1.LA TECNICA GOLOSA

1.1.Algoritmi Tipici

La tecnica *golosa* (in inglese, *greedy*) è utilizzata per risolvere *problemi di ottimizzazione*, cioè quei problemi per i quali, come abbiamo già discusso nell'Unità 4 e nella premessa dell'Unità 5, bisogna ricercare una soluzione che soddisfi alcune condizioni, dette *vincoli*, e che massimizzi o minimizzi il valore di una *funzione obiettivo*.

Un problema tipico di ottimizzazione risolvibile con tecnica golosa ha come input n tipi di oggetti, $o_1,\ldots, o_n$, ciascuno dei quali è caratterizzato da un *valore* e da un *costo di* base ed è disponibile in quantità q_i e consiste nello scegliere un certo numero di oggetti in modo da massimizzare il valore entro un certo limite di costo o minimizzare il costo a patto che un minimo livello di valore sia raggiunto. Formalmente il problema è formulato nel seguente modo:

Problema di Massimo:

$$\text{massimizzare } F = \sum_{i=1}^{n} x_i \times V_i(x_1,\ldots,x_n)$$

$$\text{con i vincoli } \begin{cases} (i)\ C(x_1,\ldots,x_n) \le Cmax \\ (ii)\ 0 \le x_i \le q_i \quad i = 1,\ldots,n \end{cases}$$

dove $Cmax$ indica la *massima disponibilità* di risorse per l'acquisizione di oggetti, $V_i(x_1,\ldots,x_n)$ è la *funzione di valore* di ciascun oggetto, che dipende non solo dal valore dell'oggetto ma anche dalla scelta globale, e $C(x_1,\ldots,x_n)$ è la *funzione di costo globale* di una scelta di oggetti.

Problema di Minimo:

$$\text{minimizzare } F = \sum_{i=1}^{n} x_i \times C_i(x_1,\ldots,x_n)$$

$$\text{con i vincoli } \begin{cases} (i)\ V(x_1,\ldots,x_n) \ge Vmin \\ (ii)\ 0 \le x_i \le q_i \quad i = 1,\ldots,n \end{cases}$$

dove $Vmin$ indica il *minimo valore di soddisfazione* che deve derivare dalla scelta degli oggetti, $C_i(x_1,\ldots,x_n)$ è la *funzione di costo* di ciascun oggetto, che dipende non solo dal costo dell'oggetto ma anche dalla scelta globale, e $V(x_1,\ldots,x_n)$ è la *funzione di valore globale* di una scelta di oggetti.

Ognuno dei due problemi di ottimizzazione può essere a *soluzione intera* (cioè un oggetto i va preso nella sua interezza o va tralasciato poiché x_i è o un intero positivo o vale 0) o a *soluzione frazionaria* (cioè di un oggetto può essere presa anche una parte poiché x_i può non essere intero).

E' facile verificare che il problema di massimo ammette sempre una soluzione mentre ciò non è garantito per il problema di minimo.

La tecnica golosa, invece di effettuare una scelta globale degli oggetti più convenienti, determina la soluzione a stadi, in ciascuno dei quali effettua una scelta parziale di ottimo, che potrebbe essere affrettata (da cui il nome "goloso") e quindi inficiare l'ottimalità. Al primo stadio determina l'oggetto che sembra essere più conveniente, tipicamente quello con *valore specifico più elevato*, dove il valore specifico è il rapporto tra la funzione di valore V_i e quella di costo C_i, ambedue calcolate in maniera approssimata come se fossero indipendenti da qualsiasi scelta. Se la disponibilità *Cmax* non è esaurita (o il livello di soddisfazione *Vmin* non è soddisfatto) si passa al secondo stadio in cui viene preso in considerazione il successivo oggetto con maggiore valore specifico, calcolando ora le funzioni V_i e C_i sulla base della scelta del primo oggetto e così via fino a quando la massima disponibilità non sia stata tutta consumata (o il livello di soddisfazione sia stato raggiunto) o non ci siano più oggetti disponibili. In generale, per l'oggetto *i*-mo, le funzioni di valore e di costo vengono calcolate in maniera approssimata sulla base delle *i*-1 scelte precedenti.

