Algoritmi e Strutture dei Dati

Pietro Grassi

2024

Indice

1 Problemi introduttivi

1.1 Segmento massimo

Si consideri un array n-dimensionale. Si scriva un algoritmo per determinare il segmento di somma massima.

Sia $A[i \cdots j] = A[i] \cdots A[j]$, si consideri $\text{somma}(i,j) = \sum_{k=i}^{j} A[k]$, trovare $\max_{0 \le i \le j < n} \text{somma}(i,j)$.

Segue una prima baseline della funzione SommaMassima.

```
SommaMassima1(a)
        int max = 0;
        for(i = 0; i < n; i++){
                for(j = i; j < n; j++){
                        int somma = 0;
                        for(k = i; k <= j; k++)
                                somma = somma+a[k];
                        if(somma > max)
                                max = somma;
                }
        }
        return max;
```

Essa è $O(n^3)$, lenta poiché contiene 3 cicli **for** annidiati.

Si noti che $\text{somma}(i, j+1) = \text{somma}(i,j) + A[j+1]$ e si sfrutti ciò per la seguente miglioria.

```
SommaMassima2(a)
        int max = 0;
        for(i = 0; i < n; i++){
                int somma = 0;
                for(j = i; j < n; j++){
                        somma = somma+a[j];
                }
                if(somma > max)
                        max = somma;
        }
return max;
```

Essa è $O(n^2)$.

1.1.1 Segmento massimale

Esso è un segmento che non si può estendere o restringere ottenendo somme più alte, cioè $A[i,j]$ massimale se $\nexists\ [i',j'] \supseteq [i,j] \ \vee\ [i',j'] \subseteq [i,j] : \text{somma}(i',j') >$

somma(i, j). Se un segmento è massimale, allora è massimo, ma il viceversa non vale.

I segmenti massimali sono disgiunti in A, cioè $A[i, j]$, $A[i', j']$ massimali, allora $[i, j] \cap [i', j'] = \varnothing$.

Sfruttando questa nozione si costruisca l'algoritmo lineare seguente.

```
SommaMassima3(a)
        max = 0;
        somma = max;
        for(j = 0; j < n; j++){
                if(somma > 0){
                        somma = somma+a[j];
                } else {
                        somma = a[j];
                }
                if(somma > max)
                        max = somma;
        }
        return max;
```

Essa è corretta ed efficiente.

1.2 2-sum

Sia A un array ordinato. Dato un valore v, stabilire se $\exists\ i < j : A[i] + A[j] = v$. Questo problema è un'istanza del difficile 3-sum che chiede se $\exists\ i, j, k : A[i] + A[j] = A[k]$, con A non ordinato, di cui non si conosce un algoritmo subquadratico.

2 Analisi del costo computazionale

2.1 Criteri per la progettazione di algoritmi

1. Correttezza;
2. Velocità ed efficienza.

2.1.1 Modello di calcolo semplificato di von Neumann

Il modello utilizzato è il Random Access Model (RAM) con accesso sequenziale a 64-bit, cioè $2^64 - 1$ celle ordinate su cui opera il Central Processing Unit (CPU), tramite:

- Operazioni di trasferimento input/output (I/O);
- Operazioni aritmetiche elementari $(+, -, *, /, \ldots)$;

- Operazioni logiche ($\&\&, \&, |, ||, \ldots$);
- Operazioni di controllo, cioè il *jump*, condizionato o meno.

Tutte queste operazioni hanno costo computazionale uniforme rispetto al numero di dati.

Il programma è in tale modello una sequenza di bit indistinguibile dai dati.

2.2 Costo computazionale

Non potendosi basare sui tempi, che dipendono dall'architettura del compilatore e della macchina, dal linguaggio e da molteplici altri fattori, si confrontano gli algoritmi sul numero di operazioni. Questo valore è variabile e dipendente dall'input, per cui si ricorre al plot dei tempi di esecuzione, intesi – appunto – come numero di operazioni elementari.

Il confronto si esegue poi analizzando il caso medio (la convergenza della serie è assicurata dal fatto che i tempi sono finiti) o il caso pessimo (l'esistenza del minimo è garantita dalla finitezza dei tempi).

La complessità asintotica è relativa al $\lim_{n\to+\infty}$ #operazioni e spesso si studia trovando due funzioni che per teorema del confronto diano informazioni sulla complessità asintotica stessa dell'algoritmo.

Il costo computazionale tiene dunque conto del tempo di esecuzione in termini del numero di operazioni e – secondariamente – dello spazio occupato, oltre che dell'energia consumata. L'analisi è asintotica: il costo è funzione del numero dei dati e della dimensione dell'input, ma le costanti moltiplicative e gli ordini inferiori vengono ignorati. Il confronto tra algoritmi è pertanto quantitativo:

- Si dice che $f(n) = O(g(n))$ se $\exists\ n_0, c > 0 : n \geq n_0 \Rightarrow f(n) \leq cg(n)$;
- Si dice che $f(n) = \Omega(g(n))$ se $\exists\ n_0, c > 0 : n \geq n_0 \Rightarrow f(n) \geq cg(n)$;
- Si dice che $f(n) = \Theta(g(n))$ se $f(n) = O(g(n)) \ \wedge \ f(n) = \Omega(g(n))$.

2.2.1 Caso pessimo del costo computazionale

- Operazioni elementari del modello RAM: $O(1)$;
- Costrutti computazionali:
  - Condizionali:

```
* if(guardia){
            blocco then;
  } else {
            blocco else;
  }
```

La guardia viene sempre valutata, mentre solo uno dei due blocchi è eseguito, per cui

$$\text{costo(if)} = \text{costo(guardia)} + \max(\text{costo(then)}, \text{costo(else)}). \quad (1)$$

- Iterativi:

  *
```
for(i = 0; i < m; i++){
        corpo(i);
}
```

  Sia $t_i = \text{costo(corpo}(i))$, vale

$$\text{costo(for)} = O\left(m + \sum_{i=0}^{m-1} t_i\right), \quad (2)$$

  dove m è legato all'esecuzione, m volte appunto, di

```
(i = 0; i < m; i++)
```

```
while(guardia(i)){
        corpo(i);
}
```

  con $0 \leq i \leq m$, dove $m = \#$volte in cui la guardia è vera, per cui si eseguono $m+1$ valutazioni (le prime m vere e l'$(m+1)$-esima falsa, che interrompe l'iterazione). Pertanto, siano $t_i = \text{costo(corpo}(i))$, $t'_i = \text{costo(guardia}(i))$, si ha

$$\text{costo(while)} = O\left(\sum_{i=0}^{m} t'_i + \sum_{i=0}^{m-1} t_i\right). \quad (3)$$

- Chiamata a funzione non ricorsiva: il costo di una funzione è definito come il costo del suo corpo.

2.3 Complessità dei problemi computazionali

Si consideri un problema computazionale $\pi : IN \to OUT$, con $n = \#$bit di π.

2.3.1 Limite superiore

Si dice che π ha limite superiore $O(f(n))$ se esiste un algoritmo che risolve π la cui complessità è $O(f(n))$.

2.3.2 Limite inferiore

Si dice che π ha limite inferiore $\Omega(f(n))$ se ogni possibile algoritmo che risolve π ha complessità $\Omega(f(n))$.

Determinare il limite inferiore richiede un'analisi rigorosa del problema.

2.3.3 Complessità di un problema

Si dice che π ha complessità $\Theta(f(n))$ se π ha limite superiore e limite inferiore coincidenti e pari a $O(f(n)) = \Omega(f(n))$.

2.3.4 Fooling argument

Se non vi sono ipotesi a priori sull'input, è necessario che questo sia letto interamente, per cui il limite inferiore del problema è di almeno $\Omega(n)$.

3 Problema dell'ordinamento

Si consideri il problema seguente:

- IN: array A di n elementi sui quali vale una relazione d'ordine;
- OUT: permutazione degli elementi di A tale che $A[0] < A[1] < \cdots < A[n-1]$;
- Gli elementi di A possono essere solo confrontati (a due a due) o copiati, senza coinvolgerli in altre operazioni.

Lo studio della complessità del problema dell'ordinamento produce i seguenti risultati:

(i) Il limite inferiore è $\Omega(n \log n)$;

(ii) Il limite superiore è $O(n \log n)$.

Pertanto, il problema dell'ordinamento ha complessità $\Theta(n \log n)$.

Dimostrazione

(i) Considerati due elementi $A[i], A[j]$, l'esito del loro confronto è uno tra $<, >, =$. Si supponga ora di disporre di una black box collegata a un led, che si accende ogni volta che si verifica un confronto, a prescindere dall'esito. Nella tabella 3 si riportano i migliori risultati deducibili da t accensioni del led.

t	massimo numero di ordinamenti distinguibili con t confronti
0	1
1	3
2	3^2
3	3^3
$\vdots$	$\vdots$
t	3^t

Tabella 1: A ogni confronto corrispondono tre possibili esiti: $<, >, =$

Due input di n elementi sono equivalenti per l'algoritmo $\mathcal{A}$ se inducono la medesima sequenza di confronti, per cui corrispondono alla stessa permutazione di S_n, in quanto $\mathcal{A}$ utilizza il solo ordine relativo degli elementi dell'input. Pertanto, $\mathcal{A}$ deve distinguere $n! = \#S_n$ permutazioni e, detto t il numero di confronti al termine dell'esecuzione di $\mathcal{A}$, allora $3^t \geq n!$, per cui $t \geq n \log n$.

(ii) La dimostrazione di questo fatto si basa sugli algoritmi di questo capitolo.

3.1 Selection Sort

Sia $\mathcal{S}$:

```
void selectionSort(vector<int> &A) {
        int n = A.size();
        for (int i=0; i < n-1; i++){
                int min = A[i];
                int indexmin = i;
                for (int j=i+1; j < n; j++){
                        if (A[j] < min){
                                min = A[j];
                                indexmin = j;
                        }
                }
                A[indexmin] = A[i];
                A[i] = min;
        }
}
```

Si ha costo($\mathcal{S}$) $= O\left(n + \sum_{i=0}^{n-1} t_i\right)$, con t_i = costo(corpo(for(i))) $= O\left(n - i + \sum_{j=i+1}^{n-1} t_j\right)$, dove t_j = costo(if) $= O(1)$, per cui $t_i = O(n-i)$ e dunque costo($\mathcal{S}$) $= O\left(n + \sum_{i=0}^{n-1}(n-i)\right) = O(n^2)$.

3.2 Insertion Sort

Sia $\mathcal{I}$:

```
void insertionSort(vector<int> &A) {
        int n = A.size();
        for (int i = 1; i < n; i++){
                int prox = A[i];
                int j = i;
                while ((j > 0) && (A[j-1] > prox)){
                        A[j] = A[j-1];
                        j--;
                }
                A[j] = prox;
        }
}
```

Si ha costo($\mathcal{I}$) $= O\left(n + \sum_{i=1}^{n-1} m_i\right)$, con $m_i =$ costo(while) e in particolare della sua guardia, essendo costo(corpo(while)) $= O(1)$, per cui $1 \leq m_i < i + 1$ e quindi

- Il caso ottimo in cui A è già ordinato dà $m_i = 1 \ \forall \ i$, quindi costo($\mathcal{I}$) $= O(n)$;
- Il caso medio in cui $m_i = \frac{i}{2}$ dà costo($\mathcal{I}$) $= O(\frac{1}{2}n^2) = O(n^2)$;
- Il caso pessimo in cui $m_i = i$ dà costo($\mathcal{I}$) $= O(n^2)$.

