## 13.1 Greedy

> *The point is, ladies and gentleman, that greed, for lack of a better word, is good. Greed is right, greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed, in all of its forms; greed for life, for money, for love, for knowledge has marked the upward surge of mankind.*
>
> Gordon Gekko, *Wall Street*

*Nota.* Non è difficile scrivere algoritmi ingordi, la parte difficile sta nel dimostrare che restituiscano la soluzione ottimale.

Molti dei problemi che vedremo li abbiamo già visti, ma in questo capitolo tratteremo dei loro casi particolari. Capita spesso che per risolvere casi generali ci si debba avvalere della programmazione dinamica, mentre per casi particolari sia meglio usare algoritmi ingordi.

## 13.2 Introduzione

Sia la programmazione dinamica che gli algoritmi greedy cercano di risolvere problemi di ottimizzazione. Gli algoritmi che li risolvono devono prendere una serie di decisioni e differiscono fra di loro da *come* queste decisioni vengono prese. Nella programmazione dinamica valutiamo tutte le possibili decisioni evitando di rivalutare decisioni (percorsi) già intraprese. Negli algoritmi ingordi, invece, selezioniamo *una sola* fra le possibili decisioni. Quale? Quella che sembra ottima (ovvero *localmente ottima*). È però necessario dimostrare che si ottiene un ottimo globale (ossia una soluzione *globalmente ottima*).

*Nota.* Questo approccio riduce la complessità di dover valutare tutte le possibilità, ma necessita di essere dimostrato.

**Quando applicarla** È consigliato applicare la tecnica greedy quando fra tutte le scelte possibili ne può essere individuata una che porta sicuramente alla soluzione ottima. Deve comunque rimanere valida (come nella programmazione dinamica) la proprietà di *sottostruttura ottima* ovvero che quando viene effettuata una scelta resti un sottoproblema con la stessa struttura del problema principale.

*Nota.* Non tutti i problemi hanno una soluzione greedy.

## 13.3 Insieme indipendente di intervalli non pesati

**Definizione del problema** Dati in input un insieme di intervalli della retta reale $S = \{1, 2, \dots n\}$. Trovare un *insieme indipendente massimale*, ovvero un sottoinsieme di cardinalità massima formato da intervalli disgiunti tra loro.

*Nota* (Proprietà degli intervalli). Gli intervalli sono chiusi a sinistra e aperti a destra.

Abbiamo già risolto questo problema (con la programmazione dinamica), ma il fatto che tutti i pesi siano pari a 1 porta ad una semplificazione del problema.

*Nota.* Per questo particolare problema non esiste un insieme di cardinalità 5 (ossia un insieme contenente cinque elementi).

**Come affrontare il problema**

1. *individuare* la sottostruttura ottima;
2. *scrivere* una definizione ricorsiva per la dimensione della soluzione ottima;

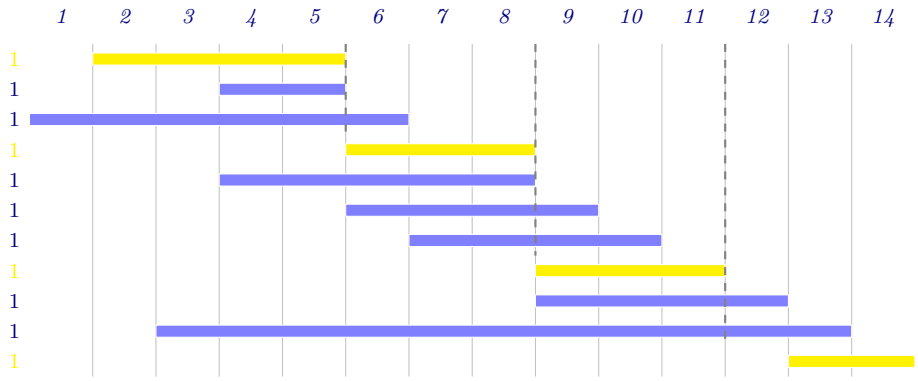



**Figura 13.1:** Una delle possibili soluzioni ingorde al problema dell'insieme indipendente di intervalli. Quali altre possibili soluzioni di cardinalità massima sono presenti? Nota che tutti gli intervalli hanno lo stesso peso.

3. *cercare* una possibile scelta ingorda;
4. *dimostrare* che la scelta presa porti alla soluzione ottima;
5. *scrivere* un algoritmo ricorsivo (spesso sono iterativi) che effettui sempre la scelta ingorda.

### 13.3.1 Individuazione sottostruttura ottima

Assumiamo che gli intervalli siano ordinati per tempo di fine ($b_1 \leqslant b_2 \leqslant \dots b_n$).

Definiamo il sottoproblema $S[i,j]$ come l'insieme di intervalli che iniziano dopo la fine di $i$ e finiscono prima dell'inizio di $j$, ovvero $S[i,j] = \{k \mid b_i \leqslant a_k < b_k \leqslant a_j\}$.

Per scopi implementativi (fungeranno da sentinelle) aggiungiamo due intervalli fittizi che singoleggiano $\pm\infty$, ovvero l'intervallo zeresimo indicato da $b_0$ ($b_0 = -\infty$) e l'intervallo successivo indicato da $n+1$ ($n+1 = +\infty$). In questi termini il problema iniziale corrisponde al problema $S[0, n+1]$.

