Capitolo 13

Greedy

Nella vita di tutti i giorni, spesso si adottano strategie che danno immediatamente buoni risultati, ma che a lungo andare si rivelano scadenti. È questa la strategia dell'ingordo (o "greedy"):

"Meglio un uovo oggi che una gallina domani".

Esempio 13.1 (Resto col numero minimo di monete). Si supponga di avere monete da 50, 10, 5 e 1 centesimo, e di dover fare un resto di 72 centesimi. Senza neanche rendercene conto, utilizziamo la filosofia dell'ingordo, scegliendo una moneta da 50 centesimi, poi due da 10, e infine due da 1. La soluzione prodotta è formata da cinque monete soltanto ed è ottima, nel senso che non ce n'è una con un numero minore di monete. L'ottimalità della soluzione, però, non dipende dalla furbizia dell'ingordo, ma da una proprietà delle monete! Si supponga di avere monete da 11, 5 e 1 centesimo, e di dover fare un resto di 15 centesimi. L'ingordo giudicherà la moneta da 11 più "appetibile" delle altre, e farà un resto con una moneta da 11 e quattro da 1, utilizzando cinque monete in tutto, mentre la soluzione ottima è data da tre monete da 5 centesimi! □

Il metodo greedy si può applicare a quei problemi di ottimizzazione in cui occorre selezionare un sottoinsieme S "ottimo" di oggetti, che verificano certe proprietà, da un insieme dato $\{a_1, \ldots, a_n\}$ di n oggetti. Uno "scheletro" di procedura che adotta tale metodo è il seguente:

```
procedure GREEDY(insieme {a_1, ..., a_n} di oggetti);
    begin
        S := ∅;
        {ordina gli a_i per "appetibilità" decrescente};
        for i := 1 to n do
            if {a_i può essere aggiunto ad S}
                then S := S ∪ {a_i};
        {restituisci S come risultato}
    end;
```

Un algoritmo greedy ordina dapprima gli oggetti in base ad un criterio di "appetibilità". La soluzione del problema è poi costruita in modo incrementale considerando gli oggetti uno alla volta e aggiungendo ogni volta l'oggetto più appetibile, se possibile. In altri termini, l'algoritmo effettua una sequenza di scelte, preferendo ogni volta la scelta che fornisce immediatamente il miglior risultato. Fatta la scelta, è successivamente risolto un sottoproblema dello stesso tipo ma di dimensione più piccola.

Il problema più piccolo dipende dalla sequenza di scelte passate, ma non dalle scelte future. Da quanto detto, affinché un algoritmo greedy fornisca la soluzione ottima di un problema, occorre che siano verificate due proprietà, tra loro correlate:

1. *Scelta greedy.* Dopo aver fornito una caratterizzazione matematica della soluzione ottima, occorre dimostrare che tale soluzione può essere modificata in modo da utilizzare una prima scelta "greedy", che riduce il problema ad un sottoproblema più piccolo dello stesso tipo, e poi mostrare per induzione che il procedimento può essere esteso ad una sequenza di scelte "greedy";
2. *Sottostruttura ottima.* Per mostrare che una scelta "greedy" riduce il problema ad un sottoproblema più piccolo dello stesso tipo, occorre dimostrare che una soluzione ottima del problema contiene al suo interno le soluzioni ottime dei sottoproblemi.

Ovviamente, non sempre esiste un algoritmo greedy che trovi la soluzione ottima di un qualsiasi problema di ottimizzazione, come abbiamo già potuto osservare per il problema del resto col minimo numero di monete. Vediamo due esempi notevoli di problemi per i quali la tecnica greedy si comporta bene: trovare il minimo albero di copertura di un grafo pesato ed individuare l'ordine migliore (schedule) col quale eseguire un insieme di programmi su un processore in modo da minimizzare il numero di programmi che non rispettano certe scadenze temporali.

13.1 Minimo albero di copertura

Si consideri il seguente problema di ottimizzazione su grafi pesati:

MINIMO ALBERO DI COPERTURA: "Dato un grafo non orientato e connesso $G = (N, A)$, con pesi non negativi sugli archi, trovare un albero di copertura per G, cioè un albero avente tutti gli n nodi di N, ma solo $n - 1$ degli m archi in A, tale che la somma dei pesi degli archi nell'albero sia la più piccola possibile".

