8 Algoritmi Greedy

Un **algoritmo greedy** ("incauto", "irruento") è un tipo di algoritmo per problemi di ottimizzazione che cerca di risolvere il problema in modo diretto, scegliendo la soluzione al sottoproblema più piccolo che al momento sembra la migliore.

Caratteristiche:

- semplice, con complessità di solito linerare;
- difficoltà di analisi;
- limitato campo di applicazione.

8.1 Critica alla Programmazione Dinamica

La programmazione dinamica è "prudente", nel senso che risolve <u>tutti</u> i sottoproblemi di una certa taglia.

8.2 Problema di Selezione di Attività Compatibili

Abbiamo:

- risorsa condivisa (e.g. aula);
- insieme di attività $S = \{a_i : 1 \leq i \leq n\}$

  $a_i = [s_i, f_i), \quad 0 \leq s_i \leq f_i \quad (s_i = \text{tempo di inizio}, f_i = \text{tempo di fine})$

Def Diciamo che a_i e a_j sono **compatibili** sse

$$[s_i, f_i) \cap [s_j, f_j) = \emptyset$$

Equivalentemente

$$f_i \leq s_j \text{ oppure } f_j \leq s_i$$

Esempio

Aula Lum250
$a_1 = [10.30, 12.00)$
$a_2 = [11.30, 13.00)$
$a_3 = [12.30, 14.00)$

a_1 e a_2 non sono compatibili
a_1 e a_3 sono compatibili
a_2 e a_3 non sono compatibili

8.2.1 Problema di Ottimizzazione

Determinare un sottoinsieme di massima cardinalità di attività mutuamente compatibili (cioè compatibili a coppie).

Ulteriore assunzione:

$$0 < f_1 \leq f_2 \leq \cdots \leq f_n$$

Voglio applicare la programmazione dinamica:

determinare i sottoproblemi

$$S_{ij} = \{a_k : f_i \leq s_k < f_k \leq s_j\}$$

Osservazione

1. $i > j \Rightarrow S_{ij} = \emptyset$

   perchè $f_i \leq s_k < f_k \leq s_j$
   ma $f_i \geq f_j$

2. S_{ij} non contiene tutte le attività di indice k con $i < k < j$

3. $|S_{ij}| \leq j - 1 - i, \quad 1 \leq i \leq n$

   Definisco due attività fittizie:

   $a_0 \rightarrow f_0 = 0$
   $a_{n+1} \rightarrow s_{n+1} = +\infty$
   $S = S_{0\,n+1}$

8.2.2 Proprietà di Sottostruttura Ottima

Sia A^*_{ij} un sottoinsieme di attività compatibili di S_{ij} di cardinalità massima.

Caso base: $S_{ij} = \emptyset \Rightarrow A_{ij} = \emptyset$

Caso generale: $S_{ij} \neq \emptyset$

1. $A^*_{ij} \neq \emptyset$
2. se $a_k \in A^*_{ij}$ allora $A^*_{ij} = A^*_{ik} \cup A^*_{kj}$
   dove A^*_{ik} e A^*_{kj} sono soluzioni ottime per S_{ik} e S_{kj}

Dimostrazione

Caso base: banale

Caso generale: $S_{ij} \neq \emptyset$

1. Un'attività è sempre compatibile con sè stessa $\Rightarrow \left|A^*_{ij}\right| \geq 1$
2. Dimostreremo che una qualsiasi attività $\in A^*_{ij}$ e $\neq a_k$ si trova o in S_{ik} o in S_{kj}

   Ho a_k, una qualsiasi attività $\in A^*_{ij}$ deve essere compatibile con a_k

   Prendiamo $a_l \in A^*_{ij},$ con $l \neq k$
   Deve valere
   $f_i \leq s_l < f_l \leq s_k \Rightarrow a_l \in S_{ik}$
   oppure
   $s_k < f_k \leq s_l < f_l \leq s_j \Rightarrow a_l \in S_{kj}$
   Quindi $A^*_{ij} = \overline{A}_{ik} \cup \{a_k\} \cup \overline{A}_{kj}$
   dove $\overline{A}_{ij}$ = tutte le attività compatibili in S_{ij}