*Nota.* L'idea è definire problemi a mano a mano sempre più piccoli. E definirli in base agli indici degli intervalli rimanenti.

**Teorema.** *Supponiamo che $A[i,j]$ sia una (poiché non è unica) soluzione ottimale di $S[i,j]$ e sia $k$ un (qualunque) intervallo che appartiene a $A[i,j]$; suddividiamo quindi il problema $S[i,j]$ in due sottoproblemi:*

- *$S[i,k]$: gli intervalli di $S[i,j]$ che finiscono prima di $k$;*
- *$S[k,j]$: gli intervalli di $S[i,j]$ che iniziano dopo di $k$*

*$A[i,j]$ contiene le soluzioni ottimali di $S[i,k]$ e $S[k,j]$ quindi:*

- *$A[i,j] \cap S[i,k]$ è la soluzione ottimale di $S[i,k]$*
- *$A[i,j] \cap S[k,j]$ è la soluzione ottimale di $S[k,j]$*

*Dimostrazione per assurdo.* Se esistesse una soluzione migliore al problema $S[i,j]$, diciamo $A'[i,k]$ e la sostituissi ad $A[i,k]$ ottenerrei una soluzione migliore anche al mio problema originale, ma questo non è possibile poiché se ottenessi una soluzione migliore allora $A[i,k]$ non sarebbe una soluzione ottima, il che è assurdo. □

### 13.3.2 Definizione ricorsiva del costo della soluzione

Partendo dalla definizione ricorsiva della soluzione

$$A[i,j] = A[i,k] \cup \{k\} \cup A[k,j]$$

non possiamo determinare $k$ a priori quindi dobbiamo necessariamente provare tutti i valori.

L'equazione di ricorrenza che si ottiene è la seguente

$$DP[i,j] = \begin{cases} 0 & S[i,j] = \varnothing \\ \max\limits_{k \in S[i,j]} \{\underbrace{DP[i,k]}_{\text{prima di } k} + \underbrace{DP[k,j]}_{\text{dopo } k} + 1\} & \text{altrimenti} \end{cases}$$

dove $DP[i,j]$ è la dimensione del più grande sottoinsieme $A[i,j] \subseteq S[i,j]$ di intervalli indipendenti.

1. se l'insieme di intervalli dati in input è vuoto ($S[i,j] = \varnothing$), allora la dimensione dell'insieme è pari a 0 (caso base);
2. altrimenti, se l'insieme di intervalli di partenza non è vuoto, consideriamo l'insieme di cardinalità massima. L'insieme viene calcolato scegliendo l'intervallo $k$-esimo fra tutti gli intervalli $k$ appartenenti all'insieme di partenza ($k \in S[i,j]$), sommando quindi 1 alla soluzione finale, e chiamando ricorsivamente il problema sugli intervalli rimanenti.

### 13.3.3 Verso una soluzione ingorda

La definizione precedente ci permette di scrivere un algoritmo basato su programmazione dinamica o su memoization di complessità $\mathcal{O}(n^3)$: bisogna necessariamente risolvere tutti i problemi con $i < j$, con costo $\mathcal{O}(n)$ nel caso pessimo.

Nella risoluzione del problema di intervalli *pesati* abbiamo visto un algoritmo di complessità $\mathcal{O}(n \log n)$, il quale è applicabile anche nella risoluzione di questo problema. Questa complessità era data dall'ordinamento del vettore che avveniva prima che i dati venissero processati indipendentemente dall'ordinamento iniziale. Tuttavia è possibile migliorare il nostro algoritmo cercando una soluzione ingorda al nostro problema evitando di valutare tutte le possibili soluzioni. Infatti non è necessario analizzare tutti i possibili valori di $k$.

**Teorema** (scelta ingorda)**.** *Sia $S[i,j]$ un sottoproblema non vuoto, ed indentifichiamo $m$ l'intervallo di $S[i,j]$ con il minor tempo di fine, allora:*

1. *il sottoproblema $S[i,m]$ è vuoto;*
2. *$m$ è compreso in qualche soluzione ottima di $S[i,j]$.*

### 13.3.4 Dimostrazione che è una soluzione ottima

*Dimostrazione.* Fai riferimento alla spiegazione su youtube qui non riportata (importante). □

**Conseguenze** Non è più necessario analizzare tutti i possibili valori di $k$ in quanto faccio una una scelta "ingorda", ma sicura: seleziono l'attività $m$ con il minor tempo di fine. A questo punto non è più necessario analizzare due sottoproblemi; eliminando tutte le attività che non sono compatibili con la scelta ingorda mi resta solo il sottoproblema $S[m,j]$ da risolvere.