Problema: ottimalità

Come già accennato, le scelte locali di ciascun stadio potrebbero non costituire una scelta ottima globale per cui un algoritmo goloso potrebbe non determinare la soluzione ottima. Un aspetto importante nell'utilizzo di un algoritmo goloso è dimostrare se esso sia capace o meno di determinare la soluzione ottima e, nel caso che non lo sia, verificare se la soluzione determinata sia comunque subottima, cioè una soluzione vicina a quella ottima. In quest'ultimo caso l'algoritmo goloso è da considerare come un algoritmo *euristico* o *approssimato* e il suo utilizzo è significativo in quelle situazioni in cui la ricerca dell'ottimo ha un costo di esecuzione estremamente elevato per cui ci si accontenta di una soluzione sub-ottima.

Presentiamo di seguito uno schema di algoritmo goloso, in cui il problema è di massimo o di minimo ed è a soluzione intera o frazionaria:

```
template <class Tx, class Tproblema>
Tsoluzione<Tx> goloso ( const Tproblema& problema )
{     Card n = numeroOggetti(problema);
      // inizializzazione della soluzione
      Tsoluzione<Tx> soluzione(n);
//    // stadi delle scelte golose
      for ( i = 1; i <= n ; i++ )
      {     // individuazione dell'indice dell'oggetto migliore
            // allo stadio i-mo
            Card jScelto = sceltaGolosa(problema,soluzione);
            // eventuale inserimento dell'oggetto jScelto
            aggiungiOggettoScelto(jScelto,problema,soluzione);
      }
      return soluzione;
}
```

A stadio della scelta go
- scelta golos
- aggiungi Scel alla soluz

Nel caso di problema di massimo, la soluzione corrente ad ogni stadio è ammissibile e il valore della funzione obiettivo *F* via via migliora avvicinandosi al valore ottimo. Nel caso di problema di minimo la soluzione corrente è non ammissibile ma con valore della funzione obiettivo *F* migliore di quello dell'ottimo e via via peggiora fino a far rispettare i vincoli e diventare, quindi, ammissibile.

Consideriamo come esempio il problema di dover fare un viaggio in auto da Reggio Calabria (RC) a Padova (PD) con un'auto che con il pieno percorre *k* km e di avere

a disposizione la carta autostradale con indicazione di tutte le n stazioni di rifornimento di benzina e la loro distanza d_i a partire da RC. Vogliamo scegliere il minor numero di rifornimenti in modo da essere capaci di percorrere tutti i K km da RC a PD, supponendo che alla partenza l'auto sia già con il pieno. Il problema può essere formulato nel seguente modo:

$$\text{minimizzare } F = \sum_{i=1}^{n} x_i$$

$$\text{con i vincoli } \begin{cases} (i)\ D_0 = k \geq d_1; \\ D_1 = \max(D_0 - (d_1 - d_0), k \times x_1) \geq d_2 - d_1 \\ \ldots \\ D_i = \max(D_{i-1} - (d - d_{i-1})_i, k \times x_i) \geq d_{i+1} - d_i \\ \ldots \\ D_n = \max(D_{n-1} - (d_n - d_{n-1}), k \times x_n) \geq K - d_n \\ (ii)\ x_i \in \{0,1\} \quad i = 1,n \end{cases}$$

D_i indica l'autonomia in chilometri all'altezza della stazione di rifornimento i — ovviamente, all'inizio, cioè per $i = 0$, si ha un'autonomia di k chilometri poichè è disponibile il pieno e il primo vincolo impone che la prima stazione di rifornimento sia raggiungibile con tale pieno. In una stazione generica di rifornimento i, l'autonomia è data o dalla autonomia residua se non viene effettuato il rifornimento nella stazione i (cioè se $x_i = 0$) o altrimenti con l'autonomia di k chilometri e il vincolo impone che con tale autonomia sia raggiungibile la prossima stazione di rifornimento. Il problema può essere risolto con la tecnica golosa nel seguente modo:

```
Vettore<Bool> viaggioRC_PD ( const Vettore<Card>& distPompe,
                             Card maxSerbatoio, Card distPD )
{     // si supponga che i rifornimenti non siano ordinati per
      // distanza crescente
      Card n = distPompe.iMax;
      // inizializzazione della soluzione
      Vettore<Bool> pompeScelte(n);
      for ( Card i = 1; i <= n; i++ )
            pompeScelte[i] = FALSO;
      Card kmPercorribili = maxSerbatoio;
      // stadi dell'algoritmo goloso
      for ( i = 1; i <= n && kmPercorribili < distPD; i++ )
      {     // scelta golosa
            Card jMax = 0;
            for ( Card j = 1; j <= n; j++ )
                  // sceglie tra le pompe raggiungibili
                  // quella con distanza maggiore
                  if ( !pompeScelte[j] &&
                       distPompe[j] <= kmPercorribili &&
                      (jMax == 0 || distPompe[j] > distPompe[jMax] )
                        jMax = j;
```

```
            assert( jMax > 0 ); // altrimenti non esiste soluzione
            kmPercorribili = distPompe[jMax] + maxSerbatoio;
            pompeScelte[jMax] = VERO;
        }
        return pompeScelte;
    }
```