   Ora dimostriamo che gli $\overline{A}$ sono anche A^*

   Per assurdo suppongo che $\overline{A}_{ik}$ non sia l'unico insieme di attività compatibili massimale in S_{ik}

   $$A^*_{ij} = \underset{\substack{\uparrow \\ A^*_{ik}}}{\cancel{\overline{A}_{ik}}} \cup \{a_k\} \cup \overline{A} \;\rightarrow\; \text{soluzione con cardinalità } > \left|A^*_{ij}\right|$$

8.2.3 Ricorrenza sui Costi

Definisco

$$c(i, j) = \left|A^*_{ij}\right|$$

$$c(i, j) = \begin{cases} 0 & \text{se } S_{ij} = \emptyset \\ 1 + \max\limits_{a_k \in S_{ij}} \{c(i, k) + c(k, j)\} & \text{se } S_{ij} \neq \emptyset \end{cases}$$

L'algoritmo bottom-up ha complessità $\Theta(n^3)$

Abbiamo visto che $A^*_{ij} = A^*_{ik} \cup \{a_k\} \cup A^*_{kj}$ ma non so chi è k.
Se conoscessi k potrei scrivere $1 + c(i, k) + c(k, j)$.

Teorema Sia a_m l'attività tale che

$$f_m = \min\{f_k : a_k \in S_{ij}\}$$

Se $S_{ij} \neq \emptyset$ allora

1. $\exists\, A^*_{ij}$ soluzione ottima tale che $a_m \in A^*_{ij}$
2. il sottoproblema $S_{im} = \emptyset$

Dimostrazione

1. Si consideri una soluzione ottima $\overline{A}_{ij}$ a S_{ij}

   Se $a_k = a_m \Rightarrow$ ho finito

   Altrimenti, $a_k \neq a_m \Rightarrow f_m \leq f_k \Rightarrow$ posso togliere a_k da $\overline{A}_{ij}$
2. banale

Strategia

1) Scegliere a_m
2) Risolvere S_{mj} (perchè $A^*_{ij} = \{a_m\} \cup A^*_{mj}$)

A noi interessa risolvere $S_{0\,n+1}$

Osservazione Devo risolvere solo problemi del tipo $S_{m\,n+1}$
$\Rightarrow$ lo spazio dei sottoproblemi è linerare (perchè il 2° indice dei sottoproblemi è fisso).

Codice ad alto livello

OPT($S_{i\,n+1}$)
if $S_{i\,n+1} \neq \emptyset$
    $a_m = f_m = \min\{f_k : a_k \in S_{i\,n+1}\}$ // molto grande
else
    return $\emptyset$

Pseudocodice

```
Rec-Sel(S, f, i)
n = S.length
m = i + 1
while (m ≤ n) and (s_m < f_i)
      m = m + 1
if m ≤ n
      return {a_m} ∪ Rec-Sel(S, f, m)
else
      return ∅
```

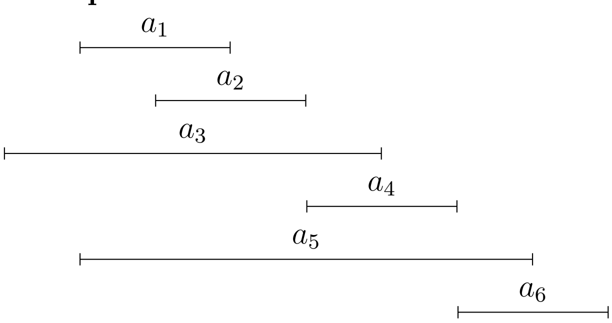
Complessità Modello di costo: confronti tra elementi s e f.

La complessità è $\Theta(n)$ perchè ogni attività viene coinvolta in un confronto una sola volta.

