Table of Contents

Fondamenti di Programmazione

Andrea Sterbini

lezione 18 - 5 dicembre 2022

RECAP:

- Ritorsione
- Merge sort

- Alberi binari
- stampa in preordine postordine e inordine
- altezza e ricerca del nodo minimo o massimo
- alberi n-ari
- scansione dell'abero come metodi della classe

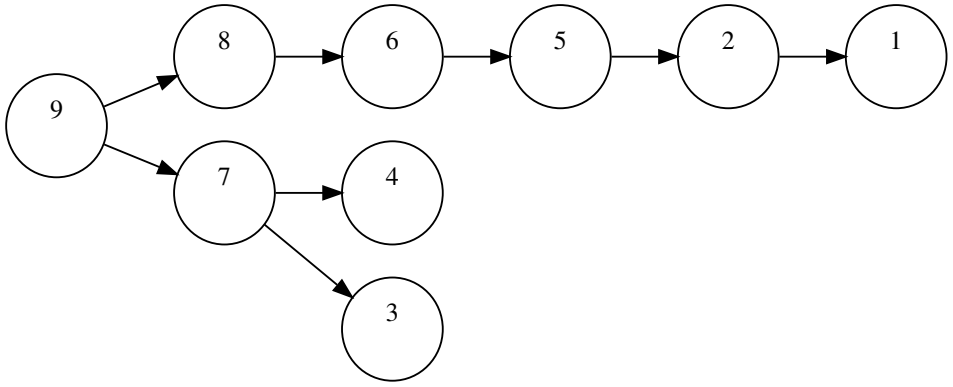
In [49]:
```
# decoratore che stampa le chiamate ed uscite di una funzione ricorsiva
from rtrace import trace
```

In [49]:
```
# mi costruisco una lista di valori casuali
import random, graphviz
lista = random.choices(range(-10000,10001), k=10)
```

In [50]:
```
class NodoBinario:
    def __init__(self, V : int, sx : 'NodoBinario' = None, dx : 'NodoBinario' = None):
        self._value = V
        self._sx = sx
        self._dx = dx
    def __repr__(self) -> str :
        return f"\"{self._value}\n{getattr(self,'_altezza','')}\n{getattr(self,'_percorso','')}\""
    def _dot(self):
        "creo l'elenco di nodi ed arghi che Graphviz sa viaualizzare"
        s = f'{self}\n'
        if self._sx and self._dx:
            s += f'{{rank=same ; {self._sx} ; {self._dx} }}\n'
        if self._sx:
            s += f'{self} -> {self._sx}\n'
            s += self._sx._dot()
        if self._dx:
            s += f'{self} -> {self._dx}\n'
            s += self._dx._dot()
        return s
    def show(self):
        "creo l'oggetto Digraph per visualizzare il grafo diretto"
        G = graphviz.Digraph()
        G.body.append('rankdir=LR\n' + self._dot())
        return G
```

In [51]:
```
n1= NodoBinario(1)
n2= NodoBinario(2, dx=n1)
n3= NodoBinario(3)
n4= NodoBinario(4)
n5= NodoBinario(5, dx=n2)
n6= NodoBinario(6, dx=n5)
n7= NodoBinario(7, n4, n3)
n8= NodoBinario(8, n6)
r= NodoBinario(9, n8, n7)
r.show()
```

Out[51]:



Diametro di un albero (lunghezza del percorso più lungo)

In [6]:
```
import graphviz
G = graphviz.Digraph()
G.body.append('''
    rankdir=LR
    A -> B -> C -> D -> E [style=bold color=red]
    A -> F -> G -> H -> I [style=bold color=red]
    B -> J -> K
    F -> L -> O
    G -> M
    J -> N
''')
```

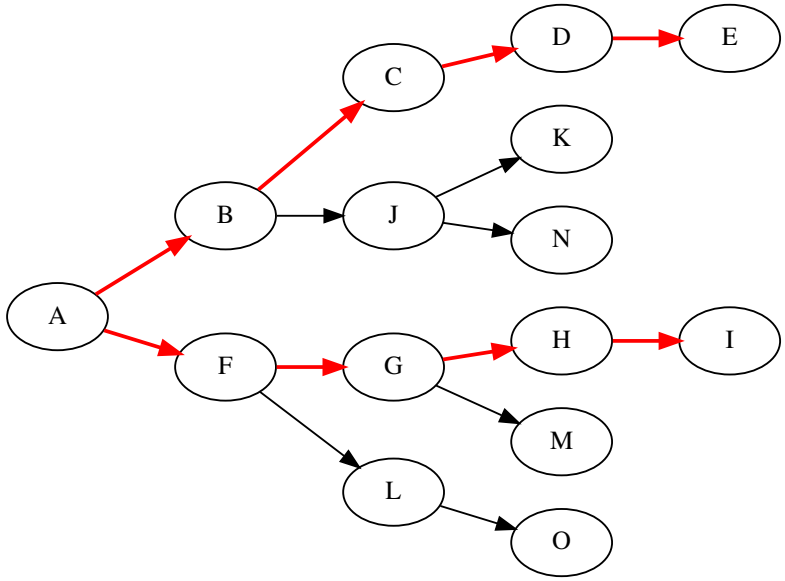
Caso 1: il percorso passa per la radice

il diametro è la somma delle due **profondità massime** dei sotto alberi + 2

In [120]:
```
print("DIAMETRO: 8 archi (9 nodi)")
G
```

```
DIAMETRO: 8 archi (9 nodi)
```