### 13.3.5 Scrittura dell'algoritmo

**Algoritmo 1:** Insieme indipendente di intervalli disgiunti

```
SET independentSet(int[] a, int[] b)
    { ordina a e b in modo che b[1] ⩽ b[2] ⩽ ··· ≤ b[n] }
    // O(n) se già ordinati, O(n log n) altrimenti
    SET S ← Set
    S.insert (1) // inserisco il primo intervallo
    int ultimo ← 1 // ultimo intervallo inserito
    from i ← 2 until n do
        if a[i] ⩾ b[ultimo] then
            // gli intervalli sono indipendenti
            S.insert (i) // lo inserisco
            ultimo ← i // lo rendo l'ultimo inserito
    return S
```

**Commento** Divido il resto per il taglio della moneta più grande trovando il numero di monete massimo di quel taglio, dopodiché tolgo il valore della somma quelle monete dal resto.

*Nota.* Questo algoritmo non è applicabile al problema dell'insieme indipendente di intervalli *pesati*.

**Ricapitolando** Abbiamo cercato di risolvere il problema della selezione delle attività tramite programmazione dinamica, prima individuando una sottostruttura ottima, poi scrivendo una definizione ricorsiva per la dimensione della soluzione ottima.

Abbiamo poi dimostrato la proprietà della scelta greedy: dimostrando che per ogni sottoproblema, esiste almeno una soluzione ottima che contiene la scelta greedy. Infine abbiamo scritto un algoritmo iterativo che effettua sempre la scelta ingorda.

## 13.4 Resto

**Definizione del problema** Dati in input un insieme di "tagli" di monete, memorizzati in un vettore di interi positivi $t[1 \dots n]$ ed un intero $R$ rappresentante il resto che dobbiamo restituire. Trovare il più piccolo numero intero di pezzi necessari per dare un resto di $R$ centesimi utilizzando i tagli di cui sopra, assumendo di avere un numero illimitato di monete per ogni taglio.

Più formalmente bisogna trovare un vettore $x$ di interi non negativi tale che

$$R = \sum_{i=1}^{n} x[i] \cdot t[i] \qquad\qquad m = \sum_{i=1}^{n} x[i] \quad \text{è minimo}$$

### 13.4.1 Individuazione sottostruttura ottima

Sia $S(i)$ il problema di dare il resto pari ad $i$. Sia $A(i)$ una soluzione ottima del problema $S(i)$, rappresentata da un multi-insieme (un insieme nel quale lo stesso indice può comparire più di una volta); sia $j \in A(i)$ ($j$ un possibile taglio). Allora, $S(i - t[j])$ è un sottoproblema di $S(i)$, la cui soluzione ottima è data da $A(i) - \{j\}$.

Quest'idea si traduce nella seguente definizione ricorsiva.

### 13.4.2 Definizione ricorsiva del costo della soluzione

Utilizziamo la tabella $DP[0 \dots R]$ per memorizzare le soluzioni e in $DP[i]$ memorizziamo il minimo numero di monete per risolvere il problema $S[i]$.

$$DP[i] = \begin{cases} 0 & i = 0 \\ \min\limits_{1 \leqslant j \leqslant n} \{DP[i - t[j]] \text{ t.c. } t[j] \leqslant i\} + 1 & i > 0 \end{cases}$$

Il numero minimo di monete è 0 se non dobbiamo dare nessun resto (caso base), altrimenti cerchiamo fra tutti i possibili tagli quello che minimizza il numero di monete restituite.

### 13.4.3 Algoritmo basato su programmazione dinamica

**Algoritmo 2:** Programmazione dinamica applicata al problema del resto

```
restoDP(int[] t, int n, int R)
    // t: tagli disponibili
    // n: numero di monete
    // R: il resto da dare

    DP ← new int[0...R]
    S ← new int[0...R]
    DP[0] ← 0 // caso base

    // Riempire la tabella DP
    from i ← 1 until R do
        DP[i] ← +∞

        from j ← 1 until n do // Riempio la tabella
            if t[j] ⩽ i and DP[i − t[j]] + 1 < DP[i] then
                // aggiorno il valore
                DP[i] ← DP[i − t[j]] + 1 // registro il valore
                S[i] ← j // la moneta da utilizzare per risolvere il problema quando il taglio è i

    while R > 0 do // ho resto da dare
        stampa t[S[R]] // stampo la moneta
        R ← R − t[S[R]] // decremento il resto
```

**Commento** $t[j] \leqslant i$ sta a significare che il taglio delle monete che possiamo scegliere dev'essere più piccolo del resto che dobbiamo dare.

**Complessità** $\mathcal{O}(nR)$ dato dai cicli innestati. Una soluzione dipendente dal resto $R$ da dare non è ottimale, si può fare meglio di così.

*Nota.* Quest'algoritmo rappresenta la soluzione generale al problema e, a differenza dell'algoritmo ingordo (che vedremo più avanti), funziona per qualsiasi insieme di tagli di monete.

### 13.4.4 Scelta ingorda

Seleziona la moneta $j$ più grande tale per cui $t[j] \leqslant R$, per poi risolvere il problema $S(R - t[j])$.