Versione iterativa:

```
Greedy-Sel(S, f)
n = S.length
A = {a_1}
last = 1 // indice dell'ultima attività selezionata
for m = 2 to n
      if s_m ≥ f_last
            A = A ∪ {a_m}
            last = m
return A
```

Esempio



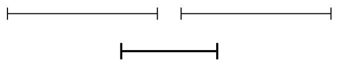
$A = a_1$	$A = a_1, a_4$	$A = a_1, a_4, a_6$
$last = 1$	$last = 4$	$last = 6$

8.2.4 Scelte Greedy Alternative

Oltre alla scelta greedy vista precedentemente, ne esistono altre:

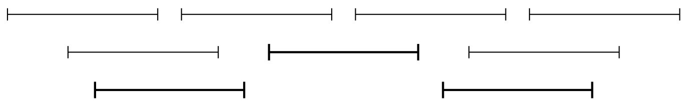
- Scegli l'attività di durata inferiore → non è ottima.

  Controesempio:

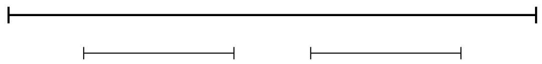
- Scegli l'attività col minor numero di sovrapposizioni → non è ottima.

  Controesempio:

- Scegli l'attività che inizia per prima → non è ottima.

  Controesempio:

- Scegli l'attività che inizia per ultima → è ottima.

8.3 Compressione dei Dati: Codici di Huffman

La compressione può essere di due tipi:

- lossy, cioè con perdita di informazione (e.g. immagine, video);
- lossless, cioè senza perdita di informazione (e.g. testo).

Esempio File di caratteri con frequenze associate:

	a	b	c	d	e	f
frequenza (%)	45	13	12	16	9	5

La codifica ASCII usa 8 bit per carattere.
File con 100K caratteri → 800 Kbit

Definisco una codifica che usa 3 bit per carattere:

	a	b	c	d	e	f
codeword	000	001	010	011	100	101

File con 100K caratteri → 300 Kbit (risparmio più del 50%)
+ spazio per una tabella di conversione/decodifica

0	'a'
1	'b'
2	'c'
3	'd'
4	'e'
5	'f'

Nella posizione i si trova il carattere la cui codifica è i stesso (in binario).

Def

C = alfabeto dei simboli presenti nel file
funzione di encoding $e \colon C \to 0, 1^*$

Proprietà che deve avere e:

- invertibile → iniettiva: se $a, b \in C, a \neq b \Rightarrow e(a) \neq e(b)$;
- ammettere algoritmi efficienti: e, e^{-1}.

Una funzione di encoding che associa ad ogni carattere di C una stringa di 0 e 1 della stessa lunghezza si chiama **fixed length encoding**.

Finora ho ignorato la frequenza dei caratteri.

Idea Associare a caratteri più frequenti codeword più corte e, di conseguenza, a caratteri meno frequenti codeword più lunghe.

	a	b	c	d	e	f
frequenza (%)	45	13	12	16	9	5
codeword	0	00	01	1	100	101

Problema $e(aa) = e(b)$
Non sono come decodificare in modo univoco perchè esistono delle codeword che sono prefissi di altre codeword.

La codifica che sto cercando deve:

- avere lunghezza variabile della codeword;
- affinchè si possa avere l'invertibilità, il codice deve essere libero da prefissi (**codice prefisso**[1]), cioè $\nexists\ a, b \in C : e(a)$ è un prefisso di $e(b)$.

Soluzione

	a	b	c	d	e	f
frequenza (%)	45	13	12	16	9	5
codeword	0	101	100	111	1101	1100

Questo è un codice a lunghezza variabile libero da prefissi ed è ottimo tra tutti i codici che associano una codeword ad ogni singolo carattere.

