lezione19

November 27, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 19 - 27 novembre 2023

2 QUESTIONARIO OPINIONI STUDENTI

Corso in presenza	**HIHES6G2**
Teledidattica	**TZF7T1J7**

3 RECAP:

- Alberi N-ari
- altezza e profondità
- diametro
- merge e merge-sort
- esercizi d'esame (combinazioni di k elementi scelti tra N)

[1]:
```
# decoratore che stampa le chiamate ed uscite di una funzione ricorsiva
from rtrace import trace
```

4 GIOCHI A TURNI

Un **gioco** è formato da: - una **situazione corrente** (configurazione, posizione delle pedine, combinazione di simboli …) - una serie di **mosse applicabili** alla situazione corrente - una **regola di terminazione** del gioco - un **criterio di vittoria o parità**

4.1 ALBERI DI GIOCO (simulazione di tutte le possibili partite)

Per capire come funziona un gioco o per definire delle strategie vincenti possiamo costruire tutte le possibili evoluzioni del gioco a partire dalla configurazione iniziale - data una configurazione iniziale ed il giocatore di turno - se il gioco è terminato calcoliamo chi ha vinto o se è patta - altrimenti individuiamo le mosse applicabili - proviamo ad applicare una mossa - ci troveremo in una nuova configurazione - ripetiamo ricorsivamente ad esplorare le nuove configurazioni finchè è possibile - se non ci sono più possibili configurazioni passiamo a provare la prossima mossa - …

5 GIOCO: somma di coppie consecutive pari o dispari

- **configurazione**: una sequenza di interi
- **mosse possibili**: sommare una coppia di numeri consecutivi pari+pari o dispari+dispari
- **terminazione**: non ci sono più coppie pari,pari o dispari,dispari

Convergenza: ad ogni passo il numero di elementi diminuisce di 1

Caso base: numeri alternati o lista di un solo elemento

GOAL: trovare tutte le sequenze finali

5.0.1 Esempio: [1, 13, 2, 7, 9, 2]

- [1, 13, 2, 7, 9, 2] -> [14, 2, 7, 9, 2] -> [16, 7, 9, 2] -> [16, 16, 2] -> [32, 2] -> [34]
- [1, 13, 2, 7, 9, 2] -> [1, 13, 2, 16, 2] -> [1, 13, 2, 18] -> [1, 13, 20] -> [14, 20] -> [34]
- ...

[2]:
```
%load_ext nb_mypy
```

```
Version 1.0.5
```

[3]:
```
from pygraphviz import AGraph
class GameNode:
    _num_nodi = 0
    def __init__(self):
        self.__class__._num_nodi += 1
        self.__id = self.__class__._num_nodi
        self._sons = []

    def dot(self, G:AGraph) -> None:
        "Costruisco la rappresentazione dell'albero da visualizzare con␣
 ↪Graphviz"
        if self._sons:
            G.add_node(self.__id, label=self) # se nodo interno colore nero
        else:
            G.add_node(self.__id, label=self, color='red', style='bold',␣
 ↪shape='box') # coloro le foglie di rosso
        for son in self._sons:
            G.add_edge(self.__id, son.__id)
            son.dot(G)

    def show(self):
        G = AGraph()
        self.dot(G)
        G.layout('dot')
        return G
```

[4]:
```
import jdc
```

```python
# configurazione: lista di valori + figli
class Sequenza(GameNode):
    def __init__(self, lista : list[int]):
        super().__init__()
        "Una configurazione contiene la lista di interi"
        self._lista = lista
        self._sons  = []

    def __repr__(self):
        "Visualizzo la lista"
        return f'{self._lista}'
```

[5]:
```python
S = Sequenza([1, 13, 2, 7, 9, 2])
S.show()
```

[5]:

$$[1, 13, 2, 7, 9, 2]$$

[ ]:
```python

```

5.0.2 Mosse valide: tutte le coppie pari o dispari

- per ogni possibile posizione **i**
  - la torniamo se i due valori hanno stesso resto diviso 2

[6]:
```python
%%add_to Sequenza

def mosse_valide(self) -> list[int]:
    "elenco di tutte le posizioni i,i+1 che possono essere sommate"
    return [ i
             for i in range(len(self._lista)-1)
             # mossa valida se resti uguali (pari+pari o dispari+dispari)
             if self._lista[i]%2 == self._lista[i+1]%2 ]
```

[7]:
```python
print(S)
S.mosse_valide() # type: ignore # (ignoro l'errore di mypy)
```

```
[1, 13, 2, 7, 9, 2]
```