3.3 Quick Sort

Esso è ricorsivo e fa uso della metodologia *divide et impera*:

1. Caso base: risolvere per $n_0 = O(1)$ dati;
2. Passo induttivo:
   (a) Dividere il problema in sottoproblemi dello stesso tipo ma con un numero di input strettamente minore;
   (b) Risolvere i sottoproblemi;
   (c) Combinare le soluzioni.

3.3.1 L'ordinamento binario

Sia B un vettore binario (cioè con elementi $0, 1$) di n bit. Si ordini B mediante confronti e spostamenti come segue: siano i, j due puntatori che scorrono l'array in direzioni opposte, rispettivamente verso destra e verso sinistra, fermandosi rispettivamente sul primo 1 o 0 incontrato. A questo punto i due elementi vengono scambiati e si procede.

3.3.2 Pivot e distribuzione

Nel caso del Quick Sort si deve ordinare da sinistra (sx) a destra (dx) all'interno dell'array A, con $sx = 0$, $dx = n-1$ all'inizio dell'algoritmo. Fissato un pivot $A[px]$, con $sx \leq px \leq dx$, nell'array $B = (A[sx], A[dx])$ si etichettino gli elementi minori di $A[px]$ con 0 e quelli maggiori di $A[px]$ con 1 e si ordini B come vettore binario, col pivot che, al termine dell'algoritmo, si trova nella posizione rango (rg). L'algoritmo descritto è detto distribuzione o partizione, $\mathcal{D}$:

```
int Distribuzione(vector <int> &A, int sx, int px, int dx){
        if (px != dx){
                scambia(A[px],A[dx]);
        }
        int i = sx;
        int j = dx-1;
        while(i <= j){
                while((i <= j) && (A[i] <= A[dx])){
                        i++;
                }
                while((i <= j) && (A[j] >= A[dx])){
                        j--;
                }
                if(i < j){
                        scambia(A[i],A[j]);
                }
        }
        scambia(A[i],A[dx]);
        return i;
}
```

Siano n_0, n_1 il numero di iterazioni dei doppi while, rispettivamente primo-secondo e primo-terzo, vale $n_0 + n_1 = O(n)$, per cui costo$(\mathcal{D}) = O(n)$.

Si è usata la semplice funzione

```
void scambia(auto &i, auto &j){
        auto temp=i;
        i=j;
        j=temp;
}
```

3.3.3 Schema ricorsivo del Quick Sort

```
void QuickSort(vector <int> &A, int sx, int dx){
        if(sx < dx){
```

```
                int px=r(sx,dx);
                int rg=Distribuzione(&A, sx, px, dx);
                QuickSort(A, sx, rg-1);
                QuickSort(A, rg+1, dx);
        }
}
```

La funzione r sceglie nel vettore $[sx \cdots dx]$ il valore di px. Si vedrà più avanti quale scelta è la migliore.

3.3.4 Costo computazionale del Quick Sort

Il caso pessimo si ha quando il pivot scelto è l'estremo (massimo o minimo) del segmento $A[sx \cdots dx]$ e ciò comporta costo($\mathcal{QS}$) $= O(n^2)$, infatti all'i-esimo livello dell'albero di chiamate ricorsive si ha un costo di $n - i$ operazioni, dunque costo($\mathcal{QS}$) $= O\left(\sum_{i=0}^{n} n - i\right) = O(n^2)$. Tuttavia, strategie deterministiche per la scelta del pivot non aiutano, per cui si ricorre a tecniche di randomizzazione, trasformando il Quick Sort in un algoritmo randomizzato: scegliendo come r la funzione myrand(a, b), che restituisce un intero nell'intervallo $[a \cdots b]$ con probabilità uniforme $\frac{1}{b-a+1}$ (con $0 < a < b$), l'algoritmo diviene meno sensibile alla distribuzione dell'input.

3.3.5 Analisi del Quick Sort randomizzato

Sia X il numero di confronti dell'algoritmo, vale costo $= O(n + X)$ (dove n deriva dalla lettura dell'input), quindi al caso medio si ha costo $= O(n + E[X])$, per cui l'idea è di scrivere $X = \sum X_i$, con X_i variabili indicatrici.

Siano $z_1 < \ldots < z_n$ gli elementi ordinati e si definisca l'evento

$$A_{i,j} = \{z_i \text{ e } z_j \text{ vengono confrontati nell'algoritmo}\}.$$

Si definisca poi la variabile indicatrice $X_{i,j} = \begin{cases} 1 \text{ se } A_{i,j} \\ 0 \text{ altrimenti} \end{cases}$.

Si noti che:

(i) Se z_i, z_j vengono confrontati, allora uno dei due elementi è pivot dell'altro;

(ii) z_i, z_j vengono confrontati al più una volta, per cui $X = \sum_{1 \le i < j \le n} X_{i,j}$;

(iii) Se z_i, z_j sono nella medesima partizione, allora anche $z_{i+1}, \ldots, z_{j-1}$ appartengono a tale partizione.

Vale dunque $E[X] = \sum_{1 \le i < j \le n} E[X_{i,j}] = \sum_{1 \le i < j \le n} P(X_{i,j} = 1) \overset{(iii)}{\le} \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$
$\le \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \approx \sum_{i} \sum_{k=2}^{n-i+1} \frac{2}{k} = O(n \log n)$.

3.4 Merge Sort

Per raggiungere il costo $O(n \log n)$ anche al caso pessimo è necessario eliminare il possibile sbilanciamento del Quick Sort, ossia i casi in cui $px \approx sx$ oppure $px \approx dx$, che rendono la ricorsione inefficiente.

Con un modello più bilanciato, ogni livello dell'albero di ricorsione ha costo n, ma il numero di livelli al caso pessimo è $\log_2 n$ (supponendo di dividere a metà ogni input), da cui si ha il costo $n \log n$.

3.4.1 Fusione

Si consideri la porzione $S = A[sx \cdots dx]$ da ordinare e sia $cx = \frac{sx+dx}{2}$, la ricorsione ordina separatamente $S_1 = S[sx \cdots cx],\ S_2 = S[cx + 1 \cdots dx]$. È necessario dunque fondere i due ordinamenti per ottenere l'ordinamento di S. Per farlo, si sfrutti l'ordinamento di S_1, S_2, come nel seguente esempio.

Siano $S_1 = \begin{bmatrix} 3 & 7 & 18 & 21 \end{bmatrix}$, $S_2 = \begin{bmatrix} 4 & 20 & 25 & 30 \end{bmatrix}$, si ordini $S = S_1 \overset{\circ}{\cup} S_2$ come segue:

1. $3 < 4 \Rightarrow 3$;
2. $7 < 4 \Rightarrow 4$;
3. $7 < 20 \Rightarrow 7$;
4. $18 < 20 \Rightarrow 18$;
5. $21 > 20 \Rightarrow 20$;
6. $21 < 25 \Rightarrow 21$;
7. S_1 è esaurito $\Rightarrow 25, 30$;
8. $S = \begin{bmatrix} 3 & 4 & 7 & 18 & 20 & 21 & 25 & 30 \end{bmatrix}$.

```
void Fusione(vector<int> &A, int sx, int cx, int dx){
        int i = sx;
        int j = cx + 1;
        int k = 0;
        while((i <= cx) && (j <= dx)){
                if(A[i] <= A[j]){
```

```
                B[k] = A[i];
                i++;
            } else {
                B[k] = A[j];
                j++;
            }
            k++;
        }
        for(/**/; i <= cx; i++, k++){
            B[k] = A[i];
        }
        for(/**/; j <= dx; j++, k++){
            B[k] = A[j];
        }
        for(i = sx; i <= dx; i++){
            A[i] = B[i-sx];
        }
}
```

Vale costo(Fusione) $= \theta(n)$ poiché a ogni confronto uno dei due indici i, j scorre e dunque il numero di confronti non può eccedere il numero di elementi.

3.4.2 Merge Sort ricorsivo

Si presenta la versione ricorsiva del Merge Sort:

```
void MergeSort(vector<int> &A, int sx, int dx){
        if(sx < dx){
            int cx = (sx + dx) / 2;
            MergeSort(A, sx, cx);
            MergeSort(A, cx+1, dx);
            Fusione(A, sx, cx, dx);
        }
}
```

Sia $T(n) =$ costo(MS), allora $T(n) \leq \begin{cases} \text{costante se } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + cn \text{ altrimenti} \end{cases}$, con $c > 0$ costante, per cui $T(n) = O(n \log n)$.

Nota Nell'implementazione si definisca una funzione non ricorsiva contenente la definizione del vettore B (inizializzato a vettore nullo di lunghezza pari a quella di A), da utilizzare nell'algoritmo Fusione, e la chiamata dell'algoritmo ricorsivo Merge Sort. Ciò evita l'allocazione di B a ogni chiamata ricorsiva:

```
void MergeAlgo(vector<int> &A){
        int n = A.size();
        vector <int> B;
        emptyA(B, n);
        MergeSort(A, 0, n-1);
}
```

con

```
void emptyA(vector<int> &A, int n){
        for (int i = 0; i < n; i++)
        A.push_back(0);
}
```

3.4.3 Scansione sequenziale

Per grandi quantità di dati la scansione sequenziale è molto efficiente, sia perché sfrutta l'architettura della macchina (livelli di cache), sia negli accessi a memorie esterne. Pertanto, implementando l'algoritmo Fusione tramite scansione seguenziale si ottiene un'ulteriore miglioria. Per farlo, si applichi il Merge Sort a porzioni di 2^k elementi, con $k = 0, \ldots, \log_2 n$, sfruttando al k-esimo passo l'ordinamento del precedente.

4 Analisi del costo ricorsivo

In alcune circostanze non si riesce a determinare il costo della chiamata ricorsiva. Tuttavia, sia $T(n)$ tale costo, talvolta si riesce a stimare

$$T(n) \leq \alpha T(\frac{n}{\beta}) + cf(n), \tag{4}$$

con α numero di chiamate ricorsive, $f(n)$ costo di *divide* e ricombina e $\frac{n}{\beta}$ numero di elementi su cui è effettuata ogni chiamata ricorsiva. In tali casi vale il seguente teorema.

4.1 Master theorem

Siano $f(n)$ non decrescente e $\alpha \geq 1,\ \beta > 1,\ n_0, c_0, c > 0$ costanti tali che

$$T(n) \leq \begin{cases} c_0 \text{ se } n \leq n_0 \\ \alpha T\left(\frac{n}{\beta}\right) + cf(n) \text{ altrimenti} \end{cases}, \tag{5}$$

se $\exists\ \gamma, n_0' > 0 : \alpha f(\frac{n}{\beta}) \leq \gamma f(n)\ \forall\ n \geq n_0'$, allora

$$T_n = \begin{cases} O(f(n)) \text{ se } \gamma < 1 \\ O(f(n)\log_\beta n) \text{ se } \gamma = 1 \\ O(n^{\log_\beta \alpha}) \text{ se } \gamma > 1 \end{cases} .$$

Nota Nel Merge Sort valgono $\alpha = 2,\ \beta = 2,\ f(n) = n$, per cui $\gamma = 1$ e dunque $T(n) = O(f(n)\log_\beta n) = O(n \log n)$.