### 13.4.5 Scrittura dell'algoritmo

**Algoritmo 3:** Approccio ingordo al problema del resto

```
restoGreedy(int[] t, int n, int R, int[] x)
    { Ordina le monete in modo decrescente }
    // O(n) se già ordinato, O(n log n) altrimenti
    from i ← 1 until n do // O(n)
        // il numero di monete di taglio massimo
        x[i] ← ⌊R / t[i]⌋
        // calcolo il resto rimanente
        R ← R − x[i] · t[i]
    return R
```

**Complessità** Questo algoritmo ha complessità lineare, ossia $\mathcal{O}(n)$.

### 13.4.6 Dimostrazione che è una soluzione ottima

*Nota.* La seguente dimostrazione si riferiscere ai tagli $\bigcirc_1 = 50$, $\bigcirc_2 = 10$, $\bigcirc_3 = 5$, $\bigcirc_4 = 1$.

Sia $x$ una qualunque soluzione ottima; quindi il resto $R$ è esprimibile tramite una certa somma dei nostri possibili tagli, dove il numero dei tagli $m$ è minimo:

$$R = \sum_{i=1}^{n} x[i] \cdot t[i] \qquad\qquad m = \sum_{i=1}^{n} x[i]$$

Sia $m_k$ *la somma delle monete di taglio inferiore* a $t[k]$:

$$m_k = \sum_{i=k+1}^{4} x[i] \cdot t[i]$$

Se dimostriamo che la somma delle monete di taglio inferiore a $k$ è minore del valore del taglio $k$-esimo ($\forall k : m_k < t[k]$), allora la soluzione (ottima) è proprio quella calcolata dall'algoritmo.

$m_*$ denota l'insieme di monete con tagli inferiore a $\bigcirc_*$ centesimi, ad esempio $m_2$ denota l'insieme di monete con tagli inferiore a ($\bigcirc_2 =$) 10 centesimi. Mentre $x[*]$ simboleggia il *numero* di monete di quel taglio presenti nel resto della soluzione ottima, ad esempio $x[3]$ simboleggia il *numero* di monete di $\bigcirc_3 = 5$ centesimi.

$$\begin{aligned}
m_4 &= 0 && < 1 = t[4] \\
m_3 &= 1 \cdot x[4] && < 5 = t[3] \\
m_2 &= 5 \cdot x[3] + m_3 && \leqslant 5 + m_3 < 5 + 5 = 10 = t[2] \\
m_1 &= 10 \cdot x[2] + m_2 && \leqslant 40 + m_2 < 40 + 10 = 50 = t[1]
\end{aligned}$$

*Nota.* Fare riferimento alla spiegazione su youtube qui non riportata.

## 13.5 Approccio ingordo

Cercheremo di risolvere i successivi problemi avendo direttamente un approccio ingordo, senza passare prima per la programmazione dinamica. Come fare? bisogna:

- *evidenziare* i passi di decisione, *trasformando* il problema di ottimizzazione in un problema di "scelte" successive;
- *evidenziare* una possibile scelta ingorda, *dimostrando* che tale scelta rispetta il principio di scelta ingorda;
- *evidenziare* la sottostruttura ottima, *dimostrando* che la soluzione ottima del problema "residuo" dopo la scelta ingorda può essere unito a tale scelta;
- *scrivendo* un algoritmo top-down (anche iterativo), in alcuni casi sarà necessario pre-processare l'input.

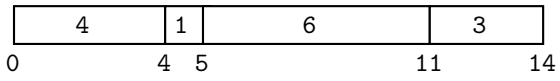
## 13.6 Scheduling

**Definizione del problema** Supponiamo di avere un processore e $n$ job da eseguire su di esso, ognuno caratterizzato da un tempo di esecuzione $t[i]$ noto a priori. Trovare una sequenza di esecuzione (permutazione) che minimizzi il *tempo di completamento medio*.

Dato un vettore $A[1 \dots n]$ contenente una permutazione di $\{1, \dots n\}$, il *tempo di completamento* dell'$h$-esimo job nella permutazione è:

$$T_A(h) = \sum_{i=1}^{h} t[A[i]]$$

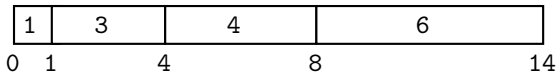
Ad esempio




dove il tempo di completamento medio è pari a

$$\frac{4 + 5 + 11 + 14}{4} = \frac{34}{4} = 8.5$$

Applicando l'algoritmo *Shortest Job First*:




dove il tempo di complemento è pari a

$$\frac{1 + 4 + 8 + 14}{4} = \frac{27}{4} = 6.75 < 8.5$$

**Teorema** (Scelta greedy)**.** *Esiste una soluzione ottima $A$ in cui il job con minor tempo di fine $m$ si trova in prima posizione ($A[1] = m$).*

**Teorema** (Sottostruttura ottima)**.** *Sia $A$ una soluzione ottima di un problema con $n$ job, in cui il job con minor tempo di fine $m$ si trova in prima posizione. La permutazione dei seguenti $n - 1$ job in $A$ è una soluzione ottima al sottoproblema in cui il job $m$ non viene considerato.*

### 13.6.1 Dimostrazione

Fai riferimento alla spiegazione grafica su youtube qui non riportata.

## 13.7 Zaino frazionario

Riproponiamo un problema visto in precedenza, ma stavolta anzichè avere la limitazione di poter prenderere o non prendere un oggetto (chiamato problema dello zaino `0/1`), possiamo prenderne anche frazioni di esso (zaino *reale*).

