Fondamenti di Programmazione 2

9 Dicembre 2022

Esercizio 1

Sia $x_1, ..., x_n$ un insieme di interi distinti. Determinare tutti i modi in cui è possibile dividere l'insieme in due sottoinsiemi S', S'' tali che:

- $S' \cup S'' = S$
- $S' \cap S'' = \{\}$
- $\sum_{x \in S'} x = \sum_{x \in S''} x$

Esercizio 2

Sia G un grafo non orientato, V l'insieme dei suoi nodi e $E \subset V \times V$ l'insieme dei suoi archi. Un insieme $W \subset V$ si dice *indipendente* se non esiste nessun arco $(a, b) \in E$ tale che $a \in W, b \in W$.

1. Scrivere un programma C++ che calcoli, utilizzando la tecnica del backtracking, un insieme indipendente per un grafo in input.
2. Modificare il programma del punto precedente per calcolare *un* insieme indipendente di G di cardinalità massima, cioè che contengono il maggior numero di nodi.

Esercizio 3

Scrivere un programma che presa in input una stringa S e una lista di stringhe W, stampi tutte le permutazioni di S che non contengono nessuna stringa in W. Si può supporre S sia composta da caratteri distinti.

Esempio Consideriamo la parola `rane`, e `W = {na, re}`. Segue la lista delle sue permutazioni *valide* e *non valide*. (Il programma può stampare anche solamente quelle valide.)

```
PERMUTAZIONI VALIDE:
rane
```

```
raen
rnea
rean
aren
aner
aenr
aern
naer
nrae
nrea
nera
near
eanr
earn
enar
enra
eran

PERMUTAZIONI NON VALIDE:
rnae
rena
arne
anre
nare
erna
```

Suggerimento: Supponiamo di saper calcolare le permutazioni di stringhe lunghe $k-1$. Data una stringa lunga k, possiamo *fissare il primo carattere* e generare le permutazione dei restanti $k-1$ caratteri, ottenendo tutte le possibili permutazioni della stringa. Inoltre, se $k \leq 1$, esiste un'unica permutazione della stringa: la stringa stessa. Ad esempio, prendiamo la stringa `abc` e indichiamo tra `[]` i caratteri *fissi*.

```
abc
    => [a]bc
        => [ab]c
            => [abc]
        => [ac]b
            => [acb]
    => [b]ac
        => [ba]c
            => [bac]
        => [bc]a
            => [bca]
    => [c]ba
        => [cb]a
            => [cba]
```

```
=> [ca]b
    => [cab]
```

Le permutazioni di `abc` sono dunque `abc`, `acb`, `bac`, `bca`, `cba`, `cab`.

Esercizio 4

Siano $s_1, ..., s_n$ un insieme di n studenti. Ogni studente ha dei corsi da seguire, da un insieme $c_1, ..., c_m$ di corsi. Ogni corso può essere pianificato in 3 slot orari differenti - `MATTINA`, `POMERIGGIO`, `SERA`. Uno studente non può seguire contemporaneamente due corsi pianificati nello stesso slot orario.

Scrivere un programma C++ che generi presa in input una lista di studenti e relativi corsi da seguire, generi un orario compatibile alle esigenze degli studenti. **Il programma dovrà inoltre segnalare se non esiste un orario compatibile con le esigenze degli studenti.**

Suggerimento: Supponiamo lo studente s_k debba seguire i corsi $c_{k,1}, c_{k,2}, ..., c_{k,m}$. Ciò significa che non può esistere nessuna coppia di corsi $(c_{k,i}, c_{k,j})$ con slot orario coincidente. Sia $G(s)$ il grafo completo (cioè con tutti gli archi possibili) che ha come nodi $c_{k,1}, ..., c_{k,m}$. Il grafo G che otteniamo come unione di $G(s_i)$ per ogni s_i, visto come un'unico grafo, rappresenta la "compatibilità di coincidenza'' tra corsi. Per trovare un assegnamento di slot orari compatibili, è sufficiente k-colorare G, dove k è il numero di slot orari disponibili.

Esercizio 5

Sia G un grafo non orientato. Una *copertura* per G è un sottoinsieme di nodi $W \subseteq V$ tale che ogni arco in G sia incidente ad almeno un nodo in V, cioè non è possibile che $(a, b) \in E$ ma $a \notin W, b \notin W$.

1. Scrivere un programma C++ che calcoli, utilizzando la tecnica del backtracking, una copertura per il grafo in input.
2. Modificare il programma del punto precedente per calcolare la copertura di G *a costo minimo*, dove il costo di un insieme è la somma dei costi dei nodi che lo compongono.