5 Moltiplicazione tra matrici

Siano $A, B \in M_n(\mathbb{C})$, si definisce $C := A \cdot B \in M_n(\mathbb{C})$ come $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$, per cui

- $\text{costo}(c_{i,j}) = \Theta(n)$;
- $\text{costo}(C) = \Theta(n^3)$.

Il problema del Matrix Multiplication (MM) ha dunque $O(n^3)$ come upper bound e $\Omega(n^2)$ come lower bound per fooling argument. Si ha quindi tale gap di un ordine di grandezza, che invece non vi è nella somma, problema $\Theta(n^2)$.

Tuttavia, vi è una congettura per la quale la complessità della MM è $O(n^\omega)$, con $\omega \approx 2.37$.

La riduzione dell'upper bound deriva dall'idea di Strassen di utilizzare un algoritmo *divide et impera* (sotto l'ipotesi di n potenza di 2) con:

- Caso base: $n = 1$, prodotto tra scalari in $O(1)$;
- Passo induttivo: $n \geq 2$

$$C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \cdot \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix},$$

  per cui $C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i,2} \cdot B_{2,j} \in M_{\frac{n}{2}}(\mathbb{C}),\ 1 \leq i, j \leq 2$.

In questo modo, a ogni chiamata ricorsiva si eseguono 8 prodotti matriciali e 4 somme, per cui la relazione di ricorrenza è $T(n) \leq 8T(\frac{n}{2}) + cn^2,\ c > 0$, per cui $\alpha = 8,\ \beta = 2,\ f(n) = n^2$, quindi $\gamma = 2$ e $T(n) = O(n^{\log_\beta \alpha}) = O(n^2)$.

In realtà, Strassen realizzò un algoritmo per il quale sono sufficienti $\alpha = 7$ moltiplicazioni, per cui $T(n) = O(n^{\log_2 7})$.

5.1 Algoritmo di Strassen

Siano $v_0 = (b-d)(g+h),\ v_1 = (a+d)(e+h),\ v_2 = (a-c)(e+f),\ v_3 = (a+b)h,\ v_4 = a(f-h),\ v_5 = d(g-e),\ v_6 = (c+d)e$, allora

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} v_0 + v_1 - v_3 + v_5 & v_3 + v_4 \\ v_5 + v_0 & v_1 - v_2 + v_4 - v_6 \end{bmatrix}. \tag{6}$$

6 Problema del nearest neighbour (NN)

Si analizzi il seguente problema della coppia di punti più vicina. Sia $S \subseteq \mathbb{R}^k$ metrico con $d : S \times S \to [0, +\infty)$, si intende trovare $x \neq y \in S : d(x, y) \leq d(x', y')\ \forall\ x' \neq y' \in S$. Ciò è cruciale, ad esempio, nella comparazione di pagine web in base al numero di parole in comune e alle loro ricorrenze.

In questa sezione si affronta il caso $k = 2,\ d$ distanza Euclidea.

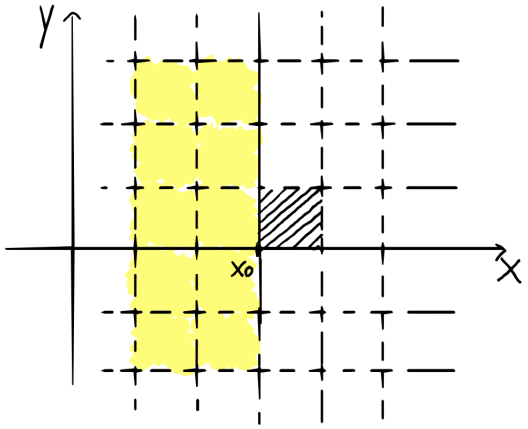
6.1 Baseline del NN

Per ogni coppia di punti si calcoli la loro distanza e si memorizzi la coppia con distanza minima. Sia $n = \#S$, si hanno $\binom{n}{2}$ possibili coppie, per cui si ottiene $O(n^2)$.

6.2 Problema del NN tramite *divide et impera*

- Caso base: $n \leq 3$, ispezione diretta in $O(1)$;
- Passo induttivo:
  1. Si ordini S in base all'ascissa ottenendo $S = S_1 \overset{\circ}{\cup} S_2$, con $S_1 = \{(x, y) \in S : x \leq x_0\},\ S_2 = \{(x, y) \in S : x > x_0\}$, con costo pari a $O(n \log n)$ (v. sezione 3.4) per la prima divisione e $O(n)$ per ogni divisione successiva;
  2. Si trovino le coppie minime in S_1 e S_2 e si confrontino con la coppia minima mista, cioè realizzata da un elemento di S_1 e uno di S_2.

È evidente che la difficoltà implementativa risiede proprio nell'individuazione della coppia minima mista. Si noti tuttavia che tra le coppie siffatte, l'interesse ricade sulle coppie $(p_1, p_2) : p_1 \in S_1,\ p_2 \in S_2,\ d(p_1, p_2) < \delta := \min\{\delta_1, \delta_2\}$, con δ_i distanza della coppia minima in S_i.



Pertanto, si effettui una suddivisione di $\mathbb{R}^2$ in quadrati di lato $\frac{\delta}{2}$, per cui ogni quadrato (che ha diagonale $\frac{\delta}{\sqrt{2}} < \delta$)contiene al più un punto. Sia $S_Y = \{(x, y) \in S : |x - x_0| \leq \delta\}$, poiché S è ordinato, con costo $O(n)$ si ottiene S_Y ordinato rispetto all'ordinata. Si scorra dunque S_Y e si confronti ogni punto di S solo coi suoi vicini, effettuando un numero costante di confronti, pari a 10, per cui

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn \tag{7}$$

a cui si deve aggiungere $O(n \log n)$ per il primo ordinamento che permette di avviare il processo di *divide*.

7 Indecidibilità e casualità

Considerati un algoritmo A e dei dati d'ingresso D, secondo il modello di von Neumann entrambi A e D sono sequenze binarie, per cui sia $A(D)$ che $A(A)$ sono scritture lecite, benché non necessariamente sensate.

7.1 Teorema della fermata di Turing

Non esiste un algoritmo/programma in grado di stabilire se $A(D)$ termini o meno.

Dimostrazione Se esistesse un algoritmo

$$\text{TERMINA}(A, D) = \begin{cases} \text{TRUE se } A(D) \text{ termina} \\ \text{FALSE altrimenti} \end{cases},$$

si consideri l'algoritmo

```
void PARADOSSO(auto x){
        while(TERMINA(x,x)){
                ; //istruzione vuota
        }
}
```

che termina se e solo se TERMINA(x, x) = TRUE, ci si chiede se PARADOSSO(PARADOSSO) termini:

- Se PARADOSSO(PARADOSSO) termina, allora per costruzione di PARADOSSO vale TERMINA(PARADOSSO,PARADOSSO) = FALSE, cioè PARADOSSO(PARADOSSO) non termina;
- Se PARADOSSO(PARADOSSO) non termina, allora per costruzione di PARADOSSO vale TERMINA(PARADOSSO,PARADOSSO) = TRUE, cioè PARADOSSO(PARADOSSO) termina.

Pertanto, si ha un assurdo.

Nota Da ciò segue che non tutti i problemi computazionali ammettono algoritmo risolutivo. Il contributo di Turing è di averne esibito uno concreto. Un altro famoso problema indecidibile è quello della sequenza, cioè di stabilire se due algoritmi siano equivalenti.

7.1.1 Esistenza di un algoritmo risolutivo

Non tutti i problemi computazionali ammettono algoritmo risolutivo.

Dimostrazione Si è visto che un problema computazionale è una funzione $\pi : \{0,1\}^* \to \{0,1\}^*$ ed essendo $\{0,1\}^* \leftrightarrow \mathbb{N}$, allora $\#\{\text{problemi computazionali}\} = \#\{f : \mathbb{N} \to \mathbb{N}\} = \#\mathbb{R}$. Invece, un algoritmo è una sequenza binaria, quindi $\#\{\text{algoritmi}\} = \#\mathbb{N}$. Pertanto, $\#\{\text{problemi computazionali}\} = \#\mathbb{R} > \#\mathbb{N} = \#\{\text{algoritmi}\}$.

7.2 Sequenze casuali

Sia $K_L(x)$ la lunghezza in bit del programma nel linguaggio L che genera la sequenza X (senza input). Vale $K_L(x) = c_L + \log_2 |x|$ e $K_L(x), K_{L'}(x)$ differiscono per una costante indipendente da $|x|$, per cui spesso si omette L. Si dice che X è una sequenza casuale se $K(x) \geq |x| - c$, cioè se il modo più veloce per generare x è scrivere x direttamente. Il concetto di casuale è dunque strettamente legato con l'incompatibilità.

7.2.1 Esistenza di stringhe casuali

Vale che $\forall\ n \in \mathbb{N}\ \exists\ x \in \{0,1\}^n : K(x) \geq n$.

Dimostrazione Sia $S := \{x \in \{0,1\}^n : K(x) < n\}$, allora $\#S = \#(A(y) = x \ \wedge\ \#A(y) < n) \leq 2^0 + 2^1 + \ldots + 2^{n-1} = 2^n - 1 < 2^n$ e ciò conclude poiché $\#\{0,1\}^n = 2^n$.

7.2.2 Incidenza delle stringhe casuali

Vale che $\forall\ n \in \mathbb{N},\ c \geq 1$ la probabilità delle stringhe casuali è $P(x \in \{0,1\}^n : K(x) \geq n - c) > 1 - \frac{1}{2^c}$.

Dimostrazione Sia $S := \{x \in \{0,1\}^n :\ K(x) < n - c\}$, allora $\#S \leq 2^0 + 2^1 + \ldots + 2^{n-c-1} = 2^{n-c} - 1$, quindi $\frac{\#S}{\#\{0,1\}^n} \leq \frac{2^{n-c}-1}{2^n} = \frac{1}{2^c} - \frac{1}{2^n} < \frac{1}{2^c}$, per cui $P(x \in \{0,1\}^n :\ K(x) \geq n - c) = 1 - P(S) \geq 1 - \frac{1}{2^c}$.

7.2.3 Indecidibilità della casualità di una stringa

È indecidibile stabilire se $K(x) \geq |x| - c$.

Dimostrazione Sia $R := \{x \in \{0,1\}^n :\ K(x) \geq |x| - c\}$, si mostri che è indecidibile stabilire se $x \in R$ o meno.

Se per assurdo esistesse un algoritmo $A_R(x) = \begin{cases}\text{TRUE se } x \in R \\ \text{FALSE altrimenti}\end{cases}$, sia B l'algoritmo che genera tutte le stringhe binarie in ordine lessicografico ($<_L$) e a ogni generazione esegue $A_R(x)$, terminando se e solo se $A_R(x) = \text{TRUE}$ e dunque stampando x. Sia ora B_n che esegue B solo per le stringhe in $\{0,1\}^n$, cioè termina quando B ha generato $1 \cdots 1$ (n cifre 1) ed eseguito $A_R(1 \cdots 1)$.

