldots, v_r]$, allora:*

1. *$v_r.d \leq v_1.d + 1$ (la differenza tra v_1 e v_r è 1).*
2. *$v_i.d \leq v_{i+1}.d \; \forall \; i \in \{1, 2, \ldots, r-1\}$.*

Questo significa che, in ogni istante, nella coda ci sono al più 2 valori diversi della distanza dalla sorgente, e i campi distanza formano una successione crescente.

Corollario 7.1. *I valori delle distanze dalla sorgente, dei vertici inseriti nella coda, sono monotoni crescenti, quindi se v_i è inserito nella coda prima di v_j, allora $v_i.d \leq v_j.d$.*

Teorema 7.1. *La BFS scopre tutti i vertici $v \in V$ raggiungibili dalla sorgente s e alla fine dell'algoritmo, $v.d = \delta(s, v)$ per ogni $v \in V$.*
Inoltre per ogni $v \neq s$, raggiungibile da s, uno dei cammini minimi da s a v è un cammino minimo da s a $v.\pi$ seguito dall'arco $(v.\pi, v)$.

Dimostrazione. Supponiamo che ci sia un vertice v che è il nodo più vicino alla sorgente che ha il campo "d" diverso dalla sua distanza dalla sorgente.

Per il Lemma 7.2, $v.d \geq \delta(s,v)$, e dato che abbiamo appena posto che v.d deve essere diverso dalla distanza, allora $v.d > \delta(s,v)$.
Inoltre v deve essere per forza raggiungibile dalla sorgente, altrimenti $\delta(s,v) = \infty \geq v.d$.

Poi, sia u il nodo che precede v in un cammino minimo da s a v, quindi:

- $\delta(s,v) = \delta(s,u) + 1$.
- $\delta(s,u) \leq \delta(s,v)$ e $u.d = \delta(s,u)$. Quest'ultima uguaglianza è vera perchè abbiamo posto che v è il nodo più vicino alla sorgente con il campo v.d errato, quindi il campo u.d sarà corretto perchè u si trova prima di v.

Dunque mettendo insieme i pezzi possiamo dire che: $v.d > \delta(s,v) = \delta(s,u)+1 = u.d + 1$ e quindi $\underline{v.d > u.d + 1}$.
Quando u viene estratto dalla coda, v può essere di colore bianco, grigio o nero:

1. Se v è bianco, allora l'algoritmo assegna $\underline{v.d = u.d + 1}$. ↯
2. Se v è nero, allora v era stato già rimosso dalla coda e per il Corollario 7.1 $\underline{v.d \leq u.d}$. ↯
3. Se v è grigio, allora v è stato scoperto da un vertice w estrato dalla coda Q prima di u e quindi v.d = w.d + 1.
   Per il Corollario 7.1 $w.d \leq u.d$, quindi $v.d = w.d+1 \leq u.d+1$, $\underline{v.d \leq u.d + 1}$.
   ↯

$$\text{Quindi } v.d = \delta(s,v).$$

Infine, tutti i vertici raggiungibili da s devono essere scoperti, altrimenti $\infty = v.d > \delta(s,v)$. ■

7.2 DFS - Visita in Profondità

7.2.1 Proprietà Foresta DF

1. u è il padre di v in un albero della foresta DF ⇔ v è stato scoperto esaminando la lista di adiacenza di u.
2. v è discendente di u nella foresta DF ⇔ v è stato scoperto quando u era Grigio.

7.3 Teoremi Principali

Teorema 7.2 (Teorema delle Parentesi)**.** *Dati gli intervalli $I_u = [u.d, u.f]$ e $I_v = [v.d, v.f]$, $\forall\, u, v \in V$ è soddisfatta una sola delle seguenti tre condizioni:*

1. $I_u \cap I_v = \varnothing \Rightarrow$ *u e v non sono discendenti uno dell'altro.*
2. $I_v \subset I_u \Rightarrow$ *v è discendente di u.*
3. $I_u \subset I_v \Rightarrow$ *u è discendente di v.*

Dimostrazione. Per ipotesi poniamo $u.d < v.d$. Ci possono essere 2 casi:

1. $u.f < v.d \Rightarrow u.d < u.f < v.d < v.f \Rightarrow I_u \cap I_v = \varnothing$.
2. $u.f > v.d \Rightarrow$ v è stato scoperto quando quando u era Grigio, quindi v è un discendente di u $\Rightarrow$ la visita di v termina prima di quella di u $\Rightarrow I_v \subset I_u$.