[7]:
```
[0, 3]
```

5.0.3 Applicare la mossa vuol dire creare una nuova sequenza

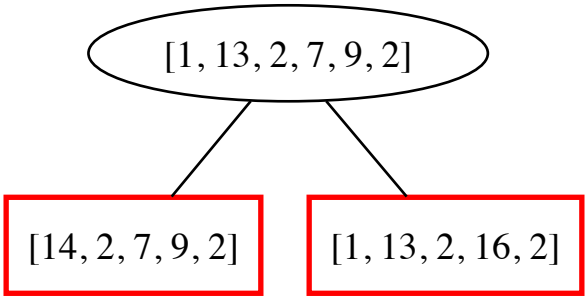
- basta sostituire i valori in posizioni i e i+1 con la somma

ATTENZIONE la lista originale NON va cambiata

[8]:
```
%%add_to Sequenza
def applica_mossa(self, i:int) -> None:
    "applico una mossa creando il nodo figlio ed aggiungendolo ai figli"
    nuova_lista = self._lista.copy()    # copio la sequenza per non modificarla
    # rimpiazzo i due valori con la loro somma usando un assegnamento a slice
    #nuova_lista[i:i+2] = [nuova_lista[i] + nuova_lista[i+1]]
    N1 = nuova_lista.pop(i)
    N2 = nuova_lista.pop(i)
    nuova_lista.insert(i, N1 + N2)
    # creo il nuovo nodo e lo aggiungo ai figli
    self._sons.append(Sequenza(nuova_lista))
```

[9]:
```
S = Sequenza([ 1, 13, 2, 7, 9, 2 ])
S.applica_mossa(0)  # type: ignore
S.applica_mossa(3)  # type: ignore
S.show()
```

[9]:



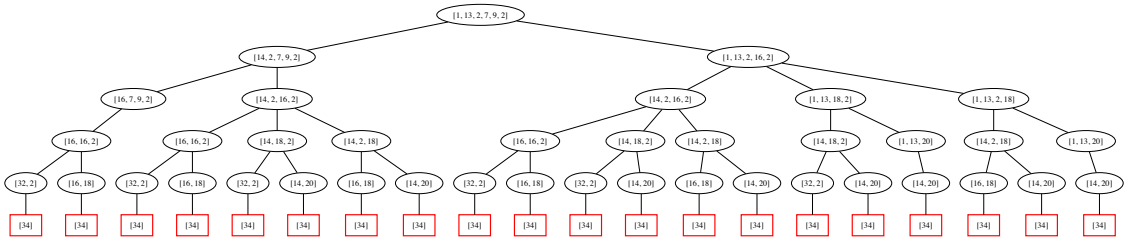
5.0.4 Generazione di tutto l'albero

- per ogni mossa possibile generiamo il nuovo figlio
- per ogni figlio generiamo le prossime mosse

[10]:
```
%%add_to Sequenza
def genera(self) -> None:
    "applicazione delle mosse valide e generazione dei sottoalberi"
    for i in self.mosse_valide():
        self.applica_mossa(i)
    for son in self._sons:
        son.genera()
```

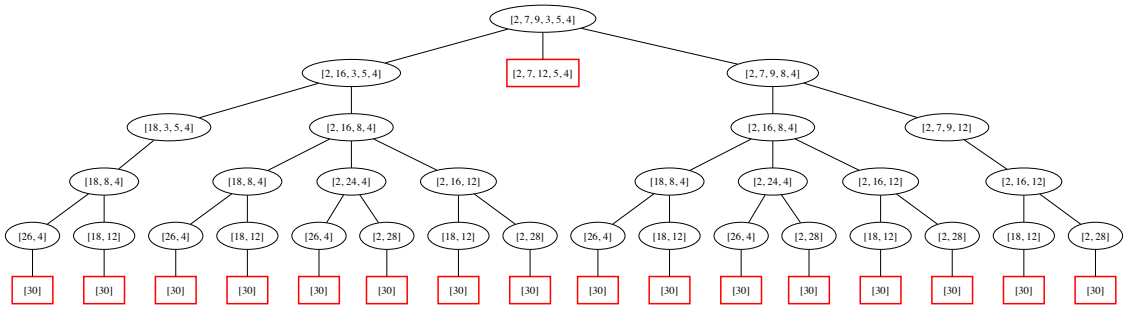
[11]:
```
S = Sequenza([ 1, 13, 2, 7, 9, 2 ])
S.genera() # type: ignore
S.show()
```

[11]:



5.0.5 MA tutti i percorsi sono sempre della stessa lunghezza?

[12]:
```
B = Sequenza([2, 7, 9, 3, 5, 4])
B.genera() # type: ignore
B.show()
```