Esempio 13.2 (Minimo albero di copertura). La Figura 13.1 illustra un grafo non orientato pesato G e due alberi di copertura per G, dei quali il secondo è il minimo albero di copertura. □

Questo problema può essere risolto con molti algoritmi, dei quali i più noti sono quelli di Kruskal (1956) e Prim (1957). Di questi, il primo si basa sul principio greedy, in cui gli archi sono ordinati per pesi crescenti e poi considerati uno alla volta, ed un generico arco è aggiunto all'albero T se non forma circuiti con gli archi precedentemente inseriti in T.

```
procedure KRUSKAL(grafo non orientato G = (N, A));
    begin
        T := ∅;
        {ordina gli m archi di G per "peso" crescente};
        for a := 1 to m do
            if {l'arco a = [i, j] non forma un circuito con gli altri archi di T}
                then T := T ∪ {a};
    end;
```

L'algoritmo di Kruskal costruisce T per unione di componenti connesse. La sua correttezza può essere dimostrata come segue. Sia F_k la minima foresta di copertura

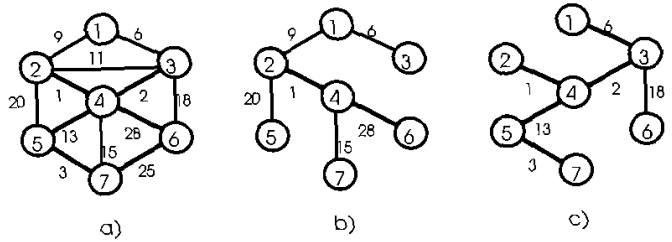

Figura 13.1: Minimo albero di copertura. a) Un grafo pesato; b) una soluzione ammissibile; c) una soluzione ottima.

per il grafo G, formata da k componenti connesse, cioè da k alberi. È facile provare per induzione che l'algoritmo computa successivamente F_n, F_{n-1}, ..., F_1, dove l'insieme A_n degli archi di F_n è vuoto, e l'insieme A_{k-1} degli archi di F_{k-1} è uguale ad $A_k \cup \{[i, j]: i \text{ e } j \text{ appartengono ad alberi distinti di } F_k\}$.

Esempio 13.3 (Algoritmo di Kruskal). La Fig. 13.2 mostra l'esecuzione dell'algoritmo di Kruskal sul grafo di Fig. 13.1. L'ordinamento degli archi è: [2,4], [3,4], [5,7], [1,3], [1,2], [2,3], [4,5], [4,7], [3,6], [2,5], [6,7], [4,6]. □

Il maggior sforzo computazionale dell'algoritmo di Kruskal è dato dall'ordinamento iniziale e dalla verifica che l'aggiunta di un nuovo arco non provochi la formazione di un circuito. L'ordinamento iniziale degli archi costa $O(m \log n)$ tempo, dato che $\log m \leq 2 \log n$. Inoltre, poiché la costruzione di T avviene per unione di insiemi disgiunti, cioè di due componenti connesse che si "fondono" in una sola con l'aggiunta del nuovo arco, è possibile utilizzare una struttura di dati di tipo "mfset" [cfr. § 5]. In questo modo, la verifica che un arco appartenga a due componenti distinte (cioè non provochi un circuito in T) ed anche il suo eventuale inserimento in T richiedono solo $O(\log n)$ tempo usando le operazioni TROVA e FONDI dell'"mfset". Poiché il ciclo **for** è ripetuto m volte, la complessità dell'algoritmo risulta essere $O(m \log n)$.

Per descrivere l'algoritmo di Kruskal con una procedura Pascal, occorre scegliere opportune realizzazioni per il grafo G in ingresso e per l'albero di copertura T fornito come risultato. Rappresentiamo il grafo semplicemente con un vettore $A[1..m]$ degli archi, dove ciascun arco $A[h]$ è un record contenente tre campi: i due nodi i e j che sono estremi dell'arco ed il peso dell'arco. Data l'usuale rappresentazione di un grafo con vettori di adiacenza, è facile vedere che il vettore degli archi può essere costruito in $O(n + m)$ tempo. L'albero T è rappresentato altrettanto semplicemente con un insieme, che può essere realizzato con una lista non ordinata, in modo che l'inserimento di un arco in T richieda tempo $O(1)$. Infine, è utilizzato un "mfset" S per mantenere le componenti connesse di T, tale che S è inizializzato con n componenti distinte, ciascuna formata da un singolo nodo del grafo.

```
type arco = record
    i, j: integer;
    peso: integer
end;

grafo = array[1..maxlung] of arco;
```


