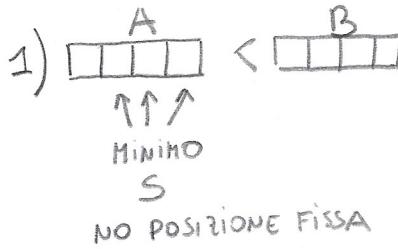


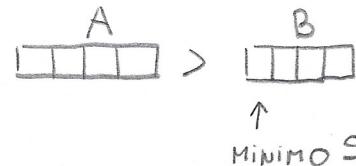
ESAME 20/06/22

①



RICERCA MINIMO: $O(m_A)$

CANCELLAZIONE MINIMO: $O(\log m_A)$



RICERCA MINIMO: $O(1)$

CANCELLAZIONE MINIMO: $O(\log m_B)$

2) CLIQUE è NP-completo

ISOMORFISMO -DI GRAFI non nota essere NP-completo

Se ISOMORFISMO-DI-GRAFI \leq_p CLIQUE:

- Non dimostrerebbe $P=NP$:
- Sarebbe solo un'altra riduzione a un problema NP-completo
- Non ci direbbe nulla sulla complessità di ISOMORFISMO stesso

Il premio richiederebbe di dimostrare che: $P=NP$ (tutti i problemi NP sono risolvibili in tempo polinomiale).

3) L'algoritmo di Dijkstra esegue esattamente m operazioni di relax su un grafo con m archi.

Questo perché:

- Ogni arco (u,v) viene considerato ESATTAMENTE una volta
- Quando un nodo u viene estraatto dalla coda di priorità, tutti i suoi archi uscenti vengono rilassati.

È possibile ridurre il numero di operazioni di RELAX senza alterarne la correttezza interrompendo l'algoritmo quando il nodo di destinazione viene estraatto.

Nel caso in cui si volesse arrivare ad un MST estraendo tutti i nodi allora non si possono ridurre le operazioni di RELAX.

PART II

4) Floyd-Warshall (W)

$m \leftarrow \text{rows}(W)$

$D^{(0)} \leftarrow W$

for $x=1$ to K

for $i=1$ to m

for $j=1$ to m

$$d_{ij}^{(x)} = \min \{ d_{ij}^{(x-1)}, d_{ix}^{(x-1)} + d_{xj}^{(x-1)} \}$$

return $D^{(m)}$

CORRETTITÀ

DIMOSTRAZIONE PER INDUZIONE

CASO BASE ($X=0$)

d_{ij}^0 rappresenta i cammini diretti (senza vertici intermedi) correttamente inizializzato con pesi degli archi e 0 sulla diagonale.

PASSO INDUTTIVO

Assumiamo che d^{x-1} contenga i cammini minimi con al più $x-1$ vertici intermedi.

Per provare d_{ij}^x consideriamo due possibilità:

- 1) Non usare il vertice x come intermedio (d_{ij}^{x-1})
- 2) Usare il vertice x come intermedio ($d_{ix}^{x-1} + d_{xj}^{x-1}$)

La scelta del minimo garantisce l'ottimalità della sottostruzione.

$$T(m) = O(K \cdot m^2)$$

3) a) METODO DELL'ALBERO DI RICORSIONE

$$2 T\left(\frac{m}{2}\right) + m^3 \quad \text{se } m > 1$$

Livello 0 Costo: m^3

Livello 1 2 nodi, ciascuno con costo $\left(\frac{m}{2}\right)^3 = \frac{m^3}{8}$ Costo totale: $2 \cdot \frac{m^3}{8} = \frac{m^3}{4}$

Livello 2 4 nodi, ciascuno con costo $\left(\frac{m}{4}\right)^3 = \frac{m^3}{64}$ Costo totale: $4 \cdot \frac{m^3}{64} = \frac{m^3}{16}$

Livello K 2^K nodi, ciascuno con costo $\left(\frac{m}{2^K}\right)^3 = \frac{m^3}{8^K}$ Costo totale: $2^K \cdot \frac{m^3}{8^K} = \frac{m^3}{4^K}$