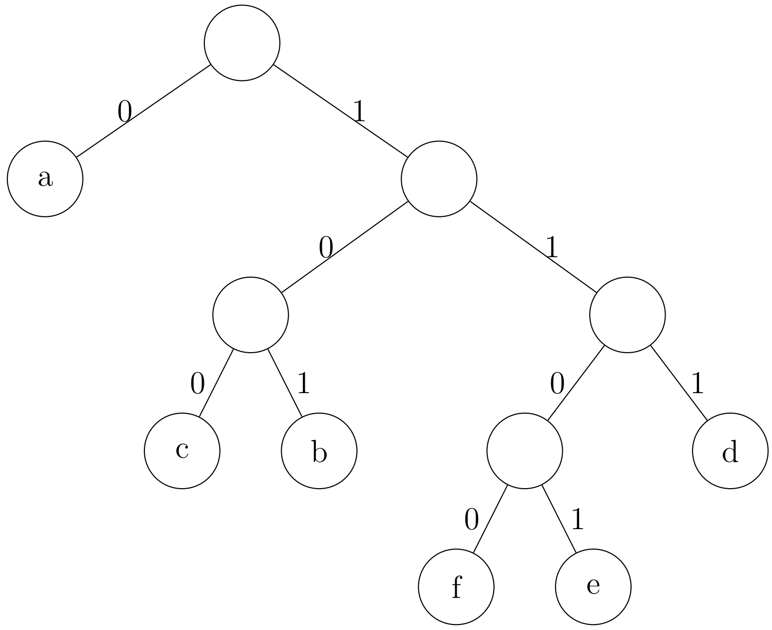
Spazio occupato da questo encoding:

$$100\text{K}\left(\frac{45}{100}\cdot 1 + \frac{13}{100}\cdot 3 + \frac{12}{100}\cdot 3 + \frac{16}{100}\cdot 3 + \frac{9}{100}\cdot 4 + \frac{5}{100}\cdot 4\right) = 224 \text{ Kbit}$$

300 Kbit $\rightarrow$ risparmio $\approx 25\%$

Serve memorizzare dell'informazione aggiuntiva per la decodifica:

[1] Attenzione: con il termine "codice prefisso" si intende un codice senza prefissi.



Stringa codificata: 110001001101
Stringa decodificata: face

Un qualsiasi codice binario può essere rappresentato in modo compatto con un albero binario.

Un codice prefisso è associato a un **albero di codifica** T in cui i caratteri da codificare appaiono tutti alle foglie.

$\forall\ c \in C$ ho $f(c)$ frequenza

Definisco

$$d_T(c) = \text{profondità di c in T}$$

$d_T(\text{'a'}) = 1$

$d_T(\text{'b'}) = d_T(\text{'c'}) = d_T(\text{'d'}) = 3$

$d_T(\text{'e'}) = d_T(\text{'f'}) = 4$

La profondità corrisponde alla lunghezza della codeword associata.

Funzione di costo:

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$$

La dimensione del file compresso è

$$|F_c| = \frac{B(T) \cdot |F|}{100}$$

dove

$$|F| = \text{taglia del file compresso (in \# di caratteri)}$$

$B(T)$, a meno di una costante, è il fattore di compressione $\min B(T)$.

Proprietà Un albero ottimo è sempre pieno (cioè i nodi interni hanno due figli).

Spazio di problemi: spazio di alberi pieni con n foglie ($n = |C|$)

Idea Prendo due nodi con frequenza minore, gli unisco in un nodo che avrà come frequenza la somma delle due, ripeto $n-1$ volte.
In questo modo, i nodi con la frequenza maggiore verranno aggiunti per ultimi alla cima dell'albero, quindi avranno una profondità bassa, ovvero la lunghezza dalla codeword corta, proprio come volevo.

Q: coda di priorità (Heap) di nodi con chiave $f(z)$

Attributi di un nodo:

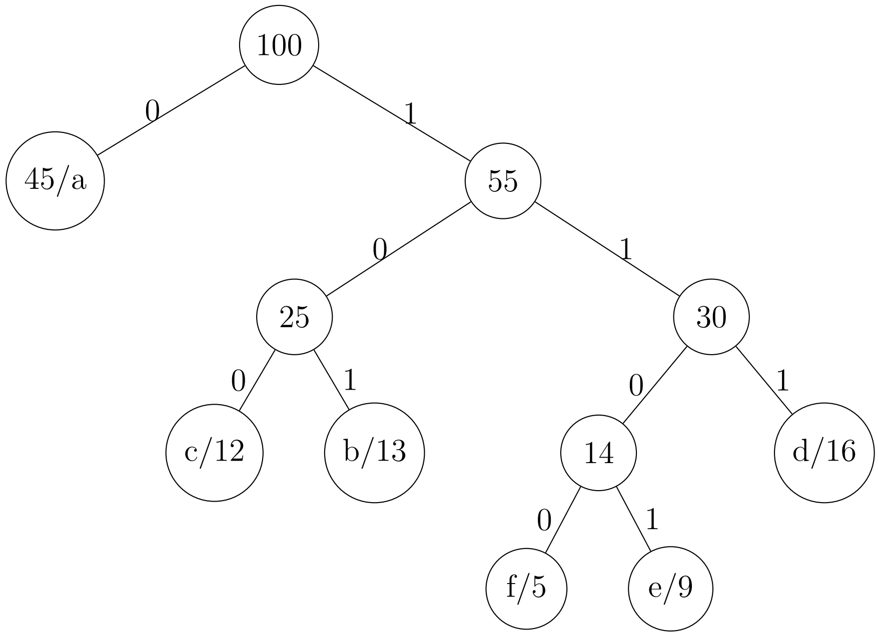
- $z.left$
- $z.right$
- $z.f$

Pseudocodice

```
Huffman(C, f)
n = |C|
Q = ∅
for each c ∈ C // inizializzazione
    z = New-Node() // crea un nuovo nodo
    z.f = f(c)
    z.left = NIL
    z.right = NIL
    Insert(Q, z) // Θ(log n)
for i = 1 to n − 1
    x = Extract-Min(Q)
    y = Extract-Min(Q)
    z = New-Node()
    z.left = x
    z.right = y
    z.f = x.f + y.f
    Insert(Q, z)
```

Complessità $\Theta\left(\sum_{i=1}^{n-1} \log(n-i)\right) = \Theta(n \log n)$

Esempio Riprendiamo l'esempio iniziale e applichiamo l'algoritmo.



	a	b	c	d	e	f
codeword	0	101	100	111	1101	1100

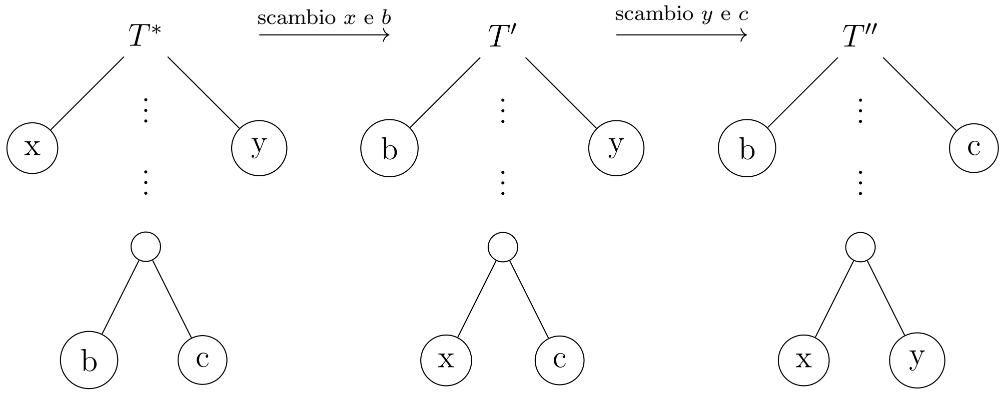
$$B(T) = \left(\frac{45}{100} \cdot 1 + \frac{13}{100} \cdot 3 + \frac{12}{100} \cdot 3 + \frac{16}{100} \cdot 3 + \frac{9}{100} \cdot 4 + \frac{5}{100} \cdot 4\right) = 224$$

8.3.1 Proprietà di Scelta Greedy

Sia C un alfabeto e siano x e y i caratteri in C di frequenza minore.
Allora esiste un codice prefisso ottimo T in cui x e y sono foglie attaccate alo stesso padre (sorelle).