[12]:


5.0.6 Trovare la giocata più corta

- basta esplorare l'albero e cercare la **foglia con profondità minima** dell'albero

5.0.7 Trovare la giocata più lunga

- basta esplorare l'albero e cercare la **foglia con profondità massima** dell'albero

[13]:
```
%%add_to Sequenza
def shortest(self):
    "minima altezza, ovvero distanza della foglia più vicina dalla radice"
    if not self._sons:
        return 0
    else:
        return min( son.shortest() for son in self._sons ) + 1

def tallest(self):
    "massima altezza, ovvero distanza della foglia più lontana dalla radice"
    if not self._sons:
        return 0
```

```
    else:
        return max( son.tallest() for son in self._sons ) + 1
```

[14]:
```
B.shortest(), B.tallest() # type: ignore
```

[14]:
```
(1, 5)
```

5.0.8 Per trovare la foglia più alta/bassa

- dobbiamo ricordare il nodo assieme alla sua profondità

[15]:
```
%%add_to Sequenza
def tallest_leaf(self):
    "massima altezza E foglia che gli corrisponde"
    if not self._sons:      # se sono una foglia
        return 0, self      # torno 1 e me stessa
    else:
        # altrimenti calcolo le distanze massime per ciascun figlio
        altezze_figli = [ son.tallest_leaf() for son in self._sons ]
        # e tra queste prendo la coppia massima con la foglia corrispondente
        massimo, nodo = max(altezze_figli, key=lambda coppia: coppia[0])
        # torno la distanza massima +1 E quella foglia
        return massimo + 1, nodo

def shortest_leaf(self):
    "minima altezza e foglia che la produce"
    if not self._sons:
        return 0, self
    else:
        altezze_figli = [ son.shortest_leaf() for son in self._sons ]
        minimo, nodo = min(altezze_figli, key=lambda coppia: coppia[0])
        return minimo + 1, nodo
```

[16]:
```
B.tallest_leaf(), B.shortest_leaf() # type:ignore
```

[16]:
```
((5, [30]), (1, [2, 7, 12, 5, 4]))
```

6 GIOCO: catena di parole

- **Configurazione**: due parole anagrammi (parola da modificare e obiettivo)
- **Mosse valide**: scambiare due lettere mettendone una a posto
- **Terminazione**: sono uguali

Generazione dell'albero: - basta cambiare la generazione delle mosse valide

GOAL: cercare il numero minimo di scambi

[17]:
```
# posizione di gioco: due stringhe
# caso base: se le due stringhe sono uguali ho trovato la soluzione, torno il␣
 ↪livello
# altrimenti provo a scambiare due caratteri in modo da metterne almeno uno a␣
 ↪posto (convergenza)
   # (cerco il primo carattere in A diverso in B, lo trovo in B e li scambio)
   # creo le configurazioni figlie di ciascun nodo creato
# alla peggio con N scambi trasformo A in B

class Anagramma(GameNode):
    # s1 ed s2 sono liste di caratteri
    def __init__(self, s1 : str, s2 : str):
        "memorizzo le sequenze di caratteri per cui devo trovare la sequenza di␣
 ↪scambi"
        assert list(sorted(s1)) == list(sorted(s2)), f"Non sono anagrammi {s1}␣
 ↪ed {s2}"
        super().__init__()
        self._s1 = s1
        self._s2 = s2
        self._sons = []

    def __repr__(self):
        "torno la stringa da stampare per visualizzare il nodo ed i figli e il␣
 ↪livello"
        return f'{self._s1}\n{self._s2}'
```

[18]:
```
A = Anagramma('ABCDEF', 'FBCDEA')
A.show()
```

[18]:

6.0.1 le mosse valide sono le coppie di posizioni di caratteri da scambiare

- scorro le posizioni
- faccio solo scambi da caratteri successivi alla posizione che sto sistemando

[19]:
```
%%add_to Anagramma
def mosse_valide(self) -> set[tuple[int,int]]:
    "Genero l'insieme di mosse valide"
    if self._s1 == self._s2:    # se le due parole sono già  uguali
        return set()            # non c'è da fare scambi
    else:
```

```
        mosse = set()           # altrimenti
        N = len(self._s1)
        #for i,(c1,c2) in enumerate(zip(self._s1,self._s2)):   # scandisco s1
↪ed s2
        for i in range(N-1):
            c1 = self._s1[i]
            c2 = self._s2[i]
            if c1 != c2:                    # se nella stessa posizione il
↪carattere di s1 è diverso
                for j in range(i+1,N):      # cerco nelle posizioni seguenti
                    if self._s1[j] == c2:   # se lo trovo
                        mosse.add((i,j))    # la aggiungo
        return mosse
```