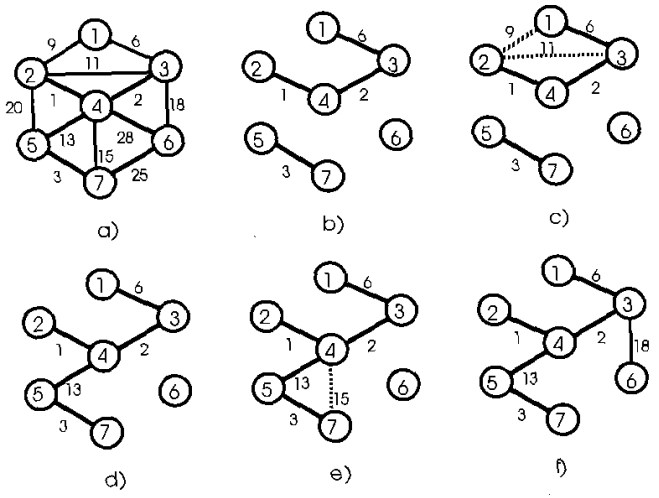

Figura 13.2: Algoritmo di Kruskal. a) Il grafo di ingresso. b) la foresta di copertura dopo l'inclusione di [2,4], [3,4], [5,7] e [1,3]; c) [1,2] e [2,3] non sono inclusi; d) inclusione di [4,5]; e) [4,7] non è incluso; f) inclusione di [3,6] e soluzione ottima.

La procedura Pascal KRUSKAL è la seguente, dove le modifiche da apportare alla procedura di ordinamento (p.e. HEAPSORT) usata per ordinare gli archi per pesi crescenti sono lasciate per esercizio. Come discusso precedentemente, la complessità di KRUSKAL è $O(m \log n)$.

```
procedure KRUSKAL(var A: grafo; n, m: integer; var T: insieme);
    var h: integer; S: mfset;
    begin
        CREAINSIEME(T);
        CREAMFSET(n, S);
        {ordina A[1], ..., A[m] in modo che A[1].peso ≤ ... ≤ A[m].peso};
        for h := 1 to m do
            if TROVA(A[h].i, S) ≠ TROVA(A[h].j, S) then begin
                FONDI(A[h].i, A[h].j, S);
                INSERISCI(A[h], T)
            end
    end;
```

Per risparmiare iterazioni inutili dopo che è stata raggiunta la soluzione ottima, la procedura KRUSKAL può essere modificata trasformando il ciclo **for** in un **while** ed introducendo una variabile intera che conta il numero di inserimenti di archi nell'albero di copertura T, uscendo dal ciclo appena sono state effettuate $n - 1$ INSERISCI. L'ordine di grandezza della complessità resta comunque $O(m \log n)$.

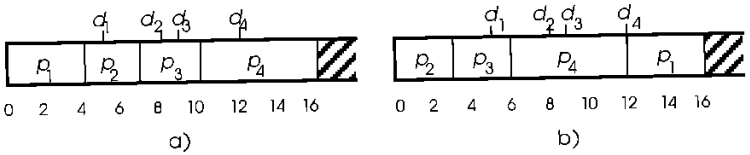

Figura 13.3: Algoritmo di Moore. a) Ordinamento iniziale; b) soluzione ottima.

13.2 Scheduling di programmi

La tecnica greedy è spesso utile per risolvere problemi di scheduling, in cui si hanno dei programmi da eseguire su un processore e si vuole trovarne l'ordine di esecuzione ottimo in base ad un prefissato criterio. A titolo di esempio, consideriamo il seguente problema.

PROGRAMMI IN RITARDO: "Dati n programmi $p_1, \ldots, p_n$, tali che ciascun p_i richiede t_i unità di tempo di esecuzione e deve essere eseguito entro una certa scadenza d_i, trovare un ordine S in cui eseguirli tutti in modo da minimizzare il numero di programmi per i quali la scadenza non è rispettata".

