



**POLITECNICO
DI TORINO**

Dipartimento
di Automatica e Informatica

Il paradigma greedy

Paolo Camurati

Il paradigma greedy

Per i problemi di ottimizzazione il paradigma greedy è un'alternativa:

- all'approccio divide et impera
 - alla programmazione dinamica
- in generale:
- di minor complessità, quindi più rapido
 - non sempre in grado di ritornare sempre una soluzione ottima.

Principi:

- a ogni passo: per *tentare* di trovare una soluzione globalmente ottima si scelgono soluzioni *localmente ottime*
- le scelte fatte ai singoli passi non vengono successivamente riconsiderate (no *backtrack*)
- scelte localmente ottime sulla base di una funzione di **appetibilità** (costo).

- Vantaggi
- algoritmo molto semplice
- tempo di elaborazione molto ridotto
- Svantaggi
 - soluzione non necessariamente ottima, in quanto non è detto che lo spazio delle possibilità sia esplorato in maniera esaustiva.

Algoritmo

Appetibilità note in partenza e non modificate:

- partenza: soluzione vuota
- si ordinano le scelte per appetibilità decrescenti
- si eseguono le scelte in ordine decrescente, aggiungendo, ove possibile, il risultato alla soluzione.

Appetibilità modificabili:

- come prima con modifica delle appetibilità e coda a priorità.

Selezione di attività (1)

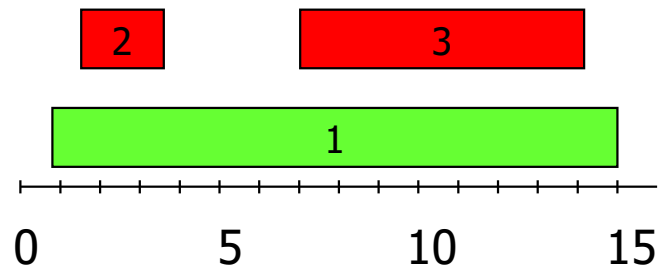
intervallo aperto a destra

- Input: insieme di n attività caratterizzate da tempo di inizio e tempo di fine $[s, f)$
- Output: insieme con il **massimo numero** di attività compatibili
- Compatibilità: $[s_i, f_i)$ e $[s_j, f_j)$ non si sovrappongono, cioè $s_i \geq f_j$ oppure $s_j \geq f_i$
- Approccio greedy: ordinamento delle attività in base a una funzione di appetibilità.

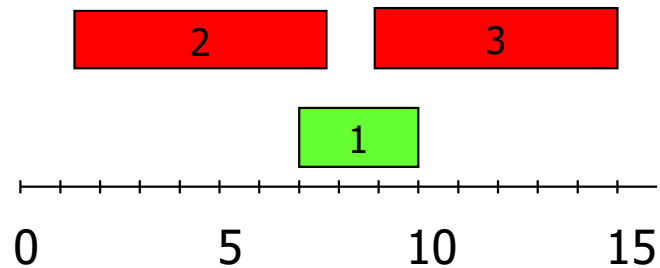


Funzioni di appetibilità

- Funzioni che non portano sempre a soluzioni ottime:
 - ordinamento per tempo di inizio crescente: un'attività che inizia presto ma dura a lungo impedisce di selezionarne altre



- Funzioni che non portano sempre a soluzioni ottime:
 - ordinamento per durata crescente: un'attività breve che interseca 2 lunghe impedisce di selezionarle

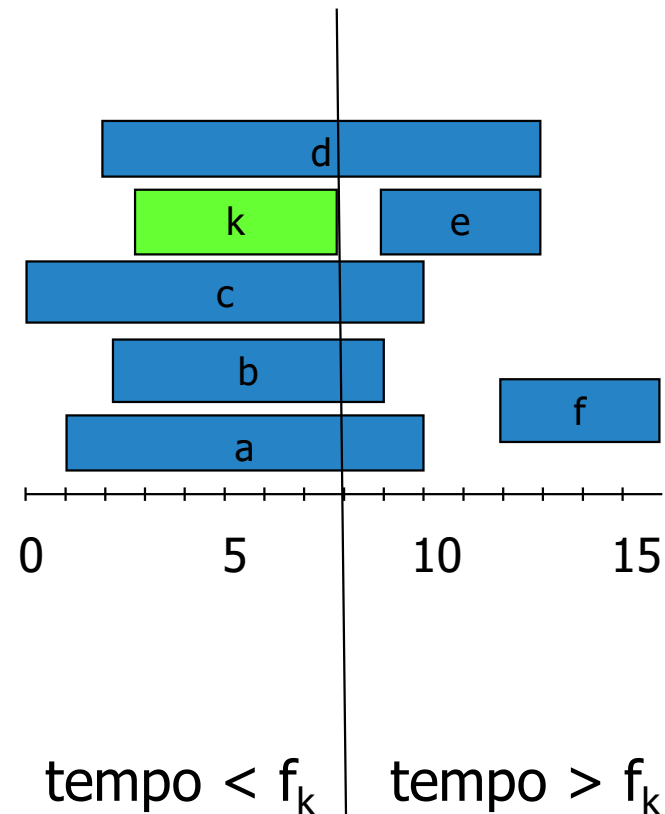


- Funzione che porta sempre a soluzioni ottime:
 - ordinamento per tempo di fine crescente:
 - supponiamo che k sia l'attività che finisce per prima:
 - scegliere k non pregiudica nulla circa le attività che iniziano dopo che è finita
 - se ci sono attività che iniziano prima di k , visto che finiscono dopo k , esse si intersecano e quindi al più se ne può scegliere una, quindi si sceglie proprio k


Si può scegliere una sola tra
a, b, c, d e k perché si
intesecono certamente tra
di loro.

Le attività a, b, c, d
potrebbero pregiudicare la
scelta di e ed f,
k certamente no.

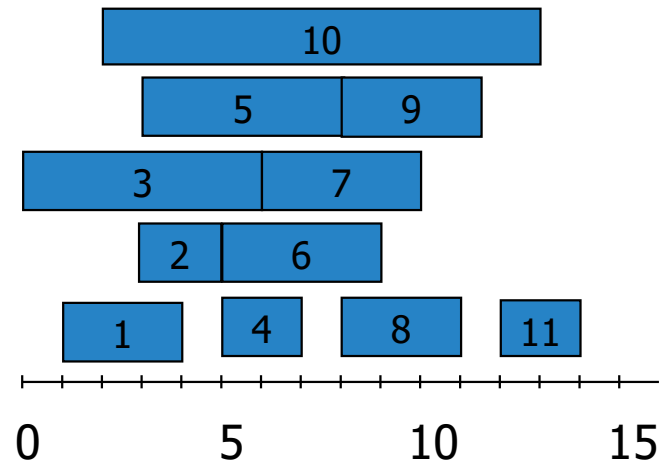
Si sceglie k.