La scelta golosa consiste nello scegliere la stazione di rifornimento più lontana tra quelle raggiungibili con il pieno a disposizione. E' facile dimostrare che la soluzione determinata è ottima. Infatti, sia G la soluzione restituita dall'algoritmo goloso e si consideri una soluzione ottima O del problema. A causa della scelta golosa, il primo rifornimento di G è a distanza maggiore o uguale a quella del primo rifornimento in O; quindi da esso è possibile raggiungere il secondo rifornimento di O. Quindi una soluzione costituita dal primo rifornimento di G e dai successivi di O è anch'essa ottima. Ciò significa che esiste sempre una soluzione ottima il cui rifornimento iniziale può essere scelto con una tecnica golosa. Consideriamo ora il percorso da questo primo rifornimento fino alla destinazione. Questo sottoproblema ha la stessa struttura del problema originario per cui anche per esso esiste una soluzione ottima che inizia con una scelta golosa, che corrisponde al secondo rifornimento in G. Il ragionamento può essere iterato per i successivi sottoproblemi, cioè con una semplice induzione sul numero di scelte fatte, è facile vedere che una scelta golosa ad ogni stadio produce una soluzione ottima.

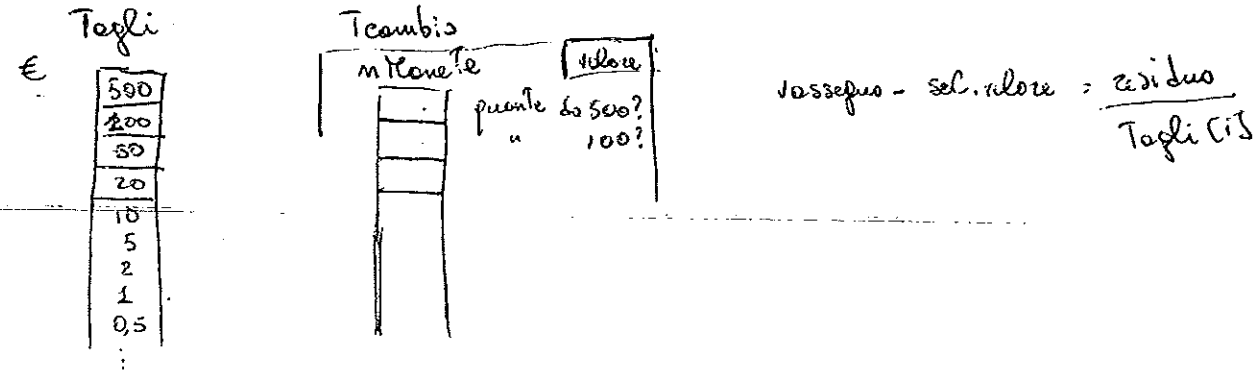
La funzione *viaggioRC_PD* ha complessità $\Theta(n^2)$.

Esercizio 1. *Riscrivere la funzione* viaggioRC_PD *in modo che essa abbia complessità* $O(n \log n)$.