Out[120]:



In [8]:
```
G2 = graphviz.Digraph()
G2.body.append('''
    rankdir=LR
    B -> C -> D -> E [style=bold color=red]
    B -> F -> G -> H -> I [style=bold color=red]
    A -> B
    C -> J
    F -> L -> O
    G -> M
''')
```

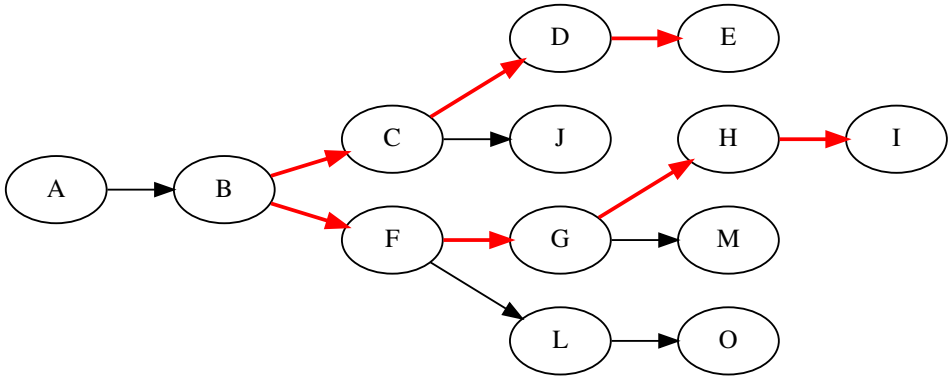
Caso 2: il percorso NON passa per la radice

il diametro è il più grande valore calcolato per ciascun nodo dell'albero

In [9]:
```
print('DIAMETRO: 7 archi (8 nodi)')
G2
```

```
DIAMETRO: 7 archi (8 nodi)
```

Out[9]:



Caso 3: il percorso NON si biforca

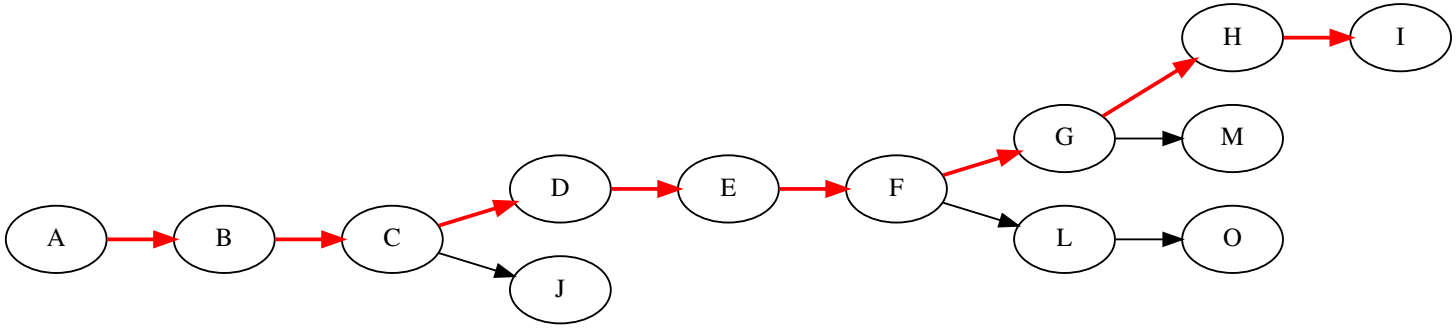
il diametro è la **profondità massima** dell'albero

In [10]:
```
G3 = graphviz.Digraph()
G3.body.append('''
    rankdir=LR
    A -> B -> C -> D -> E -> F -> G -> H -> I [style=bold color=red]
    C -> J
    F -> L -> O
    G -> M
''')
```

In [11]:
```
print("DIAMETRO 8 archi (9 nodi)")
G3
```

```
DIAMETRO 8 archi (9 nodi)
```

Out[11]:



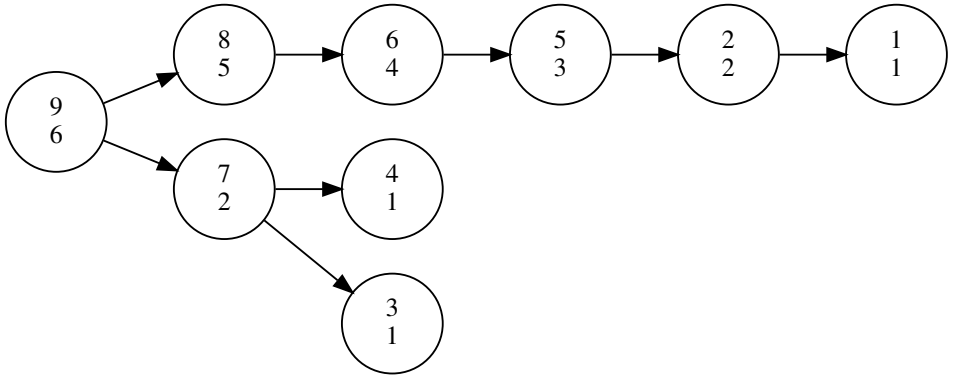
STEP 1: calcolare le altezze di tutti i sottoalberi

- visita ricorsiva
- visto che vogliamo l'**altezza** del nodo conviene lavorare **in uscita dalla ricorsione**

In [53]:
```
def aggiungi_altezze(root) -> int :
    if root is None:
        return 0
    A_sx = aggiungi_altezze(root._sx)
    A_dx = aggiungi_altezze(root._dx)
    root._altezza = max(A_sx, A_dx) +1
    return root._altezza

aggiungi_altezze(r)
r.show()
```

Out[53]:

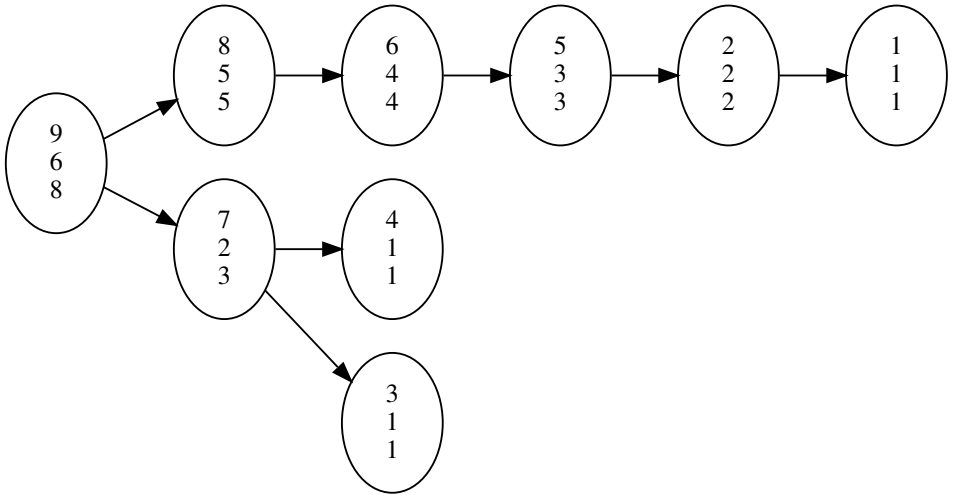


STEP 2: calcolare i percorsi supponendo che passino per ciascun nodo come radice

In [54]:
```
def aggiungi_percorsi(radice):
    "a ciascun nodo aggiungo l'attributo _percorso se passasse per quel nodo come radice"
    if radice is not None:
        A_sx = A_dx = 0
        if radice._sx:
            A_sx = radice._sx._altezza
        if radice._dx:
            A_dx = radice._dx._altezza
        radice._percorso = A_sx + A_dx + 1
        aggiungi_percorsi(radice._sx)
        aggiungi_percorsi(radice._dx)

aggiungi_percorsi(r)
r.show()
```

Out[54]:



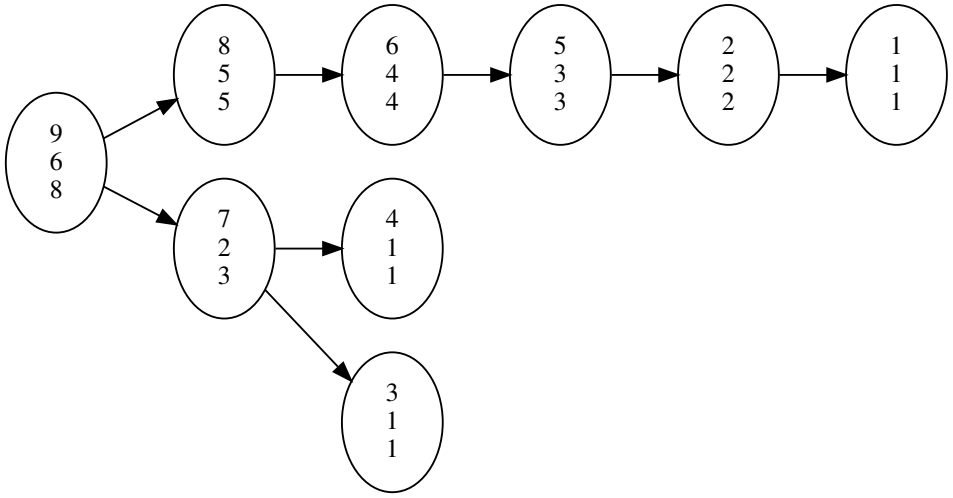
STEP 3: cercare il nodo col valore massimo di percorso o altezza

