Corso di laurea in Informatica
Progettazione d'algoritmi

Tecnica Backtracking 1

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO 1

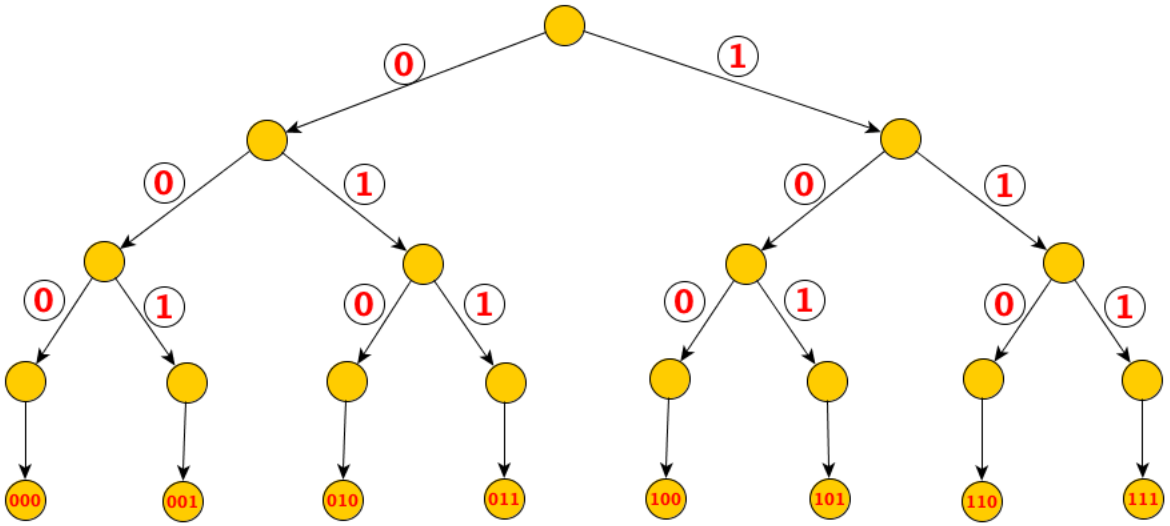
Progettare un algoritmo che prende come parametro un intero n e stampa tutte le stringhe binarie lunghe n.

Ad esempio: per $n = 3$ l'algoritmo deve stampare $2^3 = 8$ stringhe:
000
001
010
100
011
101
110
111

OSSERVAZIONE:
- Le stringhe da stampare sono 2^n
- stampare una stringa lunga n costa $\Theta(n)$

il meglio che ci si può augurare per un algoritmo che risolve questo problema è una complessità $\Omega(2^n \cdot n)$

Albero di ricerca nel caso di $n = 3$




```
def bk(n, i=0, sol=[]):
    if i==n:
        print(''.join(sol))
        return
    sol.append('0')
    bk(n,i+1,sol)
    sol.pop()
    sol.append('1')
    bk(n,i+1,sol)
    sol.pop()
```

```
>>> bk(2)
00
01
10
11
```

L'albero binario di ricorsione ha $2^n - 1$ nodi interni e 2^n foglie.

- ciascun nodo interno richiede tempo $O(1)$
- ciascuna foglia richiede tempo $O(n)$

La complessità dell'algoritmo è $O\left(2^n \cdot n\right)$

ESERCIZIO 2

Progettare un algoritmo che prende come parametro due interi n e k, con $0 \leq k \leq n$, e stampa tutte le stringhe binarie lunghe n che contengono al più k uni.