Entrambe B, B_n terminano per l'esistenza di stringhe casuali. Sia $x_n = B_n$, cioè $x_n = \min_{<_L}\{x \in R : |x| = n\}$, allora $K(x_n) \geq |x_n| - c = n - c$, quindi $|B_n| \geq K(x_n) \geq n - c$ poiché B_n genera x_n (ma non è necessariamente il più veloce a farlo), assurdo poiché $|B_n| = c' + \log_2 n < n - c$ per $n \to +\infty$, infatti la lunghezza di B_n varia solo per la codifica di n nella chiamata di B ristretto a $\{0,1\}^n$.

8 Strutture elementari

8.1 Pila e coda

La pila o stack implementa la strategia *Last In First Out* (LIFO), basandosi sugli algoritmi push(x), che aggiunge x in cima alla pila, e pop(), che restituisce l'elemento in cima alla pila e lo rimuove da essa.

La coda o queue implementa la strategia *First In First Out* (FIFO), basandosi sugli algoritmi enqueue(x), che aggiunge x in coda, e dequeue(), che restituisce l'elemento in testa e lo rimuove dalla queue.

Le implementazioni possibili sfruttano le strutture di array, liste e vector.

8.1.1 Coda con priorità e heap implicito

Nella coda con priorità a ogni elemento è assegnata una priorità, per cui:

- enqueue(x) aggiunge x in coda;
- dequeue() restituisce l'elemento con maggior priorità e lo rimuove dalla queue.

Lo heap implicito (heap significa mucchio) è un array con relazione padre-figlio legata alla sola numerazione degli elementi: considerato un nodo dell'albero corrispondente alla posizione i, allora il figlio sinistro è in posizione $2i+1$, mentre il figlio destro è in posizione $2i+2$. Viceversa, il padre del nodo in posizione i è in posizione $\lfloor \frac{i-1}{2} \rfloor$.

Secondo l'ordine heap-max vale padre>figli, mentre per l'ordine heap-min vale padre<figli.

Per controllare l'heap-max è utile partire dai figli e verificare che valga la relazione figli<padri $\forall\ i = 1, \ldots, n-1$ ($i = 0$ è la radice, che non ha padre).

La struttura della coda con priorità richiede di memorizzare n priorità e $O(1)$ celle di memoria per completare l'albero con nodi vuoti.

Si definisce altezza di un heap la lunghezza del percorso radice-foglia più lungo ed è $O(\log_2 n)$ poiché il padre è in posizione $\lfloor \frac{i-1}{2} \rfloor$, con i posizione del figlio.

Segue l'algoritmo per la riorganizzazione dell'heap secondo la priorità:

```
void RiorganizzaHeap(int i){
        while(i > 0 && heapArray[i].prio > heapArray[Padre(i)].prio){
                Scambia(i,Padre(i));
                i = Padre(i);
        }
        int migliore=0;
        while (Sx(i) < heapSize && i != MigliorePadreFigli(i)){
                migliore = MigliorePadreFigli(i);
                Scambia(i,migliore);
                i = migliore;
        }
}
```

con

```
int MigliorePadreFigli(int i){
        int j = Sx(i);
        int k = Sx(i);
```

```
        if(k + 1 < heapSize){
                k++;
        }
        if(heapArray[k].prio > heapArray[j].prio){
                j = k;
        }
        if(heapArray[i].prtio >= heapArray[j].prio){
                j = i;
        }
        return j;
}
```

dove $\mathrm{Sx}(i) = 2i + 1$, $\mathrm{Dx}(i) = 2i + 2$, $\mathrm{Padre}(i) = \lfloor \frac{i-1}{2} \rfloor$.

Pertanto, enqueue(x) aggiunge x in coda e chiama RiorganizzaHeap(n), mentre dequeue() restituisce la radice, mette l'ultima foglia come radice e chiama RiorganizzaHeap(0).

Tramite la struttura dell'heap è possibile implementare un algoritmo di ordinamento, detto HeapSort, che esegue $O(n \log n)$ confronti e richiede solo $O(1)$ spazio aggiuntivo.

8.2 Alberi binari

8.2.1 Struttura induttiva

Sia S un insieme con $\#S = n \in \mathrm{N}$, un albero T è una 3-partizione di S come $S = S_L \mathbin{\mathring{\cup}} \{r\} \mathbin{\mathring{\cup}} S_R$, con $r \in S$ detto radice. La rappresentazione implementativa è $T = \begin{bmatrix} r & sx & dx \end{bmatrix}$, con sx puntatore a S_L e dx puntatore a S_R. Ricorsivamente, cioè 3-partizionando S_L e S_R come sopra, si ottiene la struttura ad albero. Si tenga presente che:

- Se $S = \varnothing$, allora $T = \mathrm{NULL}$ (caso base);
- S_L e/o S_R possono essere vuoti (in tal caso sx e/o dx sono puntatori a NULL).

Un nodo è detto foglia se $S_L = S_R = \varnothing$.

8.2.2 Altezza e dimensione

Si definisce altezza di un albero il massimo numero di archi attraversati in un percorso radice-foglia:

```
int Altezza(u){
        if(u == NULL){
                return -1;
        } else {
                return 1 + max(Altezza(u.sx), Altezza(u.dx));
```

```
        }
}
```

Considerato un albero binario di n nodi, la sua altezza h è tale che $\log_2 n \leq h \leq n-1$.

La dimensione di un albero è il numero di nodi:

```
int Dimensione(u){
        if(u == NULL){
                return 0;
        } else {
                int dimensioneSX = Dimensione(u.sx);
                int dimensioneDX = Dimensione(u.dx);
                return dimensioneSX + dimensioneDX + 1;
        }
}
```

Questi metodi sono dei *divide et impera* applicati a un problema decomponibile (come quello dell'albero) in cui tuttavia la divisione non è stabilita dal programmatore ma dai dati e/o compilatore. Tali algoritmi sono bottom-up, cioè le informazioni sui figli vengono usate per dare informazioni sul padre.

8.2.3 Profondità

La profondità di un nodo u è il numero di archi attraversati nel cammino dalla radice a u, per cui altezza $= \max_u$ profondità(u):

```
profondita(u,d){
        if(u != NULL){
                u.profondita = d;
                profondita(u.sx, d + 1);
                profondita(u.dx, d + 1);
        }
}
```

Un livello di un albero T è un insieme di tutti e soli i nodi di T con una medesima profondità.

8.2.4 Relazione di ricorrenza di un albero

Considerato un albero con n nodi, vale

$$t(n) = c + t(x) + t(n - x - 1) \tag{8}$$

con $\#S_L = x$, per cui $\#S_R = u - x - 1$. Pertanto, $\exists\, c$ costante tale che $t(n) \leq c \cdot n$, per cui tutti i metodi sugli alberi hanno costo lineare.

8.2.5 Bilanciamento

Un albero si dice completamente bilanciato se ogni livello ha il massimo numero di nodi:

```
auto CompletamenteBilanciato(u){
        if(u == NULL){
                return <true, -1>;
        } else {
                auto <bilSX, altSX> = CompletamenteBilanciato(u.sx);
                auto <bilDX, altDX> = CompletamenteBilanciato(u.dx);
                boolean completamenteBil = bilSX && bilDX && (altSX == al
                int altezza = max(altSX, altDX) + 1;
                return <completamenteBil, alteza>;
        }
}
```

Ciò restituisce due componenti: una booleana e un intero, con il booleano che è TRUE $\iff$ l'albero è completamente bilanciato e l'intero che è l'altezza dell'albero.

Un albero binario di n nodi è bilanciato se la sua altezza h verifica $h = O(\log_2 n)$.

8.2.6 Visite

```
visita(u){
        if(u != NULL){
                [cout << u.chiave;] //visita anticipata
                visita(u.sx);
                [cout << u.chiave;] //visita simmetrica
                visita(u.dx);
                [cout << u.chiave;] //visita posticipata
        }
}
```

Nella pratica:

- La visita anticipata è quella delle chiamate ricorsive negli alberi di ricorsione;
- La visita simmetrica è quella delle chiavi ordinate negli alberi di ricerca;
- La visita positicipata è quella dell'ordine di valutazione negli alberi sintattici;
- La visita per ampiezza va in ordine di livello e dunque da sinistra a destra.

8.3 Alberi binari di ricerca

Essi sono alberi binari in cui, fissato un ordinamento tra i nodi, ogni nodo ha come sottoalbero sinistro un albero con soli nodi minori di lui e come sottoalbero destro un albero con soli nodi maggiori di lui.

8.3.1 Ricerca binaria e alberi binari di ricerca

Considerati un array ordinato A e un elemento x del tipo degli elementi di A, ci si chiede se x sia un elemento di A e, nel caso, in quale posizione compaia. Confrontando x con l'elemento centrale e dimezzando A ricorsivamente si ottiene un metodo $\Theta(\log n)$:

```
Ricerca(u,k){
        if(u == NULL){
                return NULL;
        }
        if(k == u.dato.chiave){
                return u.dato;
        } else if(k < u.dato.chiave){
                return Ricerca(u.sx, k);
        } else {
                return Ricerca(u.dx, k);
        }
}

Inserisci(u,e){
        if(u == NULL){
                u = NuovoNodo();
                u.dato = e;
                u.sx = NULL;
                u.dx = NULL;
        } else if(e.chiave < u.dato.chiave){
                u.sx = Inserisci(u.sx,e);
        } else if(e.chiave > u.dato.chiave){
                u.dx = Inserisci(u.dx,e);
        }
        return u;
}
```

All'interno della funzione NuovoNodo c'è una *new* (*malloc* in C) per l'allocazione di memoria.

Sia Ricerca che Inserisci sono $O(h)$, con h altezza dell'albero.

8.3.2 Cancellazione

- Se il nodo u è una foglia o ha un solo figlio si procede alla cancellazione come nelle liste;
- Se il nodo u ha due figli, sia w il minimo del sottoalbero radicato in $u.dx$, si sostituisca la chiave di u con quella di w e poi si cancelli w (ci si è ricondotti al caso base).

```
Cancella(u,k){
        if(u != NULL){
                if(u.dato.chiave == k){
                        if(u.sx == NULL){
                                u = u.dx;
                        } else if(u.dx == NULL){
                                u = u.sx;
                        } else {
                                w = MinimoSottoalbero(u.dx);
                                u.dato = w.dato;
                                u.dx = Cancella(u.dx, w.dato.chiave);
                        }
                } else if(k < u.dato.chiave){
                        u.sx = Cancella(u.sx,k);
                } else if (k > u.dato.chiave){
                        u.dx = Cancella(u.dx,k);
                }
        }
        return u;
}
```

dove

```
MinimoSottoalbero(u){
        while(u.sx != NULL){
                u = u.sx;
        }
        return u;
}
```

8.3.3 Caso medio randomizzato e quick sort randomizzato

L'inserimento delle chiavi di un array in un albero vuoto è analogo al quick sort randomizzato:

- pivot $\rightarrow$ radice;

- chiavi<pivot → sottoalbero left;
- chiavi>pivot → sottoalbero right.