Il problema può essere risolto con un algoritmo ideato da Moore (1968). L'algoritmo è di tipo greedy, in cui viene effettuato occasionalmente anche un backtrack.

I programmi sono ordinati in una sequenza S per scadenze crescenti. Successivamente, si cerca il primo programma p in ritardo e si elimina il programma p' avente tempo di esecuzione più grande tra quelli che stanno nella sottosequenza iniziale di S che termina con p. Il procedimento è iterato sulla sottosequenza S' ottenuta da S eliminando p', fino ad ottenere una sottosequenza S^* senza programmi in ritardo. La sequenza ottima è ottenuta eseguendo dapprima i programmi della sottosequenza S^*, nell'ordine in cui appaiono in S^*, seguiti dai programmi in ritardo in un ordine qualsiasi.

La scelta greedy dell'algoritmo di Moore è basata sul fatto che se in una sottosequenza deve esserci un programma in ritardo, allora tanto vale eliminare il programma più lungo al fine di lasciare il maggior tempo disponibile prima delle scadenze dei programmi che seguiranno quello eliminato. La dimostrazione di correttezza non è qui riportata.

Esempio 13.4 (Programmi in ritardo). La Fig. 13.3 mostra l'esecuzione dell'algoritmo di Moore per i programmi $p_1, \ldots, p_4$ tali che: $t_1 = 4, d_1 = 5, t_2 = 3, d_2 = 8, t_3 = 3, d_3 = 9, t_4 = 6, d_4 = 12$. La sequenza S iniziale è $p_1, \ldots, p_4$. Essendo p_3 il primo programma in ritardo, si elimina p_1, perché $t_1 = 5$ è massimo tra t_1, t_2 e t_3. Dopo aver eliminato p_1, nessun altro programma oltre a p_1 è in ritardo, e la sequenza ottima è p_2, p_3, p_4, p_1. Si noti che l'eliminazione di p_2 o p_3 al posto di p_1 provocherebbe il ritardo di due programmi: quello eliminato e p_4. □

Una realizzazione poco accorta dell'algoritmo porta ad una complessità di $O(n^2)$. Infatti si possono calcolare, per tutti i programmi p nella sequenza S, gli istanti in cui terminano la loro esecuzione, che sono dati dalla somma dei tempi di esecuzione di p e dei programmi che lo precedono. Tale calcolo richiede $O(n)$ tempo per tutti i programmi, effettuando una sola sommatoria. Confrontando poi gli istanti di terminazione dei programmi con le rispettive scadenze, si può determinare il primo programma p in ritardo e quindi eliminare il programma p' più lungo. Anche questa fase richiede $O(n)$ tempo. Poiché il procedimento di eliminazione di un programma dalla sottosequenza si potrebbe iterare per $O(n)$ volte, la complessità risulterebbe $O(n^2)$.

Utilizzando una coda con priorità Q, realizzata con uno heap, le cui operazioni sono opportunamente modificate per trattare elementi (programmi) che non coincidono con le priorità (tempi di elaborazione) [cfr. § 7 e 10], l'algoritmo di Moore può essere realizzato in modo che la sua complessità scenda ad $O(n \log n)$. La seguente procedura Pascal MOORE utilizza due vettori di interi $d[1..n]$ e $t[1..n]$ per le scadenze ed i tempi di esecuzione degli n programmi, rispettivamente, ed un vettore booleano $r[1..n]$ tale che $r[i]$ = **true** se e solo se il programma i è in ritardo. Dopo aver riordinato i vettori per $d[i]$ crescenti, la procedura considera ciascun programma i, per $i = 1, 2, \ldots, n$, e lo inserisce nella coda con priorità Q, usando una variabile intera T per calcolare la somma dei tempi di esecuzione dei programmi inseriti in Q. Se $T \geq d[i]$, allora il programma i è in ritardo e viene estratto da Q il programma j con priorità (tempo di esecuzione $t[j]$) massima, sottraendo quindi $t[j]$ da T, incrementando k, ed impostando $r[j]$ a **true**. Al termine dell'esecuzione, la variabile k contiene il numero dei programmi in ritardo, mentre il vettore r specifica quali programmi sono o non sono in ritardo. La sequenza ottima è ottenuta eseguendo dapprima i programmi con $r[i]$ = **false** in ordine di indici crescenti, seguiti dai programmi con $r[i]$ = **true** in un ordine qualsiasi.

```
procedure MOORE(d: vettore; n: integer; var r: vettore; var k: integer);
    var Q: prioricoda; i, j, T: integer;
    begin
        CREAPRIORICODA(Q); k := 0; T := 0;
        for i := 1 to n do r[i] := false;
        {ordina d[1..n] e t[1..n] per "scadenze" d[i] crescenti};
        for i := 1 to n do begin
            INSERISCI(i, t[i], Q);
            T := T + t[i];
            if T ≥ d[i] then begin
                j := MAX(Q); CANCELLAMAX(Q);
                T := T − t[j]; r[j] := true; k := k + 1
            end
        end
    end;
```