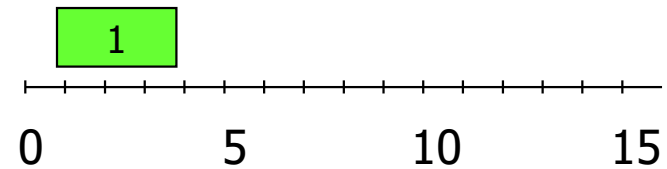
Esempio



i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

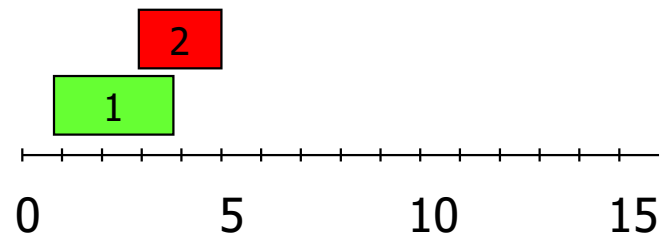


già ordinate per tempo f_i



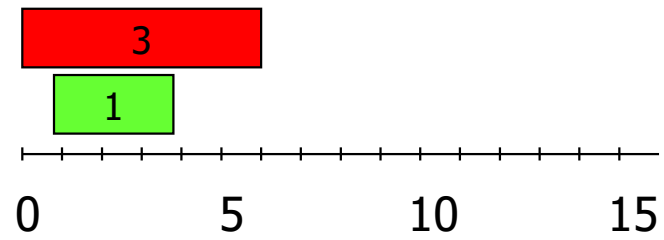


i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



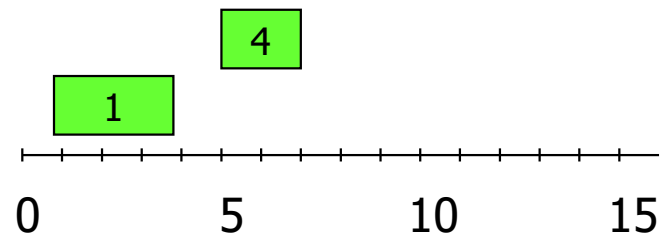


i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

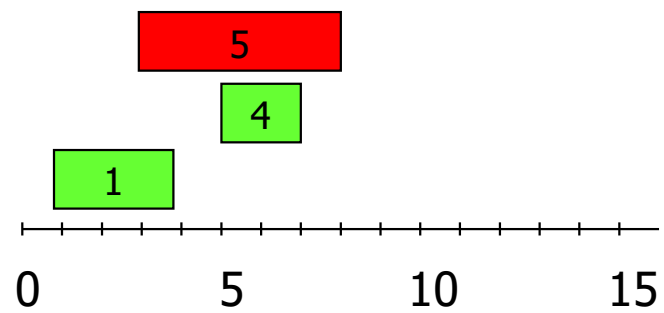




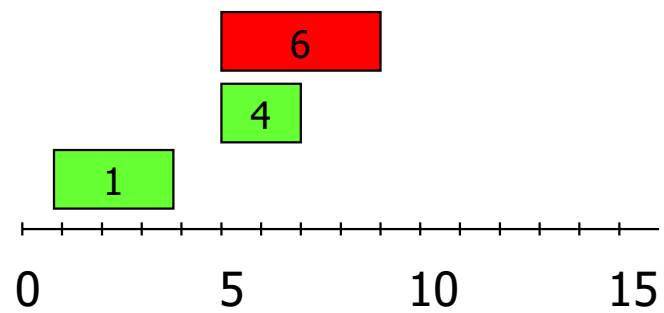
i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



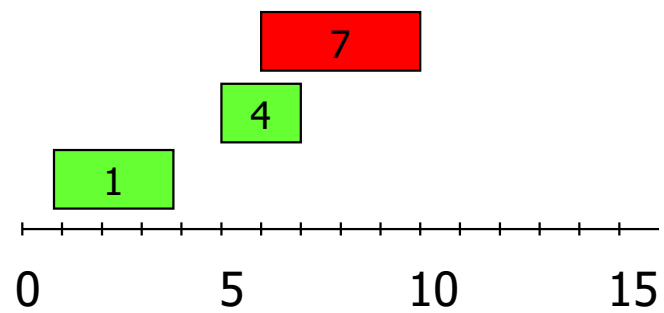
i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



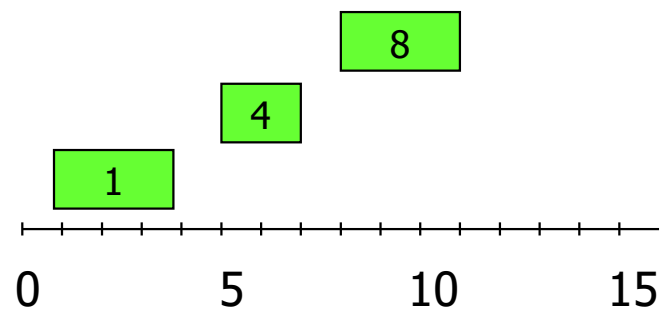
i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



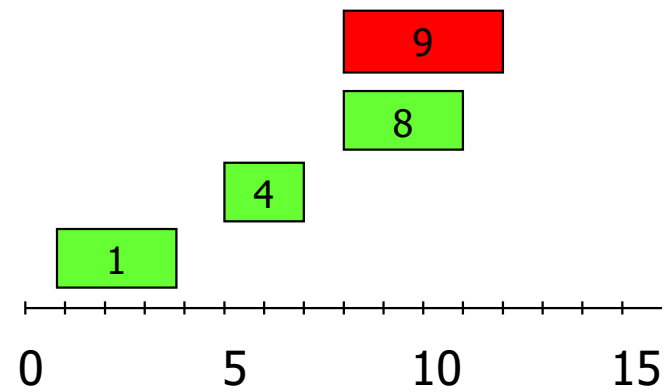
i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



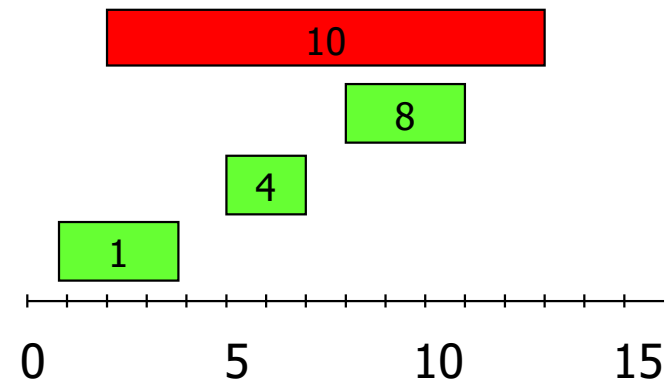
i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



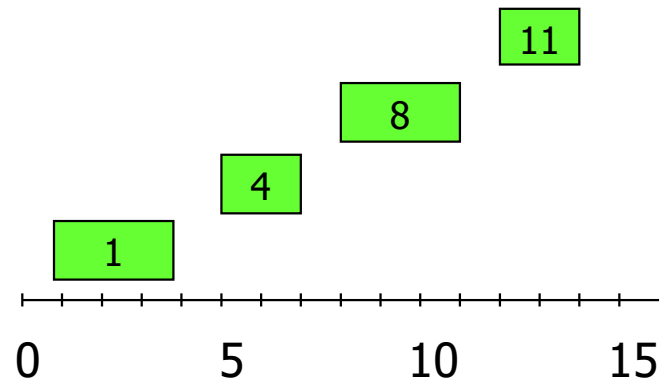
i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



Quasi ADT Item

Item

name	start	stop	selected
------	-------	------	----------

Item.h

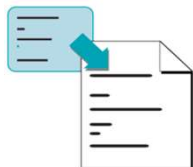
```
typedef struct {
    char name[MAXC];
    int start;
    int stop;
    int selected;
} Item;

typedef int Key;

int ITEMlt(Item A, Item B);
int ITEMgt(Item A, Item B);
Item ITEMscan();
void ITEMstore(Item A);
Key KEYgetStart(Item A);
Key KEYgetStop(Item A);
Key KEYgetSel(Item A);
void KEYsetSel(Item *pA);
```

