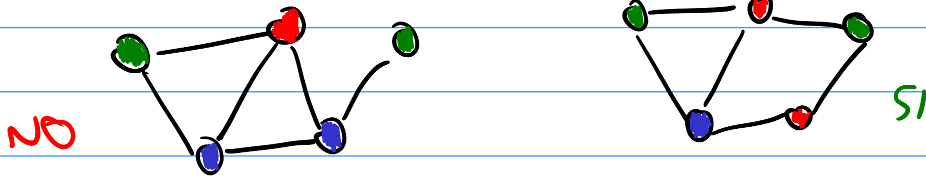
DUE PROBLEMI INTRATTABILI

- Vedremo dei problemi per cui non sono noti algoritmi efficienti
- Scriveremo degli algoritmi che esplorano lo spazio di tutte le soluzioni possibili, cercando di usare delle funzioni di taglio per ridurre lo spazio cercato.

PROBLEMA DELLA 3-COLORAZIONE

Dato un grafo $G = (V, E)$ semplice non orientato, una **COLORAZIONE** è un assegnamento di colori ai vertici in modo che non ci siano vertici adiacenti dello stesso colore



- Per le applicazioni il numero più piccolo possibile di colori è preferibile tuttavia
  - 1 - colore, possibile solo per grafi senza archi
  - 2 colori, grafi bipartiti
  - 3 colori, ???
- Determinare se 3-colori sono sufficienti per colorare un grafo è un problema DIFFICILE, e addirittura si sospetta INTRATTABILE

PROBLEMA Dato un grafo $G = (V, E)$ rappresentato come liste di adiacenza determinare se esiste o meno un assegnamento

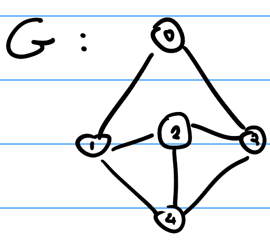
$$col : V \longrightarrow \{R, G, B\}$$

tale che $\forall \{u, v\} \in E \quad col(u) \neq col(v)$.

Assumiamo $V = \{0,1,2,..,n-1\}$ e quindi $|V|=n$

- Ci sono 3^n possibili modi di colorare gli n-vertici, quindi un approccio di forza bruta è impossibile per n grandi
- Scriviamo ora un algoritmo che risolve il problema e oltretutto
  - se G è 3-colorabile, stampa tutte le colorazioni lecite
  - se G non è colorabile, non stampa nulla

G :



0 → 1,3
1 → 0,2,4
2 → 1,3,4
3 → 0,2,4
4 → 1,2,3

```
[0, 1, 0, 1, 2]
[0, 1, 2, 1, 0]
[0, 2, 0, 2, 1]
[0, 2, 1, 2, 0]
[1, 0, 1, 0, 2]
[1, 0, 2, 0, 1]
[1, 2, 0, 2, 1]
[1, 2, 1, 2, 0]
[2, 0, 1, 0, 2]
[2, 0, 2, 0, 1]
[2, 1, 0, 1, 2]
[2, 1, 2, 1, 0]
```

Le 12 possibili colorazioni

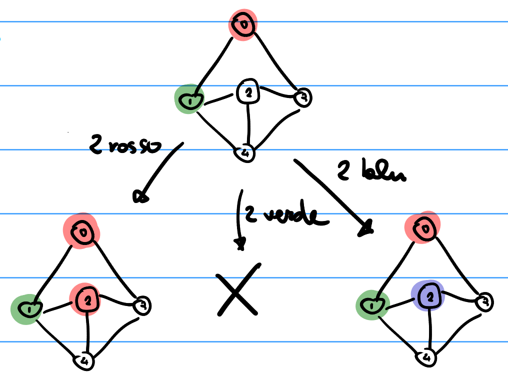
```
 4 def tc_rec(G,col,v):
 5
 6     if v == len(col):
 7         print(col)
 8         return
 9
10     # colori permessi per il vertice v
11     allowed=[True,True,True]
12     for u in G[v]:
13         if col[u] is not None:
14             allowed[ col[u] ] = False
15     for c in range(3):
16         if allowed[c]:
17             col[v]=c
18             tc_rec(G,col,v+1)
19     col[v] = None
20
21 def three_col(G):
22     soluzione=[None]*len(G)
23     tc_rec(G,soluzione,0)
```