L'ordinamento iniziale richiede $O(n \log n)$ tempo. Poiché il ciclo **for** è ripetuto n volte, ed al suo interno le operazioni più gravose sono INSERISCI e CANCELLAMAX, che costano $O(\log n)$, il ciclo richiede $O(n \log n)$ tempo. Inoltre, la sequenza ottima può essere ottenuta da r in $O(n)$ tempo tramite una scansione. Pertanto, la complessità della procedura MOORE è $O(n \log n)$.

13.3 Matroidi

Moltissimi problemi (ma non tutti) per i quali un algoritmo greedy permette di trovare la soluzione ottima possono essere formulati come problemi su una struttura matematica chiamata matroide. La teoria dei matroidi è stata introdotta da Whitney

(1935) per studiare le proprietà algebriche delle dipendenze lineari, e viene richiamata molto brevemente in questo paragrafo. Una trattazione completa si può trovare nel testo di Lawler (1976).

Un matroide è una coppia $M = (S, F)$, dove S è un insieme finito di elementi ed F è una famiglia di sottoinsiemi di S, chiamati insiemi indipendenti di S, che soddisfa le proprietà seguenti:

1. *Ereditarietà.* L'insieme vuoto appartiene ad F e tutti i sottoinsiemi propri di un insieme di F appartengono anch'essi ad F, ovvero: $\emptyset \in F$, e se $I \in F$ e $J \subseteq I$, allora $J \in F$.
2. *Scambio.* Se due insiemi di F non hanno la stessa cardinalità, allora un elemento che sta soltanto nel sottoinsieme più grande può essere aggiunto all'insieme più piccolo, e l'insieme risultante appartiene ancora ad F, ovvero: se $I \in F$, $J \in F$, e $|I| < |J|$, allora esiste $x \in J - I$ tale che $I \cup \{x\} \in F$.

M è un matroide di una matrice A se S corrisponde all'insieme delle colonne di A, ed un insieme I di colonne appartiene ad F se le colonne in I sono linearmente indipendenti. M è un matroide di un grafo non orientato $G = (N, E)$ se S corrisponde all'insieme E degli archi, mentre un insieme I di archi sta in F se e solo se gli archi in I non formano un circuito. In altri termini, ciascun insieme indipendente di un matroide di un grafo forma una foresta. Il matroide di un grafo G è equivalente al matroide della matrice A di incidenza nodi-archi di G [cfr. § 9], dove gli elementi 0 e 1 della matrice A sono considerati nel campo degli interi modulo 2.

Sui matroidi sono state dimostrate moltissime proprietà, delle quali ne enunciamo nel seguito solo alcune e senza dimostrazione.

Dati un matroide M ed un insieme $I \in F$, un elemento $x \notin I$ è una estensione di I se può essere aggiunto ad I mantenendone l'indipendenza, cioè se $I \cup \{x\} \in F$. Per esempio, se M è un matroide di un grafo, allora un'estensione è un arco che può essere aggiunto ad un insieme di archi senza introdurre un circuito. Un insieme I per il quale non esiste alcuna estensione è detto massimale. Una proprietà molto utile dei matroidi, che si può facilmente dimostrare per assurdo, è che tutti i suoi insiemi indipendenti massimali hanno la stessa cardinalità.