[20]:
```
A1 = Anagramma("BEDCA", "ABCDE")
print(A1.mosse_valide()) # type: ignore
A1.show()
```

```
{(2, 3), (0, 4)}
```

[20]:

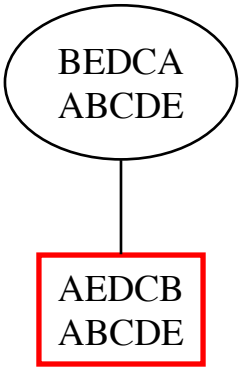
6.0.2 Per applicare la mossa

- costruisco una lista di caratteri
- scambio i due
- li trasformo in stringa
- creo il nuovo figlio

[21]:
```
%%add_to Anagramma
def applica_mossa(self, i: int, j: int) -> None:
    "Applico una mossa generando un figlio"
    nuova_s1 = list(self._s1)
    # scambio i valori che stanno nei due indici
    nuova_s1[i], nuova_s1[j] = nuova_s1[j], nuova_s1[i]
    nuova_s1 = ''.join(nuova_s1)
    self._sons.append(Anagramma(nuova_s1, self._s2))
    # creo la nuova configurazione e la aggiungo ai figli
```

[22]:
```
A1.applica_mossa(4,0) # type: ignore
A1.show()
```

[22]:



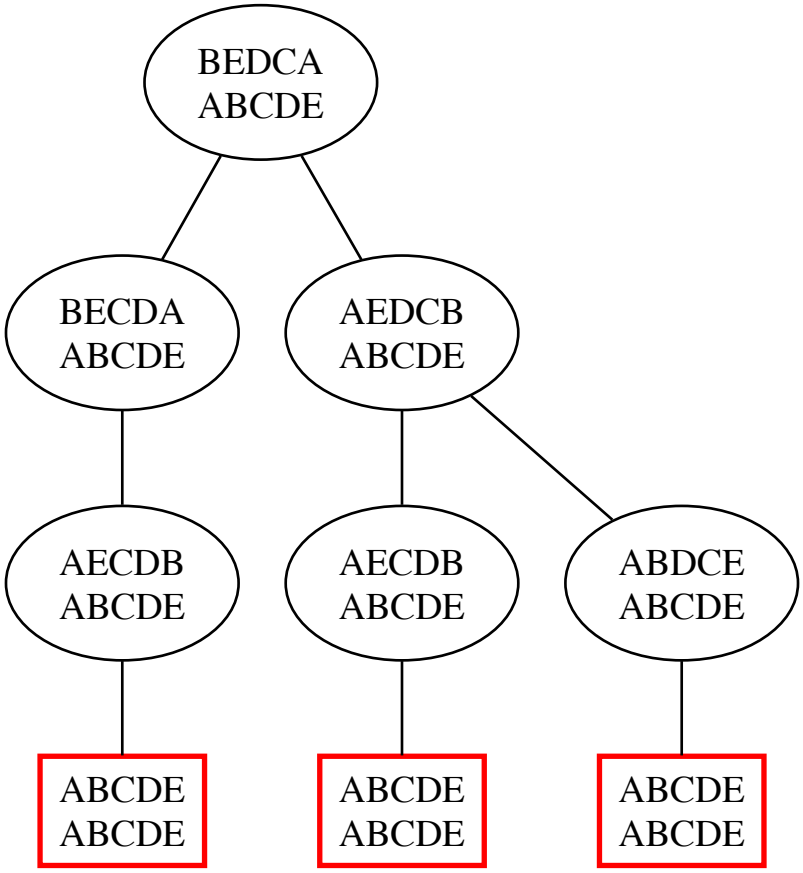
6.0.3 Per generare tutto l'albero

- genero tutti i figli applicando le mosse valide
- per ogni figlio genero il resto

[23]:
```
%%add_to Anagramma
def genera(self) -> None:
    "Applico tutte le mosse valide e poi lo faccio sui figli generati"
    for i,j in self.mosse_valide():
        self.applica_mossa(i,j)
    for son in self._sons:
        son.genera()
```

[24]:
```
A1 = Anagramma("BEDCA", "ABCDE")
A1.genera()  # type: ignore
A1.show()
```

