Esercizio. Dobbiamo comprare una certa quantità P di piatti pregiati. Esistono solo n rivenditori di tali piatti. L' i-esimo rivenditore, con $i \in \{1, \ldots, n\}$, ha un numero limitato $d[i]$ di piatti disponibili alla vendita e ognuno di questi $d[i]$ piatti ha un prezzo $p[i]$. Bisogna calcolare il budget minimo necessario per comprare i P piatti. Progettare quindi un algoritmo che, dato il numero intero P, il vettore di interi $d[1 \ldots n]$ (piatti disponibili presso ognuno degli n rivenditori), il vettore di numeri non negativi $p[1 \ldots n]$ (costo di un piatto presso ognuno degli n rivenditori), restituisce la cifra minima necessaria per acquistare P piatti. L'algoritmo deve restituire $+\infty$ nel caso in cui i rivenditori non abbiano globalmente un numero sufficiente di piatti.

Soluzione. Il problema può essere risolto semplicemente comprando i piatti con il prezzo unitario minore. Un modo greedy per implementare questo approccio è di ordinare i rivenditori in ordine non decrescente di prezzo unitario, poi si iniziano a comprare i piatti disponibili partendo dal rivenditore con il prezzo unitario minimo, passando a rivenditori successivi (secondo l'ordinamento) fino al raggiungimento della quantità prevista. Tale soluzione prevede di ordinare l'intero elenco di rivenditori e avrà quindi un costo computazionale $\Omega(n \log n)$, considerando la complessità pseudolineare del problema dell'ordinamento. Nel caso in cui il numero di rivenditori utilizzati fosse molto inferiore rispetto al numero totale di rivenditori, risulterebbe inutile aver ordinato tutti i rivenditori in quanto si utilizzerebbe solo una minima parte iniziale del vettore ordinato dei rivenditori. In questi casi, risulta più efficiente procedere come riportato nell'Algoritmo PIATTI: preventivamente si popola un min-heap con i rivenditori considerando il loro ordinamento rispetto al prezzo unitario e poi si estrae solo la quantità di rivenditori sufficiente per comprare tutti i piatti richiesti. Sia q questa quantità di rivenditori utilizzati. Considerando il costo $O(n)$ per popolare il min-heap, ed il costo $O(\log n)$ per l'estrazione del minimo da un min-heap, tale algoritmo risulta avere costo computazionale $O(n + q \log n)$ che risulterebbe $O(n)$ nel caso in cui $q = O(\frac{n}{\log n})$ (in caso contrario, avremmo comunque $q = O(n)$ che implicherebbe un costo computazionale $O(n \log n)$ non superiore alla soluzione che preventivamente ordina tutti i rivenditori).

Algorithm 1: PIATTI(INT P, INT[1..n] d, REAL[1..n] p) $\rightarrow$ REAL

```
/* Inizializza res piatti residui, tot costo totale, A coppie prezzo/rivenditore (p[i], i) */
REAL res ← P, tot ← 0, daComprare
(REAL,INT)[1..n] A
for i ← 1 to n do
    A[i] ← (p[i], i)

/* Generazione dell'heap contenente le coppie (p[i], i) ordinate secondo il primo campo    */
minHeap[(REAL,INT)]H ← A.heapify()

/* Estrazione dall'heap dei rivenditori da cui comprare i piatti                           */
while res > 0 and (not H.empty()) do
    (REAL,INT) ( p, j ) ← H.findMin()
    H.deleteMin()
    daComprare ← min(res, d[j])
    res ← res − daComprare
    tot ← tot + (d[j] × daComprare)

/* Controllo se si sono comprati tutti i piatti                                            */
if res = 0 then
    return tot
else
    return +∞
```

Esercizio. Si consideri una variante del problema dello zaino, in cui ci sono due zaini a disposizione invece di uno solo. Nello specifico, sono dati n oggetti, ognuno identificato da un indice compreso fra 1 e n, e aventi un peso intero positivo p_i e un valore v_i (con $i \in \{1, \ldots, n\}$), e due zaini con capacità massima C_1 e C_2, rispettivamente (con C_1 e C_2 interi positivi). Progettare un algoritmo che, forniti questi dati in input, stampa gli indici di un sottoinsieme di oggetti, che possono essere inseriti nei due zaini, e che massimizza il valore complessivo.
Suggerimento: Si consideri un generico problema $P(i, j_1, j_2)$ in cui si prendono in analisi i oggetti e due zaini di capacità j_1 e j_2, rispettivamente.

Soluzione. Seguendo il suggerimento, si considera la classe di problemi $P(i, j_1, j_2)$, con $i \in \{0, \ldots, n\}$, $j_1 \in \{0, \ldots, C_1\}$ e $j_2 \in \{0, \ldots, C_2\}$, che consistono nel trovare il valore massimo che si riesce ad ottenere considerando i primi i oggetti e i due zaini di capacità j_1 e j_2, rispettivamente.