Domanda 1. *Qual'è la complessità del problema* viaggioRC_PD *nell'ipotesi che le stazioni di rifornimento siano già ordinate per distanza crescente?*

Come esempio di problema in cui è possibile scegliere più copie dello stesso tipo di oggetto, consideriamo il problema di dover cambiare un assegno bancario con il minor numero di monete. La tecnica golosa consiste nel cercare di utilizzare il più possibile monete di taglio grosso. Anche questo problema è di minimo e di tipo intero, ma non 0/1, e può essere risolto nel seguente modo:

```
struct Tcambio
{     Vettore<Card> nMonete; Card valore;
      Tcambio( Card n ): nMonete(n)
      {valore = 0; for ( Card i = 1; i <= n; i++ ) nMonete[i] = 0;}
};
Tcambio cambio ( const Vettore<Card>& tagli, Card valoreAssegno )
{     // i tagli disponibili delle monete sono
      // in ordine decrescente di valore
      Card n = tagli.iMax;
      Tcambio sol(n);
      for ( Card i = 1; i <= n && sol.valore < valoreAssegno; i++ )
      {     sol.nMonete[i] = (valoreAssegno-sol.valore)/tagli[i];
            sol.valore += sol.nMonete[i] * tagli[i];
      }
```



```
        // verifica che la soluzione sia ammissibile cioè
        // il controvalore sia >= del valore dell'assegno
        if ( sol.valore < valoreAssegno )
        {     sol.nMonete[n]++; sol.valore += tagli[n];
        }
        return sol;
    }
```

Nell'algoritmo si è supposto che, nel caso non possa essere corrisposto l'importo esatto dell'assegno, esso viene approssimato per eccesso a vantaggio del correntista. L'algoritmo ha complessità lineare $\Theta(n)$.

Domanda 2. *Qual'è la formulazione matematica del problema del "cambio" nel caso si voglia cambiare un assegno di valore V avendo a disposizione tagli di monete da 10000, 5000, 1000, 500, 100 e 50?*

L'esperienza ci suggerisce che, con i tagli di moneta attualmente disponibili in Italia, il problema del "cambio"" è un problema di minimo intero che ha soluzione ottima e una soluzione ottima è determinata con l'algoritmo goloso. La nostra esperienza ci dice anche che se l'assegno non è multiplo di 50 lire (e tra poco di 100!) dopo aver effettuato le scelte golose, il vincolo non è soddisfatto per cui è necessario arrotondare la cifra alle 50 lire superiori (o, più spesso, a quelle inferiori!). Supponiamo ora che in Italia sia anche disponibile una moneta da 4000 lire e di dover cambiare un assegno da 8000 lire. L'algoritmo goloso restituirebbe una moneta da 5000, una moneta da 2000 e una moneta da 1000, cioè una soluzione sub-ottima in quanto l'ottimo è dato da due monete da 4000. Cioè per alcuni tagli di monete il problema del cambio è risolto in maniera approssimata.

1.2.Complessità e Correttezza dell'Algoritmo Tipico

Consideriamo l'algoritmo tipico goloso che abbiamo schematizzato nel paragrafo precedente. La complessità temporale di esso è $\Theta(n*(K_1+K_2))$ dove K_1 è la complessità della funzione *sceltaGolosa* e K_2 è la complessità della funzione *aggiungiOggettoScelto*. In molte situazioni K_1 è lineare in n mentre K_2 è costante, cioè la complessità tipica di un algoritmo tipico goloso è $\Theta(n^2)$.

Un caso particolare e anche abbastanza frequente (caso "statico") è quando il valore specifico di un oggetto su cui è basata la scelta golosa non dipende dalla soluzione corrente e può essere determinato una volta per tutte prima di iniziare le scelte golose. In questo caso conviene ordinare gli oggetti per ordine crescente di valore specifico in modo che la funzione *sceltaGolosa* possa essere banalmente realizzata in tempo costante. I vari stati sono quindi eseguiti in tempo lineare e la complessità dell'algoritmo dipende dalla funzione di ordinamento, cioè è $\Theta(n \log n)$. Addirittura, se gli oggetti sono già ordinati per valore specifico crescente, la complessità si riduce a $\Theta(n)$ così come abbiamo visto per il problema del *cambio*.

Dall'analisi effettuata appare evidente che gli algoritmi golosi sono abbastanza efficienti. La contropartita è che non sempre determinano la soluzione ottima. Questo limite dipende dal fatto che ogni scelta golosa può fare utilizzo delle scelte precedenti ma non di quelle future per cui la soluzione è ottima solo se il problema ha

una *sottostruttura ottimale*, cioè effettuando la prima scelta golosa esiste una soluzione ottima che contiene l'oggetto scelto e i sottoproblemi successivi mantengono la struttura del problema originale, in particolare anche per essi la prima scelta golosa non inficia la possibilità di determinare l'ottimo. La parte complicata nell'utilizzo della tecnica golosa è dimostrare la sua eventuale ottimalità o, in subordine, definire un limite allo scostamento della soluzione determinata da quella ottima.