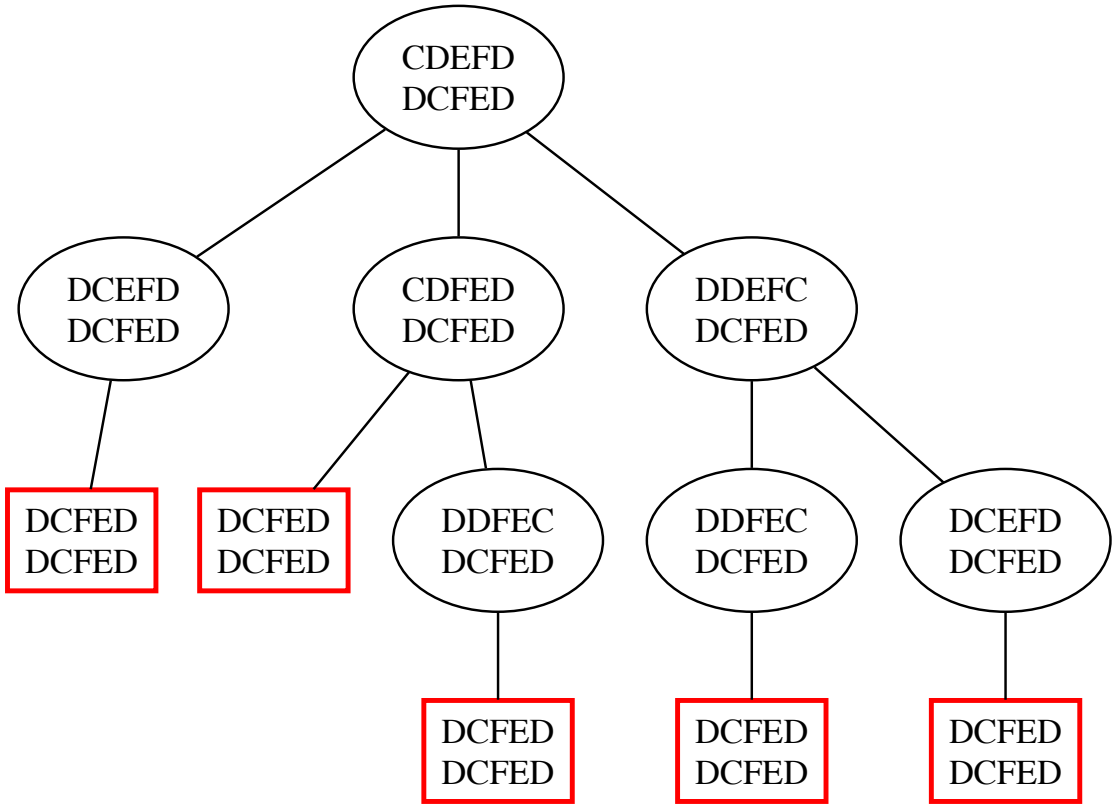
[24]:



6.0.4 Sembrerebbe che tutte le soluzioni abbiano la stessa lunghezza .... è vero?

[25]:
```
A2 = Anagramma("CDEFD", "DCFED")
A2.genera() # type: ignore
A2.show()
```

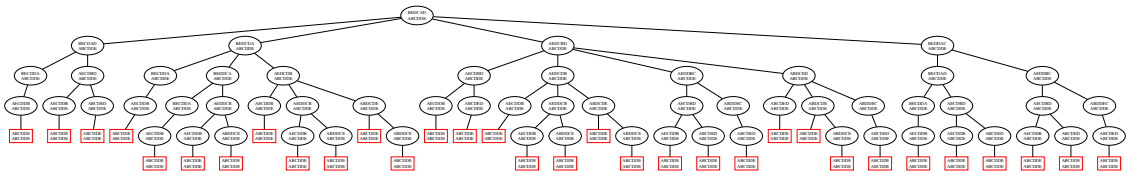
[25]:



[26]:
```
%%add_to Anagramma
def min_mosse(self):
    "altezza minima (come numero di archi)"
    if not self._sons:
        return 0
    else:
        return 1 + min(son.min_mosse() for son in self._sons)
```

[27]:
```
A1 = Anagramma("BEDCAD", "ABCDDE")
A1.genera()  #type: ignore
print(A1.min_mosse())  #type: ignore
A1.show()
```

```
4
```

[27]:


7 GIOCO: Dare il resto con certi tipi di monete

- dato un valore intero **N** (resto da dare)
- ed una lista ordinata **L** di valori interi di monete che contiene sempre **1** (es [10, 5, 2, 1])
- trovare tutti i diversi modi di dare il resto