L'albero termina quando $\frac{m}{2^K} = 1 \rightarrow K = \log_2 m$

$$T(m) = \sum_{K=0}^{\log_2 m} \frac{m^3}{4^K} = m^3 \sum_{K=0}^{\log_2 m} \frac{1}{4^K} = m^3 \cdot \frac{1 - \left(\frac{1}{4}\right)^{\log_2 m + 1}}{\frac{3}{4}} = \frac{4}{3} m^3 \left[1 - \left(\frac{1}{4}\right)^{\log_2 m + 1} \right] =$$

$$\left(\frac{1}{4}\right)^{\log_2 n+1} = \underbrace{\left(\frac{1}{4}\right)^{\log_2 n}}_{n^{\log_2 \frac{1}{4}} = n^{-2}} \cdot \frac{1}{4} = \frac{1}{4} n^{-2}$$

$$= \frac{4}{3} n^3 \left(1 - \frac{1}{4} n^{-2}\right) = \underbrace{\frac{4}{3} n^3}_{\text{TERMINI DOMINANTE}} - \frac{1}{3} n$$

$$T(n) = \Theta(n^3)$$

b) METODO DELL'ITERAZIONE

$$2T\left(\frac{m}{2}\right) + 1 \quad \text{se } m \geq 1$$

$$T(m) = 2T\left(\frac{m}{2}\right) + 1 \quad \text{PRIMO PASSO}$$

$$T(m) = 2\left(2T\left(\frac{m}{4}\right) + 1\right) = 4T\left(\frac{m}{4}\right) + 2 + 1 \quad \text{SECONDO PASSO}$$

$$T(m) = 4\left(2T\left(\frac{m}{8}\right) + 1\right) + 2 + 1 = 8T\left(\frac{m}{8}\right) + 4 + 2 + 1 \quad \text{TERZO PASSO}$$

$$T(m) = 2^K T\left(\frac{m}{2^K}\right) + \sum_{i=0}^{K-1} 2^i \quad \text{PASSO K-ESIMO}$$

$$\text{CASO BASE : } \frac{m}{2^K} = 1 \rightarrow m = 2^K \rightarrow \log_2 m = K$$

$$T(m) = 2^{\log_2 m} T(1) + \underbrace{\sum_{i=0}^{\log_2 m - 1} 2^i}_{\frac{2^{\log_2 m - 1 + 1} - 1}{2 - 1}} = m - 1$$

$$= m + m - 1 = 2m - 1$$

$$T(m) = \Theta(m)$$

1) void eliminaMinonK (PTree t, int k) {

PNode iter, temp;

iter = t->root;

while (iter != nullptr && iter->key != k) {

if (iter->key < k) {

// Elimina il sottoalbero sinistro (tutti i nodi minori)

rimuovi (iter->left);

iter->left = nullptr;

// Sposta il nodo corrente al sottoalbero destro

temp = iter;

transplant (T, iter, iter->right);

iter = iter->right;

// Libera la memoria del nodo eliminato

delete (temp);

}

else {

// Procedi nel sottoalbero sinistro

iter = iter->left;

}

// Se trovato il nodo con chiave k, elimina eventuali residui minori

if (iter != nullptr) {

rimuovi (iter->left);

iter->left = nullptr;

}

void rimuovi (PNode u) {

if (u != nullptr) {

rimuovi (u->left);

rimuovi (u->right);

delete u;

}

typedef struct Node {

int key;

Node* left;

Node* right;

} * PNode;

typedef struct Tree {

PNode root;

} * PTree;

2) a) Per soddisfare la condizione $|A[i] - A[i+1]| \geq |A[i+1] - A[i+2]|$, gli elementi nelle prime due posizioni devono essere gli elementi massimo e minimo dell'array, in qualsiasi ordine.