→ 3 colorazione trovata (righe 6–8)

→ CALCOLA QUALI SONO I COLORI GIÁ ASSEGNATI AI VICINI DI V E QUINDI VIETATI (righe 11–14)

→ FUNZIONE DI TAGLIO (riga 16)

→ RICORSIONE (riga 18)



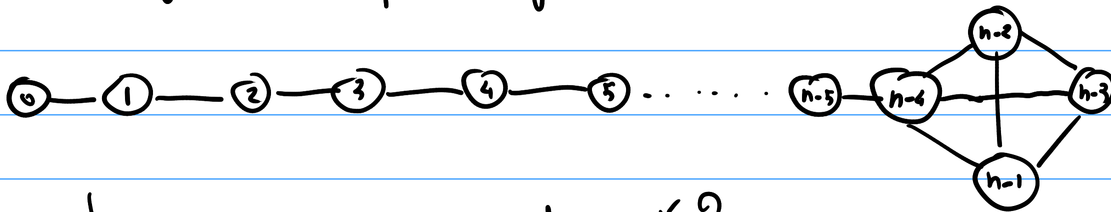
- La complessità dell'algoritmo è alla peggio $\Theta(n \cdot 3^n)$ poiché la ricorsione è un albero ternario di altezza n quindi:

$$\#\text{nodi} = \sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{3-1} = \Theta(3^n)$$

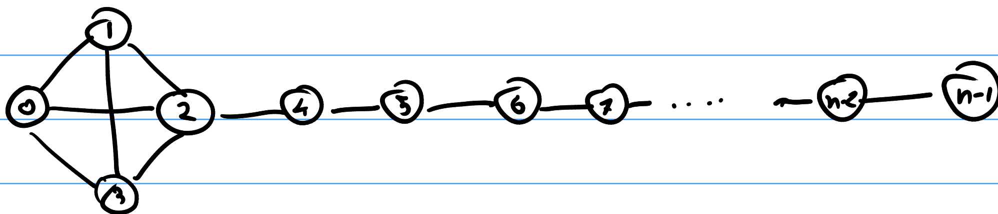
in ogni nodo si esplora il vicinato oppure si stampa la colorazione: $\Theta(n)$

É POSSIBILE FARE STIME PIÚ PRECISE?

ESERCIZIO Quante chiamate ricorsive impiegherá l'algoritmo per accorgersi che questo grafo non é 3-colorabile?



E se i vertici sono invece numerati cosí?



CONCLUSIONE: Anche per grafi senza colorazioni lecite, il tempo di esecuzione puó variare moltissimo

IL PROBLEMA DELLO ZAINO

- Abbiamo risolto il problema dello zaino in tempo $\Theta(n \cdot C)$, ma C, la capacità dello zaino, è un numero di grandezza esponenziale rispetto alla sua scrittura in bit.
- A meno che C non sia MOLTO più piccolo di 2^n, questo algoritmo non è efficiente in generale.
- Vediamo un approccio con BACKTRACKING

Problema Dati n oggetti con pesi $p_0, \dots, p_{n-1}$ e valori $v_0, \dots, v_{(n-1)}$ e uno zaino con capacità C,
trovare un insieme $S \subseteq \{0, \dots, n-1\}$ tale che

- $\sum_{i \in S} p_i \leq C$
- $\sum_{i \in S} v_i$ sia massima

Tentativo 1: ricerca esaustiva.

Esploriamo tutte le possibili scelte di oggetti (che sono 2^n, come le stringhe binarie) con una procedura ricorsiva.

Nota: nel programma che segue la funzione ricorsiva è definita all'interno della funzione principale, e ne condivide le variabili.