Un approccio ingordo è quello di ordinare gli oggetti in ordine di *profitto specifico decrescente* (profitto su volume) e prendere quante più frazioni possibili degli elementi fino a riempire l'intera capacità dello zaino.

**Algoritmo 4:** Approccio ingordo al problema del resto

```
zaino(float[] p, float[] v, float C, int n, float[] x)
    { ordina p e v in modo che p[1]/w[1] ⩾ p[2]/w[2] ⩾ ··· ⩾ p[n]/w[n] }
    // O(n) se già ordinato, O(n log n) altrimenti
    from i ← 1 until n do
        x[i] ← min(C / w[i], 1)   // ne prendo solo una frazione?
        C ← C − x[i] · w[i]       // aggiorno la capacità residua
```

$x[i]$ rappresenta la frazione dell'oggetto $i$-esimo che deve essere presa.

### 13.7.1 Correttezza dell'algoritmo

Dimostrazione informale:

1. assumiamo che gli oggetti siano ordinati per profitto specifico decrescente;
2. sia $x$ una soluzione ottima;
3. supponiamo che $x[1] < \min\left(\frac{C}{w[i]}, 1\right) < 1$;
4. allora possiamo costruire una nuova soluzione in cui $x'[1] = \min\left(\frac{C}{w[i]}, 1\right)$ e la proporzione di uno o più oggetti è ridotta di conseguenza;
5. otteniamo così una soluzione $x'$ di profitto uguale o superiore, visto che il profitto specifico del primo oggetto è massimo.

## 13.8 Compressione di Huffman

Per rappresentare in modo efficiente i dati (minimizzare lo spazio su disco occupato, minimizzare il tempo di trasferimento su disco...) abbiamo bisogno di adottare una qualche tecnica di compressione. Una fra le tante possibili è la *codifica dei caratteri*.

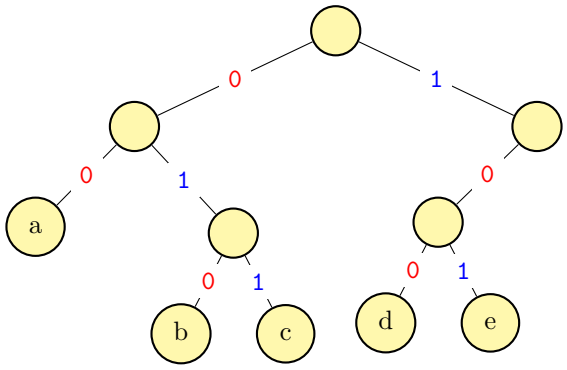
*Nota.* La codifica dei caratteri ha una complessità dipendente in percentuale dalle dimensioni del file di origine.

*Nota.* Un codice non deve essere mai prefisso di un altro codice, in quanto è una condizione necessaria per distinguerli.

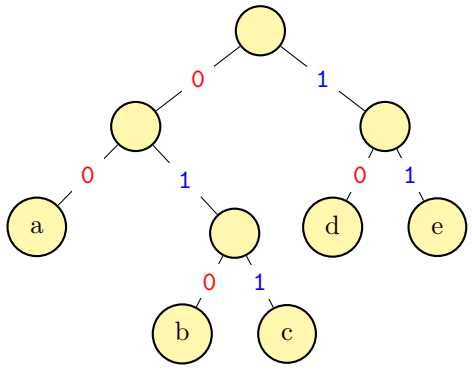
### 13.8.1 Rappresentazione ad albero per la decodifica

Rappresenteremo il nostro risultato tramite un albero. La rappresentazione sulla sinistra non è la migliore possibile. Ed è possibile ottimizzare l'albero comprimendo i cammini per i caratteri `d` e `e` (come puoi vedere

nella figura a destra). Modificando di conseguenza anche la codifica dei caratteri ed ottenendo una codifica ottimizzata.




| a | b | c | d | e |
|---|---|---|---|---|
| 00 | 010 | 011 | 100 | 111 |




| a | b | c | d | e |
|---|---|---|---|---|
| 00 | 010 | 011 | 10 | 11 |

**Definizione formale del problema** Dati in input un file $F$ composto da caratteri di un certo alfabeto (che chiameremo $\Sigma$). Dobbiamo cercare di rappresentare il nostro file con il minor numero di bit possibili. Supponiamo che l'albero $T$ rappresenti la nostra codifica. Per ogni carattere appartente all'alfabeto ($c \in \Sigma$), definiamo come $d_T(c)$ la profondità della foglia che rappresenta il carattere $c$. Quindi il codice per rappresentare $c$ richiederà $d_T(c)$ bit. Infine se $f[c]$ è il numero di occorrenze di $c$ in $F$, allora la *dimensione della codifica* è

$$C[F,T] = \sum_{c \in \Sigma} f[c] \cdot d_T(c)$$

- $f[c]$: frequenza del carattere $c$ nell'alfabeto $\Sigma$;
- $d_T(c)$: profondità del nodo $c$ nell'albero $T$, ovvero i bit necessari per la codifica di $c$.