Ad esempio: per $n = 4$ e $k = 2$, delle $2^4 = 16$ stringhe lunghe n bisogna stampare le seguenti 11:

0000 0001 0010
0100 1000 0011
0101 1001 0110
1010 1100

Un possibile algoritmo che risolve il problema in $\Theta(2^n \cdot n)$

```
def bk1(n, k, i=0, sol=[]):
    if i==n:
        if sol.count('1') <=k:
            print(''.join(sol))
        return
    sol.append('0')
    bk1(n, k, i+1, sol)
    sol.pop()
    sol.append('1')
    bk1(n, k, i+1, sol)
    sol.pop()
```

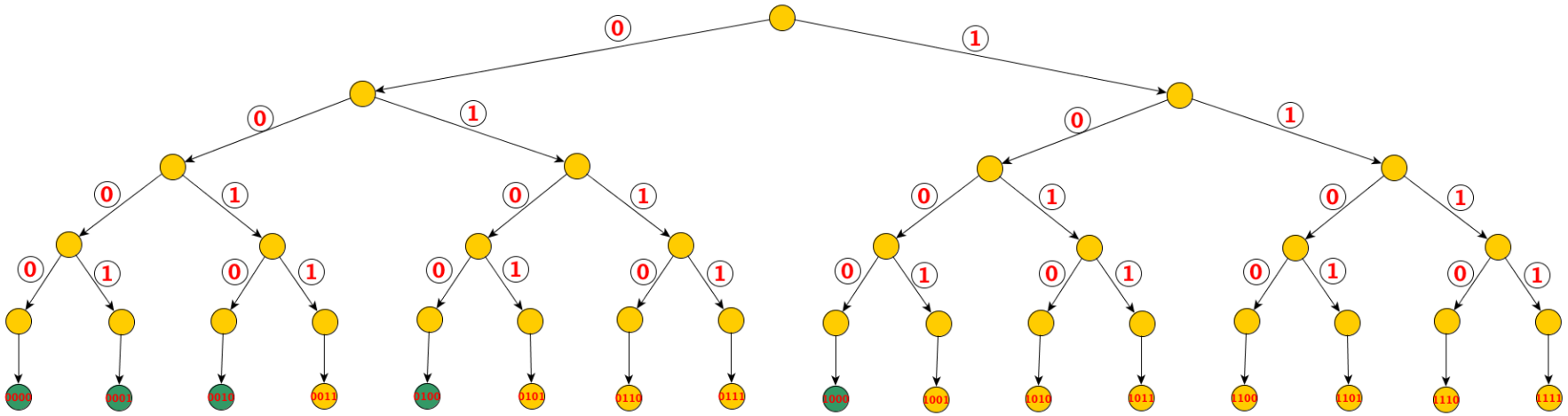
```
>>> bk1(4,2)
0000
0001
0010
0011
0100
0101
0110
1000
1001
1010
1100
```

Indichiamo con $S(n,k)$ il numero di stringhe che bisogna stampare.
Un buon algoritmo per questo problema dovrebbe avere una complessità proporzionale alle stringhe da stampare, vale a dire $O(S(n,k) \cdot n)$ (poche stringhe = poco tempo).

Ad esempio per $k = 2$ si ha

$$S(n,k) = 1 + n + \binom{n}{2} = \Theta(n^2)$$

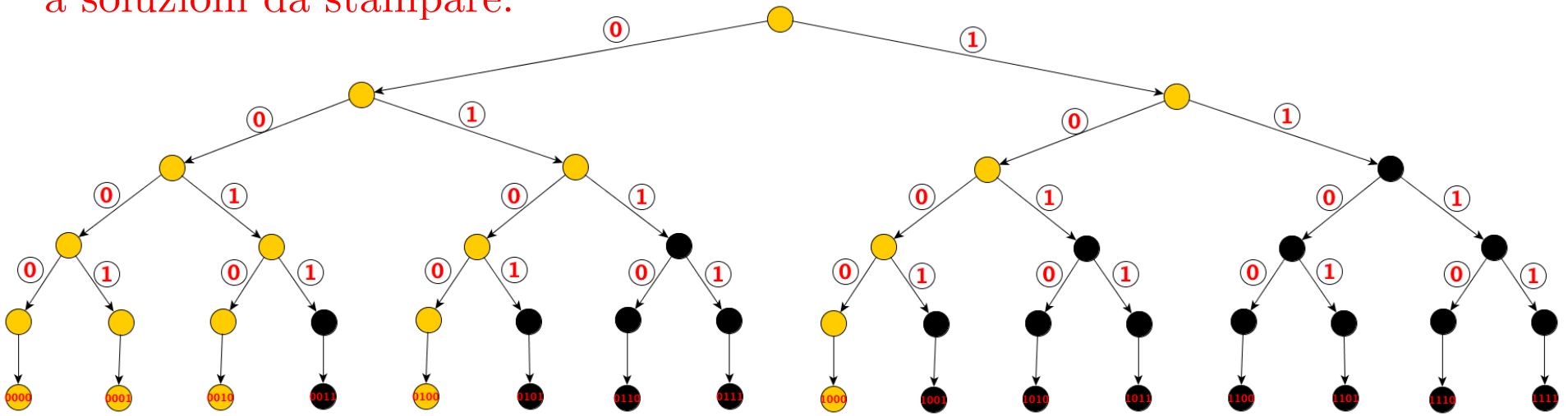
e quindi un buon algoritmo per $k = 2$ dovrebbe avere complessità polinomiale $O(n^3)$ mentre l'algoritmo proposto ha complessità esponenziale $\Theta(2^n n)$ (indipendente da k)