3

Tipologia 3
nelle slide di
programmazione



01activityselection

Item.c

```
Item ITEMscan() {
    Item A;
    printf("name, start, stop: ");
    scanf("%s %d %d", A.name, &A.start, &A.stop);
    return A;
}

void ITEMstore(Item A) {
    printf("name= %s \t start= %d \t stop = %d \n",
        A.name, A.start, A.stop);
}

int ITEMlt(Item A, Item B) {
    return (A.stop < B.stop);
}

int ITEMgt(Item A, Item B) {
    return (A.stop > B.stop);
}
```

```
Key KEYgetStop(Item A) {  
    return A.stop;  
}  
  
Key KEYgetStart(Item A) {  
    return A.start;  
}  
  
Key KEYgetSel(Item A) {  
    return A.selected;  
}  
  
void KEYsetSel(Item *pA){  
    pA->selected = 1;  
}
```


client.c

```
void select(Item *act, int n) {  
    int i, stop;  
  
    KEYsetSel(&act[0]);  
    stop = KEYgetStop(act[0]);  
  
    for (i=1; i<n; i++)  
        if (KEYgetStart(act[i]) >= stop) {  
            KEYsetSel(&act[i]);  
            stop = KEYgetStop(act[i]);  
        }  
}
```

```
int main() {
    int n, i;
    Item *act;
    printf("No. of activities: "); scanf("%d", &n);
    act = calloc(n, sizeof(Item));
    printf("Input activities: \n");
    for (i=0; i<n; i++) act[i] = ITEMscan();

    MergeSort(act, n);

    select(act, n);

    printf("Selected activities: \n");
    for (i=0; i<n; i++)
        if (KEYgetSel(act[i]) == 1)
            ITEMstore(act[i]);
    return 0;
}
```

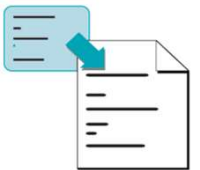
Selezione di attività (2)

- Input: insieme di n attività caratterizzate da tempo di inizio e tempo di fine $[s, f)$ e durata $f-s$
- Output: insieme con **somma delle durate massima**
- L'approccio greedy con ordinamento delle attività per tempo di fine crescente non dà sempre una soluzione ottima

Il cambiamonete

- Input: monetazione, resto da erogare
- Output: resto con numero minimo di monete
- Appetibilità: valore della moneta
- Approccio greedy: a ogni passo moneta di maggior valore inferiore al resto residuo.

```
for (i=0; i < numden; i++) {  
    coins[i] = amount / den[i];  
    amount = amount - (amount/den[i])*den[i];  
    printf("n. of  %d cent coins = %d\n",den[i],coins[i]);  
}
```



02changemachine

Esempio

Monetazione:

25, 10, 5, 1

Resto:

67

Passo Resto residuo

0 17

1 10

2 7

3 2

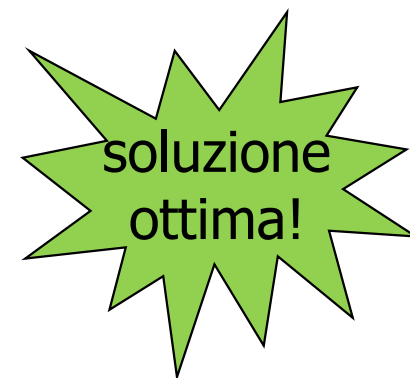
Moneta scelta

2 x 25

1 x 10

1 x 5

2 x 1



Esempio

Monetazione:

25, 10, 1

Resto:

30

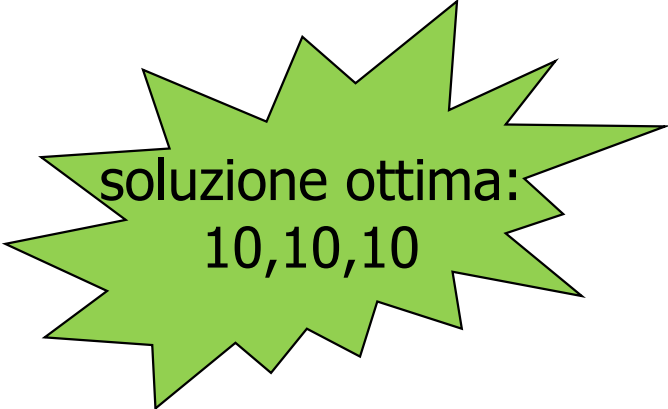
Passo	Resto residuo	Moneta scelta
-------	---------------	---------------

0	5	
---	---	--


1	0	
---	---	--

1x25

5x1



**soluzione ottima:
10,10,10**



**soluzione
non ottima!**

Il problema dello zaino (discreto)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap , determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq cap$
- $\sum_{j \in S} v_j x_j = MAX$
- $x_j \in \{0,1\}$

Ogni oggetto o è preso ($x_j=1$) o lasciato ($x_j=0$).

Appetibilità: valore specifico v_j / w_j decrescente.

Approccio greedy: a ogni passo aggiungo l'oggetto a massimo valore specifico compatibile con il peso disponibile.

Esempio

cap = 50

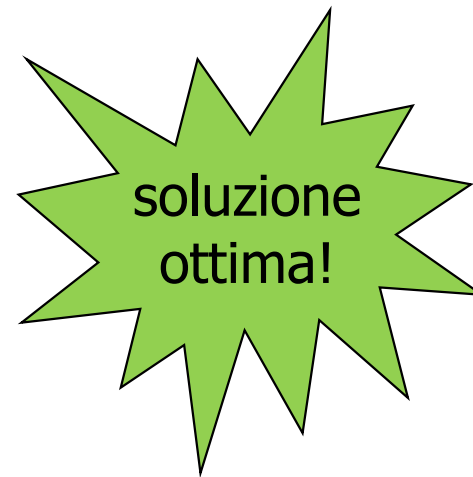
	i=1	i=2	i=3
Valore v_i	60	100	120
Peso w_i	10	20	30
Val. spec. v_i/w_i	6	5	4

Passo	Residuo	Oggetto	Valore
0	50	1	60
1	40	2	100
2	20		

Oggetti: 1 e 2
Valore 160:



Oggetti: 2 e 3
Valore 220:



Il problema dello zaino (continuo)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap , determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq P$
- $\sum_{j \in S} v_j x_j = \text{MAX}$
- $0 \leq x_j \leq 1$

Ogni oggetto può essere preso per una frazione x_j .

Appetibilità: valore specifico v_j / w_j decrescente.

Approccio greedy: ad ogni passo aggiungo la frazione di oggetto a massimo valore specifico compatibile con il peso disponibile.

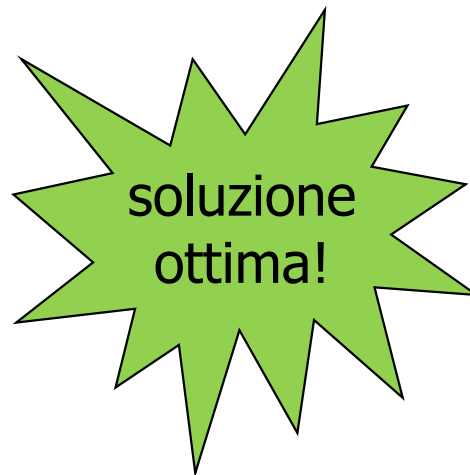
Esempio

cap = 50

	i=0	i=1	i=2
Valore v_i	60	100	120
Peso w_i	10	20	30
Val. spec. v_i/w_i	6	5	4

Passo	Residuo	Oggetto	Valore	Frazione
0	50	1	60	1.00
1	40	2	100	1.00
2	20	3	120	0.66

Oggetti: 1, 2 e $\frac{2}{3}$ di 3
Valore 240:



Quasi ADT Item

item

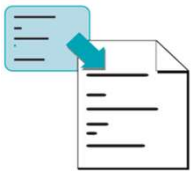
n(ame)	w(eight)	v(alue)	r(atio)	f(ract)
--------	----------	---------	---------	---------

3

Item.h

```
typedef struct {char n[MAX]; float w; float v;  
               float r; float f; } Item;  
  
typedef float Key;  
  
int     ITEMeq(Item A, Item B);  
int     ITEMgt(Item A, Item B);  
Item    ITEMscan();  
void    ITEMstore(Item A);  
Key     KEYgetW(Item A);  
Key     KEYgetV(Item A);  
Key     KEYgetF(Item A);  
void    KEYsetF(Item *pA, float f);
```