La funzione ricorsiva mantiene
- i : prossimo oggetto da scegliere
- choices : gli oggetti in $\{0, \dots, i-1\}$ scelti

```
160 def zaino1(weights,values,capacity):
161
162     assert len(weights)==len(values)
163     opt_choices, opt_value=[],0
164     n = len(weights)
165     calls = 0
166
167     def zr(i,choices):
168         nonlocal opt_value,opt_choices,n
169         nonlocal calls
170         calls+=1
171         if i == n:
172             value  = sum( values[i] for i in choices)
173             weight = sum(weights[i] for i in choices)
174             if value > opt_value and weight <= capacity:
175                 opt_value = value
176                 opt_choices = choices[:]
177         else:
178             # non prendere i
179             zr(i+1,choices)
180             # prendi i
181             choices.append(i)
182             zr(i+1,choices)
183             choices.pop()
184
185     zr(0,[])
186     print(opt_choices,opt_value,calls)
```

Annotazioni:
- (righe 165, 170) contiamo il numero di nodi nella ricorsione
- (righe 171–175) Foglia: Aggiorniamo la migliore soluzione trovata, se non viola capacità.
- (riga 179) Ramo 0: non prendiamo i
- (riga 182) Ramo 1: prendiamo i
- (riga 185) chiamata iniziale della funzione ricorsiva