Albero di ricorsione generato dall'algoritmo proposto nel caso di $n = 4$ e $k = 1$ con in verde le foglie che vanno stampate.

Osservazione:

Inutile generare nell'albero di ricorsione nodi che non hanno possibilità di portare a soluzioni da stampare.

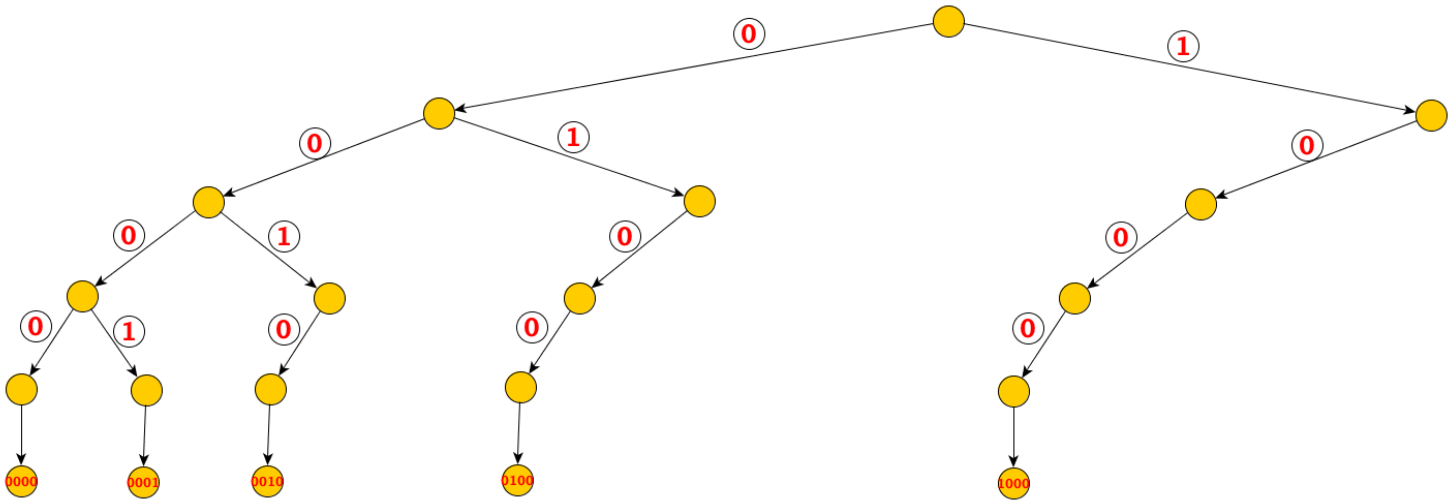


In nero i nodi che potrei evitare di generare nel caso in cui $n = 4$ e $k = 1$

Algoritmo alternativo con un controllo sul numero di uni presenti nel prefisso delle stringhe che si vanno via via generando

```
def bk2(n, k, i=0, tot1 = 0, sol=[]):
    if i==n:
        print(''.join(sol))
    return
    sol.append('0')
    bk2(n, k, i+1, tot1, sol)
    sol.pop()
    if tot1<k:
        sol.append('1')
        bk2(n, k, i+1, tot1+1, sol)
        sol.pop()
```