Tipologia 3
nelle slide di
programmazione



03fractionalknapsack

```

. . .
Item ITEMscan() {
    printf("name, weight, value: ");
    scanf("%s %f %f", A.name, &(A.w), &(A.v));
    A.r = A.v/A.w;
    return A;
}
void ITEMstore(Item A) {
    printf("name= %s \t weight= %.2f \t value = %.2f \t
          fract= %.2f \n", A.name, A.w, A.v, A.f);
}
int  ITEMeq(Item A, Item B) {return (A.r == B.r);}
int  ITEMgt(Item A, Item B) {return (A.r > B.r);}
Key  KEYgetW(Item A) {return (A.w);}
Key  KEYgetV(Item A) {return (A.v);}
Key  KEYgetF(Item A) {return (A.f);}
void KEYsetF(Item *pA, float f) {pA->f = f;}

```

main.c

...

```
float knapsack(int n, Item *objects, float cap) {  
    float stolen = 0.0, res = cap;  
    int i;  
  
    for (i=0; i<n && (KEYgetW(objects[i]) <= res); i++) {  
        KEYsetF(&objects[i], 1.0);  
        stolen = stolen + KEYgetV(objects[i]);  
        res = res - objects[i].weight;  
    }  
  
    KEYsetF(&objects[i], res/KEYgetW(objects[i]));  
    stolen = stolen + KEYgetF(objects[i])*KEYgetV(objects[i]);  
  
    return stolen;  
}
```

```

int main() {
    int n, i; float cap, stolen=0.0; Item *objects;
    printf("No. of objects: "); scanf("%d", &n);
    objects = calloc(n, sizeof(Item));
    printf("Input objects: \n");
    for (i=0; i<n; i++)
        objects[i] = ITEMscan();
    printf("Capacity of knapsack: "); scanf("%f", &cap);
    MergeSort(objects, n);
    stolen = knapsack(n, objects, cap);
    printf("Results: \n");
    for(i = 0; i < n; i++)
        ITEMstore(objects[i]);
    printf("Total amount stolen: %.2f \n", stolen);
    return 0;
}

```


Codici di Huffman (1950)

- Codice: stringa di bit associata a un simbolo $s \in S$
 - a lunghezza fissa
 - a lunghezza variabile
- Codifica: da simbolo a codice
- Decodifica: da codice a simbolo.



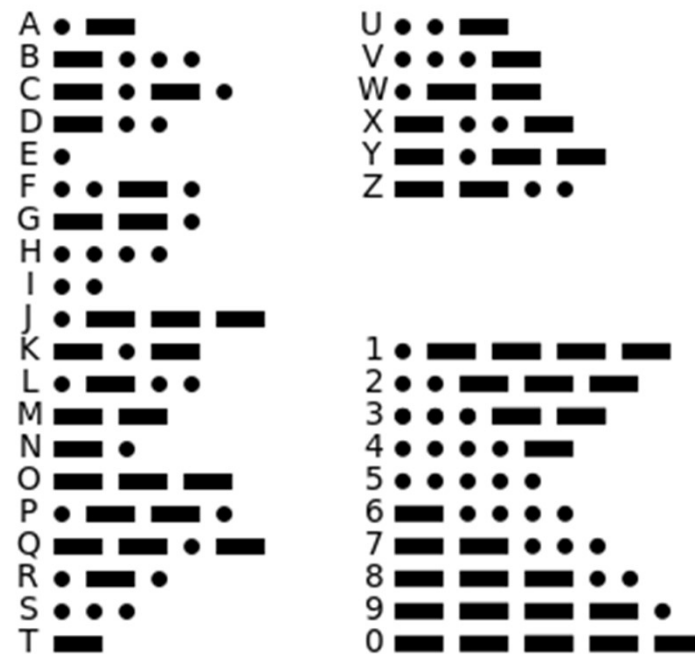
Codici a lunghezza fissa:

- numero di bit $n = \lceil \log_2 (\text{card}(S)) \rceil$
- vantaggio: facilità di decodifica
- uso: simboli isofrequenti

Codici a lunghezza variabile:

- svantaggio: difficoltà di decodifica
- vantaggio: risparmio di spazio di memoria
- uso: simboli con frequenze diverse.

Esempio: alfabeto Morse con durata specificata di lineette e punti e pause tra simboli (punti e lineette) e tra parole.



Esempio

	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
codice fisso	000	001	010	011	100	101
codice variabile	0	101	100	111	1101	1100

file con 100.000 caratteri

codice fisso: $3 \times 100.000 = 300.000$ bit

codice variabile:

$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1.000 = 224.000$ bit

Codici prefissi

Codice (libero da) prefisso:

nessuna parola di codice valida è un prefisso di un'altra parola di codice.

Codifica: giustapposizione di stringhe

Decodifica: percorrimiento di albero binario.

PS: il codice Morse non è libero da prefisso, ma ci sono le durate e le pause.

Esempio

a = 0

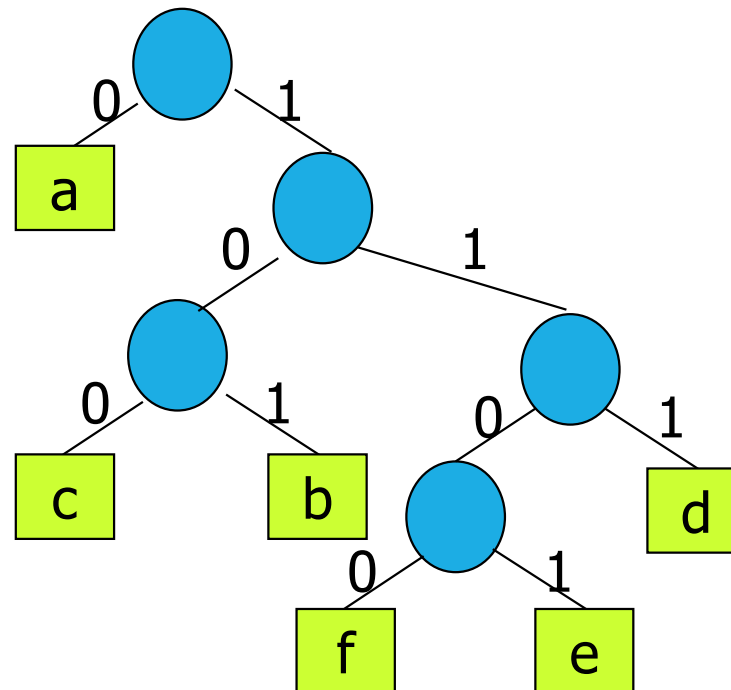
b = 101

c = 100

d = 111

e = 1101

f = 1100



Codifica:

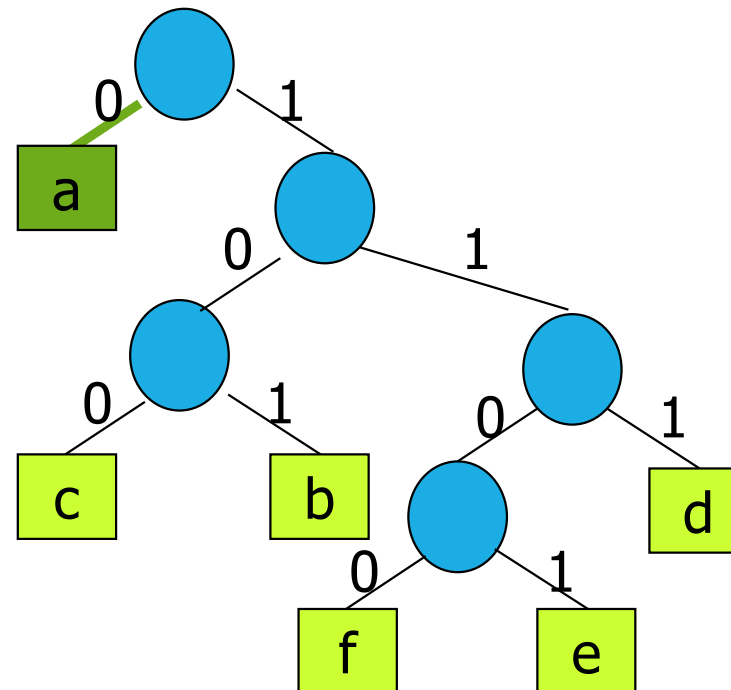
a b f a a c

0101110000100

Decodifica:

0101110000100

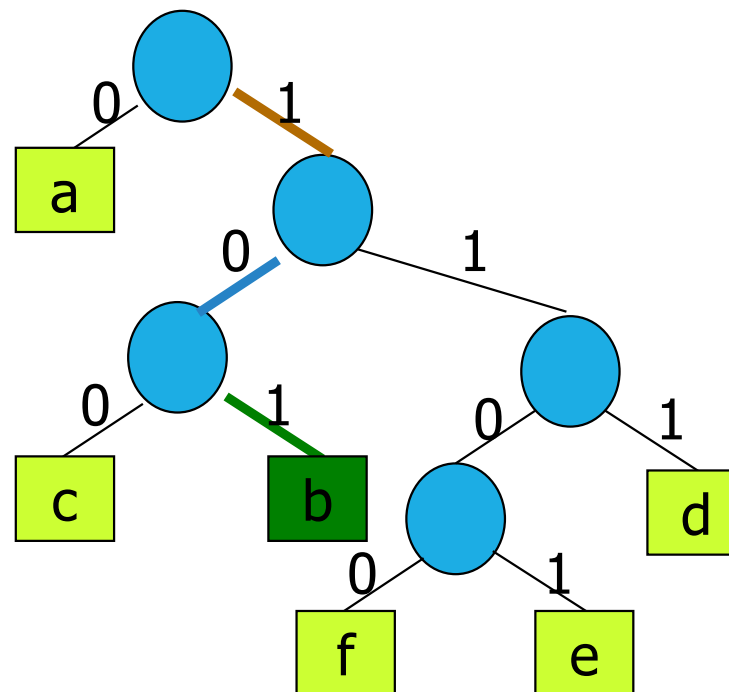
a



Decodifica:

101110000100

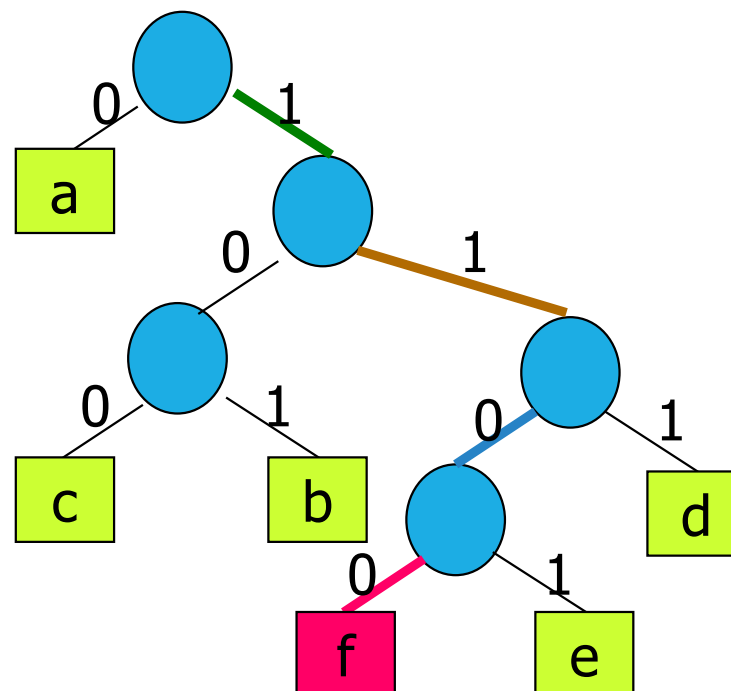
ab



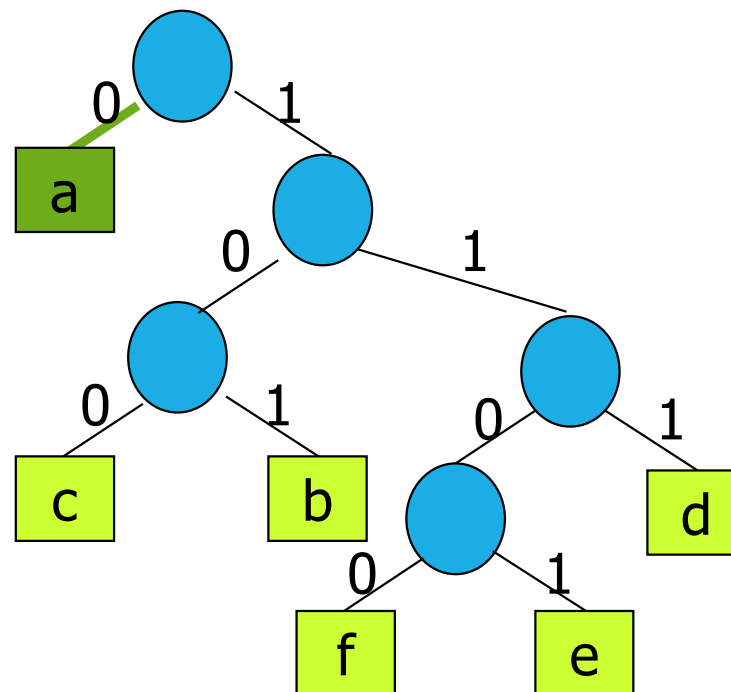
Decodifica:

110000100

abf



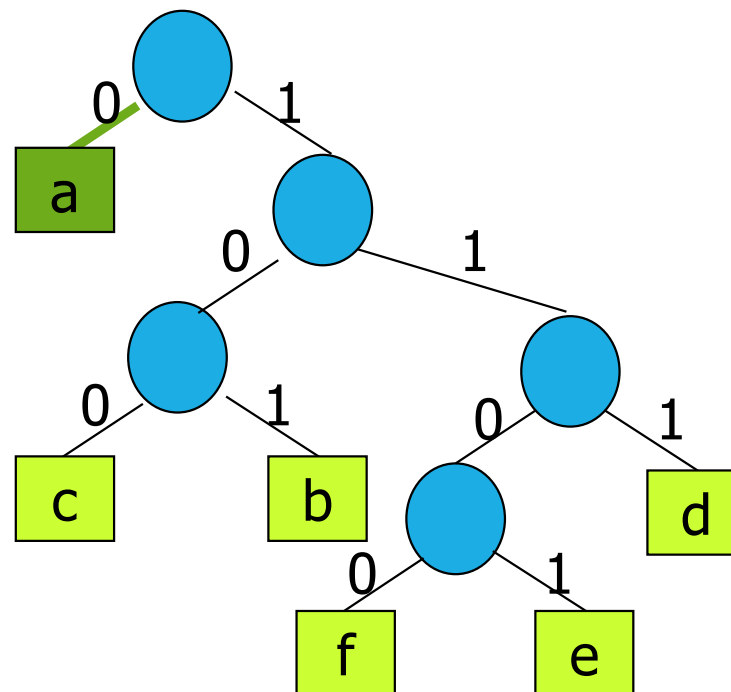
Decodifica:
00100
abfa



Decodifica:

0100

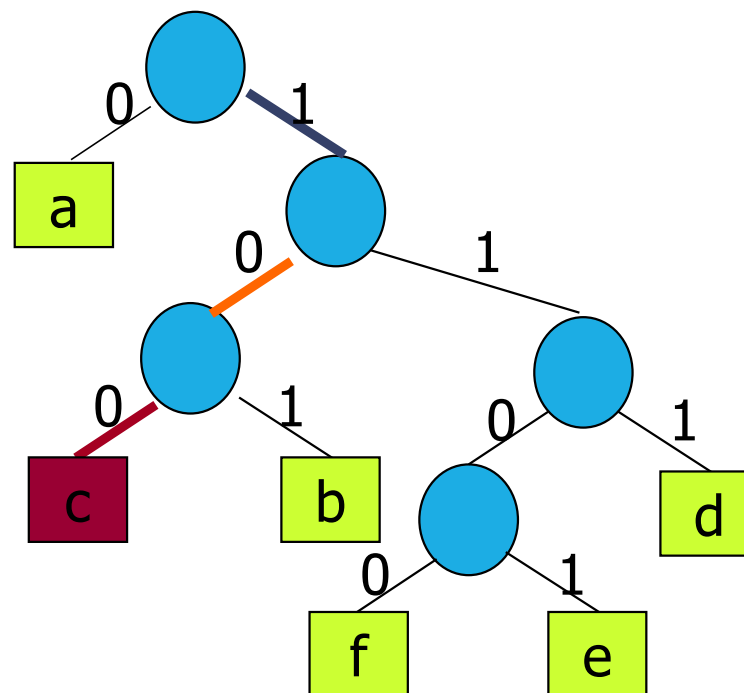
abfa



Decodifica:

100

abfaac



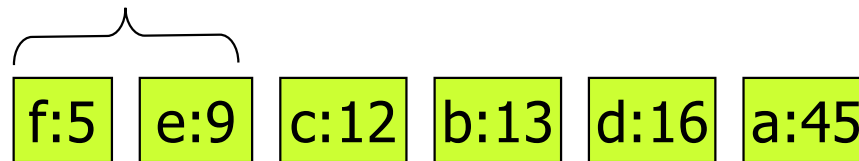
Costruzione dell'albero binario

Struttura dati: coda a priorità (frequenze crescenti, proprietà greedy)

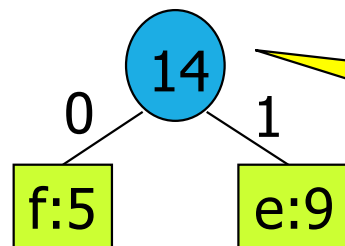
- Inizialmente: simbolo = foglia
- Passo i-esimo:
 - estrazione dei 2 simboli (o aggregati) a minor frequenza
 - costruzione dell'albero binario (aggregato di simboli): nodo = simbolo o aggregato, frequenza = somma delle frequenze
 - inserimento nella coda a priorità
- Terminazione: coda vuota.