Esempio: N = 9, L = [10, 5, 2, 1] - 9 = 5 + 2 + 2 - 9 = 5 + 2 + 1 + 1 - 9 = 5 + 1 + 1 + 1 + 1 - 9 = 2 + 2 + 2 + 2 + 1 - 9 = 2 + 2 + 2 + 1 + 1 + 1 - 9 = 2 + 2 + 1 + 1 + 1 + 1 + 1 - 9 = 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1 - 9 = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

7.1 Approccio ricorsivo

- **casi base**:
  - **N == 0** : soluzione []
  - **N == 1** : soluzione [**1**]
  - **M == [1]** : soluzione **[1]*N**
- **Se M[0] > N**: la prima moneta è troppo grande
  - tolgo la moneta **M -> M[1:]** e torno la sottosoluzione
- altrimenti:
  - provo ad usarlo: **N -> N - M[0]** e aggiungo **M[0]** alla sottosoluzione
  - provo a non usarlo più: **M -> M[1:]** e torno la sottosoluzione

Convergenza: tolgo sempre qualcosa da N o da M

[28]:
```
class Resto(GameNode):
    def __init__(self, N : int, LM : list[int], mossa : int|None = None):
        "una configurazione contiene un valore e la lista di monete disponibili
 ↪e puÃ² ricordare la moneta usata per arrivare qui"
        super().__init__()
        self._N = N
        self._LM = LM
        self._sons = []
        self._mossa = mossa        # quando serve ci ricordiamo la mossa che ci
 ↪ha portato qui

    def __repr__(self):
        "torno la stringa da stampare per visualizzare il nodo ed i figli"
        if self._mossa is None:
            return f'N: {self._N}\nMonete: {self._LM}'
        else:
            return f'N: {self._N}\nMonete: {self._LM}\nMossa: {self._mossa}'
```

[29]:
```
R = Resto(9, [5,2,1])
R.show()
```

[29]:

N: 9
Monete: [5, 2, 1]

7.1.1 Mosse valide

Come rappresentare una "mossa"? Vogliamo aggiornare **N** oppure **M**

Le rappresento con una coppia: **valore da sottrarre**, **nuovo insieme di monete** e ritorno

- nessuna mossa se N == 0
- (1, M) se M == [1]
- (0, M[1:]) se M[0] > N
- altrimenti le due coppie:
  - (M[0], M) provo ad usare la prima moneta
  - (0 , M[1:]) smetto di usare la prima moneta

[30]:
```
%%add_to Resto
def mosse_valide(self):
    "ciascuna mossa Ã¨ rappresentata dalla coppia: valore da sottrarre, elenco␣
 ↪di monete disponibili"
    if self._N == 0:                # se N == 0
        return []                   # nessuna mossa
    if len(self._LM) == 1:          # se ho solo 1 tipo di moneta (1)
        return [ (1,self._LM) ]     # tolgo 1 e continuo con quella moneta
    if self._LM[0] > self._N:       # se la prima moneta è troppo grossa
        return [ (0,self._LM[1:]) ] # la ignoro (sottraggo 0 e continuo senza␣
 ↪quella moneta)
    else:
        prima, *altre = self._LM    # altrimenti ho due possibilitÃ
        return [ (prima,self._LM),  # sottraggo la prima moneta e continuo con␣
 ↪lo stesso elenco di monete
                 (0, altre)  ]      # oppure non la sottraggo e smetto di usarla
```

[31]:
```
print(R.mosse_valide())  # type: ignore
R.show()
```

```
[(5, [5, 2, 1]), (0, [2, 1])]
```

[31]:

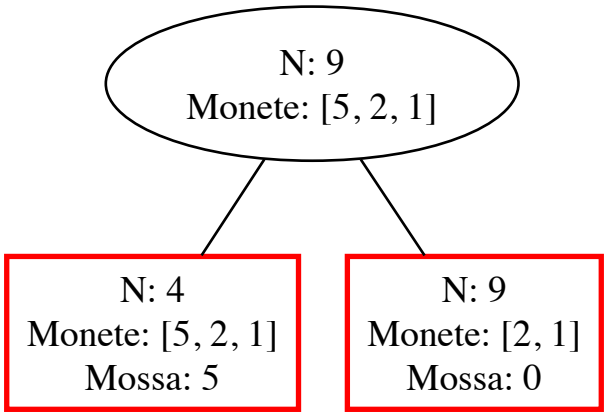
N: 9
Monete: [5, 2, 1]