Albero di ricorsione generato dalla funzione con *n=4* e *k=1* grazie all'introduzione della **funzione di taglio**



per calcolare i tempi di calcolo del proprio programmino:

```
from time import time

def tempi():
    from time import time
    start = time()
    #invoca  qui il programmino per il quale vuoi calcolare i tempi
    end= time()
    print(end -start)
```

se invoco *bk1(24, 2)* o *bk2(24, 2)* in entrambi i casi verranno stampate le $1 + 24 + \frac{24 \cdot 23}{2} = 301$ stringhe binarie con al più 2 uni.

I tempi di calcolo delle due funzioni saranno però molto diversi. Commentando le istruzioni di print (funzioni di output piuttosto costose) che influiscono in antrambi i programmi allo stesso modo, si ottiene quanto segue:

- tempo di calcolo per *bk1(24, 2)*: 18.11323380
- tempo di calcolo per *bk2(24, 2)*: 0.00232100

Si consideri un algoritmo di enumerazione basato sul backtraking dove l'albero di ricorsione ha altezza h, il costo di una foglia è $g(n)$ e il costo di un nodo interno è $O(f(n))$.
Se l'algoritmo gode della seguente proprietà:

un nodo viene generato solo se ha la possibilità di portare ad una foglia da stampare.

Allora la complessità dell'algoritmo è proporzionale al numero di cose da stampare $S(n)$, più precisamente la complessità dell'algoritmo è:

$$O\left(S(n) \cdot h \cdot f(n) + S(n) \cdot g(n)\right))$$

questo perché:

- il costo totale dei nodi foglia sarà $O(S(n) \cdot g(n))$ (in quanto solo le foglie da enumerare verranno generate).
- i nodi interni dell'albero che verranno effettivamente generati saranno $O(S(n) \cdot h)$ (in quanto ogni nodo interno generato apparterrà ad un cammino che parte dalla radice e arriva ad una delle $S(n)$ foglie da enumerare).

```python
def bk2(n, k, i=0, tot1 = 0, sol=[]):
    if i==n:
        print(''.join(sol))
    return
    sol.append('0')
    bk2(n, k, i+1, tot1, sol)
    sol.pop()
    if tot1<k:
        sol.append('1')
        bk2(n, k, i+1, tot1+1, sol)
        sol.pop()
```

Per l'algoritmo codificato con la funzione $bk2(n,k)$. La proprietà di generare un nodo solo se questo può portare ad una delle $S(n,k)$ foglie da stampare è rispettata.
Inoltre $h = n$, $g(n) = \Theta(n)$, $f(n) = O(1)$.
Quindi la complessità dell'algoritmo è:

$$S(n,k) \cdot n \cdot O(1) + S(n,k) \cdot \Theta(n) = \Theta(S(n,k) \cdot n)$$

e l'algoritmo risulta ottimale.

La complessità dell'algoritmo è $O\left(n^{k+1}\right)$
infatti: $S(n,k) = \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{k} < 2 \cdot n^k$