estrazione

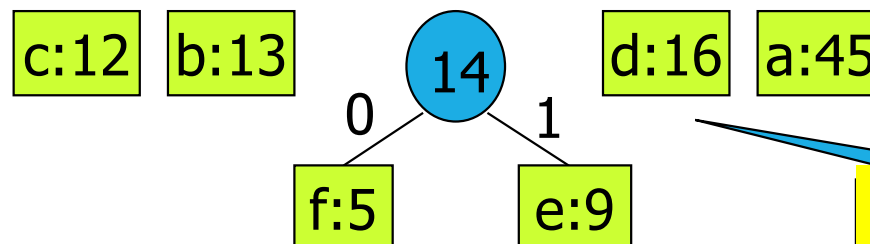
coda a priorità come
vettore ordinato



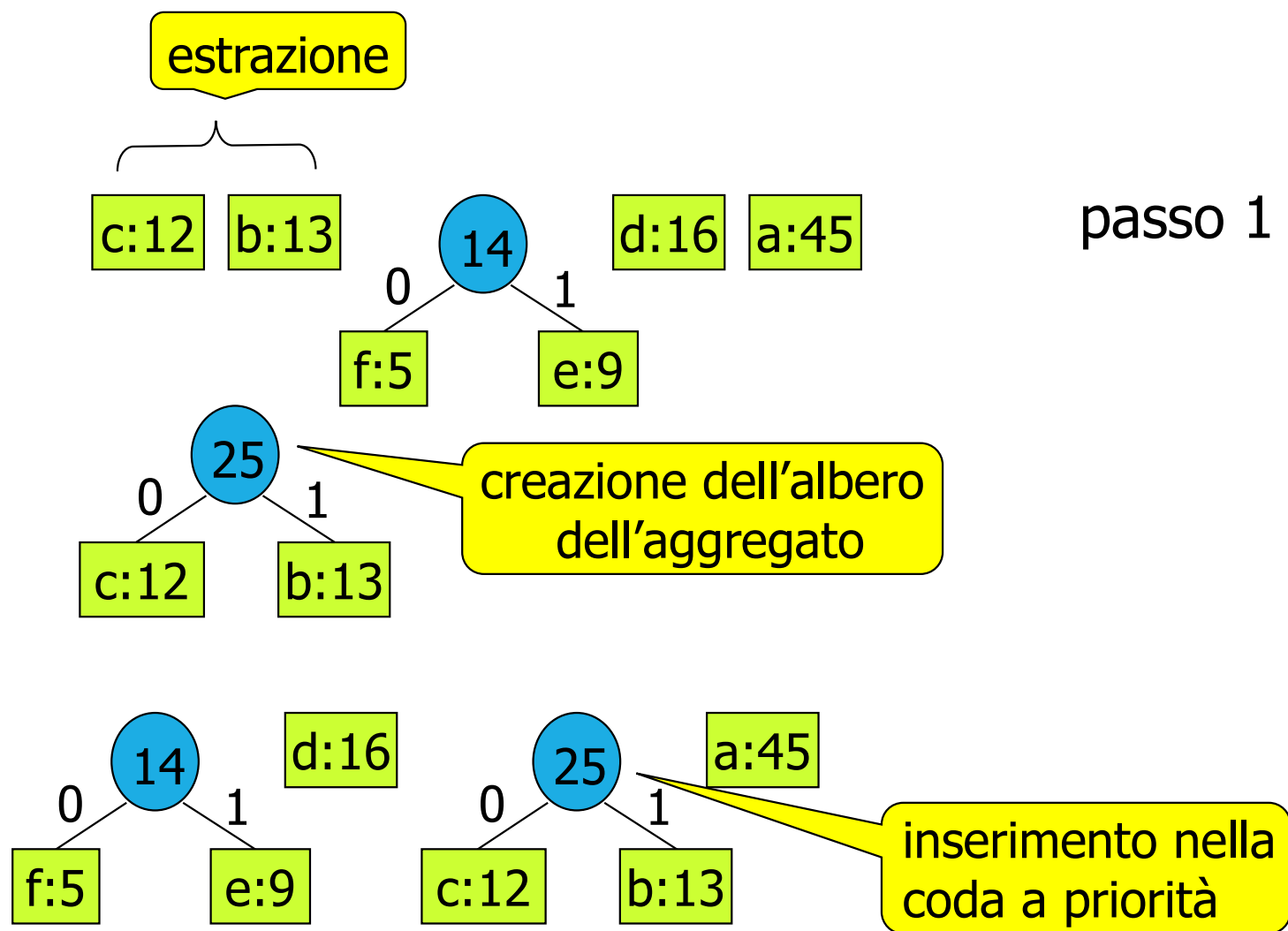
passo 0

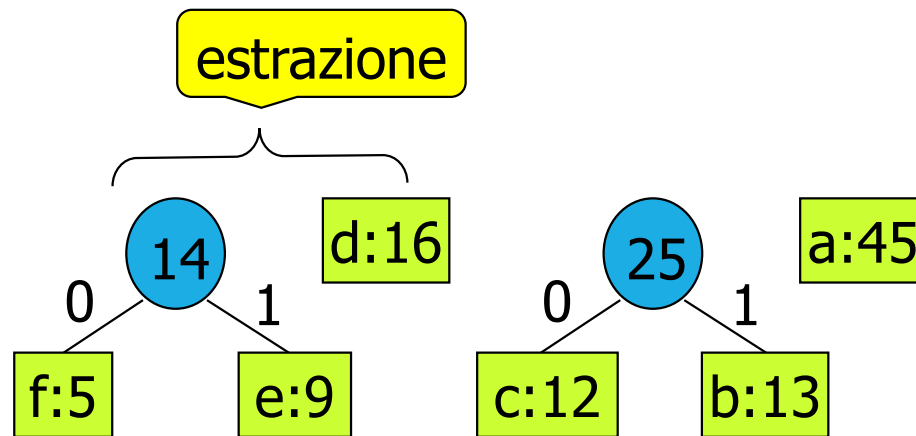


creazione dell'albero
dell'aggregato

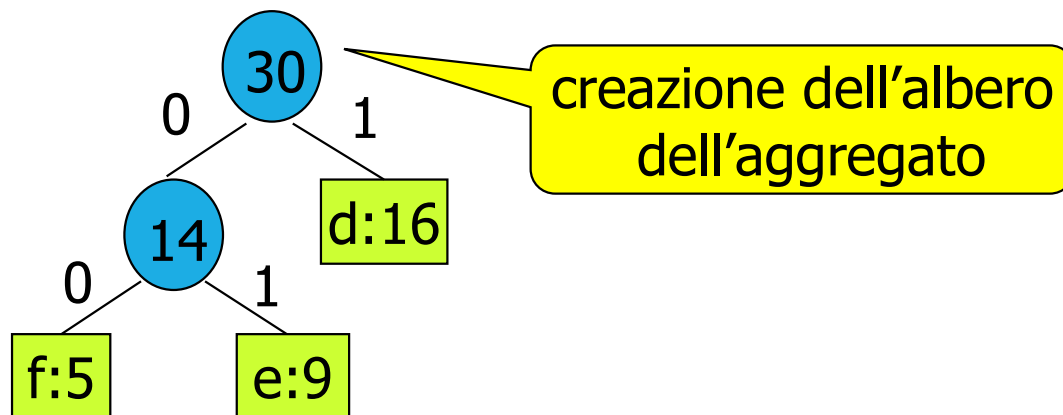


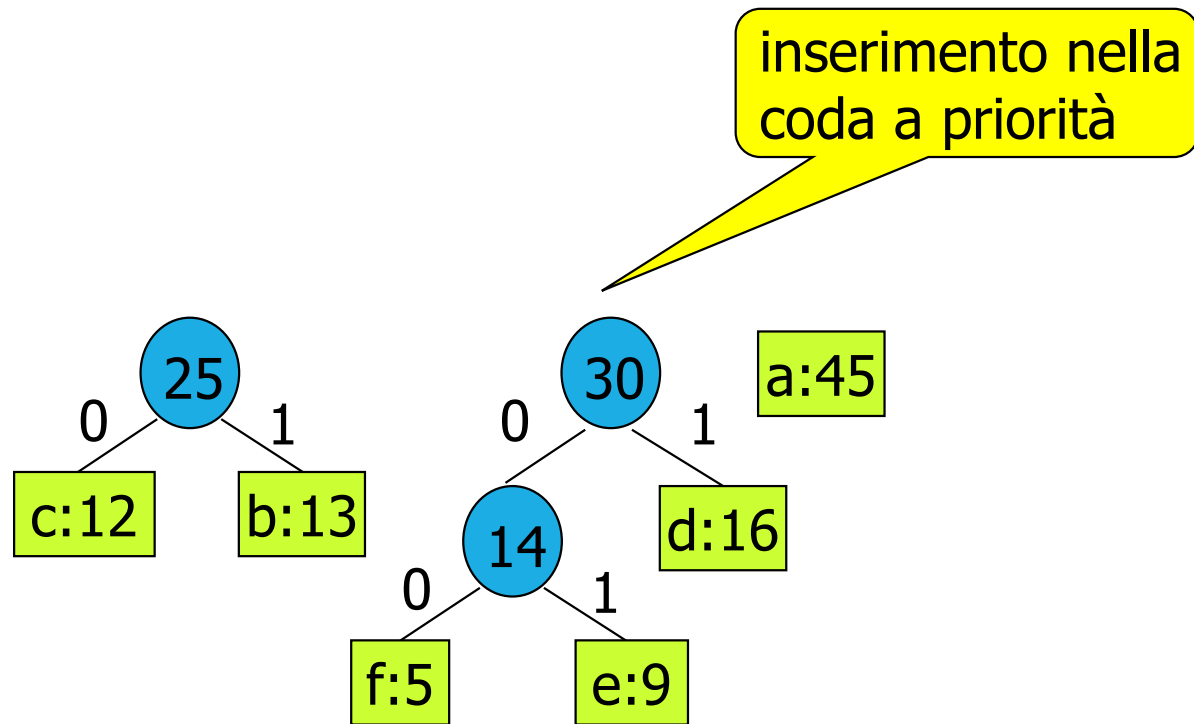
inserimento nella
coda a priorità

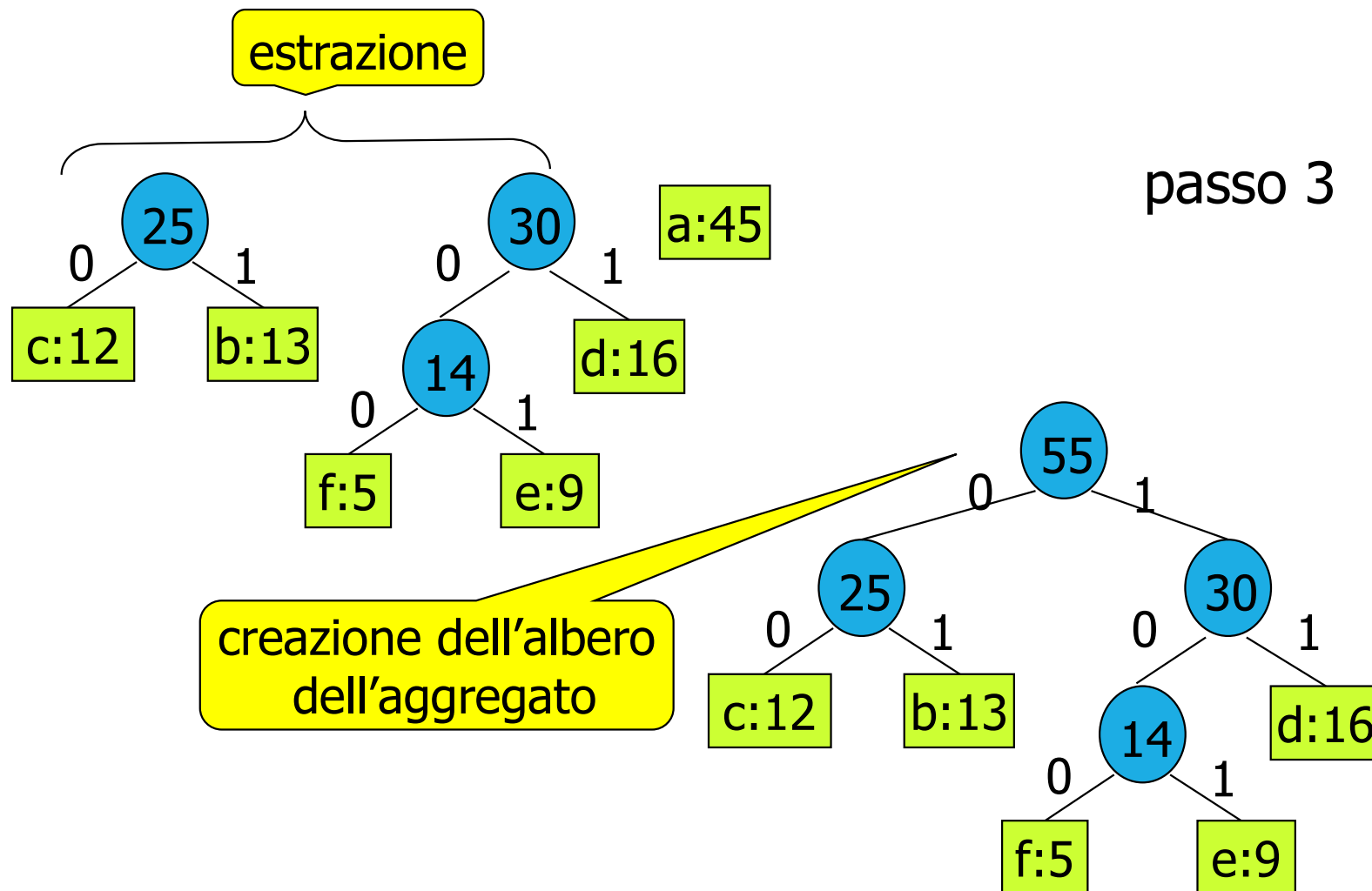