MOTIVAZIONE

- La differenza massima possibile nell'array è $\max(A) - \min(A)$
- Per massimizzare la prima differenza e permettere alle successive di decrescere, dobbiamo iniziare con queste coppia
- Quindi $A[1] = \max(A)$ e $A[2] = \min(A)$ oppure $A[1] = \min(A)$ e $A[2] = \max(A)$

b) PSEUDO-CODICE:

1. Trova il minimo (\min) e il massimo (\max) in $A \rightarrow O(n)$

2. Ordina A in ordine crescente $\rightarrow O(n \log n)$

3. Inizializza due puntatori: $\text{left} = 0$, $\text{right} = n-1$

4. Crea un nuovo array result

5. Alternà elementi grandi e piccoli

- $\text{result}[0] = \max$

- $\text{result}[1] = \min$

- $\text{result}[2] = \text{secondo max}$

- $\text{result}[3] = \text{secondo min}$

- e così via...

6. Restituisce result

COMPETTITIVITÀ

$O(n \log n)$ per l'ordinamento

$O(n)$ per la costruzione

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

```

std::vector<int> orangeArrayGeneral (std::vector<int>& A) {
    // Ordina l'array
    sort (A.begin(), A.end());
    int n = A.size();
    std::vector<int> result;
    int left = 0, right = n - 1;
    while (left <= right) {
        if (left == right) {
            result.push_back (A[right]);
        }
        else {
            result.push_back (A[right]);
            result.push_back (A[left]);
        }
        left++;
        right--;
    }
    // Aggiustamento finale per l'ultima coppia
    if (result.size() >= 2 && abs(result[result.size() - 2] - result.back()) > abs(result.back() - result[result.size() - 2])) {
        swap (result[result.size() - 1], result[result.size() - 2]);
    }
    return result;
}

```

CORRETTEZZA

Ordinando e alternando massimi e minimi, garantiamo che le differenze decrescano:
 $(\max - \min) \geq (\text{secondo}-\max - \text{secondo}-\min) \geq \dots$

ESEMPIO

$$A = [5, 2, 7, 4, 1]$$

$$1) \text{ ORDINATO } [1, 2, 4, 5, 7]$$

$$2) \text{ COSTRUZIONE ALTERNATA } [7, 1, 5, 2, 4]$$

$$3) \text{ DIFFERENZE } |7-1|=6, |1-5|=4, |5-2|=3, |2-4|=2 \text{ (decrescenti)}$$

c) PSEUDOCODICE

1. Trova min e max in A $\rightarrow O(n)$
2. Usa counting sort per ordinare A $\rightarrow O(n+k)$ dove $K = \max - \min < n \rightarrow O(n)$
3. Costruisci l'array risultato alternando elementi degli estremi come nel punto b $\rightarrow O(n)$

COMPLESSITÀ

$$T(n) = O(n)$$

```
std::vector<int> orangeArraySmallRange (const std::vector<int>& A) {
```

```
    int min_val = *min_element(A.begin(), A.end());  
    int max_val = *max_element(A.begin(), A.end());  
    int range = max_val - min_val + 1;
```

// Counting Sort

```
    std::vector<int> count(range, 0);
```

```
    for (int num : A) {
```

```
        count[num - min_val]++;
```

}

// Ricostroisci l'array ordinato

```
    std::vector<int> sorted_A;
```

```
    for (int i = 0; i < range; ++i) {
```

```
        for (int j = 0; j < count[i]; ++j) {
```

```
            sorted_A.push_back(i + min_val);
```

}

}

// Costruisci risultato alternando

```
    int m = sorted_A.size();
```

```
    std::vector<int> result;
```

```
    int left = 0, right = m - 1;
```

```
    while (left <= right) {
```

```
        if (left == right)
```

```
            result.push_back(sorted_A[right]);
```

```
        else {
```

```
            result.push_back(sorted_A[right]);
```

```
            result.push_back(sorted_A[left]);
```

}

```
        left++;

```

```
        right--;

```

}

```
    return result;
```

CORRETTEZZA

- Sfruttiamo il fatto che $\max - \min < m$ per usare Counting Sort
- La costruzione dell'array risultato è identica, ma l'ordinamento è più efficiente

ESEMPIO

$A = [10, 11, 12]$ (differenza massima $2 < 3$)

1) COUNTING SORT $[10, 11, 12]$

2) COSTRUZIONE AUTERNATA $[12, 10, 11]$

3) DIFFERENZE $|12-10|=2, |10-11|=1$ (decrescenti)