Un matroide è pesato se c'è un peso positivo associato ad ogni elemento $x \in S$. Il peso di un insieme $I \in F$ è dato dalla somma dei pesi degli elementi che appartengono ad I. Molti problemi per i quali la soluzione ottima può essere trovata con un algoritmo greedy possono essere formulati come problemi di trovare un insieme indipendente massimale di peso massimo in un matroide pesato. Per esempio, il problema di trovare il minimo albero di copertura di un grafo G è equivalente al problema di trovare un insieme indipendente massimale di peso massimo nel matroide M del grafo G. Infatti, se c_a è il peso dell'arco a nel grafo G, allora il peso p_a dello stesso arco nel matroide M è definito come $p_a = C - c_a$, dove C è un numero più grande del massimo peso degli archi di G. In questo modo, i pesi di M sono positivi e trovare l'insieme indipendente massimale di peso massimo in M corrisponde a trovare il minimo albero di copertura di G. Infatti, $p(I) = \sum_{a \in I} p_a$ è il peso di un insieme indipendente I di M, mentre $c(I) = \sum_{a \in I} c_a$ è quello di un albero di copertura di G formato dagli stessi archi. Per costruzione, $p(I) = (n-1)C - c(I)$, dove n è il numero di nodi di G, e quindi massimizzare $p(I)$ è equivalente a minimizzare $c(I)$.

La caratteristica più importante dal punto di vista algoritmico dei matroidi pesati è che il metodo greedy verifica le proprietà della "scelta greedy" e della "sottostruttura ottima" e che pertanto trova sempre l'insieme indipendente massimale di peso massimo! Riformulato su un matroide pesato $M = (S, F)$ con funzione peso p (positiva), l'algoritmo greedy diventa:


```
procedure GREEDY-MATROID(matroide M = (S, F) con funzione peso p);
    begin
        I := ∅;
        {ordina gli elementi di S per peso decrescente};
        for ciascun x ∈ S considerato per peso p(x) decrescente do
            if I ∪ {x} ∈ F
                then I := I ∪ {x};
        {restituisci I come risultato}
    end;
```

La complessità dell'algoritmo dipende essenzialmente dal tempo richiesto per verificare che $I \cup \{x\}$ sia indipendente. Se tale tempo è $O(f(n))$, dove n è la cardinalità di S, allora la complessità risultante è $O(n \log n + nf(n))$.

13.4 Esercizi

Esercizio 13.1 (Zaino reale). Dati un insieme di n oggetti, con profitti $\{p_1, \ldots, p_n\}$ e volumi $\{v_1, \ldots, v_n\}$, tutti interi positivi, e un intero positivo c, la capacità dello zaino, il problema dello ZAINO REALE consiste nello scegliere oggetti o porzioni di essi in modo da massimizzare il profitto degli oggetti scelti senza superare la capacità dello zaino, ovvero determinare un insieme $\{x_1, \ldots, x_n\}$ di reali compresi tra 0 e 1 tali che $\sum_{1 \le i \le n} p_i x_i$ sia massima e $\sum_{1 \le i \le n} v_i x_i \le c$. Si progetti un algoritmo di tipo greedy per risolvere in tempo polinomiale il problema dello ZAINO REALE.

Si assuma che i profitti e i volumi siano memorizzati nei vettori $p[1..n]$ e $v[1..n]$. Per applicare la tecnica greedy si riordinano dapprima tali vettori per rapporto profitto/volume decrescente. Si cerca poi di scegliere la porzione più grande possibile dell'oggetto che dà il maggior profitto per unità di volume, senza superare la capacità c, ed una volta effettuata la scelta c è aggiornato con la capacità residua nello zaino. Se lo zaino non è riempito, allora si ripete il procedimento col secondo oggetto che dà maggior profitto per unità di volume, e così via, sempre aggiornando la capacità residua. Alla fine, saranno scelti per intero gli oggetti più appetibili, può essere scelta una frazione dell'oggetto immediatamente successivo, ed i rimanenti oggetti non saranno scelti. La seguente procedura ZAINO restituisce il vettore reale $x[1..n]$ e richiede tempo $O(n \log n)$.

```
procedure ZAINO(var p, v: vettore-intero; c: integer; var x: vettore-reale);
    var i: integer;
    begin
        {ordina p e v in modo che p[1]/v[1] ≤ p[2]/v[2] ≤ ... ≤ p[n]/v[n]};
        for i := 1 to n do x[i] := 0;
        i := 1;
        while (i ≤ n) and (c ≥ 0) do begin
            if v[i] ≥ c then begin x[i] := c/v[i]; c := 0; end
            else begin x[i] := 1; c := c − v[i]; end;
            i := i + 1;
        end;
    end;
```

Esercizio 13.2 (Zaino 0-1). Si dimostri con un controesempio che, al contrario dello ZAINO REALE, il problema dello ZAINO 0-1 non si può risolvere con una tecnica greedy (nello ZAINO 0-1 si richiede di prendere o lasciare un oggetto senza la possibilità di sceglierne una frazione, cioè x_i deve essere un intero uguale a 0 oppure ad 1, per $1 \leq i \leq n$).