Sia $X_{i,j} = \begin{cases} 1 \text{ se } z_i, z_j \text{ vengono confrontati} \\ 0 \text{ altrimenti} \end{cases}$, se $z_i = p$ (pivot), allora $Y = \sum_j X_{i,j}$ indica il numero di chiavi con cui p viene confrontato. Se p foglia, allora Y indica la profondità di p, per cui $E[\text{altezza}] = E[Y] = \sum_j E[X_{i,j}] = \sum_j \frac{2}{|j-i+1|} = O(\log n)$, con la penultima uguaglianza analoga al QS randomizzato e l'ultima per serie armonica.

8.4 Alberi AVL: 1-bilanciamento

Tali alberi garantiscono $h = O(\log n)$ al caso pessimo. Sia $h(u)$ l'altezza del nodo u, cioè la massima lunghezza di un cammino da u a una delle sue foglie, si dice che un albero è 1-bilanciato se per ogni nodo u vale $|h(u.sx) - h(u.dx)| \leq 1$.

L'1-bilanciamento è una proprietà locale e dunque verificabile ricorsivamente. Gli alberi 1-bilanciati sono detti AVL.

Per gli alberi 1-bilanciati vale $h = O(\log n)$.

Dimostrazione Vale $n \geq n_h \geq c^h$, per cui $h = O(\log n)$, con n_h numero di nodi dell'albero di Fibonacci di altezza h, per cui le due disuguaglianze valgono per definizione e costruzione degli alberi di Fibonacci (v. sezione 8.4.1).

8.4.1 Alberi di Fibonacci

Si definisce albero di Fibonacci di altezza h il più piccolo (in termini di numero di nodi) albero AVL tale che se si rimuove una foglia si ottiene un albero non più 1-bilanciato o di altezza $h - 1$.

La definizione ricorsiva degli alberi di Fibonacci è la seguente:

Figura 1: $h = 0$

Figura 2: $h = 1$

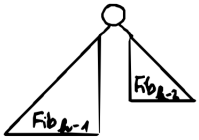


Figura 3: $h \geq 2$

8.4.2 Operazioni sugli alberi AVL

- Ricerca: come negli alberi binari di ricerca (v. sezione 8.3.1), ottenendo $O(h) = O(\log n)$;
- Cancellazione: si marchino i nodi da cancellare e si ricostruisca l'albero con le sole chiavi non marchiate quando il numero di marchi è una frazione di n costante (stabilita a priori);
- Inserimento: si crei una nuova foglia e si ruoti l'albero se ha perso la proprietà di 1-bilanciamento. Cioè, considerata la foglia f inserita, si definisce nodo critico il nodo antenato di f più vicino a f tale che $|h(u.sx) - h(u.dx)| = 2$ (non necessariamente un tale nodo esiste):

```
Inserisci(u,e){
        if(u == NULL){
                return f = NuovaFoglia(e);
        } else if(e.chiave < u.dato.chiave){
                u.sx = Inserisci(u.sx, e);
                if(Altezza(u.sx) - Altezza(u.dx) == 2){
                        if(e.chiave > u.sx.dato.chiave){
                                u.sx = RuotaAntiOraria(u.sx);
                        }
                        u = RuotaOraria(u);
                }
        } else if(e.chiave > u.dato.chiave){
                u.dx = Inserisci(u.dx, e);
                if(Altezza(u.dx) - Altezza(u.sx) == 2){
                        if(e.chiave < u.dato.chiave){
                                u.dx = RuotaOraria(u.dx);
                        }
                        u = RuotaAntiOraria(u);
                }
        }
        u.altezza = max(Altezza(u.sx), Altezza(u.dx)) + 1;
        return u;
}
```

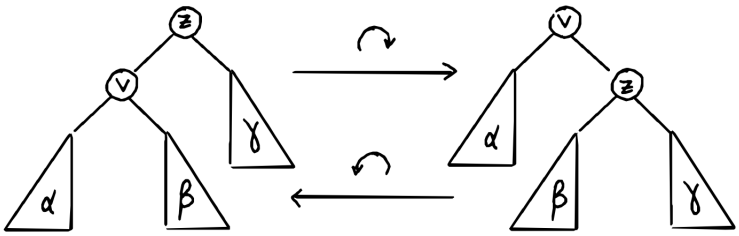
con

```
int Altezza(u){
        if(u == NULL){
                return -1;
        }
        return u.altezza
}

NuovaFoglia(e){
        u = NuovoNodo();
        u.dato = e;
        u.altezza = 0;
        u.sx = NULL;
        u.dx = NULL;
        return u;
}
```

per cui è necessario un dato aggiuntivo altezza nella definizione della struct albero.

Le funzioni di rotazione viste per l'operazione di inserimento servono a rendere l'albero nuovamente 1-bilanciato nel caso in cui l'inserimento della foglia lo abbia sbilanciato, cioè abbia generato un nodo critico:

```
RuotaOraria(z){
        v = z.sx;
        z.sx = v.dx;
        v.dx = z;
        z.altezza = max(Altezza(z.sx), Altezza(z.dx)) + 1;
        v.altezza = max(Altezza(v.sx), Altezza(v.dx)) + 1;
        return v;
}

RuotaAntiOraria(v){
        z = v.dx;
        v.dx = z.sx;
        z.sx = v;
        v.altezza = max(Altezza(v.sx), Altezza(v.dx)) + 1;
        z.altezza = max(Altezza(z.sx), Altezza(z.dx)) + 1;
        return z;
}
```



9 Problema del dizionario

Siano U universo delle chiavi e $S \subseteq U$ insieme delle chiavi memorizzate (class map), si considerino i seguenti problemi:

- Appartenenza (find): dire se per un fissato $x \in U$ valga $x \in S$;
- Inserimento (insert/add): $S \leftarrow S \cup \{x\}$;
- Cancellazione (delete/remove): $S \leftarrow S \setminus \{x\}$.

Un dizionario si dice statico se supporta solo appartenenza, dinamico se supporta anche inserimento e cancellazione.

Se U è ordinato, si hanno le seguenti operazioni:

- $\text{pred}(x) = \max\{y \in S : y \leq x\}$;
- $\text{succ}(x) = \min\{y \in S : y \geq x\}$;
- $\text{intervallo}(a, b) = \{y \in S : a \leq y \leq b\},\ \text{count}(a, b) = \#\text{intervallo}(a, b)$.

9.1 Dizionario per $n = |S|$ chiavi

Struttura di dati	find	insert	delete
Array non ordinato	$O(n)$	$O(1)$	$O(1)$
Array ordinato	$O(\log n)$	$O(n)$	$O(n)$
Lista non ordinata	$O(n)$	$O(1)$	$\begin{cases} O(1) \text{ se } \leftrightarrow \\ O(n) \text{ se } \rightarrow \end{cases}$
Lista ordinata	$O(n)$	$O(n)$	$\begin{cases} O(1) \text{ se } \leftrightarrow \\ O(n) \text{ se } \rightarrow \end{cases}$
Heap	$O(n)$	$O(\log n)$	$O(\log n)$
Albero binario di ricerca di altezza $\log n \leq h \leq n$	$O(h) = O(\log n)$ al caso medio		
AVL	$O(h) = O(\log n)$ al caso pessimo		
Tabelle Hash	$O(1)$ al caso medio		

9.2 Hash

9.2.1 Funzione hash

Sia U universo delle chiavi, una funzione hash è $h : U \to [m]$, con $[m]$ insieme di cardinalità $m \ll \#U$, in modo che ogni elemento di U venga ridotto da sequenza di bit (intero molto grande) a numero relativamente piccolo in $[m]$. Poiché $\#U \gg m$, per principio della piccionaia $\exists\ k_1, k_2 \in U :\ k_1 \neq k_2$, ma $h(k_1) = h(k_2)$, cioè h non iniettiva.

Pertanto, le tabelle hash riducono il numero di chiavi e non operano confronti, scendendo sotto la soglia di $\log n$ per la ricerca.

A livello implementativo, funzioni hash comuni sono:

- $h(x) = x\%m$;
- Sia $p \in [m+1,\ 2m]$ primo e siano $a \in \mathbb{Z}_p^*,\ b \in \mathbb{Z}_p$ scelti casualmente e indipendentemente (con probabilità uniforme), si definisca la hash $h_{a,b}(x) = ((ax+b)\%p)\%m$, detta hash universale. Vale $\underset{a,b}{P}[h_{a,b}(k_1) = h_{a,b}(k_2)] = \frac{1}{m}$, ottimale poiché uniformemente distribuita.

Un hash si dice perfetto se la funzione hash non dà collisioni su S.

9.2.2 Folding

Per chiavi lunghe si impiega la tecnica del ripiegamento o folding, che spezza x in $x_1 \cdots x_k$, con x_1 in 64 bit, definendo $h(x) = \text{xor}_{i=1}^{k} h'(x_i)$, con $h' : U' \to [m]$.

Nel caso particolare delle stringhe, spesso si usano $x = a_0 \cdots a_{n-1}$ e $h(x) = \sum_{i=0}^{n-1} (a_i \sigma^i)\%m$, con σ dimensione dell'alfabeto (per ASCII si usa $\sigma = 131$).

In generale, il Karp-Rabin fingerprint $h(x_{i+1,j+1}) = ((h(x_{i,j}) - a_i\sigma^{n-1})\sigma + a_{j+1})\%m$ fa sì che sia necessario $O(1)$ per calcolare l'hash successivo.

9.2.3 Liste concatenate o di trabocco

In questo modello ogni casella i di una tabella hash T è una lista contenente tutte le chiavi x tali che $h(x) = i$. Pertanto:

- insert(x) = $T[h(x)]$.push_back(x);
- find(x) = scan$T[h(x)]$;
- delete(x){
  find $i :\ T[h(x)][i] == x$;
  swap($T[h(x)][i],\ T[h(x)]$[last]);
  $T[h(x)]$.pop_back;
  }

$x \in S$	B	A	D	C	F	G	E	H
$h(x)$	3	2	3	2	0	10	1	3

0	$\to F$
1	$\to E$
2	
3	$\to A, C, H$
4	$\to B, D$
5	
6	
7	
8	
9	
10	$\to G$

Pertanto, ogni casella della tabella hash si comporta come un vector non ordinato e la tabella è ottimizzata se la distribuzione delle chiavi è uniforme, come nell'hash universale

$$h_{a,b}(x) = ((ax + b)\%p)\%m. \tag{9}$$

Sia $\alpha := \frac{n}{m}$, detto fattore di carico, con n numero di chiavi, il valore α rappresenta la lunghezza media di una lista/vector. Al caso medio il costo è $O(1 + \alpha)$.

In genere $m \approx 2n$, per cui $\alpha = \frac{1}{2}$, ottenendo $O(1)$ al caso medio.

9.2.4 Indirizzamento aperto

Sia $m > n$ (spesso $m \approx 2n$), per scansione lineare si effettui il seguente inserimento: ogni chiave viene inserita nel primo posto libero dopo il proprio (se quest'ultimo è occupato). Vale l'effetto Pacman circolare.

$x \in S$	B	A	D	C	F	G	H
$h(x)$	3	2	3	2	0	10	3

F		A	B	D	C	H				G
0	1	2	3	4	5	6	7	8	9	10

Un gruppo di chiavi senza posti liberi in mezzo o a sinistra o a destra è detto cluster. Nell'esempio, i cluster sono

A	B	D	C	H
2	3	4	5	6

G	F
10	0

Un cluster è di fatto l'unione di liste di trabocco adiacenti.