ESERCIZIO 3

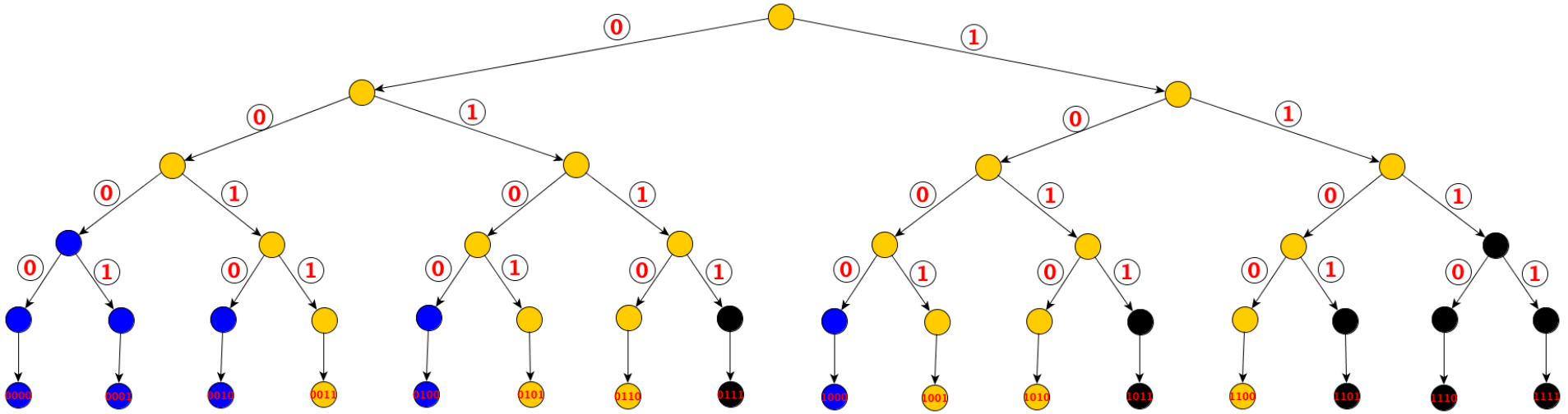
Progettare un algoritmo che prende come parametro due interi n e k, con $0 \leq k \leq n$, e stampa tutte le stringhe binarie lunghe n che contengono ESATTAMENTE k uni.

Ad esempio: per $n = 6$ e $k = 3$, delle $2^6 = 64$ stringhe lunghe n bisogna stampare le seguenti 20:

000111	001011	001101	001110
010011	010101	010110	011001
011010	011100	100011	100101
100110	101001	101010	101100
110001	110010	110100	111000

rispetto all'esercizio precedente aggiungiamo ora una funzione di taglio anche nel caso in cui al prefisso viene aggiunto uno zero. Bisogna assicurarsi infatti che sia sempre possibile completare quel prefisso in modo da ottenere una stringa valida da stampare.

```python
def bk3(n, k, i=0, tot1 = 0, sol=[]):
    if i==n:
        print(''.join(sol))
    return
    if tot1 + n - (i+1) >=k:
        sol.append('0')
        bk3(n, k, i+1, tot1, sol)
        sol.pop()
    if tot1<k:
        sol.append('1')
        bk3(n, k, i+1, tot1+1, sol)
        sol.pop()
```



l'albero di ricorsione generato da $bk3(n,k)$ con $n = 4$ e $k = 2$ con in nero i nodi che evito di generare grazie alla funzione di taglio sugli 1 ed in blu i nodi che evito di generare grazie alla funzione di taglio sugli 0.

Per l'algoritmo codificato con la funzione $bk3(n,k)$. La proprietà di generare un nodo solo se questo può portare ad una delle $S(n,k)$ foglie da stampare è rispettata.
Inoltre

- altezza dell'albero di ricorsione $h = n$,
- tempo richiesto da una foglia $g(n) = \Theta(n)$
- tempo richiesto da un nodo interno $f(n) = O(1)$

Quindi la complessità dell'algoritmo è:

$$S(n,k) \cdot n \cdot O(1) + S(n,k) \cdot \Theta(n) = \Theta(S(n,k) \cdot n)$$

e l'algoritmo risulta ottimale.

La complessità dell'algoritmo è $O\left(n^{k+1}\right)$
infatti: $S(n,k) = \binom{n}{k} < n^k$

ESERCIZIO. esame settembre 2020.
Progettare un algoritmo che, dato un intero n, stampa tutte le stringhe sull'alfabeto dei 3 simboli a, b e c, in cui il numero delle b supera quello di ciascuno degli altri due simboli.
L'algoritmo proposto deve avere complessità $O(nD(n))$ dove $D(n)$ è il numero di stringhe da stampare.