Quindi una codifica $C$ è data da un particolare albero $T$ dove viene rappresentata e da un particolare file $F$.

### 13.8.2 Principio dell'algoritmo di Huffman

Vogliamo minimizzare la lunghezza della codifica per caratteri che compaiono più frequentemente: assegnando ai caratteri con frequenza minore codici corrispondenti a percorsi più lunghi all'interno dell'albero e a quelli di frequenza maggiore un percorso più corto.

*Nota.* Una certa codifica è progettata per un file specifico.

### 13.8.3 Algoritmo

**Algoritmo 5:** Approccio ingordo al problema del resto

```
TREE huffman(int[] c, int[] f, int n)
    // c[]1n:  caratteri dell'alfabeto
    // f[1...n]:  frequenze dei caratteri
    // n:  dimensione dell'alfabeto
    PRIORITYQUEUE Q ← MinPriorityQueue
    from i ← 1 until n do //                                   O(n)
        Q.inserisci(f[i], Tree(f[i], c[i])) //                 O(log n)
    from i ← 1 until n − 1 do // n:  radice                    O(n)
        // estraggo i 2 caratteri meno frequenti
        z1 ← Q.deleteMin
        z2 ← Q.deleteMin
        // Creo un nuovo nodo
        z ← Tree(z1.f + z2.f, nil)
        z.left ← z1
        z.right ← z2
        // Lo inserisco nella coda
        Q.inserisci(z.f, z)
    return Q.deleteMin
```

Siccome ogni volta devo estrarre i due elementi più piccoli faccio affidamento ad una MinPriorityQueue.

1. inserisco tutte le lettere con la priorità data dalla loro frequenza e con un nodo contenente sia la frequenza che la lettera associata. Dopodiché estraggo i due elementi più piccoli, credo un nuovo nodo contenente i due elementi appena estratti e restituisco il nodo così fatto al chiamante.

**Funzionamento dell'algoritmo**

1. rimuovo i due nodi con frequenza minore;
2. creo un nodo un'etichetta nulla e con frequenza pari alla somma delle frequenze dei nodi eliminati;
3. collego i due nodi con il nodo creato;
4. aggiungo il nodo così creato alla lista, mantenendo l'ordine;
5. si termina quando reasta un solo nodo nella lista;
6. al termine, si etichettano gli archi dell'albero con bit `0`, `1`.

**Complessità** Effettuo $n$ volte operazioni che costano $\log n$, come le operazioni di inserisci o di deleteMin per una complessità totale di $\mathcal{O}(n \log n)$.

### 13.8.4 Dimostrazione di correttezza

**Teorema.** *L'output dell'algoritmo Huffman per un dato file è un codice a prefisso ottimo.*

Scegliere i due elementi con la frequenza più bassa conduce sempre ad una soluzione ottimale. Dato un problema sull'alfabeto $\Sigma$. È possibile costruire un sottoproblema con un alfabeto più piccolo in cui togliamo due caratteri e ne aggiungiamo uno fittizio.

### 13.8.5 Scelta ingorda

`MATERIALE MANCANTE`

## 13.9 Albero di copertura minimi

**Problema** Dato un grafo pesato, determinare come interconnettere tutti i suoi nodi minimizzando il costo del peso associato ai suoi archi (agli archi che andiamo a scegliere). Questo problema prende vari nomi, come albero di copertura minimo o albero di connessione minimo, in inglese *Minimum Spanning Tree*.

**Risorse** If you are curious about who is the cookie monster, then watch the MIT lecture on Minimum Spanning Tree of Professor Erik Demaine.

**Definizione del problema** Dati in input un grafo non orientato e connesso $G = (V, E)$ ed una funzione di peso (che determina il costo di connessione) $w: V \times V \to \mathbb{R}$ (data una coppia di nodi restituisce un numero reale che rappresenta il peso).

*Nota.* Poiché $G$ non è orientato possiamo dire che $w(u, v) = w(v, u)$.

Un albero di copertura di $G$ è un sottografo $T = (V, E_T)$ tale che $T$ è un albero non radicato, i lati dell'albero siano un sottoinsieme di quelli del grafo ($E_T \subseteq E$) e che contenga tutti i vertici di $G$. Il problema consiste quindi nel trovare l'albero di copertura il cui *peso totale* sia minimo rispetto a ogni altro albero di copertura, dove il *peso totale* è dato da:

$$w(T) = \sum_{e \in T} w(e) = \sum_{[u,v] \in E_T} w(u, v)$$

*Nota.* Non è detto che l'albero di copertura minimo sia univoco.

### 13.9.1 Algoritmo generico

L'idea è di accrescere un sottoinsieme $A$ di archi in modo tale che venga sempre rispettata la seguente invariante: $A$ è un sottoinsieme di qualche albero di connessione minimo.