In [56]:
```
def diametro(radice):
    if not radice:
        return 0
    D_sx = diametro(radice._sx)
    D_dx = diametro(radice._dx)
    return max(D_sx, D_dx, radice._percorso, radice._altezza)

print(diametro(r))
r.show()
```

```
8
```

Out[56]:



E sugli Alberi N-ari? (TODO)

Un percorso massimo può passare:

- per la radice ed i due sottoalberi più profondi **SE ALMENO 2 FIGLI**
- per la radice ed il solo sottoalbero più profondo **SE UN SOLO FIGLIO**
- per un nodo interno ...

Soluzione: come prima

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**

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
- ...

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

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]
- ...

In [15]:
```
import graphviz
class GameNode:
    _num_nodi = 0
    def __init__(self):
        self.__class__._num_nodi += 1
        self.__id = self.__class__._num_nodi
    def dot(self):
        "Costruisco la rappresentazione dell'albero da visualizzare con Graphviz"
        if self._sons:
            s = f'{self.__id} [label={self}]\n'          # se non foglia colore nero
        else:
            s = f'{self.__id} [label={self} color=red, style=bold, shape=box] \n' # coloro le foglie di rosso
        for son in self._sons:
            s += f'{self.__id} -> {son.__id}\n'          # archi padre -> figlio
            s += son.dot()
        return s

    def show(self):
        "Creo l'oggetto Digraph che visualizza il grafo diretto"
        G = graphviz.Digraph()
        G.body.append('rankdir=LR\n' + self.dot())
        return G
```

In [63]:
```
import jdc

# 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}"'


```

In [59]:
```
S = Sequenza([1, 13, 2, 7, 9, 2])
S.show()
```

Out[59]:

[1, 13, 2, 7, 9, 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

In [64]:
```
%%add_to Sequenza

def mosse_valide(self):
    "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 ]
```

In [61]:
```
print(S)
S.mosse_valide()
```

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

Out[61]: [0, 3]

Applicare la mossa vuol dire creare una nuova sequenza

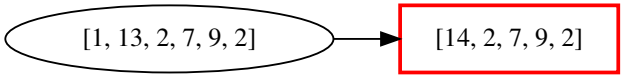
- basta sostituire i valori in posizioni 1 e i+1 con la somma

ATTENZIONE la lista originale NON va cambiata

In [66]:
```
%%add_to Sequenza
def applica_mossa(self, i):
    "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))
```

In [68]:
```
S = Sequenza([ 1, 13, 2, 7, 9, 2 ])
S.applica_mossa(0)
S.show()
```

Out[68]:



Generazione di tutto l'albero

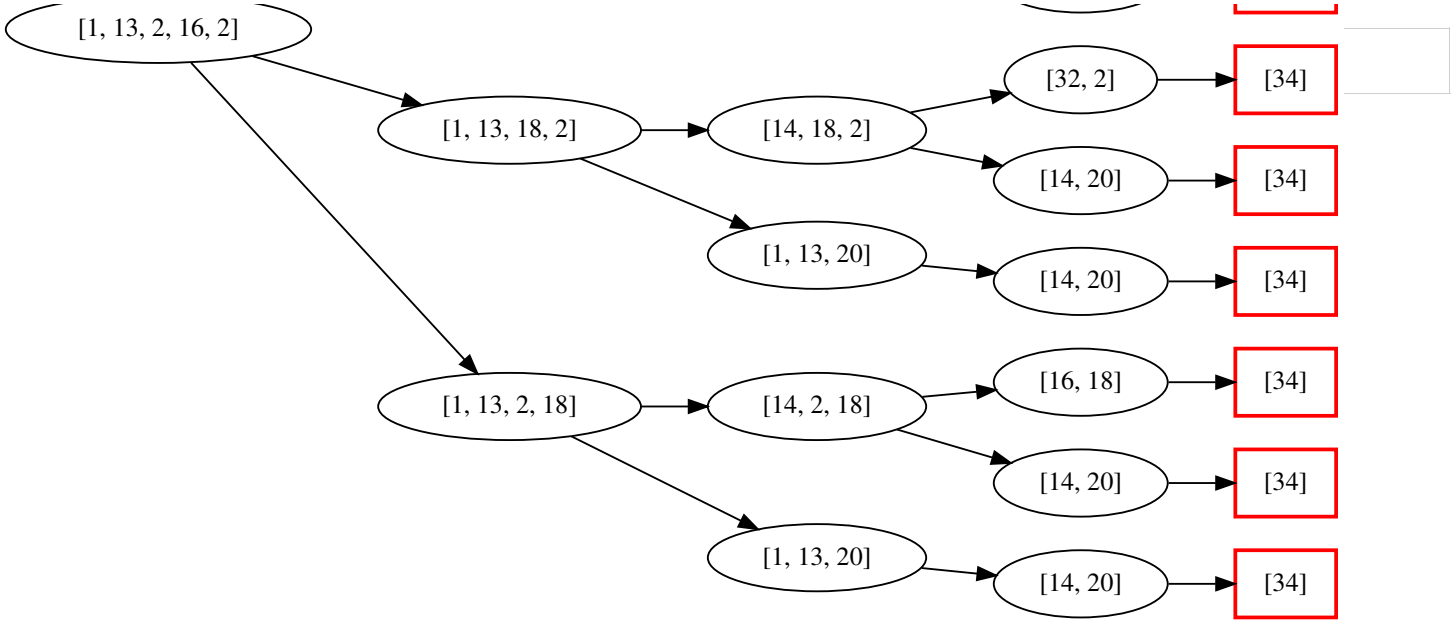
- per ogni mossa possibile generiamo il nuovo figlio
- per ogni figlio generiamo le prossime mosse