Dimostrazione
Sia T^* una soluzione ottima arbitraria $\Rightarrow B(T^*)$ è minima.
Non so dove siano x e y in T^*
Siano b e c le foglie sorelle di profondità massima in T^*

1. $d_{T^*}(b) = d_{T^*}(c) \geq d_{T^*}(x), d_{T^*}(y)$
2. $f(x) \leq f(b), \quad f(y) \leq f(c)$



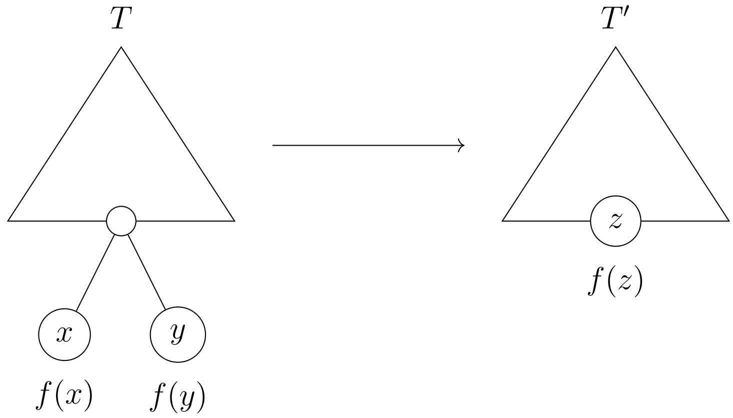
$T^* \to T'$
$B(T^*) \geq B(T')$
$B(T^*) - B(T') \geq 0$

$$\begin{aligned}
B(T^*) - B(T') &= \sum_{c \in C} f(c)\, d_{T^*}(c) - \sum_{c \in C} f(c)\, d_{T'}(c) \\
&= f(b)\, d_{T^*}(b) + f(x)\, d_{T^*}(x) - f(b)\, \underbrace{d_{T'}(b)}_{d_{T^*}(b)} - f(x)\, \underbrace{d_{T'}(x)}_{d_{T^*}(x)} \\
&= f(b)\, d_{T^*}(b) + f(x)\, d_{T^*}(x) - f(b)\, d_{T^*}(b) - f(x)\, d_{T^*}(x) \\
&= f(b)\big(d_{T^*}(b) - d_{T^*}(x)\big) - f(x)\big(d_{T^*}(b) - d_{T^*}(x)\big) \\
&= \Big(\underbrace{f(b) - f(x)}_{\geq 0 \text{ perchè } f(x) \leq f(b)} \Big)\Big(\underbrace{d_{T^*}(b) - d_{T^*}(x)}_{\geq 0 \text{ per il punto 1}} \Big) \geq 0
\end{aligned}$$

$T' \to T''$ si dimostra analogamente.

8.3.2 Proprietà di Sottostruttura Ottima

Sia T un codice prefisso ottimo che contiene la scelta greedy (sui caratteri x e y).
Sia z un nuovo carattere (cioè $\notin C$) con $f(z) = f(x) + f(y)$.
Allora il codice prefisso $T' = T \setminus \{x, y\}$ è ottimo per $C \setminus \{x, y\} \cup \{z\}$



T ha n foglie.
T' ha $n-1$ foglie.
Cerco una relazione tra $B(T)$ e $B(T')$.

Osservazione

1. $\forall\ c \in C \setminus \{x,y\},\ c \neq z : d_{T'}(c) = d_T(c)$
2. $d_{T'}(z) = d_T(x) - 1$

Dimostrazione

$$
\begin{aligned}
B(T) &= \sum_{c \in C} f(c)\, d_T(c) && \big(d_T(x) = d_T(y)\big) \\
&= \sum_{c \in C\setminus\{x,y\}} f(c)\, d_T(c) + \big(f(x)+f(y)\big) d_T(x) \\
&= \sum_{c \in C\setminus\{x,y\}} f(c)\, d_T(c) + \big(\underbrace{f(x)+f(y)}_{f(z)}\big)\big(\underbrace{d_T(x)-1}_{d_{T'}(z)}\big) + \big(f(x)+f(y)\big) \\
&= \sum_{c \in C\setminus\{x,y\}\cup\{z\}} f(c)\, d_{T'}(c) + f(x) + f(y) \\
&= B(T') + f(x) + f(y)
\end{aligned}
$$

Suppongo per assurdo che T non sia il codice prefisso ottimo per il sottoproblema $C \setminus \{x,y\} \cup \{z\}$.
Allora $\exists\ T''$ di costo strettamente inferiore.

$$B(T) = \underbrace{B(T'')}_{<B(T')} + f(x) + f(y)$$

Assurdo!