**Teorema 1.** *Un arco $[u, v]$ è detto sicuro per $A$ se $A \cup \{[u, v]\}$ è ancora un sottoinsieme di qualche albero di connessione minimo.*

**Algoritmo 6:** Algoritmo generico per generare un albero di copertura minimo

```
SET mst-generico(GRAPH G, int[] w)
    SET A ← ∅ // parto da un insieme vuoto
    while A non forma un albero di copertura do
        Trova un arco sicuro [u, v]
        A ← A ∪ {[u, v]} // lo aggiungo all'albero
    return A
```

### 13.9.2 Dimostrazione

La dimostrazione banale è lasciata come esercizio al lettore. Scherzo, scherzo! (ma non la riporto comunque)

### 13.9.3 Algoritmo di Kruskal

**Idea** L'idea è quella di ingrandire sottoinsieme disgiunti (qualche idea sulla struttura dati da utilizzare?) di un albero di copertura minimo connettendoli fra di loro fino ad avere l'albero complessivo. Si individua quindi un arco sicuro scegliendo un arco *di peso minimo* fra tutti gli archi che connettono due alberi distinti (distinti da componenti connesse) della foresta.

L'algoritmo è ingordo perchè ad ogni passo si aggiunge alla foresta un arco con il peso minore.

**Implementazione** Per l'implementazione si usa una struttura Merge-Find Set (Mfset).

L'input non è un grafo $G$, ma un vettore di archi (EDGE[ ]) perché abbiamo bisogno di ordinare gli archi in base la peso. La rappresentazione dei grafi (per liste di adiacenza) non permette di ordinare gli archi in base al peso. La trasformazione non viene rappresentata nel seguente algoritmo.

**Algoritmo 7:** Algoritmo di Kruskal per la generazione di un MST

```
    SET kruskal(EDGE[] A, int n, int m)
        // EDGE[]: vettore di archi
(1)     SET T ← Set // insieme (inizialmente vuoto) che conterrà gli archi dell'albero minimo
        MFSET M ← Mfset(n) // insieme disgiunto grande
        // ordino per peso crescente
(2)     { ordina A[1][m] in modo che A[1].peso ⩽ ··· ⩽ A[m].peso }
        int c ← 0 // quanti archi ho aggiunto
        int i ← 1 // quale arco sto guardando
(3)     while c < n − 1 and i ⩽ m do // Termina quando l'albero è costruito
            // c < n − 1: ho raggiunto tutti gli archi necessari per fare un albero
            // i ⩽ m: ho esaurito tutti gli archi da guardare (per controllo)
            if M.find(A[i].u) ≠ M.find(A[i].v) then // non fanno parte dello stesso albero
                M.merge(A[i].u, A[i].v) // unisco gli insiemi disgiunti
                T.insert(A[i]) // inserisco l'arco all'albero
                c ← c + 1 // ho aggiunto un altro arco
            i ← i + 1 // guardo il prossimo arco
        return T // Ritorna l'albero di copertura minimo
```

**Complessità** Il tempo di esecuzione per l'algoritmo di Kruskal dipende dalla realizzazione della struttura dati Merge-Find Set. Utilizziamo la versione con *euristica sul rango più compressione dei cammini* che rende tutte le operazioni con costo ammortizzato costante pari a $\mathcal{O}(1)$.

① l'inizializzazione richiede $\mathcal{O}(n)$ in quanto devono essere creati tutti gli alberi separati;

② l'ordinamento degli archi ($m$) richiede $\mathcal{O}(m \log n)$, siccome il numero degli archi è limitato superiormente da $n^2$ posso scrivere $\mathcal{O}(m \log n^2)$, per le proprietà dei logaritmi $\mathcal{O}(m \log n)$;

③ nel caso peggiore vengono eseguite $\mathcal{O}(m)$ operazioni sulla foresta di insiemi disgiunti (due find ed una merge), con tempo ammortizzato $\mathcal{O}(1)$. Per un totale di $\mathcal{O}(n + m \log n + m) = O(m \log n)$.

## 13.10 Algoritmo di Prim

**Idea** A differenza dell'algoritmo di Kruskal che formava molti alberi (rappresentati da insiemi disgiunti) che uniti successivamente, l'algoritmo di Prim procede mantenendo in $A$ un singolo albero.

*Nota.* Durante l'esecuzione dell'algoritmo esiste un solo albero che rappresenta il quale rappresenta un sottoinsieme di un albero di copertura minimo del grafo totale.

L'albero parte da un vertice arbitrario $r$ (la radice) e cresce fino a quando non ricopre tutti i vertici. Ad ogni passo viene aggiunto un arco leggero che collega un vertice in $V_A$ con un vertice in $V - V_A$, dove $V_A$ è l'insieme di nodi raggiunti da archi in $A$.

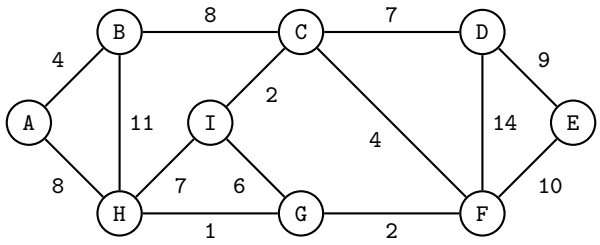
**Implementazione** Abbiamo bisogno di una struttura dati per i nodi non ancora presenti nell'albero. Durante l'esecuzione, i vertici che devono essere inseriti si trovano in una coda con min-priorità $Q$ ordinata in base alla seguente definizione di priorità: "la priorità del nodo $v$ è il peso minimo di un arco che collega $v$ ad un vertice nell'albero, o $+\infty$ se tale arco non esiste".