In [69]:

```
%%add_to Sequenza
def genera(self):
    "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()
```

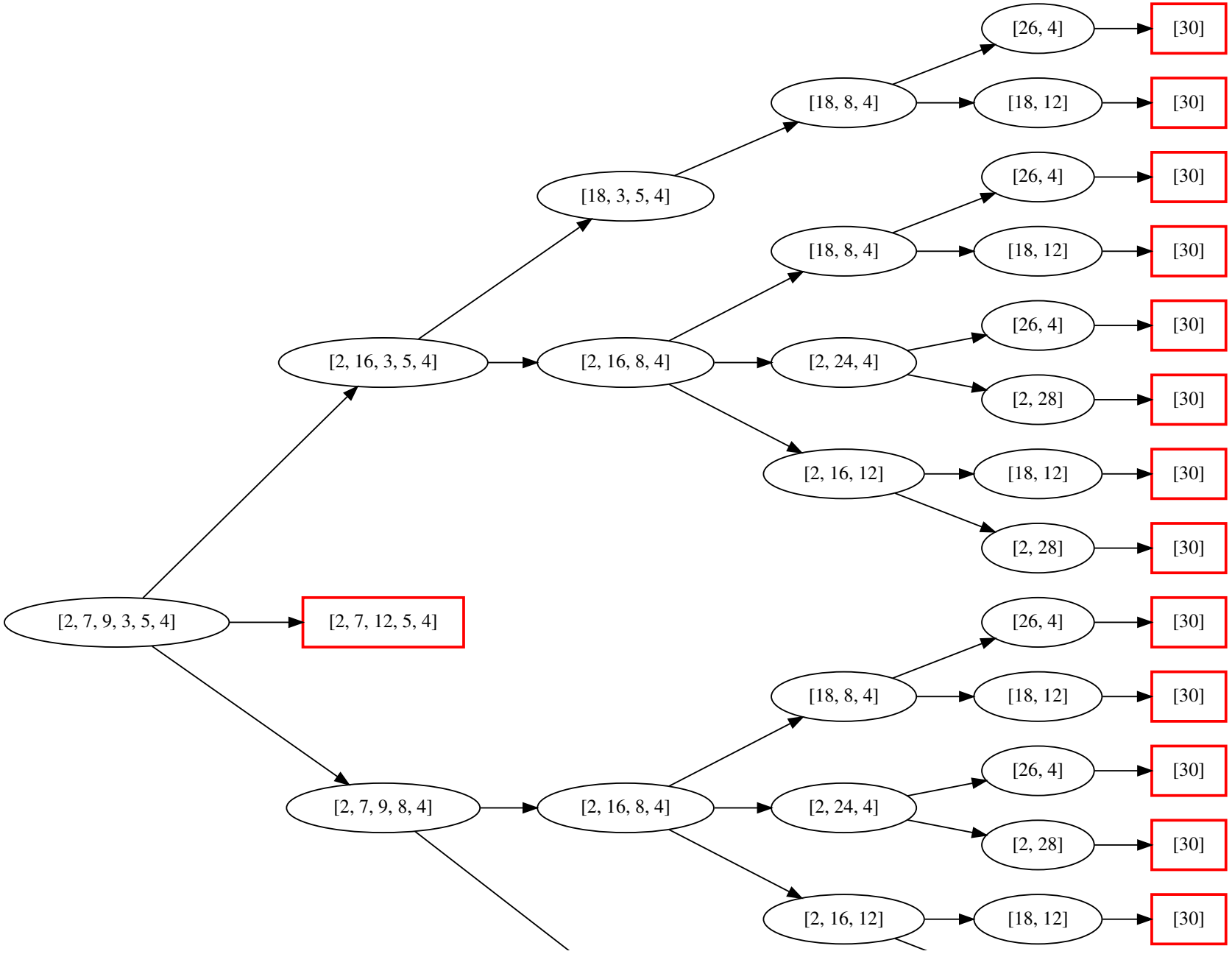
In [70]:
```
S = Sequenza([ 1, 13, 2, 7, 9, 2 ])
S.genera()
S.show()
```