I problemi $P(i, j_1, j_2)$ possono essere risolti considerando come casi banali i casi $P(0, j_1, j_2) = 0$, mentre per $i > 0$ si procede in modo ricorsivo considerando che $P(i, j_1, j_2) =$

$$\begin{cases} P(i-1, j_1, j_2) & \text{se } p_i > j_1 \text{ e } p_i > j_2 \\ max\{P(i-1, j_1, j_2), P(i-1, j_1 - p_i, j_2) + v_i\} & \text{se } p_i \leq j_1 \text{ e } p_i > j_2 \\ max\{P(i-1, j_1, j_2), P(i-1, j_1, j_2 - p_i) + v_i\} & \text{se } p_i > j_1 \text{ e } p_i \leq j_2 \\ max\{P(i-1, j_1, j_2), P(i-1, j_1 - p_i, j_2) + v_i, P(i-1, j_1, j_2 - p_i) + v_i\} & \text{se } p_i \leq j_1 \text{ e } p_i \leq j_2 \end{cases}$$

L'Algoritmo 2 risolve i problemi $P(i, j_1, j_2)$ inserendo le soluzioni nella tabella $P[0..n, 0..C_1, 0..C_2]$. L'esercizio richiede di stampare l'utilizzo degli oggetti nel caso ottimale, e non il valore complessivo ottimale. A tal fine l'algoritmo, una volta riempita la tabella P, la ripercorre partendo dalla cella contenente il valore ottimale $P[n, C_1, C_2]$ per valutare come ogni oggetto i-esimo venga utilizzato dalla soluzione ottimale, ovvero, se viene inserito nel primo zaino, nel secondo zaino, oppure se non viene utilizzato. Il costo computazionale di tale algoritmo risulta essere in $\Theta(n \times C_1 \times C_2)$ in quanto le operazioni elementari hanno tutte costo costante, ed il blocco di maggiore costo è quello che contiene i tre **for** annidati.

Cont.

Algorithm 2: DueZaini(Int $p[1..n]$, Number $v[1..n]$, Int C_1, Int C_2)

```
Number P[0..n, 0..C_1, 0..C_2]
// Inizializzazione con i casi base di P
for j_1 ← 0 to C_1 do
    for j_2 ← 0 to C_2 do
        P[0, j_1, j_2] ← 0
// Riempimento delle restanti celle di P
for i ← 1 to n do
    for j_1 ← 0 to C_1 do
        for j_2 ← 0 to C_2 do
            if p[i] ≤ j_1 and p[i] ≤ j_2 then
                P[i, j_1, j_2] ← max{P[i−1, j_1, j_2], P[i−1, j_1 − p[i], j_2] + v[i], P[i−1, j_1, j_2 − p[i]] + v[i]}
            else if p[i] ≤ j_1 then
                P[i, j_1, j_2] ← max{P[i−1, j_1, j_2], P[i−1, j_1 − p[i], j_2] + v[i]}
            else if p[i] ≤ j_2 then
                P[i, j_1, j_2] ← max{P[i−1, j_1, j_2], P[i−1, j_1, j_2 − p[i]] + v[i]}
            else
                P[i, j_1, j_2] = P[i−1, j_1, j_2]
if P[n, C_1, C_2] ≠ 0 then
    // Esiste una soluzione e viene stampata
    i ← n; j_1 ← C_1; j_2 ← C_2
    while i > 0 do
        if P[i, j_1, j_2] == P[i, j_1 − p[i], j_2] + v[i]          // oggetto i-esimo in primo zaino
        then
            stampa("oggetto "+i+" in zaino 1")
            j_1 ← j_1 − p[i]
        else if P[i, j_1, j_2] == P[i, j_1, j_2 − p[i]] + v[i]     // oggetto i-esimo in secondo zaino
        then
            stampa("oggetto "+i+" in zaino 2")
            j_2 ← j_2 − p[i]
        i ← i − 1
```

Esercizio. Assumiamo di rappresentare tramite un grafo non orientato un tabellone di un gioco costituito da caselle (corrispondenti ai vertici del grafo), in cui possono essere collocate delle pedine, e linee di collegamento tra caselle (corrispondenti agli archi del grafo). Ad ogni mossa, una pedina può essere spostata dalla cella corrente ad un'altra cella collegata tramite una linea. Ad ogni giocatore vengono assegnate in modo casuale tre caselle che per comodità chiamiamo i, m e f. Ad inizio partita, un giocatore colloca la propria pedina nella cella i, poi tramite mosse deve far raggiungere alla pedina la cella m, ed infine la cella f. Vogliamo calcolare il numero minimo di mosse necessarie. Bisogna quindi progettare un algoritmo che dato $G = (V, E)$ e tre vertici $i, m, f \in V$, restituisce il numero minimo di mosse necessarie per spostare la pedina da i a f, passando per m. Nel caso in cui i, m ed f non siano fra loro collegate, l'algoritmo deve restituire ∞.