Ad esempio per $n = 2$ l'algoritmo deve stampare la sola stringa bb mentre per $n = 3$ le stringhe da stampare sono le seguenti:

$$bbb, \quad abb, \quad bab, \quad bba, \quad cbb, \quad bcb, \quad bbc$$

Motivare bene la correttezza e la complessità dell'algoritmo proposto

- Per la stampa delle stringhe eseguiremo una funzione di backtracking.
- Al passo ricorsivo i-esimo possiamo potenzialmente inserire uno dei tre simboli b, a e c dell'alfabeto. Per ottenere una complessità proporzionale alle $D(n)$ stringhe da stampare la funzione di backtraking controlla che la scelta di inserire uno dei tre simboli venga fatta solo se la soluzione parziale che si ottiene porta ad un elemento da stampare. A questo proposito utilizziamo una sempilce funzione di taglio che si avvale di due contatori (na con il numeor delle a presenti nella soluzione parziale e nc con il numero delle c presenti nella soluzione parziale):
  - **inseriamo il simbolo** b questo simbolo puó sempre essere inserito
  - **inseriamo il simbolo** a solo se l'inserimento di a lascia spazio sufficiente per inserire eventualmente altri simboli di tipo b in modo da rendere alla fine il numero di b presenti nella stringa superiore a quello delle a e delle b. Il numero delle a a seguito dell'inserimento diventa $na+1$ il numero delle b potenziali é $n-(na+nc+1)$ il numero delle c resta nc. Deve quindi aversi $n-(na+nc+1) > na+1$ e anche $n-(na+nc+1) > nc$.
  - **inseriamo il simbolo** c solo se l'inserimento di c lascia spazio sufficiente per inserire eventualmente altri simboli di tipo b in modo da rendere alla fine il numero di b presenti nella stringa superiore a quello delle c e quello delle a. Il numero delle c a seguito dell'inserimento diventa $nc+1$ il numero delle b potenziali é $n-(na+nc+1)$ quello delle a resta na. Deve quindi aversi $n-(na+nc+1) > nc+1$ e anche $n-(na+nc+1) > na$

```python
def es(n, i=0, na=0, nc = 0, sol=[]):
    if i == n:
        print(''.join(sol))
        return
    sol.append('b')
    es(n, i+1, na, nc, sol)
    sol.pop()
    if n - (na + nc + 1) > na + 1 and n - (na + nc + 1) > nc:
        sol.append('a')
        es(n, i+1, na+1, nc, sol)
        sol.pop()
    if n - (na + nc + 1) > nc + 1 and n - (na + nc + 1) > na:
        sol.append('c')
        es(n, i+1, na, nc + 1, sol)
        sol.pop()
```

```
>>> es(4)
bbbb
bbba
bbbc
bbab
bbac
bbcb
bbca
babb
babc
bacb
bcbb
bcba
bcab
abbb
abbc
abcb
acbb
cbbb
cbba
cbab
cabb
```

- Nota che nell'albero di ricorsione prodotto dall'esecuzione di $es2R$ un nodo viene generato solo se porta ad una foglia da stampare.
- Possiamo quindi dire che la complessità dell'esecuzione di $es2R(0, n, 0, 0, [\,],)$ richiederà tempo

$$O(D(n) \cdot h \cdot f(n) + D(n) \cdot g(n))$$

  dove:

  - $h = n$ è l'altezza dell'albero.
  - $f(n) = O(1)$ è il lavoro di un nodo interno.
  - $g(n) = O(n)$ è il lavoro di una foglia
- Quindi il costo di $es2R(0, n, 0, 0, [\,],)$ è $O(nD(n))$.
- Otteniamo quindi che il costo di $es2(n)$ è $O(nD(n))$.

ESERCIZIO. esame marzo 2023.
Fissiamo n, k e T positivi con $T \leq n\dot{k}$. Definiamo **valida** una sequenza di lunghezza n contenente interi da 0 a k e la cui somma è T.

Ad esempio per $n = 6$, $k = 4$ e $T = 12$ allora la sequenza 132042 è **valida** mentre 121213 **non è valida**.

Trovate un algoritmo che dati n, k e T stampi tutte e sole le sequenze valide. L'algoritmo deve avere complessità $O(n \cdot k \cdot S(n, k))$ dove $S(n, k)$ è il numero di sequenze valide esistenti.

Motivare BENE la correttezza e la complessità dell'algoritmo proposto.