Esercizio 13.3 (Algoritmo di Kruskal). Si assuma che i pesi degli archi di un grafo non orientato G siano compresi nell'intervallo $1..n$, dove n è il numero di nodi di G. È possibile abbassare la complessità dell'algoritmo di Kruskal?

Esercizio 13.4 (Algoritmo di Prim). Si consideri il seguente algoritmo per costruire il minimo albero di copertura T di un grafo G:

```
procedure PRIM(grafo non orientato pesato G = (N, A));
  begin
    considera T formato da un nodo qualsiasi e nessun arco;
    while esistono nodi non in T adiacenti a un nodo in T do begin
      seleziona l'arco di peso minimo tra quelli che connettono un
          nodo in T e un nodo non in T;
      aggiungi a T sia l'arco selezionato che il nodo che non era in T;
    end;
    restituisci T come il minimo albero di copertura di G;
  end;
```

Si esegua l'algoritmo di Prim sul grafo dell'Esempio 13.1. Si dimostri che l'algoritmo è corretto provando per induzione che l'insieme di archi selezionato ad ogni iterazione è un sottoinsieme degli archi di un minimo albero di copertura. Si realizzi l'algoritmo con una procedura Pascal in modo che abbia complessità $O(n^2)$. Quando è più vantaggioso l'algoritmo di Prim e quando invece quello di Kruskal?

Esercizio 13.5 (Programmi in ritardo). Si dimostri che ogni sequenza ottima per il problema dei PROGRAMMI IN RITARDO può essere trasformata in una sequenza che comprende dapprima i programmi non in ritardo ordinati per scadenze crescenti seguiti da quelli in ritardo in un ordine qualsiasi.

Data una sequenza ottima S_{OTT}, essa può essere sempre trasformata nella sequenza ottenuta con le scelte "greedy", dove i programmi in orario (cioè non in ritardo) sono ordinati in una sottosequenza iniziale per scadenze crescenti, mentre tutti i programmi in ritardo seguono quelli in orario. Infatti, se in S_{OTT} un programma in ritardo precede uno in orario, allora, spostando il programma in ritardo all'ultimo posto di S_{OTT}, il numero di programmi in ritardo non aumenta. Pertanto tutti i programmi in orario possono precedere quelli in ritardo. Consideriamo nella sottosequenza iniziale dei programmi in orario la minima scadenza d_j tale che p_j è preceduto da p_i con $d_i > d_j$ (se ci sono più p_i, allora sono considerati uno alla volta per d_i crescenti). Togliendo p_i e spostandolo immediatamente dopo p_j, i tempi di terminazione dei programmi che seguivano p_i fino a p_j (incluso) diminuiscono, mentre quelli di p_i e dei programmi che lo seguono restano immutati, e quindi il numero dei programmi in ritardo non cambia. Ripetendo il procedimento considerando sempre la minima scadenza d_j, tutti i programmi in orario possono essere ordinati per scadenze crescenti.

Esercizio 13.6 (Matrice di incidenza). Sia data la matrice di incidenza nodi-archi di un grafo non orientato. Si dimostri che un insieme di colonne della matrice è linearmente indipendente se e solo se gli archi corrispondenti a tali colonne non formano un circuito.

Esercizio 13.7 (Cardinalità insiemi indipendenti massimali). Si dimostri per assurdo che tutti gli insiemi indipendenti massimali di un matroide hanno la stessa cardinalità.

Esercizio 13.8 (Massimizzazione e minimizzazione). Si dimostri se MASSIMO ALBERO DI COPERTURA e CAMMINI MASSIMI sono equivalenti oppure no a MINIMO ALBERO DI COPERTURA e CAMMINI MINIMI.