Soluzione. Il problema corrisponde alla ricerca della lunghezza minima di un cammino che inizia da i arriva ad m e poi continua fino ad f, in un grafo non orientato. Questo cammino si ottiene concatenando due cammini di lunghezza minima, uno da i ad m, ed uno da m a f. Questi due cammini avranno lunghezza rispettivamente uguale alla distanza tra i ed m, e alla distanza tra m ed f. Concludendo, la lunghezza del cammino da ricercare risulta essere uguale alla somma della distanze tra i ed m, e della distanza tra m ed f. Queste distanze possono essere calcolate facendo una unica visita in ampiezza BFS del grafo, partendo dal vertice m.

Seguendo quanto detto sopra, l'Algoritmo 3 effettua la visita BFS partendo dal vertice m, terminandola appena vengono visitati sia i sia f. Al termine della visita, si controlla se i ed f sono entrambi raggiungibili da m, e in tal caso viene restituita la somma delle due distanze. Se la visita termina senza visitare entrambi i vertici si restituisce

Cont.

Algorithm 3: LunghezzaMinima(Graph $G = (V, E)$, Vertex i, Vertex m, Vertex f) $\rightarrow$ Intero

```
Queue q ← new Queue()                                          // Esecuzione BFS

for x ∈ V do
    x.dist ← ∞
m.dist ← 0
q.enqueue(m)
while !q.isEmpty() ∧ (i.dist = ∞ ∨ f.dist = ∞)      // Termina BFS se si raggiungono sia i sia f
 do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if w.dist == ∞ then
            w.dist ← u.dist + 1
            q.enqueue(w)
if i.dist ≠ ∞ ∧ f.dist ≠ ∞ then
    return i.dist + f.dist                          // i, m ed f sono tra loro raggiungibili
else
    return ∞                                      // i, m ed f NON sono tra loro raggiungibili
```

∞, in quanto i tre vertici non sono tra loro raggiungibili. Il costo computazionale nel caso ottimo si ha quando i tre vertici i, m ed f coincidono: in tal caso il costo è costante, ovvero $\Theta(1)$. Nel caso pessimo, ovvero se i tre vertici non sono fra loro raggiungibili, viene effettuata l'intera BFS che ha costo $O(n+m)$, con $n = |V|$ e $m = |E|$, assumendo implementazione del grafo tramite liste di adiacenza.

Esercizio. Considerate un grafo orientato $G = (V, E)$ tale che per ogni $v \in V$ esiste un campo $v.nat$ che contiene un numero naturale (ovvero, un intero positivo). Progettare un algoritmo che dato G e due vertici $s \in V$ e $t \in V$, stampa (se esiste) il cammino $v_0, v_1, v_2, \ldots, v_k$ da s in t (ovvero, $v_0 = s$ e $v_k = t$) che minimizza la somma dei numeri associati ai nodi del cammino (ovvero, minimizza $\sum_{i=0}^{k} v_i.nat$).

Soluzione. Il problema può essere risolto usando una variante dell'algoritmo di Dijkstra che calcola il costo dei cammini sommando i valori associati ai vertici invece di considerare i pesi degli archi. È possibile utilizzare l'algoritmo di Dijkstra in quanto non ci sono valori negativi associati ai vertici. Conoscendo il vertice di destinazione t, si può interrompe l'esecuzione dell'algoritmo appena si trova il cammino di costo minimo per raggiungere tale vertice.

L'algoritmo CamminoMinimo (si veda Algoritmo 4) adotta la solita convenzione per cui i vertici sono rappresentati tramite i numeri compresi fra 1 e n; questa convenzione permette di rappresentare l'albero dei cammini minimi tramite il cosiddetto *parent vector*. L'algoritmo ha il medesimo costo computazionale dell'algoritmo di Dijkstra, quindi $T(n, m) = O(m \log n)$ con $n = |V|$ e $m = |E|$, assumendo l'utilizzo di liste di adiacenza.

Cont.

Algorithm 4: CamminoMinimo(Grafo $G = (V, E)$, Vertex s, Vertex t)

```
// inizializzazione strutture dati
n ← G.numNodi()
Int pred[1..n], v, u
Number D[1..n]
for v ∈ V do
    D[v] ← ∞
    pred[v] ← −1
D[s] ← s.nat
MinPriorityQueue[Int, Number] Q ← new MinPriorityQueue[Int, Number]()
Q.insert(s, D[s])

// esecuzione algoritmo di Dijkstra (modificato)
while not Q.isEmpty() do
    u ← Q.findMin()
    if u = t then
        printPath(pred, t)
        break
    Q.deleteMin()
    for v ∈ u.adjacent() do
        if D[v] = ∞ then
            // prima volta che si incontra v
            D[v] ← D[u] + v.nat
            Q.insert(v, D[v])
            pred[v] = u
        else if D[u] + v.nat < D[v] then
            // scoperta di un cammino migliore per raggiungere v
            Q.decreaseKey(v, D[v] − D[u] − v.nat)
            D[v] = D[u] + v.nat
            pred[v] = u


// funzione ausiliaria di stampa del cammino
procedure printPath(Int pred[1..n], Vertex d)
if pred[d] ≠ −1 then
    printPath(pred, pred[d])
print d
```

The End.