E' infine interessante rilevare che la tecnica golosa è di fatto un caso particolare della tecnica Divide et Impera che genera ad ogni passo un unico sottoproblema e in cui la ricomposizione della soluzione del problema originario a partire da quella del sottoproblema è immediata. Tale semplificazione permette di sostituire naturalmente la ricorsione con l'iterazione.

2.PROBLEMA DELLA BISACCIA

2.1.Problema della Bisaccia 0-1

Un problema classico in cui viene usato la tecnica golosa è il problema della *bisaccia* (in inglese *knapsack*). Questo problema intende modellare la situazione in cui, dato un insieme di oggetti di dimensione e valore diverso, si desidera scegliere un sottoinsieme di oggetti da inserire in una bisaccia in modo da massimizzare il valore trasportato.

Un utilizzatore frequente, ancorché inconsapevole, della tecnica golosa è il ladro di appartamenti che deve scegliere gli oggetti da rubare. Egli comincerà a prendere gli oggetti di maggior valore specifico, misurato come rapporto tra valore e volume. Ad esempio un televisore ha probabilmente maggior valore di un anello ma occupa un volume molto maggiore. Ovviamente ogni oggetto, se scelto, va preso nella sua interezza.

La formulazione generale del problema della bisaccia è la seguente. Abbiamo n oggetti, ciascuno con valore v_i e costo c_i, e abbiamo una disponibilità massima C per coprire i costi. Dobbiamo:

$$\text{massimizzare } F = \sum_{i=1}^{n} v_i \times x_i$$

$$\text{con i vincoli } \begin{cases} \text{(i)} \ \sum_{i=1}^{n} c_i \times x_i \le C \\ \text{(ii)} \ x_i \in \{0,1\} \quad i = 1,\ldots,n \end{cases}$$

v_i valore
$x_i \in \{0,1\}$ prendo/lascio l'oggetto
c_i costo (dimensione)
C costo massimo

Il valore 1 per la variabile x_i significa che l'oggetto i-mo è stato scelto mentre il valore 0 indica la sua esclusione. Una soluzione ammissibile è un vettore di 0 e 1 che rispetti il vincolo (i) — per questa ragione si parla di *problema della bisaccia 0-1*. Una soluzione ottima è una soluzione che massimizzi F.

Può capitare che nessuno di essi venga usato

Riconosciamo subito che le funzioni di costo e di valore sono costanti, cioè non dipendono dalle scelte effettuate. Pertanto un algoritmo goloso si riduce a ordinare gli oggetti in ordine decrescente di valore specifico, misurato come rapporto tra valore e costo, e scegliere gli oggetti in quest'ordine fino a saturare la disponibilità C. L'algoritmo è quindi

```
struct ToggettoK { Card c, v;};
struct TsoluzioneK_01
{     Card F, Cspeso; Vettore<Bool> x;
      TsoluzioneK_01( Card n ): x(n)
      {     F = Cspeso = 0;
            for ( Card i = 1; i <= n; i++ ) x[i] = FALSO;
      }
};
struct IndOggValK
{     Card ind; float vSpec;
      Bool operator< ( const IndOggValK ov )
            return Bool( vSpec < ov.vSpec );
```

```
struct  Toggetto {int costo, valore;}

Vettore<bool> knapsack_01 (Vettore<Toggetto> & oggetti, int CMax)
{       int n=oggetti.n();
        ordinaPerValoreSpecifico(oggetti);  // val. specifico= valore/costo

        Vettore<bool> sol(1,n);
        inizializza(sol, false);
        int Cspeso=0;
        for (int i=1; i<=n && Cspeso<CMax; i++)
        {       if (Cspeso+oggetti[i].costo <CMax)
                { sol[i]=true;
                  Cspeso+=oggetti[i].costo;
                }
        }
        return sol;
}
```

```
);
TsoluzioneK_01 knapsack_01 ( const Vettore<ToggettoK>& oggetti,
                                                    Card C )
{     Card n = oggetti.iMax;
      // calcolo valori specifici
      Vettore<IndOggValK> indiciOggettiOrdinati(n);
      for ( Card i = 1; i <= n; i++ )
      {     assert( oggetti[i].c > 0 );
            indiciOggettiOrdinati[i].ind = i;
            indiciOggettiOrdinati[i].vSpec = float(oggetti[i].v)/
                                             oggetti[i].c;
      }
      // ordinamento indici oggetti per valori specifici decrescenti
      heapSort(indiciOggettiOrdinati,1,n,
                                 TipoOrd<IndOggValK>(DISC));
      // inizializzazione della soluzione
      TsoluzioneK_01 S(n);
      // scelte golose
      for ( i = 1; i <= n && S.Cspeso < C; i++ )
      {     // selezione dell'oggetto migliore allo stadio i-mo
            Card io = indiciOggettiOrdinati[i].ind;
            // l'oggetto è preso se non viola il vincolo di costo
            if ( S.Cspeso + oggetti[io].c <= C )
            {     S.x[io] = VERO;
                  S.Cspeso += oggetti[io].c; S.F += oggetti[io].v;
            }
      }
      return S;
}
```