La dimostrazione è speculare nel caso $v.d < u.d$. ■

Corollario 7.2 (Corollario di Annidamento degli Intervalli)**.** *v è discendente di u nella foresta DF* ($u \neq v$) $\Leftrightarrow I_v \subset I_u$.

Teorema 7.3 (Teorema del Cammino Bianco)**.** *v è un discendente di u nella foresta DF* $\Leftrightarrow$ *al tempo u.d, v può essere raggiunto da u lungo un cammino di soli vertici Bianchi.*

Dimostrazione.

$\Rightarrow$

- $v = u$ In questo caso il cammino $u \rightsquigarrow v$ contiene solo il nodo u che è Bianco al tempo u.d.
- $v \neq u$ v è un discendente diretto di u. Per il Corollario 7.2 se $I_v \subset I_u$ allora $u.d < v.d$. Quindi v è scoperto dopo u, e per questo v è Bianco all'istante u.d. Se v non è un discendente diretto di u, applicando in modo induttivo il ragionamento precedente su tutti i vertici lungo l'unico cammino nella foresta DF da u a v, essi saranno tutti Bianchi al tempo u.d.

$\Leftarrow$ Per assurdo diciamo che esiste un cammino Bianco $u \rightsquigarrow v$ al tempo u.d, ma che v non è discendente di u. Scegliamo v come il vertice più vicino a u che non è discendente di u. Inoltre scegliamo w come il vertice che precede direttamente v sul cammino. w è discendente di u, quindi $I_w \subseteq I_u$ e $w.f \leq u.f$ (w e u possono essere lo stesso vertice). Inoltre sappiamo che v è Bianco al tempo u.d. Quindi:
$u.d < v.d < w.f \leq u.f$. Il primo '$<$' è vero perchè v è Bianco al tempo u.d. Il secondo '$<$', invece, è vero perchè $v \in Adj[w]$, quindi la visita di w è ancora in corso quando v viene scoperto. Il terzo '$\leq$' è vero perchè w è discendente di u.
Per il Teorema delle Parentesi, dato che $v.f < u.f$, allora $I_v \subset I_u$ e v è discendente di u. ↯

■

7.4 Archi

Teorema 7.4. *In una DFS su un grafo non orientato, gli archi sono solo archi d'albero e archi all'indietro.*

Dimostrazione. $(u, v) \in E$. Supponiamo che sia stato scoperto prima u, allora $u.d < v.d$. Quindi v diventa Grigio e successivamente Nero, invece u è Grigio. Ci sono 2 casi:

1. (u, v) è esplorato la prima volta da u verso v, allora v è Bianco e (u, v) diventa arco d'albero.
2. (u, v) è esplorato la prima volta da v verso u, allora u è Grigio e (u, v) è un arco all'indietro.

■

Teorema 7.5. *Un grafo G è ciclico ⇔ G contiene almeno un arco all'indietro.*

Dimostrazione.

⇐ (u, v) è un arco all'indietro di un grafo orientato o non orientato e v è un antenato di u. Il cammino $v \rightsquigarrow u$ in un albero DF, unito all'arco (u, v) forma un ciclo.

⇒ Se il grafo non è orientato, gli archi di un ciclo non possono tutti essere d'albero, quindi ci dev'essere almeno un arco all'indietro.
Se il grafo è orientato, poniamo v come il primo vertice di un ciclo ad essere scoperto e che diventa Grigio, allora quando si scopre v, gli altri nodi sono Bianchi. Poniamo u, invece, come il nodo che precede v nel ciclo. Al tempo v.d tutti i vertici sul cammino $v \rightsquigarrow u$ sono Bianchi.
Per il Teorema del Cammino Bianco, u diventa un discendente di v, quindi v è un antenato di u e (u, v) è un arco all'indietro.

■

7.5 Ordinamento Topologico

Teorema 7.6. $\forall\ (u, v) \in E$, *se u precede v nell'ordinamento, allora u deve precedere v nella lista; quindi u deve essere inserito in lista dopo v. Affinchè questo avvenga,* $u.f > v.f$.

Dimostrazione. Quando si ispeziona l'arco (u, v), (u è Grigio), ci sono 3 casi:

1. Se v è Bianco, allora (u, v) è un arco d'albero, quindi v è discendente di u e $v.f < u.f$.
2. Se v è Nero, allora la visita di v è già finita, mentre quella di u è ancora in corso, quindi $v.f < u.f$.
3. Se v è Grigio, allora G contiene un ciclo, ma questo è impossibile perchè G deve essere un DAG.

■