Esercizio 1 Si consideri il seguente problema. Dato un grafo $G = (V, E)$ non diretto ed aciclico trovare una copertura tramite nodi di G di cardinalità minima. Si ricorda che una copertura C è un sottoinsieme dei nodi tale che ogni arco del grafo è incidente ad almeno uno dei nodi in C. Si propone il seguente algoritmo:

```
INPUT un grafo non diretto ed aciclico G = (V, E)
    SOL ← ∅
    A ← E
    WHILE A ≠ ∅ DO
        Scegli un arco {u, v} in A tale che u non è incidente ad altri archi in A
        SOL ← SOL ∪ {v}
        Elimina da A tutti gli archi incidenti in v
    ENDWHILE
OUTPUT SOL.
```

1. *(max 15 punti)* Dire se l'algoritmo risolve il problema e in caso affermativo dimostrare la correttezza dell'algoritmo mentre in caso negativo fornire un controesempio.

2. *(max 10 punti)* Assumendo che il grafo venga rappresentato tramite liste di adiacenza, descrivere una implementazione dell'algoritmo con complessità asintotica **strettamente** inferiore a $O(n^2)$, dove n è il numero dei nodi.

Soluzione Esercizio 1

1. L'algoritmo risolve il problema.
   Per provare che l'algoritmo è corretto è sufficiente dimostrare che ogni nuova estensione della soluzione parziale (tramite un passo del WHILE) mantiene la proprietà di essere contenuta in una soluzione ottima. Sia SOL_i la soluzione costruita fino all'i-esimo passo del WHILE e sia $SOL_0 = \emptyset$.

   Per ogni passo $i \geq 0$ *del* WHILE *esiste una soluzione ottima* SOL^* *tale che* $SOL_i \subseteq SOL^*$.

   Dimostrazione: per induzione su i.
   Per $i = 0$ è banalmente vero. Supponiamo ora che sia vero per i e dimostriamolo per $i+1$. Per ipotesi induttiva esiste una soluzione ottima SOL^* tale che $SOL_i \subseteq SOL^*$. Se vale anche $SOL_{i+1} \subseteq SOL^*$ allora abbiamo fatto. Altrimenti $SOL_{i+1} \not\subseteq SOL^*$. Sia $\{u, v\}$ l'arco scelto all'$(i+1)$-esimo passo e v il nodo aggiunto alla soluzione. Risulta $v \notin SOL^*$, perchè l'arco $\{u, v\}$ sia coperto dalla soluzione SOL^* deve necessariamente aversi $u \in SOL^*$. Definiamo $\overline{SOL} = (SOL^* - \{u\}) \cup \{v\}$. Poichè per ogni nodo scelto dall'algoritmo tutti gli archi ad esso incidenti vengono eliminati dall'insieme A, la presenza in A dell'arco $\{u, v\}$ al passo $(i+1)$-esimo del WHILE implica che $u \notin SOL_i$ quindi per ipotesi induttiva si ha $SOL_{i+1} \subseteq \overline{SOL}$. Resta quindi solo da dimostrare che $\overline{SOL}$, al pari di SOL^*, è un ricoprimento di G.
   Gli unici archi di E che potrebbero risultare non coperti al seguito dell'eliminazione di u da SOL^* sono quelli incidenti in u. Di tutti questi archi uno solo compare in A ed è l'arco $\{u, v\}$ che è certamente coperto in $\overline{SOL}$ grazie al nodo v. Gli eventuali altri archi incidenti in u ed appartenenti ad $E - A$ sono certamente coperti da nodi in SOL_i in quanto l'algoritmo elimina un arco da A solo quando un estremo dell'arco viene inserito nella soluzione. Quindi tutti questi archi sono coperti da nodi in $SOL_i - \{u\}$ e di conseguenza, poichè $SOL_i - \{u\} \subset \overline{SOL}$, sono coperti anche da nodi in $\overline{SOL}$.
   Abbiamo appena dimostrato che $\overline{SOL}$ è una copertura, il fatto che sia anche di cardinalità minima segue immediatamente notando che SOL^* era una copertura di cardinalità minima e $|SOL^*| = |\overline{SOL}|$.

Rimane da far vedere che la soluzione prodotta non solo è contenuta in una soluzione ottima ma coincide con una soluzione ottima. Per questo è sufficiente osservare che la soluzione prodotta è una copertura.