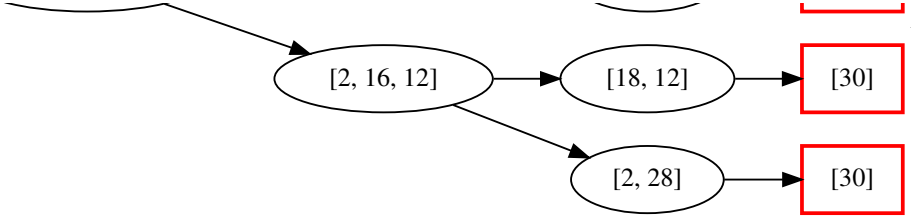
Out[70]:


[1, 13, 2, 16, 2]
[1, 13, 18, 2]
[14, 18, 2]
[32, 2]
[34]
[14, 20]
[34]
[1, 13, 20]
[14, 20]
[34]
[1, 13, 2, 18]
[14, 2, 18]
[16, 18]
[34]
[14, 20]
[34]
[1, 13, 20]
[14, 20]
[34]

In [71]:
```
1 B = Sequenza([2, 7, 9, 3, 5, 4])
2 B.genera()
3 B.show()
```

Out[71]:





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

In [72]:
```
%%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
```

In [73]:
```
B.shortest(), B.tallest()
```

Out[73]: (1, 5)

Per trovare la foglia più alta/bassa

- dobbiamo ricordare il nodo assieme alla sua profondità

In [74]:
```
%%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
```

In [75]:
```
B.tallest_leaf(), B.shortest_leaf()
```

Out[75]:
```
((5, "[30]"), (1, "[2, 7, 12, 5, 4]"))
```

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

In [76]:
```python
# 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, s2):
        "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}"'
```

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

In [78]:
```python
%%add_to Anagramma
def mosse_valide(self):
    "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):
            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
```

In [79]:
```python
A1 = Anagramma("BEDCA", "ABCDE")
A1.mosse_valide()
```

Out[79]:
```
{(0, 4), (2, 3)}
```

Per applicare la mossa

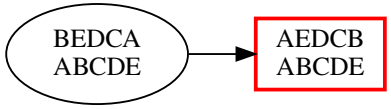
- costruisco una lista di caratteri
- scambio i due

- li trasformo in stringa
- creo il nuovo figlio

In [80]:
```
%%add_to Anagramma
def applica_mossa(self, i, j):
    "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
```

In [81]:
```
A1.applica_mossa(4,0)
A1.show()
```

Out[81]:



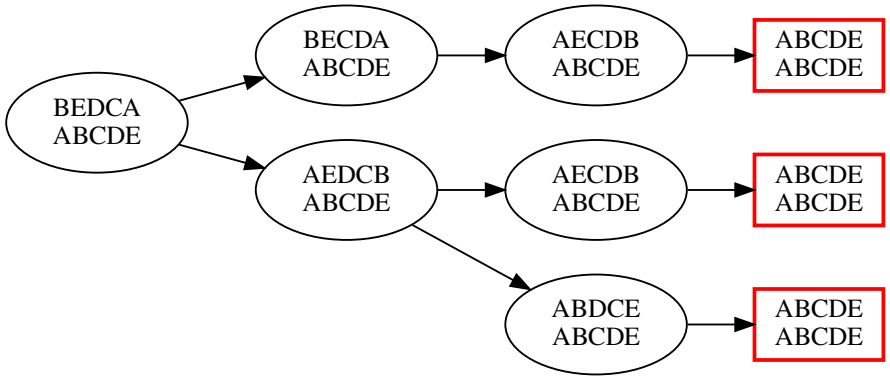
Per generare tutto l'albero

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

In [82]:
```
%%add_to Anagramma
def genera(self):
    "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()
```