7.1.2 Per applicare la mossa aggiorno sia N che M

[32]:
```
%%add_to Resto
def applica_mossa(self, moneta, LM):
    "applicazione di una mossa"
    N1 = self._N - moneta   # detraggo la moneta dal resto
    # aggiungo un nuovo figlio per il nuovo valore e con le monete indicate e␣
 ↪mi ricordo che moneta ho sottratto
    self._sons.append(Resto(N1, LM, moneta))
```

[33]:
```
R.applica_mossa(5, [5,2,1]) # type: ignore
R.applica_mossa(0, [2,1])   # type: ignore
R.show()
```

[33]:

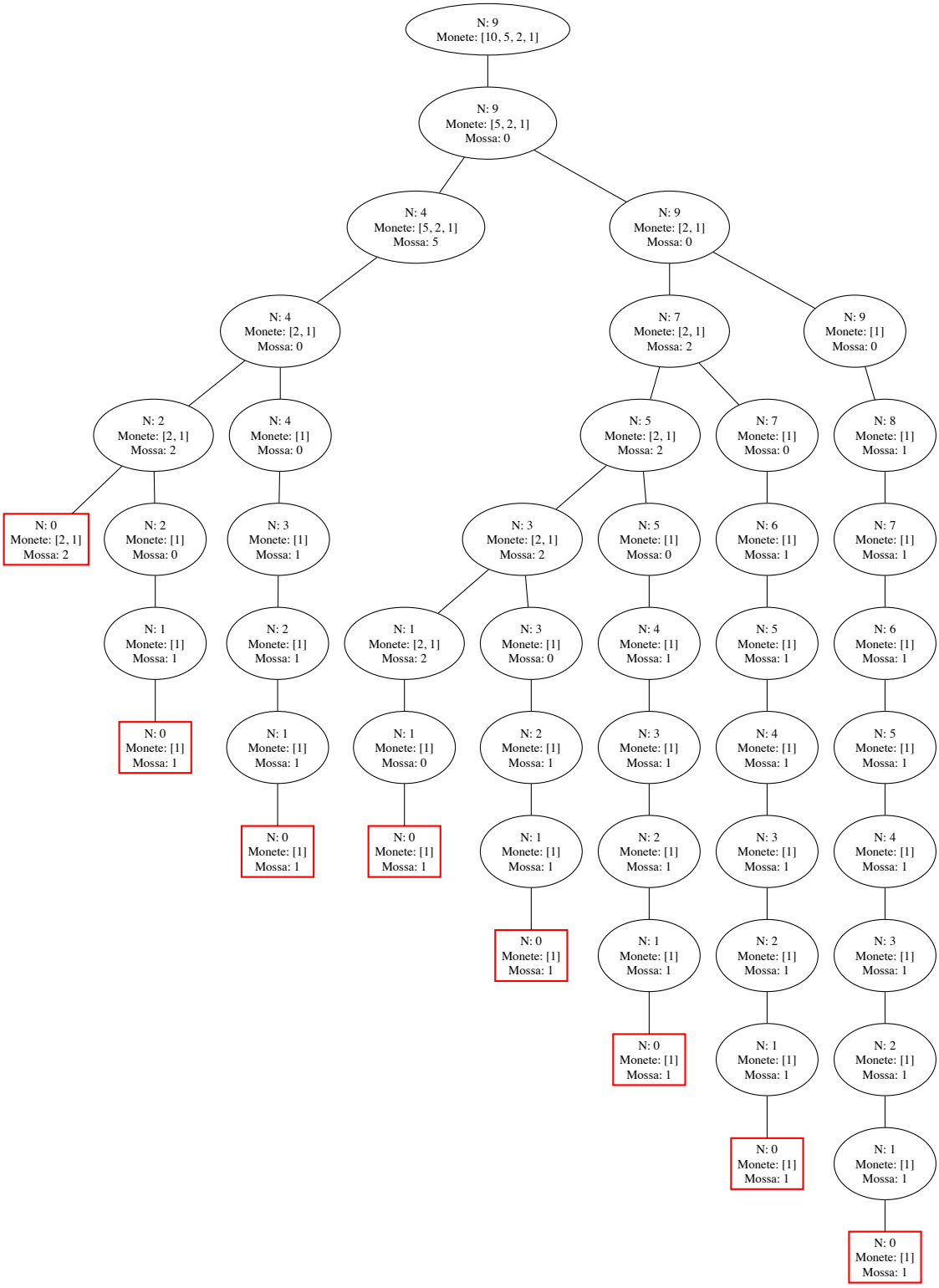


7.1.3 Come al solito genero l'albero applicando ricorsivamente le mosse valide

[34]:
```
%%add_to Resto
def genera(self):
    "Applico tutte le mosse valide e poi lo faccio sui figli generati"
    for M,LM in self.mosse_valide():
        self.applica_mossa(M,LM)
    for son in self._sons:
        son.genera()
```