2. Un modo per dare un'implementazione efficiente dell'algoritmo è il seguente. Usiamo un heap H per mantenere i nodi incidenti ad archi in A ordinati per grado di modo che in cima all'heap ci sia un nodo di grado minimo. Usiamo un vettore P tale che, per ogni nodo v, se v è nell'heap allora $P[v]$ è la posizione di v in H, altrimenti $P[v]$ è -1. La soluzione SOL può essere rappresentata tramite una lista. Ad ogni iterazione del WHILE si estrae il nodo u in cima all'heap. Si aggiorna $P[u]$ ponendolo uguale a -1. Nella lista degli adiacenti a u si cerca il nodo v che è in H vale a dire l'unico nodo v, adiacente a u, tale che $P[v] \neq -1$. Si aggiunge v a SOL. Poi si scandisce la lista degli adiacenti di v e, per ogni adiacente x, se $P[x] \neq -1$ si decrementa x in H e se il grado diventa zero x viene estratto da H e si pone $P[x] = -1$.

   L'estrazione del nodo dall' heap ha un costo $O(n \log n)$. Le operazioni di ricerca del nodo v costano complessivamente $O(n)$ perchè ogni lista di adiacenza viene scandita al più una volta. Le operazioni di decremento e aggiornamento di H sono al più $n-1$ perchè ogni arco può dar luogo ad al più un decremento. Quindi siccome ogni singolo decremento costa $O(\log n)$, complessivamente i decrementi costano $O(n \log n)$. E la complessità dell'implementazione è $O(n \log n)$.

Esercizio 2

1. *(max 8 punti)* Scrivere in pseudo–codice una procedura che preso in input due interi positivi n e k con $1 \leq k \leq n$, stampi tutte le sequenze binarie di lunghezza n che non contengono più di k simboli uguali consecutivi. Ad esempio se $n = 5$ e $k = 2$ allora la procedura deve stampare (non necessariamente in quest'ordine): 00100, 11011, 10100, 10010, 01100, 01010, 01001, 00110, 00101, 01011, 01101, 10011, 10101, 10110, 11001, 11010. La complessità della procedura **deve essere** O($nD(n)$), dove $D(n)$ è il numero di sequenze da stampare.

2. *(max 3 punti)* Spiegare perché la procedura proposta ha la complessità richiesta.

Soluzione Esercizio 2

1. Un algoritmo che risolve il problema e che si basa sulla tecnica del backtracking è il seguente. L'algoritmo BCD, essendo ricorsivo, prende in input gli interi n e k, il vettore SOL in cui viene memorizzata la sequenza da stampare, il numero i di elementi inseriti nella sequenza fino a questo momento, il numero $zeri$ rappresentante il numero di zeri che compaiono in coda alla sequenza fino ad ora costruita ed il numero uni rappresentante il numero di uni che compaiono in coda alla sequenza fino ad ora costruita. Quindi la prima chiamata sarà BCD(n, k, SOL, 0, 0, 0).

```
BCD:  INPUT gli interi n e k, il vettore SOL e gli interi i, zeri, uni
    IF i = n THEN stampa la sequenza SOL[1], SOL[2], ..., SOL[n]
    ELSE
        IF zeri < k THEN
            SOL[i + 1] ← 0
            BCD(n, k, SOL, i + 1, zeri + 1, 0)
        ENDIF
        IF uni < k THEN
            SOL[i + 1] ← 1
            BCD(n, k, SOL, i + 1, 0, uni + 1)
        ENDIF
    ENDIF
```

2. Il contatori $zeri$ e uni tengono traccia del numero di zeri e uni presenti in coda alla soluzione parziale che si va costruendo e viene aggiornato coerentemente alle scelte fatte: se si aggiunge uno zero alla soluzione parziale allora $zeri$ viene incrementato, in caso contrario viene incrementato uni.
   Onde evitare la generazione di sequenze inutili, si procede come segue: il simbolo viene accodato se e solo se non sono presenti in coda alla sequenza gia costruita k simboli ad esso uguali.

   Per quanto detto una chiamata ricorsiva sarà effettuata se e solo se la soluzione parziale fino a quel momento costruita può estendersi ad una sequenza da stampare. In questo modo la complessità risultante sarà O($nD(n)$) dove $D(n)$ è il numero di sequenze da stampare.