In [83]:
```
A1 = Anagramma("BEDCA", "ABCDE")
A1.genera()
A1.show()
```

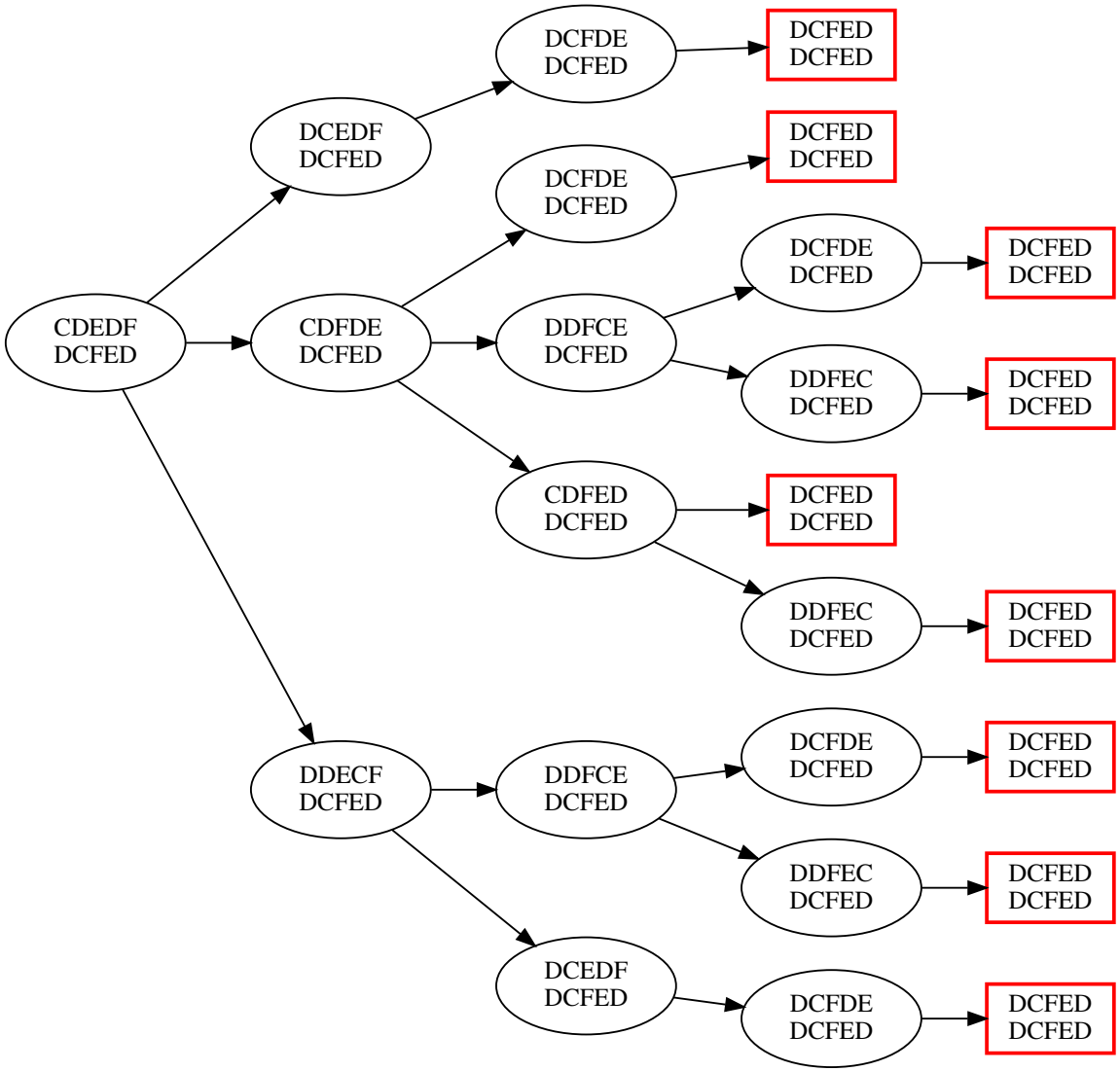
Out[83]:



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

In [84]:
```
A2 = Anagramma("CDEDF", "DCFED")
A2.genera()
A2.show()
```

Out[84]:



In [85]:
```
%%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)
```

In [86]:
```
A1 = Anagramma("BEDCA", "ABCDE")
A1.genera()
print(A1.min_mosse())
```

```
3
```

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

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

In [88]:
```
class Resto(GameNode):
    def __init__(self, N, LM, mossa=0):
        "una configurazione contiene un valore e la lista di monete disponibili e puÃ² ricordare la moneta usata per
        super().__init__()
        self._N = N
        self._LM = LM
        self._sons = []
        self._mossa = mossa

    def __repr__(self, livello=0):
        "torno la stringa da stampare per visualizzare il nodo ed i figli"
        return f'"{self._N} {self._LM}\n{self._mossa}"'
```

In [89]:
```
R = Resto(9, [5,2,1])
R.show()
```

Out[89]:
9 [5, 2, 1]
0

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

In [90]:
```
%%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, *resto = self._LM    # altrimenti ho due possibilitÃ
        return [ (prima,self._LM),  # sottraggo la prima moneta e continuo con lo stesso elenco di monete
                 (0, resto)  ]      # oppure non la sottraggo e smetto di usarla
```