Per quanto riguarda la ricerca, si parta dalla posizione $h(x)$ in V e si avanzi in modo circolare finché si trova x oppure si raggiunge un posto libero.

Per la cancellazione, infine, si marchi la chiave come cancellata e una volta raggiunta uno stabilito numero di marchi di cancellazione si ricostruisca V con le sole chiavi non marchiate.

9.2.5 Considerazioni probabilistiche sugli hash

Sia $\tilde{T}(n, m)$ il tempo medio (cioè il numero medio di posizioni esaminate) per inserire una chiave in una tabella hash di m posizioni di cui $n < m$ sono occupate. Il fattore di carico $\alpha = \frac{n}{m}$ è la probabilità di trovare una posizione occupata. Vale $\tilde{T}(n, m) = \alpha(1 + \tilde{T}(n-1, m-1)) + (1-\alpha) \cdot 1$, con caso base $\tilde{T}(0, m) = 1$, cioè $\tilde{T}(n, m) = 1 + \alpha T(n-1, m-1)$, per cui $\tilde{T}(n, m) \leq \frac{m}{m-n} = \frac{1}{1-\alpha}$.

9.2.6 Cuckoo hashing

Questo metodo produce $O(1)$ al caso pessimo per ricerca e cancellazione e $O(1)$ al caso medio ammortizzato per l'inserimento. Si avvale di 2 funzioni hash, e di un meccanismo di inserimento a rischio loop, risolto tramite il rehashing:

- ricerca(x) verifica se $T[h_1(x)] == x \ \vee \ T[h_2(x)] == x$;
- cancellazione(x) agisce come ricerca ma sostituisce x con $\varnothing$;
-

```
Inserimento(x){
        if(T[h_1(x)] == x || T[h_2(x)] == x){
                return;
        } else if(T[h_1(x)] == vuoto || T[h_2(x)] ==
                inserisci x;
                return;
        } else {
                int i = h_1(x) (oppure h_2(x));
                int c = 0;
                while(T[i] != 0 && c <= n + 1){
                        inserisci x in T[i];
                        memorizza in x la chiave cac
                        i = {h_1(x), h_2(x)}-{i};
                        c++;
                }
        }
```

```
            if(c > n+1){
                rehash;
            }
        }
```

con rehash che svuota la tabella, sceglie in modo random due nuove funzioni h_1 e h_2 e ricomincia l'inserimento con le nuove funzioni hash.

9.2.7 Costo computazionale dell'inserimento randomizzato

(i) Se T ha una cella vuota, si ha $O(1)$;

(ii) Se $T[h_1(x)] \neq \varnothing$ e si attraversa un cammino di posizioni $h_1(x), i', \ldots, j$ di lunghezza l (con $l+1$ indici), con $T[j]$ vuota prima di inserire x. Sia $m > 2nc$, con $c > 2$ costante, per induzione su l si ha

$$P(h_1(x) \underset{l}{\to} j) \leq \frac{1}{c^l m} \ (*),$$

quindi la lunghezza media è

$$\sum_{l \geq 1} \frac{l}{c^l m} \leq \frac{c}{(c-1)^2} \cdot \frac{1}{m}$$

e dunque al caso medio si ha $O\left(1 + \frac{c}{m(c-1)^2}\right) = O(1)$;

(iii) Come in (ii) ma con $T[j] \neq \varnothing$, per cui si effettua il rehash. Vale $P(\text{rehash}) \leq P(\text{esiste un loop}) = \sum_{i=0}^{m-1} P(i \to j = i) \leq \sum_{i=0}^{m-1} \sum_{l \geq 1} P(i \underset{l}{\to} j = i) \overset{(*)}{\leq} \sum_{i=0}^{m-1} \sum_{l \geq 1} \frac{1}{c^l m} \leq \frac{1}{m} \sum_{i=0}^{m-1} \frac{1}{c-1} = \frac{1}{c-1} =: p$. Il numero medio di rehashing è $\sum_t t p^t = O(1)$.

10 Grafi

Un grafo è una coppia $G = (V, E)$, con V insieme dei nodi o vertici ed $E \subseteq V \times V$ insieme degli archi. Se gli archi si considerano non orientati, cioè $(a, b) = (b, a) \in E$, allora $0 \leq \#E \leq \binom{\#V}{2}$. Si definisce dimensione del grafo G il valore

$$\dim G = \#V + \#E. \tag{10}$$

Array, liste e alberi sono dunque casi particolari di grafi su cui sussiste un ordinamento totale per quanto riguarda array e liste e parziale nel caso degli alberi.

10.1 Lettura di un grafo

Per leggere un grafo è sufficiente una matrice $(\#E + 1) \times 2$, in cui la prima riga ha come entrate $(\#V, \#E)$, mentre le restanti $\#E$ righe hanno come entrate gli archi, cioè coppie $(v, w) \in E$.

10.2 Vicinato e grado di un nodo

Considerato un nodo $u \in V$ si definisce il neighbourhood (o vicinato) di u come

$$N(u) := \{v \in V : \ (u, v) \in E\}. \tag{11}$$

Si definisce poi grado di u il valore

$$d(u) := \#N(u). \tag{12}$$

Pertanto, per il teorema dell'handshaking vale

$$\sum_{u \in V} d(u) = 2 \cdot \#E. \tag{13}$$

10.3 Grafi orientati

Un grafo $G = (V, E)$ si dice orientato se gli archi in E si considerano come unidirezionali, cioè $(a, b) \neq (b, a)$, per cui $0 \leq \#E \leq \#V \cdot (\#V - 1)$.

10.4 Matrice di adiacenza e gradi in uscita ed entrata

Si definisce la matrice di adiacenza $A = (\delta_{i,j}) \in M_{\#V}$ data da $\delta_{i,j} = \begin{cases} 0 \text{ se } (i,j) \notin E \\ 1 \text{ se } (i,j) \in E \end{cases} \quad \forall\, 0 \leq i, j \leq \#V - 1$.

Se il grafo è non orientato, la matrice di adiacenza è simmetrica e vale $d(u) = \sum_{i=0}^{\#V-1} A_{i,u}$.

Se il grafo è orientato, si definiscono grado in uscita d^+ e grado in entrata d^- del nodo u i valori

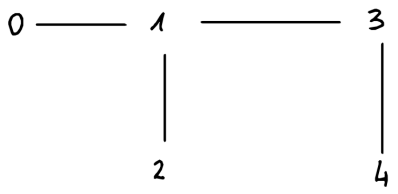
$$\begin{aligned} d^+(u) &:= \sum_{j=0}^{\#V-1} A_{u,j} \\ d^-(u) &:= \sum_{i=0}^{\#V-1} A_{i,u}. \end{aligned} \tag{14}$$

Per il teorema dell'handshaking vale

$$\sum_{u \in V} d^+(u) = \sum_{u \in V} d^-(u) = 2 \cdot \#E. \tag{15}$$

10.5 Liste di adiacenza

Un'altra rappresentazione di un grafo è tramite le sue liste di adiacenza: si costruisca un array di $\#V$ elementi in cui ogni nodo u è collegato a una lista di $d^+(u)$ elementi, contenenti $N(u)$ (ordinato in modo crescente).



0	$\to 1$
1	$\to 0, 2, 3$
2	$\to 1$
3	$\to 1, 4$
4	$\to 3$

10.6 Visita di un grafo: i cammini

Si definisce cammino di lunghezza k una k-upla di nodi $u_1, \ldots, u_k \in V$ con $k-1$ archi $a_1, \ldots, a_{k-1} \in E$, con $a_i = (u_i, u_{i+1})$. Un cammino è detto ciclo se $u_1 = u_k$ e $a_i \neq a_j \ \forall \ i, j = 1, \ldots, k-1$.

Si dice che un grafo G è connesso se $\forall \ u, v \in V \ \exists \ C$ cammino tale che $u, v \in C$. Si dice che un grafo G è ciclico se G ammette un ciclo, altrimenti è detto aciclico. Un albero è un grafo non orientato aciclico e connesso.

10.7 La Depth First Search o visita in profondità

La Depth First Search (DFS) permette di attraversare un grafo G tramite le liste di adiacenza visitando ciascun nodo u una sola volta:

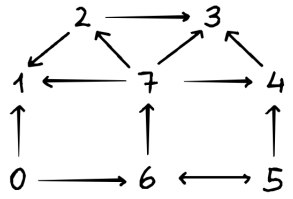
```
DFSric(u){
        raggiunto[u] = TRUE;
        for(auto v: listaAdj[u]){
                if(!raggiunto[v]){
                        DFSric(v);
                }
        }
}
```

richiamata nel semplice algoritmo

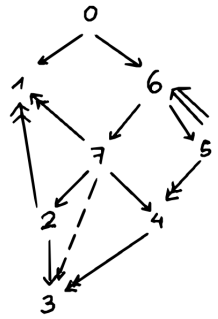
```
Scansione(G){
        for(s = 0; s < n; s++){
                raggiunto[s] = FALSE;
        }
        for(s = 0; s < n; s++){
                if(!raggiunto[s]){
                        DFSric(s);
                }
        }
}
```

Si consideri ad esempio il grafo seguente.

0	$\to 1, 6$
1	$\to \varnothing$
2	$\to 1, 3$
3	$\to \varnothing$
4	$\to 3$
5	$\to 4, 6$
6	$\to 7, 5$
7	$\to 1, 2, 3, 4$



L'applicazione di DFSric(0) produce



con la seguente legenda:

$\rightarrow$ tree edge: albero costruito tramite la DFSric;

$\dashrightarrow$ forward edge: da antenato, non genitore, a discendente;

$\Rightarrow$ backward edge: da discendente ad antenato;

$\twoheadrightarrow$ cross edge: da un nodo visitato dopo a un nodo visitato prima.

Nota Nei grafi non orientati gli archi (u, v) forward e cross non esistono poiché vengono attraversati a partire da v invece che da u.

10.7.1 Connessione e ciclicità tramite la DFS

Tramite la DFS si possono stabilire connessione e ciclicità di un grafo.

Per quanto riguarda la connessione, si analizzino i seguenti due casi:

- Se G non è orientato, è sufficiente partire da qualunque nodo u e verificare al termine di DFSric(u) che tutti i nodi siano stati raggiunti. Ciò garantisce la connessione o meno di G;
- Se G è orientato, esiste una nozione più forte, quella di componente fortemente connessa (SCC), cioè $\forall\ u, v \in \text{SCC}\ \exists\ C$ cammino orientato da u a v. Le SCC si trovano tramite un'implementazione più sofisticata della DFS.

Per quanto riguarda la ciclicità, sono fatti equivalenti:

(i) G ciclico;

(ii) Ogni albero DFS su G ha un arco backward;

(iii) Esiste un albero DFS su G con un arco backward.

Dimostrazione

(ii)$\Rightarrow$(iii) Ovvio;

(iii)$\Rightarrow$(i) Il fatto che un albero DFS su G abbia un backward edge implica l'esistenza di un ciclo in G, per cui G ciclico;

(i)$\Rightarrow$(ii) Sia $u_1, \ldots, u_k, u_1$ un ciclo in G, con u_1 primo nodo scoperto da DFSric(u) in tale ciclo, allora u_k è discendente di u_1 e dunque l'arco (u_k, u_1) è un backward edge.