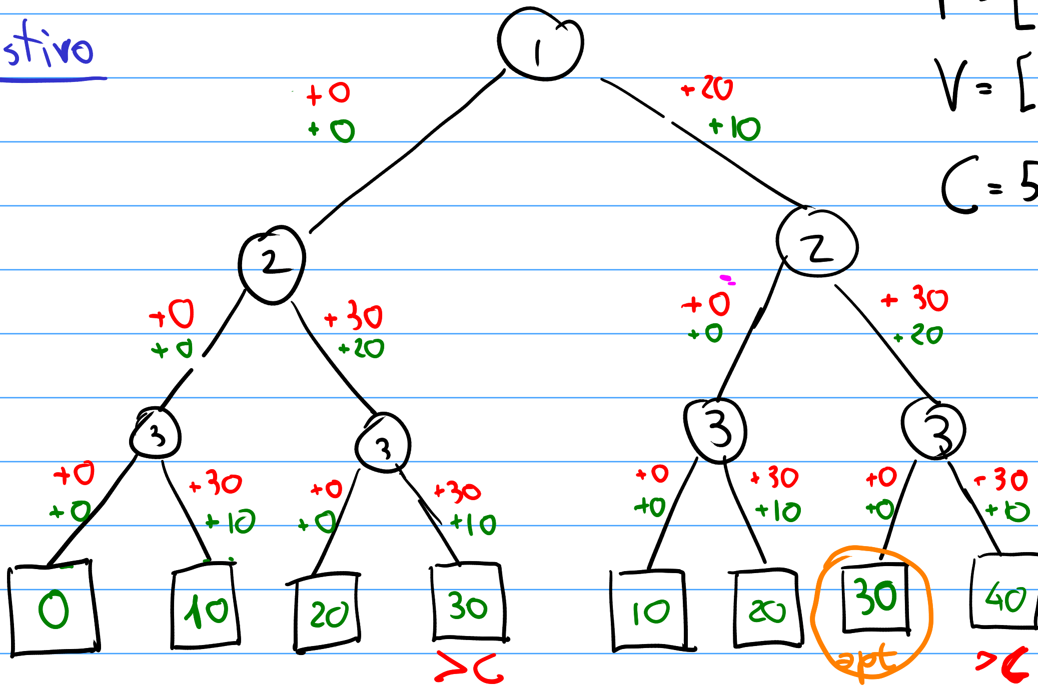
Esempio di problema dello zaino

Esaustivo

$P = [20, 30, 30]$

$V = [10, 20, 10]$

$C = 50$



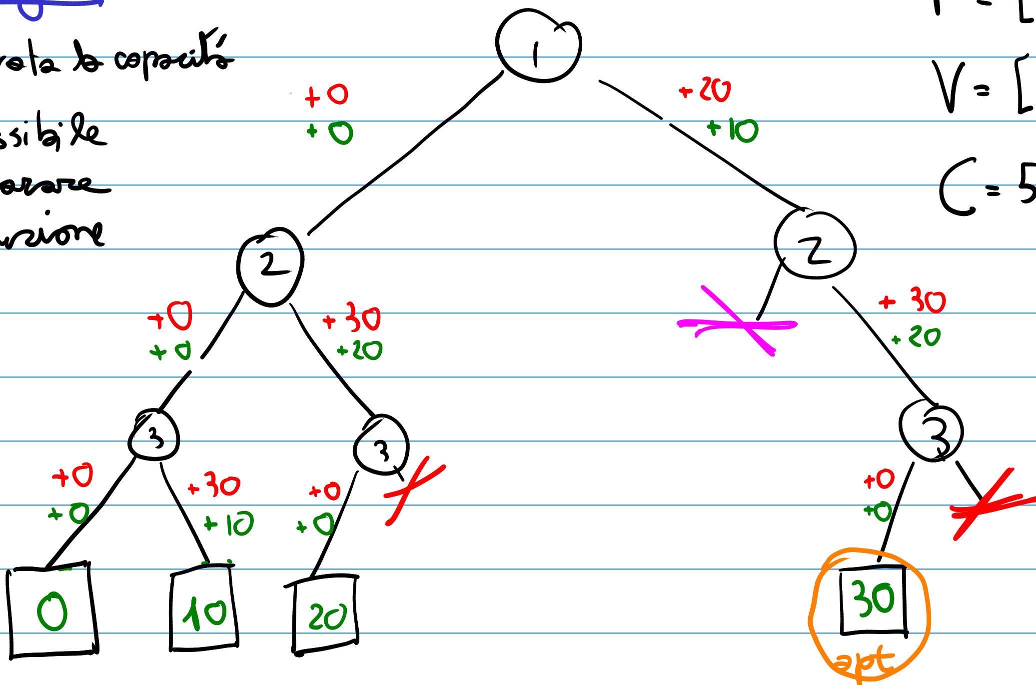
Con tagli

- superata la capacità
- impossibile migliorare la soluzione

$P = [20, 30, 30]$

$V = [10, 20, 10]$

$C = 50$



Sulla strategia 1 ci sta poco da dire: fa 2^n tentativi e per ognuno ricalcola il peso e il valore totale. Eventualmente aggiorna la scelta migliore vista fino a quel momento

Tentativo 2:
Aggiungiamo una funzione di taglio: inutile inserire oggetti che fanno superare la capacità massima

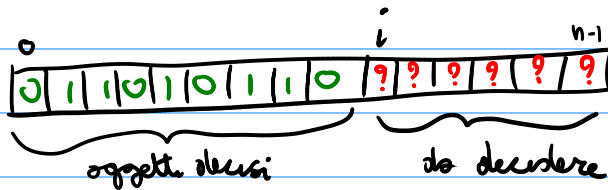
- Se p_i + peso corrente > C allora i non è scelto
- Visto che ci siamo, non ha senso ricalcolare ad ogni foglia valore e peso. Si possono mantenere i parziali negli argomenti della funzione ricorsiva
  → In questo modo le foglie che non migliorano la soluzione ottima costano $O(1)$

```
187
188 def zaino2(weights,values,capacity):
189
190     assert len(weights)==len(values)
191     opt_choices, opt_value=[],0
192     n = len(weights)
193     calls = 0
194
195     def zr(i,choices,weight,value):
196         nonlocal opt_value,opt_choices,n
197         nonlocal calls
198         calls+=1
199         if i == n:
200             if value > opt_value:
201                 opt_value = value
202                 opt_choices = choices[:]
203         else:
204             # non prendere i
205             zr(i+1,choices,weight,value)
206             # prendi i
207             if weights[i]+weight<=C:
208                 choices.append(i)
209                 zr(i+1,choices,
210                    weights[i]+weight,
211                    value+values[i])
212                 choices.pop()
213
214     zr(0,[],0,0)
215     print(opt_choices,opt_value,calls)
216
```

Annotazioni:
- (weight, value) → parziali che vengono aggiornati nelle chiamate ricorsive
- (righe 199–202) → Foglia: la soluzione ottima viene aggiornata
- (riga 205) → Ramo 0: i non è scelto
- (riga 207) → Funzione di taglio
- (righe 208–211) → Ramo 1: i è scelto
- (riga 214) → chiamata iniziale