In [91]:
```
R.mosse_valide()
```

Out[91]:
```
[(5, [5, 2, 1]), (0, [2, 1])]
```

Per applicare la mossa aggiorno sia N che M

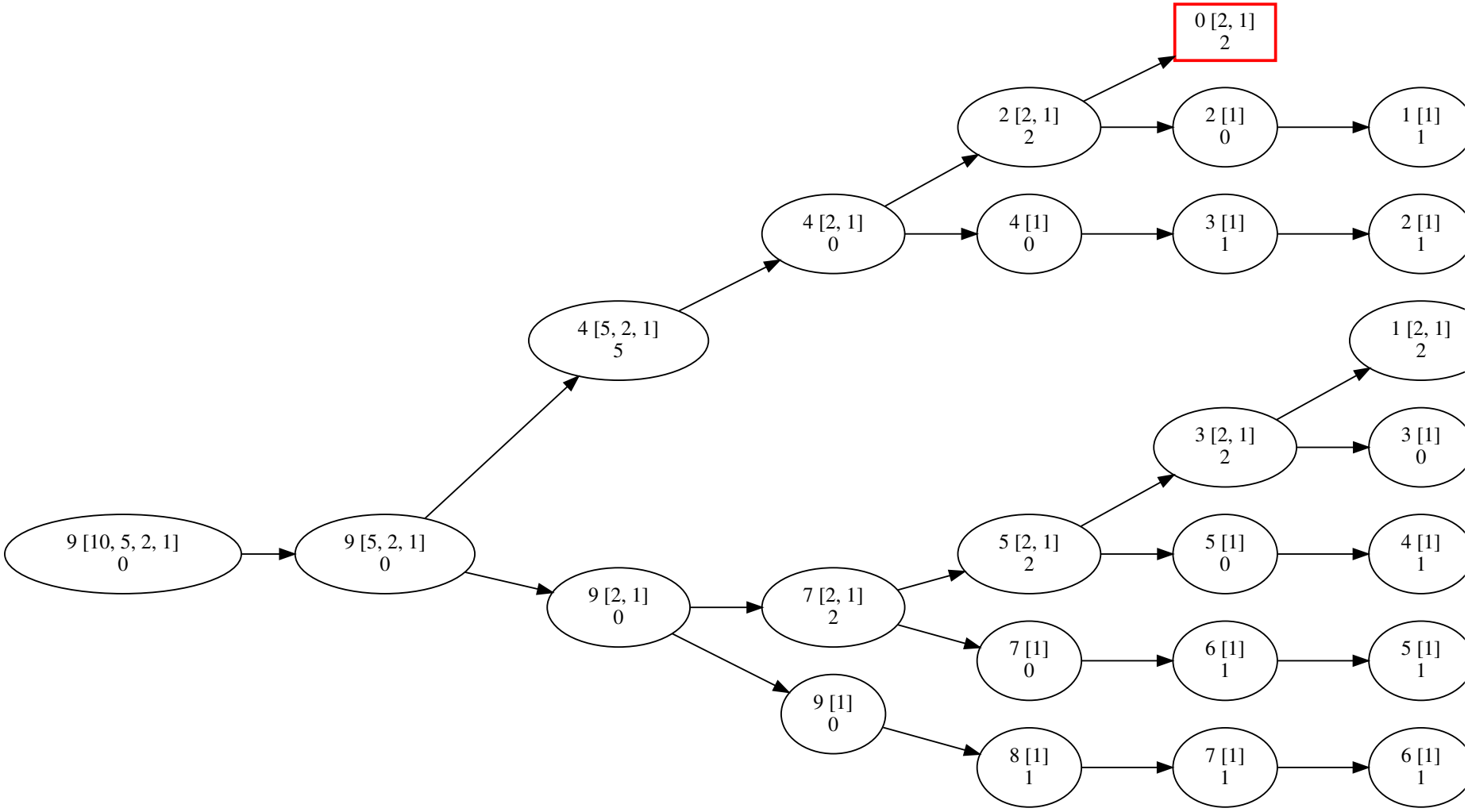
In [92]:
```
%%add_to Resto
def applica_mossa(self, moneta, LM):
    "applicazione di una mossa"
    N1 = self._N - moneta   # detraggo la moneta dal resto
    # aggiungo un nuovo figlio per il nuovo valore e con le monete indicate e mi ricordo che moneta ho sottratto
    self._sons.append(Resto(N1, LM, moneta))
```

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

In [93]:
```
%%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()
```

In [94]:
```
R = Resto(9, [10,5,2,1])
R.genera()
R.show()
```

Out[94]:



Per trovare le soluzioni

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

In [95]:
```
%%add_to Resto
def soluzioni(self):
    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 ] ]


```

In [96]:
```
print(*R.soluzioni(), sep='\n')

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

[ [ m for m in soluzione if m] for soluzione in R.soluzioni()]
```

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

Out[96]:
```
[[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]]
```

In [48]:
```
Sequenza._num_nodi, Anagramma._num_nodi, Resto._num_nodi
```

Out[48]:
```
(110, 45, 50)
```