L'algoritmo goloso è di tipo statico e ha quindi una complessità $\Theta(n \log n)$, cioè la parte più costosa è l'ordinamento. Purtroppo la soluzione trovata non è necessariamente ottima. Ad esempio si considerino una disponibilità di 50 e tre oggetti, il primo con valore 60 e costo 10 (quindi con valore specifico 6), il secondo con valore 100 e costo 20 (quindi con valore specifico 5), e il terzo con valore 120 e costo 30 (quindi con valore specifico 4). L'algoritmo goloso sceglie il primo e secondo oggetto ottenendo $F = 160$; è facile verificare che la soluzione ottima è invece rappresentata dalla scelta del secondo e terzo oggetto per la quale $F = 220$.

La ragione per cui l'algoritmo goloso non sempre determina la soluzione ottima è che la scelta di un oggetto con più elevato valore specifico può comportare l'esclusione di un gruppo di oggetti di valore specifico più basso ma che, presi insieme, risultano più convenienti dell'elemento scelto.

Mostriamo ora che con piccole modifiche dell'algoritmo goloso è sempre possibile ottenere una soluzione che non è mai inferiore al 50% di quella ottima. A tale scopo utilizziamo il seguente importante risultato. Sia p l'indice dell'oggetto con valore massimo (attenzione non necessariamente con valore specifico massimo). Allora si può dimostrare che se $F^G \le F^*/2$ allora $v_p > F^*/2$, dove F^* è la soluzione ottima e F^G è la soluzione dell'algoritmo goloso.

Es. CMax = 50

Oggetti: v_1 60 c_1 10 (v spec. = 6)
v_2 100 c_2 20 (v spec. = 5)
v_3 120 c_3 30 (v spec. = 4)

l'algo restituisce → Cspeso = 30
MA è questa la sol. ottima

Esercizio 2. *Dimostrare che se* $F^G \leq F^*/2$ *allora* $v_p \geq F^*/2$.