Algoritmo:

- Utilizziamo un algoritmo di backtracking per la stampa di tutte le stringhe lunghe n sull'alfabeto $\{0, \ldots, k\}$ con l'aggiunta di una funzione di taglio.
- Per la funzione di taglio possiamo ragionare come segue: ad ogni inserimento nella sequenza che via via costruiamo teniamo traccia in una variabile tot della somma degli interi utilizzati. Dopo l'inserimento dei primi $i-1$ valori nella sequenza, l'inserimento del valore j, con $0 \leq j \leq k$ avverrà se e solo se la sequenza potrà poi completarsi in modo che la somma dei suoi elementi sia almeno T. In altri termini l'inserimento di j avviene se e solo se entrambi i seguenti vincoli sono soddisfatti:
  - $tot + j + (n - i - 1) * k \geq T$
  - $tot + j \leq T$
- grazie alla funzione di taglio la soluzione parziale via via costruita è sempre un prefisso di una soluzione da stampare.

```python
def es(n,k,T,i=0,tot=0, sol=[]):
    if i == n:
        print(sol)
        return
    for j in range(k+1):
        if tot + j + (n-i-1) * k >= T and tot+j <=T:
            sol.append(j)
            es(n, k, T, i+1, tot+j, sol)
            sol.pop()
```

```
>>> es(3,3,6)
[0, 3, 3]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 2, 2]
[2, 3, 1]
[3, 0, 3]
[3, 1, 2]
[3, 2, 1]
[3, 3, 0]
```

- Nota che nell'albero di ricorsione prodotto dall'esecuzione dell'algoritmo un nodo viene generato solo se porta ad una foglia da stampare.
- Possiamo quindi dire che la complessità dell'esecuzione di $es3(n, k, T, i = 0, tot = 0, sol = [\,])$ richiederà tempo

$$O(S(n,k) \cdot h \cdot f(n) + S(n,k) \cdot g(n))$$

dove:

  - $h = n$ è l'altezza dell'albero.
  - $f(n) = \Theta(k)$ è il lavoro di un nodo interno.
  - $g(n) = \Theta(n)$ è il lavoro di una foglia

- Quindi il costo è $O(n \cdot k \cdot S(n,k))$.

ESERCIZIO

Progettare un algoritmo che, data una stringa X lunga n sull'alfabeto ternario $\{'0', '1', '2'\}$, stampa tutte le stringhe che è possibile ottenere da X sostituendo il simbolo $'*'$ ad alcuni dei suoi caratteri in modo che i caratteri rimanenti risultino in ordine strettamente crescente.

Ad esempio:

- per $X = '021'$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le stringhe:

$$***,\ 0**,\ 02*,\ 0*1,\ *2*,\ **1$$

per $X = '2110'$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le stringhe:

$$****,\ 2***,\ *1**,\ **1*,\ ***0$$

L'algoritmo proposto deve avere complessità $O(nS(X))$ dove $S(X)$ è il numero di stringhe da stampare.

Motivare BENE la correttezza e la complessità dell'algoritmo proposto.

Algoritmo:

Sostanzialmente nelle soluzioni che si vanno costruendo in ciascuna posizione o inseriamo il simbolo * o lasciamo il simbolo della stringa X. Partiamo dunque dall'algoritmo per la stampa delle stringhe binarie e aggiungiamo delle funzioni di taglio che assicurino che uno dei due simboli viene aggiunto alla soluzione parziale solo se quella soluzione potrà poi completarsi in una stringa da stampare.

- Il simbolo * può sempre essere aggiunto quindi non c'è bisogno di funzione di taglio
- Il simbolo della stringa può essere aggiunto solo se è il primo di questa nuova stringa o è maggiore dell'ultimo simbolo diverso da * inserito finora. C'è dunque in questo caso bisogno di una funzione di taglio.

Per far si che la funzione di taglio sia calcolabile in tempo $O(1)$ utilizziamo una variabile p che tiene traccia dell'ultimo simbolo diverso da * inserito nella stringa che si va costruendo ($p = *$ se la stringa non contiene simboli diversi da *).