Funzioni di Taglio per Problemi di Ottimizzazione

- Fino ad ora abbiamo visto tagli di rami che non portano a soluzioni ammissibili (e.g. capienza dello zaino superata, troppi 1 nelle stringhe binarie, ecc...)
- Osserviamo che in un problema di ottimizzazione un ramo della ricorsione è utile **SOLO SE MIGLIORA** la soluzione trovata fino a quel momento



- Al passo i-esimo la quantità MASSIMA di valore aggiuntivo che si può avere è $\sum_{j=i}^{n-1} v_j$ (e non è neppure detto che questa quantità sia ammissibile)
- Se $\text{ValoreCorrente} + \sum_{j=i}^{n-1} v_j <$ Valore della migliore soluzione trovata allora possiamo tagliare il ramo

Tentativo 3

```
217 def zaino3(weights,values,capacity):
218 
219     assert len(weights)==len(values)
220     opt_choices, opt_value=[],0
221     n = len(weights)
222     calls = 0
223 
224     def zr(i,choices,weight,value,remaining):
225         nonlocal opt_value,opt_choices,n
226         nonlocal calls
227         calls+=1
228         if i == n:
229             if value > opt_value:
230                 opt_value = value
231                 opt_choices = choices[:]
232         else:
233             # non prendere i
234             if value + remaining - values[i] > opt_value:
235                 zr(i+1,choices,
236                    weight,
237                    value,
238                    remaining-values[i])
239             # prendi i
240             if weights[i]+weight<=C:
241                 choices.append(i)
242                 zr(i+1,choices,
243                    weights[i]+weight,
244                    value+values[i],
245                    remaining-values[i])
246                 choices.pop()
247 
248     zr(0,[],0,0,sum(values))
249     print(opt_choices, opt_value, calls)
```

- `remaining` → Rappresenta la quantità di valore ancora in gioco
- Riga 234 → Taglio del ramo 0: soluzione non migliorabile
- Riga 240 → Taglio del ramo 1: capacità superata

Le funzioni di taglio "potano l'albero di ricerca"
(prune the search tree)

ESEMPIO pesi = [23, 31, 29, 89, 44, 53, 38, 63, 85, 82]
valori = [92, 57, 49, 87, 68, 60, 43, 67, 84, 72]
capacità: 166

- Tutte e tre le varianti ottengono una soluzione ottima {0,1,2,4,6} di valore 309
- osservate che abbiamo inserito un contatore di chiamate
  - tentativo 1 : 2047 chiamate
  - tentativo 2 : 546 chiamate
  - tentativo 3 : 388 chiamate

Quali funzioni di taglio adoperare?

- Complessità delle funzioni di taglio
  - → più onerose da eseguire
  - → tagliano più rami

Va trovato (e va testato!!) un giusto compromesso

BACKTRACKING e OTTIMIZZAZIONE : USO DI UNA SOLUZIONE INIZIALE

- Nel problema dello zaino si parte con una SOLUZIONE NULLA e ad ogni nuova foglia si MIGLIORA LA SOLUZIONE, se possibile
- In presenza di funzioni di taglio che scartano soluzioni riconoscibilmente non ottime, può essere conveniente partire con una SOLUZIONE INIZIALE di discreta qualità, ottenuta in fretta magari con
  - METODI GREEDY
  - ALGORITMI DI APPROSSIMAZIONE

Un strategia greedy può essere quella di prendere gli oggetti con $\frac{v_i}{p_i}$ (rapporto valore/peso) più grande possibile fino ad esaurimento della capacità

- Nell'esempio visto questo produce $\{0,1,2,4\}$ con valore 266
- Utilizzando questa soluzione iniziale l'algoritmo a pag 6 fa 314 chiamate

BACKTRACKING: Sequenze di decisione

Nei nostri esempi abbiamo visto che al passo i-esimo della ricorrenza si decide per l'elemento i-esimo dell'input

- È possibile che un ordine diverso delle scelte aiuti (es. Esercizio a pag. 3)
- Non è neppure necessario che l'ordine sia rigido e che sia lo stesso per tutti i rami. Ogni nodo può decidere il prossimo elemento da considerare