L'albero è memorizzato tramite un *vettore dei padri*, in cui $A$ è mantenuto implicitamente, infatti è definito $A = \{[v, p[v]] \mid v \in V - Q - \{r\}\}$ ($Q$ archi non ancora raggiunti, $\{r\}$ radice).

**Algoritmo 8:** Algoritmo di Prim per la generazione di un MST

```
prim(GRAPH G, NODE r, int[] p)
    // r:  nodo dalla quale parto
    // p:  vettore dei padri
    PRIORITYQUEUE Q ← MinPriorityQueue
    PRIORITYITEM[] POS ← new PRIORITYITEM[1...G.n]
    // inserisco i nodi nella coda, memorizzando la loro posizione
(1) foreach u ∈ G.V() − {r} do
        POS[u] ← Q.inserisci(u, +∞)
    // Inserisco il "nodo di partenza"
    POS[r] ← Q.inserisci(r, 0)
    p[r] ← 0  // convenzione per indicare che non ha padre
(2) while not Q.isEmpty do // non ci sono più nodi
        NODE u ← Q.deleteMin  // cancello e restituisco il nodo
        POS[u] ← nil // non considero più quel nodo
        // per ciascun nodo adiacente a quello considerato
(3)     foreach v ∈ G.adj(u) do
            if POS[v] ≠ nil and w(u,v) < POS[v].priority then
                // POS[v] ≠ nil:  è già stato visitato
                // w(u,v) < POS[v].priority:
                Q.decrease(POS [v], w(u,v))  // commento
                p[v] ← u  // commento
```

**Esempio** Fai riferimento alla spiegazione grafica, qui viene riportato solo lo schema sulla quale si basa.




**Analisi della complessità** $\mathcal{O}(m \log n)$

L'efficienza dell'algoritmo di Prim dipende dalla coda con priorità. Può essere implementato tramite heap binario oppure con un vettore non ordinato.

**heap binario** Nel caso si utilizzi un heap binario allora:

① l'inizializzazione costa $\mathcal{O}(m \log n)$;

② il ciclo principale viene eseguito $n - 1$ per una complessità di $\mathcal{O}(n)$ volte dove ogni operazione di deleteMin costa $\mathcal{O}(\log n)$;

③ il ciclo interno viene eseguito $\mathcal{O}(m)$ volte dove ogni operazione di decrease costa $\mathcal{O}(\log n)$.

Per un totale di $\mathcal{O}(n + m \log n + n \log n) = \mathcal{O}(m \log n)$.

*Nota.* L'algoritmo risulta asintoticamente uguale a quello di Kruskal.

**vettore non ordinato** Nel caso si utilizzi vettore un non ordinato:

① l'inizializzazione costa $\mathcal{O}(n)$;

② il ciclo principale viene eseguito $n-1$ per una complessità di $\mathcal{O}(n)$ volte dove ogni operazione di deleteMin costa $\mathcal{O}(n)$;

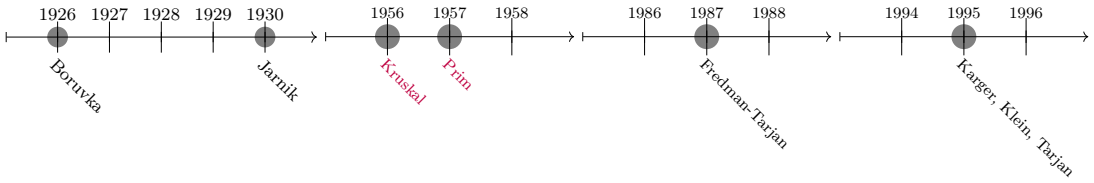
③ il ciclo interno viene eseguito $\mathcal{O}(m)$ volte dove ogni operazione di decrease costa $\mathcal{O}(1)$.

Per un totale di $\mathcal{O}(n + n^2 + m \cdot 1) = \mathcal{O}(n^2)$.

*Nota.* Cambiando la struttura dati cambia la complessità dell'algoritmo.

In definitiva se il grafo è *sparso* conviene utilizzare l'heap binario ($\mathcal{O}(m \log n)$), mentre se il grafo è *denso* (o addirittura completo) conviene utilizzare il vettore non ordinato ($\mathcal{O}(n^2)$).

### 13.10.1 Cenni storici




L'algoritmo di Fredman-Tarjan sfrutta un heap di Fibonacci che abbassa di molto la complessità dell'algoritmo ma ha dei costi associati molto grandi e quindi non viene utilizzato.

Nel '95 Karger, Klein e Tarjan hanno ideato un algoritmo probabilistico che risolve il problema in $\mathcal{O}(m+n)$ (molto spesso andare "a caso" conviene).

Se questo problema si possa risolvere in tempo lineare in modo deterministico è una questione ancora aperta.

### 13.10.2 Conclusioni

Gli algoritmi ingordi sono semplici da programmare e molto efficienti. Inoltre quando è possibile dimostrare la proprietà di scelta ingorda, danno la soluzione ottima. Una soluzione sub-ottima è comunque accettabile.