[35]:
```
R = Resto(9, [10,5,2,1])
R.genera()   # type: ignore
R.show()
```

[35]:


N: 9
Monete: [10, 5, 2, 1]
N: 9
Monete: [5, 2, 1]
Mossa: 0
N: 4
Monete: [5, 2, 1]
Mossa: 5
N: 9
Monete: [2, 1]
Mossa: 0
N: 4
Monete: [2, 1]
Mossa: 0
N: 7
Monete: [2, 1]
Mossa: 2
N: 9
Monete: [1]
Mossa: 0
N: 2
Monete: [2, 1]
Mossa: 2
N: 4
Monete: [1]
Mossa: 0
N: 5
Monete: [2, 1]
Mossa: 2
N: 7
Monete: [1]
Mossa: 0
N: 8
Monete: [1]
Mossa: 1
N: 0
Monete: [2, 1]
Mossa: 2
N: 2
Monete: [1]
Mossa: 0
N: 3
Monete: [1]
Mossa: 1
N: 3
Monete: [2, 1]
Mossa: 2
N: 5
Monete: [1]
Mossa: 0
N: 6
Monete: [1]
Mossa: 1
N: 7
Monete: [1]
Mossa: 1
N: 1
Monete: [1]
Mossa: 1
N: 2
Monete: [1]
Mossa: 1
N: 1
Monete: [2, 1]
Mossa: 2
N: 3
Monete: [1]
Mossa: 0
N: 4
Monete: [1]
Mossa: 1
N: 5
Monete: [1]
Mossa: 1
N: 6
Monete: [1]
Mossa: 1
N: 0
Monete: [1]
Mossa: 1
N: 1
Monete: [1]
Mossa: 1
N: 1
Monete: [1]
Mossa: 0
N: 2
Monete: [1]
Mossa: 1
N: 3
Monete: [1]
Mossa: 1
N: 4
Monete: [1]
Mossa: 1
N: 5
Monete: [1]
Mossa: 1
N: 0
Monete: [1]
Mossa: 1
N: 0
Monete: [1]
Mossa: 1
N: 1
Monete: [1]
Mossa: 1
N: 2
Monete: [1]
Mossa: 1
N: 3
Monete: [1]
Mossa: 1
N: 4
Monete: [1]
Mossa: 1
N: 0
Monete: [1]
Mossa: 1
N: 1
Monete: [1]
Mossa: 1
N: 2
Monete: [1]
Mossa: 1
N: 3
Monete: [1]
Mossa: 1
N: 0
Monete: [1]
Mossa: 1
N: 1
Monete: [1]
Mossa: 1
N: 2
Monete: [1]
Mossa: 1
N: 0
Monete: [1]
Mossa: 1
N: 1
Monete: [1]
Mossa: 1
N: 0
Monete: [1]
Mossa: 1

7.1.4 Per trovare le soluzioni

- esploro l'albero
- a ciascuna soluzione di un sottoproblema aggiungo la moneta che ha portato a questo nodo

[36]:
```
%%add_to Resto
def soluzioni(self) -> list[list[int]]:
    if self._sons:
        # raccolgo tutte le soluzioni dei figli e gli aggiungo la mossa
        return [ [ self._mossa ] + sol for son in self._sons
                                        for sol in son.soluzioni() ]
    else:
        return [ [ self._mossa ] ]
```

[37]:
```
print(*R.soluzioni(), sep='\n')  # type: ignore

# MA: **non ci interessano** le mosse in cui non si sottrae nulla da N
```

```
[None, 0, 5, 0, 2, 2]
[None, 0, 5, 0, 2, 0, 1, 1]
[None, 0, 5, 0, 0, 1, 1, 1, 1]
[None, 0, 0, 2, 2, 2, 2, 0, 1]
[None, 0, 0, 2, 2, 2, 0, 1, 1, 1]
[None, 0, 0, 2, 2, 0, 1, 1, 1, 1, 1]
[None, 0, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1]
[None, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

[38]:
```
[ [ m for m in soluzione if m] for soluzione in R.soluzioni()] # type: ignore
```

[38]:
```
[[5, 2, 2],
 [5, 2, 1, 1],
 [5, 1, 1, 1, 1],
 [2, 2, 2, 2, 1],
 [2, 2, 2, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

[39]:
```
Sequenza._num_nodi, Anagramma._num_nodi, Resto._num_nodi
```

[39]:
```
(111, 115, 52)
```