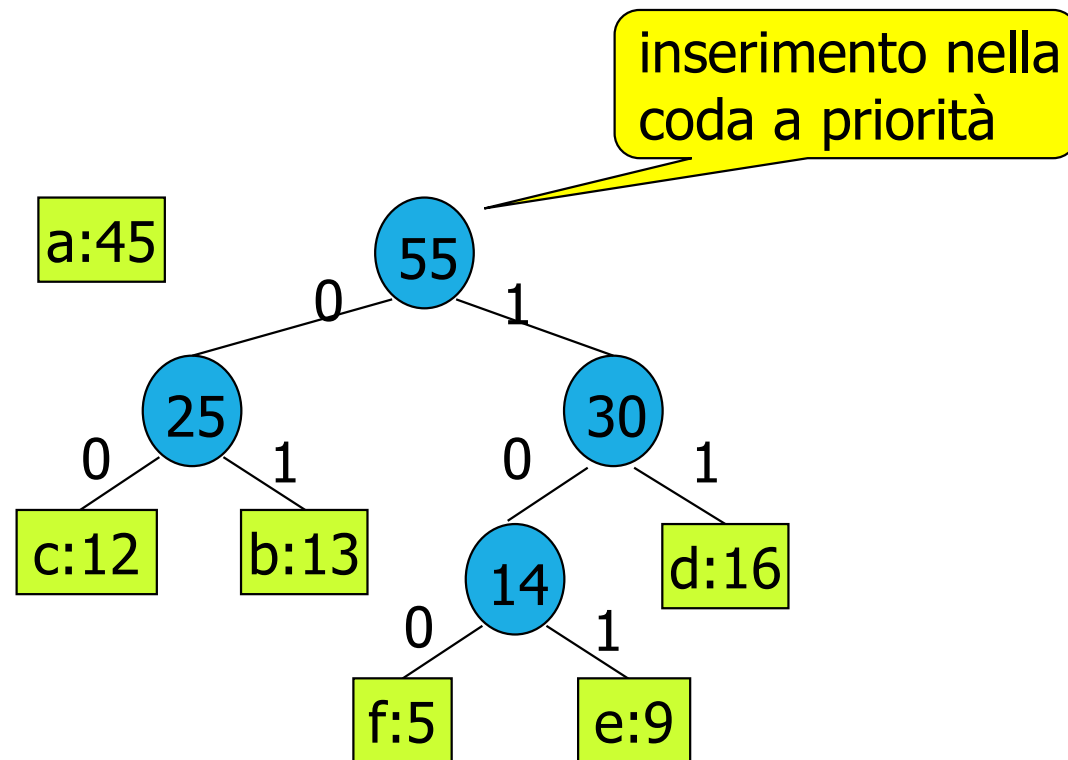


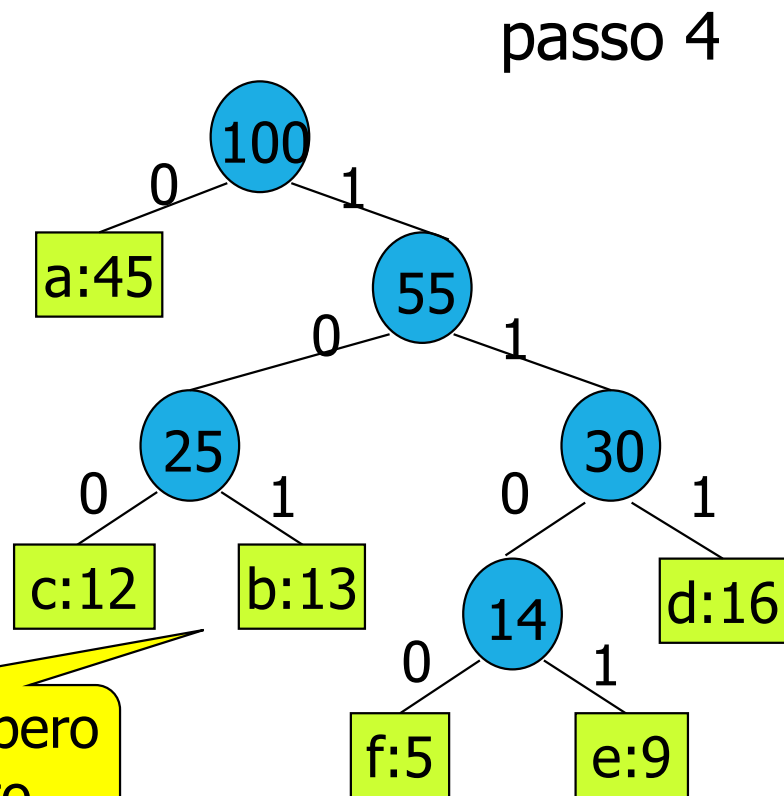
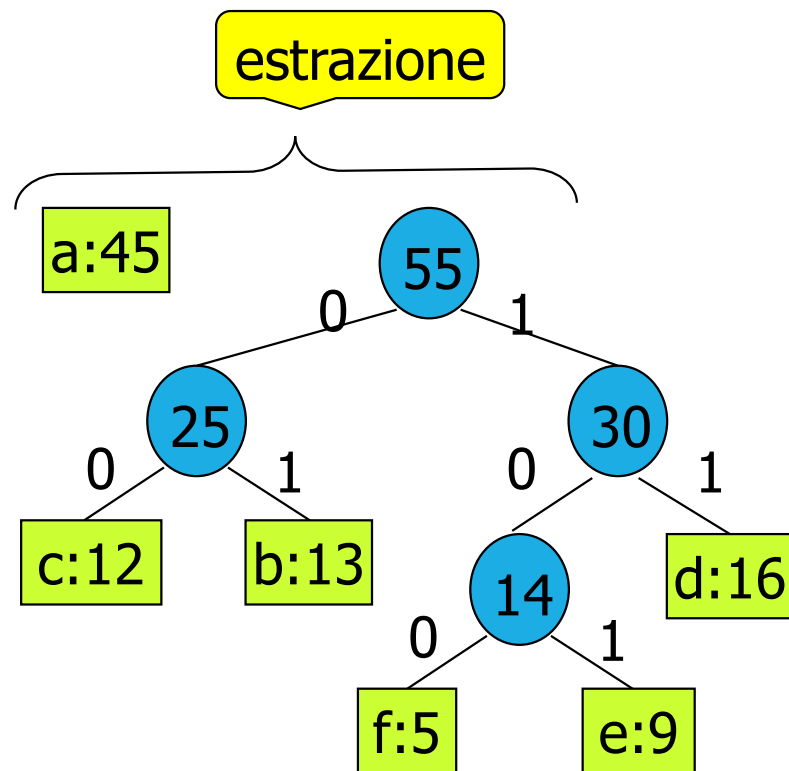
passo 2











creazione dell'albero
dell'aggregato

Riferimenti

- Selezione di attività:
 - Cormen 17.1
- Problema del ladro e dello zaino:
 - Cormen 17.2
- Codici di Huffman
 - Cormen 17.3

Esercizi di teoria

- 9. Paradigma greedy
 - 9.1 Activity selection
 - 9.2 Codici di Huffman