Nota Pertanto, si implementi nel DFS un flag che indichi la presenza o meno di backward tramite due vector booleani aperto-chiuso in modo da identificare la zona del grafo che si sta analizzando.

10.8 Ordinamento topologico

Sia G un grafo orientato e aciclico (DAG), un ordine topologico è una mappa $\eta : V \to A$, con A array di dimensione $n = \#V$, tale che $(u, v) \in E \Rightarrow \eta(u) < \eta(v)$.

Nota Si noti che la condizione di aciclicità di G è necessaria per la buona definizione di η.

10.8.1 Algoritmo di ordinamento topologico e DFS ordinante

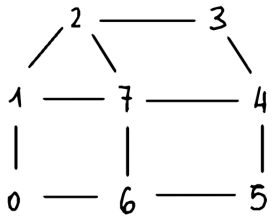
```
OrdinamentoTopologico(){
        for(s = 0; s < n; s++){
                raggiunto[s] = FALSE;
        }
        contatore = n - 1;
        for(s = 0; s < n; s++){
                if(!raggiunto[s]){
                        DFSricOrdina(s);
                }
        }
}
```

con

```
DFSricOrdina(u){
        raggiunto[u] = TRUE;
        for(x = listaAdj[u].inizio; x != NULL; x = x.succ){
                v = x.dato;
                if(!raggiunto[v]){
                        DFSricOrdina(v);
                }
        }
        eta(u) = contatore;
        contatore--;
}
```

10.9 La Breadth First Search o visita in ampiezza

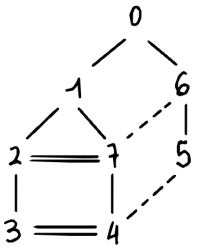
Sia $d : V \times V \to \mathbb{N},\ (x, y) \mapsto$ lunghezza del cammino più breve da x a y.



La funzione BFS visita in ordine:

1. 0;
2. 1, 6;
3. 2, 5, 7;
4. 3, 4,

cioè a ogni passo si visitano i nodi a distanza 1 dal passo precedente, ossia l'ordine di visita è dato da $d(0, \cdot)$.




```
BFS(s){
        for(u = 0; u < n; u++){
                visitato[u] = FALSE;
        }
        C.enqueue(s);
        visitato[s] = TRUE;
        d[s] = 0;
        while(!C.empty()){
                u = C.dequeue();
                for(auto v : adj[u]){
                        if(!visitato[v]){
                                C.enqueue(v);
                                d[v] = d[u] + 1;
                                visitato[v] = TRUE;
                        }
```

```
                    }
            }
}
```

con C che indica la coda e d la distanza da s.

10.10 Complessità di DFS e BFS

Per l'handshaking lemma sia DFS che BFS sono algoritmi $O(n+1)$.

10.11 Distanza media e diametro

Sia G grafo, si definiscono distanza media

$$\sum_{u \neq v} \frac{d(u,v)}{n(n-1)} \tag{16}$$

e diametro

$$\max_{u \neq v} d(u,v) = \max_{s} \max_{v} d(s,v), \tag{17}$$

con $\max d(s,v)$, detta eccentricità, data da BFS(s). Ciò richiede $O(n^2 + nm)$, per cui:

- Se G è denso, si usano metodi basati sulla moltiplicazione tra matrici;
- Se G è sparso, si ha di fatto $O(n^2)$.

10.12 Grafi pesati

Sia $G = (V, E, W)$, con $W : E \to \mathbb{R}$ funzione peso, si hanno la matrice di adiacenza

$$A_{i,j} = \begin{cases} W(i,j) \text{ se } (i,j) \in E \\ 0 \text{ altrimenti} \end{cases} \tag{18}$$

e le liste di adiacenza

i	$\to \ldots, (j, W(i,j)), \ldots$

Conseguentemente, i cammini divengono pesati, per cui $C = (u_1, \ldots, u_k)$ ha peso $\sum_{i=1}^{k-1} W(u_i, u_{i+1})$.

Si definisce la distanza $d : V \times V \to \mathbb{R}$, $(u,v) \mapsto$ peso del cammino da u a v di peso minimo. Pertanto, il BFS non riesce a stabilire la distanza di grafi pesati. Per farlo si utilizza l'algoritmo di Dijkstra.

10.12.1 Algoritmo di Dijkstra per i cammini minimi

Sia C un cammino minimo da s a t, allora qualsiasi porzione di C è il cammino minimo tra i suoi estremi.

Fissato $s \in V$, si partizioni $V = V_1 \overset{\circ}{\cup} V_2$ con V_1 nodi di cui già si conosce $d(s, \cdot)$ e V_2 i restanti. Tale partizione è detta taglio. Si definisca ora $E(V_1, V_2) := \{(u, v) \in E : \ u \in V_1, \ v \in V_2\}$ e si consideri $v \in V_2$ tale che $\exists \ u \in V_1 : \ (u, v)$ è l'arco minimo di $E(V_1, V_2)$. Si proceda ricorsivamente con $V_1 \leftarrow V_1 \cup \{v\}, \ V_2 \leftarrow V_2 \setminus \{v\}$:

```
Dijkstra(s){
        for(u = 0; u < n; u++){
                pred[u] = -1;
                dist[u] = infinito;
        }
        pred[s] = s;
        dist[s] = 0;
        for(u = 0; u < n; u++){
                elemento.peso = dist[u];
                elemento.dato = u;
                PQ.enqueue(elemento);
        }
        while(!PQ.empty()){
                e = PQ.dequeue();
                v = e.dato;
                for(x = listaAdj[v].inizio; x != NULL; x = x.succ){
                        u = x.dato;
                        if(dist[u] > dist[v] + x.peso){
                                dist[u] = dist[v] + x.peso;
                                pred[u] = v;
                                PQ.decreasekey(u, dist[u]);
                        }
                }
        }
}
```

con infinito = somma dei pesi+1 e PQ priority queue, che contiene i nodi da scoprire in ordine.

10.13 Minimum spanning tree

Sia $G = (V, E, W)$ grafo pesato, un sottoinsieme $T \subseteq E$ si dice spanning tree se è aciclico (cioè è un albero) e $\#T = \#V - 1$, cioè tutti i nodi di G sono in T. Gli alberi DFS, BFS e Dijkstra sono spanning tree.

Per grafi completi di n nodi, il numero di spanning tree è almeno $(n-1)!$, per cui minimizzare il costo di spanning tree per grafi pesati appare computazionalmente complesso.

10.13.1 Regola del ciclo

Sia $T = \text{MST}(G),\ \forall\ e \in E \setminus T$ vale $w(e) \geq w(e')\ \forall\ e' \neq e$ nel ciclo formato da e in $T \cup \{e\}$, con $w(\cdot)$ funzione peso, altrimenti $T' = T \cup \{e\} \setminus \{e'\}$ sarebbe $\text{MST}(G)$, assurdo per minimalità di T per ipotesi.

10.13.2 Regola del taglio

Siano $T = \text{MST}(G),\ E(V_1, V_2) = \{(u, v) \in E :\ u \in V_1,\ v \in V_2\}$ taglio su G (cioè $V = V_1 \mathring{\cup} V_2$), allora $T \cap E(V_1, V_2) \neq \varnothing$, infatti $\exists\ e \in T \cap E(V_1, V_2) :\ \forall\ e' \in E(V_1, V_2)$ vale $w(e) \leq w(e')$.

Dimostrazione Se fosse $w(e') < w(e)$, sia $\hat{e} \in T \cap E(V_1, V_2)$ l'arco nel ciclo formato da e' in $T \cup \{e'\}$, allora $w(T \cup \{e'\} \setminus \{e\}) < w(T)$, assurdo.

Nota Non necessariamente $\hat{e} = e$, ma se $w(\hat{e}) \geq w(e)$, allora $w(\hat{e}) > w(e')$.

10.13.3 Algoritmo di Jarnik-Prim

```
JarnikPrim(){
        for(int u = 0; u < n; u++){
                incluso[u] = FALSE;
                pred[u] = u;
                elemento.peso = peso[u] = infinito;
                elemento.dato = u;
                PQ.enqueue(elemento);
        }
        while(!PQ.empty()){
                elemento = PQ.dequeue();
                v = elemento.dato;
                incluso[v] = TRUE;
                mst.InserisciFondo(<pred[v],v>);
                for(x = listaAdj[v].inizio; x != NULL; x = x.succ){
                        u = x.dato;
                        if(!incluso[u] && x.peso < peso[u]){
                                pred[u] = v;
                                peso[u] = x.peso;
                                PQ.decreaseKey(u, peso[u]);
```

```
                        }
                }
        }
}
```

Si sceglie dunque, a ogni taglio, il minimo degli archi minimi sfruttando la regola del taglio. Tra due nodi già presi si ignorano gli archi per la regola del ciclo.

Ciò richiede $O(n \log n)$.

10.13.4 Algoritmo di Kruskal

Si ordinino gli archi in modo crescente (in base al peso) e si costruisca un albero che contenga tutti i nodi del grafo e che a ogni passo includa nodi ancora non scoperti:

```
Kruskal(){
        for(int u = 0; u < n; u++){
                for(x = listaAdj[u].inizio; x != NULL; x = x.succ){
                        v = x.dato;
                        elemento.dato = <u,v>;
                        elemento.peso = x.peso;
                        PQ.enqueue(elemento);
                }
                set[u] = NuovoNodo();
                Crea(set[u]);
        }
        while(!PQ.empty()){
                elemento = PQ.dequeue;
                <u,v> = elemento.dato;
                if(!Appartieni(set[u], set[v])){
                        Unisci(set[u], set[v]);
                        mst.InserisciFondo(<u,v>);
                }
        }
}
```

10.13.5 Union-find

Considerati n insiemi disgiunti, ciascuno dei quali è un singoletto, si utilizzino le seguenti operazioni:

- unisci(u, v): se set$[u] \neq$ set$[v]$ si aggiunge set$[u] \mathbin{\mathring{\cup}}$ set$[v]$ alla lista degli insiemi e si rimuovono set$[u]$, set$[v]$. Ogni set è un vector rappresentato dal primo elemento, per cui unisci deve modificare il rappresentante a tutti gli elementi di uno dei due set, sostituendolo con quello dell'altro set (per ottimizzare tale processo si modifica il rappresentante degli elementi del set più corto);

- appartieni(set[u], set[v]): verifica se set[u] = set[v] in $O(1)$ confrontando i rappresentanti.

Per quanto riguarda il costo di unisci, poiché si possono effettuare al più $n-1$ unisci e benché ognuno abbia $O(n)$ al caso pessimo, il costo totale è $O(n \log n)$, infatti, dopo i unioni, ogni elemento è in un set di taglia almeno 2^i, in quanto se a un elemento viene cambiato rappresentante, allora esso finisce in un set di dimensione almeno doppia rispetto a quello di partenza (si ricordi che si modifica il set più corto). Poiché ogni insieme ha dimensione al più n, allora $2^i \leq n$, per cui $i \leq \log_2 n$, quindi ogni elemento cambia rappresentante $O(\log n)$ volte e dunque il costo di $n-1$ unisci è $O(n \log n)$.