IMPLEMENTAZIONE:

```
def es(X, i = 0, p = '*', sol = []):
    if i== len(X):
        print(''.join(sol))
        return
    sol.append('*')
    es(X, i+1, p, sol)
    sol.pop()
    if p=='*' or X[i] > p:
        sol.append(X[i])
        es(X, i+1, X[i], sol)
        sol.pop()
```

```
>>> X='0123'
>>> es(X)
****
***3
**2*
**23
*1**
*1*3
*12*
*123
0***
0**3
0*2*
0*23
01**
01*3
012*
0123
```

- Nota che nell'albero di ricorsione prodotto dall'esecuzione dell'algoritmo un nodo viene generato solo se porta ad una foglia da stampare.
- Possiamo quindi dire che la complessità dell'algoritmo richiederà tempo

$$O(S(X) \cdot h \cdot f(n) + S(X) \cdot g(n))$$

  dove:

  - $h = n$ è l'altezza dell'albero.
  - $f(n) = O(1)$ è il lavoro di un nodo interno.
  - $g(n) = \Theta(n)$ è il lavoro di una foglia

- Quindi il costo è $\Theta(n \cdot S(X))$.

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO. esame ottobre 2020.
Progettare un algoritmo che, data una stringa X di lunghezza n, sull'alfabeto dei 3 simboli 0, 1 e 2, stampa tutte le stringhe di lunghezza n sullo stesso alfabeto che differiscono da X in ciascuna posizione.
L'algoritmo proposto deve avere complessità $O(nD(n))$ dove $D(n)$ è il numero di stringhe da stampare.

Ad esempio per $X =' 020'$ l'algoritmo deve stampare le seguenti 8 stringhe (non necessariamente nello stesso ordine):

$$101,\ 102,\ 111,\ 112,\ 201,\ 202,\ 211,\ 212$$

Motivare bene la correttezza e la complessità dell'algoritmo proposto

ESERCIZIO. esame gennaio 2021.
Progettare un algoritmo che, dato un intero n, stampa tutte le stringhe binarie lunghe n in cui non appaiono mai 3 simboli uguali consecutivi.
L'algoritmo proposto deve avere complessità $O(nD(n))$ dove $D(n)$ è il numero di stringhe da stampare.

Ad esempio per $n = 3$ l'algoritmo dve stampare, non necessariamente nello stesso ordine, le seguenti 6 stringhe:

$$001,\ 010,\ 011,\ 100,\ 101,\ 110$$

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

ESERCIZIO. esame gennaio 2021.
Progettare un algoritmo che, dato un intero n, stampa tutte le stringhe binarie lunghe n in cui non appaiono mai 3 simboli uguali consecutivi.
L'algoritmo proposto deve avere complessità $O(nD(n))$ dove $D(n)$ è il numero di stringhe da stampare.

Ad esempio per $n = 3$ l'algoritmo dve stampare, non necessariamente nello stesso ordine, le seguenti 6 stringhe:

$$001,\ 010,\ 011,\ 100,\ 101,\ 110$$

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

ESERCIZIO. esame gennaio 2023.
Data una stringa binaria s, la differenza tra il numero di 1 e il numero di 0 nelle prime i posizioni di s è detto vantaggio alla posizione i in s.

Ad esempio per $s = 010010111000$ il vantaggio nelle dodici posizioni è rispettivamente: $-1,\ 0,\ -1,\ -2,\ -1,\ -2,\ -1,\ 0,\ 1,\ 0,\ -1,\ -2$.

Dati due interi positivi n ed a, $a \geq 1$, con $S(n,\, a)$ definiamo l'insieme di stringhe binarie di lunghezza n il cui vantaggio in ogni posizione cade nell'intervallo che va da $-a$ ad a.

Ad esempio: $S(5, 2) = \{01010\ \ 01011\ \ 01100\ \ 01101\ \ 10010\ \ 10011\ \ 10100\ \ 10101\}$

Trovate un algoritmo che dati n ed a stampi tutte le stringhe in $S(n,\, a)$. La complessità dell'algoritmo deve essere $O(n \cdot |S(n,\, a)|)$.

Motivare BENE la correttezza e la complessità dell'algoritmo proposto.