La modifica all'algoritmo goloso consiste nel verificare se l'oggetto escluso con valore massimo abbia valore maggiore della soluzione golosa; se ciò accade, la soluzione golosa va modificata inserendo in essa l'oggetto con valore massimo al posto di tanti oggetti quanti sono necessari per rispettare il vincolo sui costi. E' da notare che, nella determinazione dell'oggetto escluso con valore massimo, vanno ovviamente trascurati eventuali oggetti con costo di per se stesso maggiore della massima disponibilità. L'algoritmo modificato diventa il seguente:

```
TsoluzioneK_01 knapsack_01 ( const Vettore<ToggettoK>& oggetti,
                                                    Card C )
{     Card n = oggetti.iMax;
      // calcolo valori specifici
      Vettore<IndOggValK> indiciOggettiOrdinati(n);
      for ( Card i = 1; i <= n; i++ )
      {     assert( oggetti[i].c > 0 );
            indiciOggettiOrdinati[i].ind = i;
            indiciOggettiOrdinati[i].vSpec = float(oggetti[i].v)/
                                             oggetti[i].c;
      }
      // ordinamento indici oggetti per valori specifici decrescenti
      heapSort(indiciOggettiOrdinati,1,n,
                              TipoOrd<IndOggValK>(DISC));
      // inizializzazione della soluzione
      TsoluzioneK S(n);
      // inizializzazione variabili per la determinazione
      // dell'oggetto con valore massimo
      Card iMax = 0; Card vMax = 0
      // scelte golose
      for ( i = 1; i <= n ; i++ )
      {     // selezione dell'oggetto migliore allo stadio i-mo
            Card io = indiciOggettiOrdinati[i].ind;
            // l'oggetto è preso se non viola il vincolo di costo
            if ( S.Cspeso + oggetti[io].c <= C )
            {     S.x[io] = VERO;
                  S.Cspeso += oggetti[io].c; S.F += oggetti[io].v;
            }
            else
                  // aggiornamento dell'oggetto escluso
                  // con valore massimo
                  if ( oggetti[io].v > vMax && oggetti[io] <= C )
                  {     vMax = oggetti[io].v; iMax = i;
                  }
      }
      if ( iMax > 0 && vMax > S.F)
      {     // viene determinata una soluzione alternativa attraverso
            // l'inserimento dell'oggetto escluso con valore massimo
            Card ioMax = indiciOggettiOrdinati[iMax].ind;
            S.x[ioMax] = VERO;
            S.Cspeso += oggetti[ioMax].c; S.F += oggetti[ioMax].v;
            for ( i = n; i >= 1 && S.Cspeso > C ; i-- )
            {     // vengono eliminati oggetti per far spazio
                  // a quello con valore massimo
```

```
                    Card io = indiciOggettiOrdinati[i].ind;
                    if ( i != ioMax && S.x[io] )
                    {     S.x[io] = FALSO;
                          S.Cspeso -= oggetti[io].c;
                          S.F -= oggetti[io].v;
                    }
               }
          }
          return S;
}
```

La soluzione restituita dall'algoritmo non è mai inferiore al 50% della soluzione ottima; in pratica essa è abbastanza vicina alla soluzione ottima di cui costituisce una buona approssimazione. La complessità dell'algoritmo rimane $\Theta(n \log n)$. La rilevanza dell'algoritmo goloso, nonostante il suo comportamento approssimato, è dovuta al fatto che il problema della bisaccia 0-1 è *NP-arduo*, cioè esso appartiene a una classe di problemi per i quali, come abbiamo visto nell'Unità 4, non è noto allo stato attuale alcun algoritmo capace di determinare la soluzione ottima in tempo polinomiale e un eventuale futura individuazione di un tale algoritmo comporterebbe notevoli stravolgimenti in tutta la teoria dell'informatica. Quindi l'algoritmo goloso non ha praticamente alternative a meno che la dimensione del problema (cioè n è abbastanza piccolo) non permetta il ricorso ad algoritmi con complessità esponenziale.

Un'estensione del problema della bisaccia 0-1 è considerare che per ciascun oggetto i siano a disposizione più copie $q_i \geq 1$(*problema della bisaccia generalizzato*) per cui la formulazione diventa:

$$\text{massimizzare } F = \sum_{i=1}^{n} v_i \times x_i$$

$$\text{con i vincoli} \begin{cases} \text{(i)} \displaystyle\sum_{i=1}^{n} c_i \times x_i \leq C \\ \text{(ii)} \; x_i \in \{0,1,\ldots,q_i\} \quad i = 1,\ldots,n \end{cases}$$

Esercizio 3. *Scrivere la funzione* knapsack_gen *che risolve il problema della bisaccia a numeri interi in tempo* $\Theta(n \log n)$ *e che restituisca una soluzione che non è mai inferiore del 50% di quella ottima.*

Una variazione interessante del problema della bisaccia è ottenuta allorquando si hanno a disposizione più bisacce in cui inserire gli oggetti e si vuole minimizzare il numero di bisacce necessarie per trasportare tutti gli oggetti, *problema dell'impacchettamento* (in inglese, *bin packing*). Poichè tutti gli oggetti vanno comunque scelti, il valore di essi non va più preso in considerazione; invece i costi (cioè i volumi) continuano ad avere un ruolo cruciale per valutare il vincolo sulla capacità C delle singole bisacce. Il problema può essere risolto con un algoritmo goloso andando ad allocare gli oggetti in ordine decrescente di volume nella prima bisaccia capace di contenerli. Quest'algoritmo è noto in letteratura con il nome di *First Fit Decreasing* e può essere scritto nel seguente modo:

```
struct ToggettoBP { Card c;};
```