11 Programmazione dinamica

```
int Fib(int n){
        if(n <= 1){
                return n;
        }
        return Fib(n - 1) + Fib(n - 2);
}
```

Si ha il problema dell'esplosione combinatoria delle chiamate ricorsive che calcolano lo stesso valore, per cui:

```
int Fib(int n){
        array<int> F(n + 1);
        F[0] = 0;
        F[1] = 1;
        for(int i = 2; i <= n; i++){
                F[i] = F[i - 1] + F[i - 2];
        }
        return F[n];
}
```

che richiede $O(n) = O(2^k)$, con $k = \log_2 n$ numero di bit nella rappresentazione binaria di n.

11.1 Longest common subsequence

Siano $A = a_0, \ldots, a_{m-1}$, $B = b_0, \ldots, b_{n-1}$ sequenze ordinate, si definisce sottosequenza$(A) = \{(a_{i_1}, \ldots, a_{i_k}) : 0 \leq i_1 < \ldots < i_k \leq m-1\}$ e analogo per B.

Si dice che X è sottosequenza comune di A, B se è sottosequenza sia di A che di B. Nella ricerca della sottosequenza comune più lunga (LCS), poiché il numero delle

sottosequenze comuni cresce esponenzialmente, si fa ricorso alla programmazione dinamica.

11.1.1 Regola ricorsiva

- Caso base: $A = \varepsilon \ \vee \ B = \varepsilon$ (stringa vuota), allora $\text{LCS}(A, B) = \varepsilon$;
- Passo induttivo: $A, B \neq \varepsilon$, nell'analisi di $A[i-1], B[j-1]$:
  - Se $A[i-1] = B[j-1]$, allora $\text{LCS}(i, j) := \text{LCS}((a_0, \ldots, a_{i-1})(b_0, \ldots, b_{j-1})) = 1 + \text{LCS}(i-1, j-1)$;
  - Se $A[i-1] \neq B[j-1]$, allora $\text{LCS}(i, j) = \max\{\text{LCS}(i-1, j),\ \text{LCS}(i, j-1)\}$.

```
LCS(a,b){
 for(i = 0; i <= m; i++){
  lunghezza[i][0] = 0;
 }
 for(j = 0; j <= n; j++){
  lunghezza[0][j] = 0;
 }
 for(i = 1; i <= m; i++){
  for(j = 1; j <= n; j++){
   if(a[i - 1] == b[j - 1]){
    lunghezza[i][j] = lunghezza[i - 1][j - 1]+1;
   } else if(lunghezza[i][j - 1] > lunghezza[i - 1][j]){
    lunghezza[i][j] = lunghezza[i][j - 1];
   } else {
    lunghezza[i][j] = lunghezza[i - 1][j];
   }
  }
 }
}
```

con lunghezza matrice $m \times n$.

```
StampaLCS(i,j){
        if((i > 0) && (j > 0)){
                <i',j'> = indice[i][j];
                StampaLCS(i',j');
                if((i' == i - 1) && (j' == j - 1)){
                        print a[i - 1];
                }
        }
```

}

con chiamata StampaLCS(m, n). La sintassi $< i', j' >$ indica il make_pair, per cui $i' = x$.first, $j' = x$.second.

11.2 Knapsack problem

Si considerino n oggetti numerati, ognuno dei quali ha un peso $p_i \in \mathbb{N}$ e un valore $v_i \in \mathbb{N}$. Si vogliono prendere gli oggetti in $I \subseteq \{0, \ldots, n-1\}$ e inserirli in uno zaino di possanza P in modo da ottenere

$$\max_I \sum_{i \in I} v_i$$

con $\sum_{i \in I} p_i \leq P$.

Ciò richiede $O(nP)$, pseudo-polinomiale.

11.2.1 Relazione ricorsiva

Considerato l'oggetto i, o lo si prende oppure no:

```
Bisaccia(peso, valore, possanza){
 for(i = 0; i <= n; i++){
  for(j = 0; j <= possanza; j++){
   V[i][j] = 0;
  }
 }
 for(i = 1; i <= n; i++){
  for(j = i; j <= possanza; j++){
   V[i][j] = V[i - 1][j];
   if(j >= peso[i - 1]){
    m = V[i - 1][j - peso[i - 1]] + valore[ i - 1];
    if(m > V[i][j]){
     V[i][j] = m;
    }
   }
  }
 }
 return V[n][possanza];
}
```

con $V[i][j] = \max(V[i-1][j],\ V[i-1][j-p_{i-1}] + v_{i-1})$ massimo valore in uno zaino di possanza j usando gli oggetti $0, \ldots, i-1$.

12 Intrattabilità computazionale

12.1 Alcuni problemi intrattabili

12.1.1 Ciclo hamiltoniano

Sia G un grafo, si definisce ciclo hamiltoniano un ciclo che attraversa tutti i nodi di G una e una sola volta. Di questo problema si ha una soluzione in $O(n!)$, ma non in tempo polinomiale.

Cicli e cammini euleriani Si definisce ciclo euleriano un ciclo che attraversa tutti gli archi una e una sola volta e analoga definizione si ha per il cammino euleriano (che a differenza del ciclo non deve necessariamente terminare nel punto di partenza).

Se G è connesso, allora:

- Esiste ciclo euleriano $\iff$ tutti i nodi hanno grado pari;
- Esiste cammino euleriano $\iff$ tutti i nodi tranne eventualmente 2 hanno grado pari.

12.1.2 Colorazione di mappe planari: il problema dei tre colori

Si consideri una mappa planare e la si modelli tramite un grafo con $V = \{\text{paesi}\}$, $E = \{(i,j) \in V^2 : \ i,j \text{ confinano}\}$. Il fatto che si possa o meno colorare la mappa con 3 colori (due paesi confinanti non possono avere medesimo colore) è intrattabile, ossia non esiste un algoritmo in poly-time.

12.1.3 Satisfaction problem

Si considerino n variabili booleane $x_1, \ldots, x_n \in \{0,1\}$, si definiscano:

- Letterale: $x_i, \overline{x_i}$;
- Clausola: or di letterali, ad esempio $x_3 \vee \overline{x_4} \vee x_7 \vee x_8$;
- Formula in forma normale congiuntiva (CNF): and di clausole, ad esempio $\varphi(x_1,x_2,x_3,x_4) = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$;
- Assegnamento: funzione $\tau : \{x_1, \ldots, x_n\} \to \{0,1\}$. Si dice che τ soddisfa φ se $\varphi(\tau(x_1), \ldots, \tau(x_n)) = 1$.

Il fatto che esista o meno τ che soddisfa φ è computazionalmente intrattabile.

12.2 Problemi decisionali

Un problema si dice decisionale se la sua risposta è binaria (sì/no, $0 \vee 1$). Anche l'input è una stringa binaria in $\{0,1\}^*$, per cui i problemi decidibili sono $\Pi \subseteq \{0,1\}^*$, cioè $\Pi = \{x \in \{0,1\}^* : \ \Pi(x) = 1\}$.

Si definiscono:

- $P = \text{poly} := \{\Pi \subseteq \{0,1\}^* : \ x \overset{?}{\in} \Pi$ richiede tempo $\text{poly}(|x|)\}$;
- $NP := \{\Pi \subseteq \{0,1\}^* : \ x \overset{?}{\in} \Pi$ ammette certificato $\text{poly}(|x|)\}$, dove con certificato poly si intende che fissata un'istanza $x \in \{0,1\}^*$ si è in grado di stabilire in tempo poly se $x \in \Pi$:
  - Se $x \in \Pi$, allora $\exists\, y \in \{0,1\}^* : \ |y| = p(|x|)$ e $V(x,y) = 1$;
  - Se $x \notin \Pi$, allora $\forall\, y \in \{0,1\}^*$ vale $V(x,y) = 0$,

  dove V è un verificatore polinomiale.

Vale $P \subseteq NP$. Il problema è stabilire se valga anche $P \supseteq NP$.

12.2.1 Problemi NP-completi

Siano $\Pi_1, \Pi_2 \subseteq \{0,1\}^*$, si dice che $\Pi_1 \propto \Pi_2$ se $\exists\, T$ algoritmo deterministico poly tale che $\forall\, x \in \{0,1\}^*$ vale $x \in \Pi_1 \iff T(x) \in \Pi_2$.

La realzione $\propto$ è riflessiva e transitiva, ma non è noto se sia simmetrica.

Sia $\Pi_1 \propto \Pi_2$, allora $\Pi_2 \in P \Rightarrow \Pi_1 \in P$, per cui $\Pi_1 \notin P \Rightarrow \Pi_2 \notin P$. Questo risultato è noto come teorema di Cook-Levin.

A Randomizzazione e probabilità

A.1 Variabili aleatorie e valore atteso

Sia (Ω, P) uno spazio di probabilità discreto, il valore atteso della variabile aleatoria X è

$$E[X] = \sum_{x \in \Omega} x \cdot P(X = x) \tag{19}$$

e vale $E[aX + bY] = aE[X] + bE[Y]\ \forall\, a, b$ indipendenti dall'esperimento.

Si definisce variabile indicatrice dell'evento A la variabile aleatoria

$$X = \begin{cases} 1 \text{ se } A \\ 0 \text{ altrimenti} \end{cases}. \tag{20}$$

Vale $E[X] = P(A)$.

A.1.1 Problema dello streaming o del segretario

Si considerino n servizi streaming, tra cui si intende scegliere il migliore. Si adotti la seguente strategia: si analizzi un servizio alla volta e si acquisti il nuovo se migliore di quello in uso (il primo si acquista a prescindere). Ad esempio, nella stringa

$$1\ 7\ 2\ 3\ 4\ 6\ 5$$

si effettuano due acquisti: 1 e 7 (i valori indicano i livelli di gradimento). Inece, nella stringa

$$2\ 5\ 3\ 4\ 6\ 7\ 1$$

si effettuano quattro acquisti: 2, 5, 6, 7.

Sia $X_i = \begin{cases} 1 \text{ se si acquista l'}i\text{-esimo servizio} \\ 0 \text{ altrimenti} \end{cases}$ $\forall\ i = 1, \dots, n$. Sia poi $X = \sum_{i=1}^{n} X_i$, essa restituisce il numero di volte in cui si effettuano acquisti. Vale $E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} P(X_i = 1)$. Fissato i, la probabilità che si acquisti l'i-esimo servizio coincide con la probabilità che esso sia migliore degli $i-1$ precedenti, ovvero

$$P(X_i = 1) = \frac{\text{casi favorevoli}}{\text{casi possibili}} = \frac{\text{permutazioni di } i \text{ elementi in cui l'ultimo è il massimo}}{\text{permutazioni di } i \text{ elementi}} = \frac{(i-1)!}{i!} = \frac{1}{i},$$

per cui $E[X] = \sum_{i=1}^{n} \frac{1}{i} \sim \log n$.

A.1.2 Generazione di una permutazione di n elementi con probabilità uniforme

```
void perm(vector <int> &A){
        int n = A.size();
        for(i = 0; i < n-1; i++){
                j = random(i, n-1);
                scambia(i,j);
        }
}
```