

FONDAMENTI DI INFORMATICA

Corso di Laurea in Ingegneria Meccanica

Università degli Studi di Firenze

A. A. 2006-2007

APPUNTI INTEGRATIVI

**Strutture dati e Algoritmi di ricerca e ordinamento:
modi alternativi di espressione in C**

Prof. Alessandro Fantechi

1. Lista realizzata con strutture e puntatori: funzioni *car* (ritorna il valore in testa alla lista) , *cons* (inserisce un elemento in testa alla lista) , *cdr* (toglie un elemento dalla testa della lista), visita della lista con stampa.

```
typedef struct EL {
    tipoatomi val;
    struct EL *next;
};
typedef struct EL elemlista;
typedef elemlista *tipolista;

tipoatomi car (tipolista lis)
{
    if (lis == NULL)
        printf("operazione non eseguibile");
    else
        return lis->val;
}

void cons (tipoatomi e, tipolista *lis)
{
    tipolista temp;
    temp = malloc(sizeof(elemlista));
    temp->val = e;
    temp->next = *lis;
    *lis = temp;
}

void cdr (tipolista *lis)
{
    tipolista temp;
    if (*lis == NULL)
        printf("operazione non eseguibile");
    else
    {
        temp = *lis;
        *lis = (*lis)->next;
        free(temp);
    }
}
```

(Versione di cdr senza effetti laterali)

```
tipolista cdr (tipolista lis)
{
    if (lis == NULL)
        printf("operazione non eseguibile");
    else
        return lis->next;
}

void stampa (tipolista lis)
{
    while (lis != NULL)
    {
        printf("%d", lis->val);
    }
}
```

```

        lis = lis -> next
    }
}

```

Alternativamente:

```

void stampa (tipolista lis)
{
    tipolista p;
    for (p= lis; p != NULL; p = p -> next)
        printf("%d", p->val);
}

```

2. Pila realizzata con un vettore, : funzioni *top* (ritorna il valore in testa alla lista) , *push* (inserisce un elemento in testa alla lista) , *pop* (toglie un elemento dalla testa della lista),

```

typedef struct{
    tipoatomi elementi[nmax];
    int top;
} pila;

```

```

tipoatomi top(pila p)
{
    if (p.top== -1)
        printf("operazione non eseguibile\n");
    else
        return p.elementi[p.top];
}

```

```

void push(tipoatomi e, pila *p)
{
    if((p->top == namx-1)
        printf("operazione non eseguibile\n");
    else {
        p->top++;
        p->elementi[p->top] = e;
    }
}

```

```

void pop(pila *p)
{
    if(p->top == -1)
        printf("operazione non eseguibile\n");
    else {
        p->top--;
    }
}

```

3. Coda FIFO realizzata con strutture e puntatori: funzioni *incoda* (inserisce un elemento in coda) , *outcoda* (toglie un elemento dalla testa).

```

typedef struct EL {
    tipoelem val;
}

```

```

    struct EL *next;
    };
typedef struct EL elemcoda;
typedef elemcoda *coda;

void incoda(coda *p,*u, tipoelem e)
{
    coda temp;
    temp = malloc(sizeof(elemcoda));
    temp->val = e;
    temp->next = NULL;
    if (*p == NULL)
    {
        *p = temp;
        *u = temp;
    }
    else
    {
        (*u)->next = temp;
        *u = (*u)->next;
    }
}

```

```

void outcoda (coda *p,*u)
{
    coda aux;
    if (*p == NULL)
        printf("operazione non eseguibile");
    else
    {
        aux = *p;
        *p = (*p)->next;
        if (*p == NULL)
            *u = NULL
        free(aux);
    }
}

```

Versione di outcoda che ritorna come risultato il valore dell'elemento eliminato.

```

void outcoda (coda *p,*u, tipoelem *e)
{
    coda aux;
    if (*p == NULL)
        printf("operazione non eseguibile");
    else
    {
        *e = (*p)->val;
        aux = *p;
        *p = (*p)->next;
        if (*p == NULL)
            *u = NULL
        free(aux);
    }
}

```

4. Coda realizzata con vettore circolare

```
typedef struct {
    tipoelem elem[nmax];
    int p, u;
} coda;
```

u indica la posizione in cui inserire il prossimo elemento, p indica la posizione in cui è presente l'elemento da togliere. Inizialmente p e u sono posti a 0.

```
void incoda (coda *q, tipoelem e)
{
    if ((q->u+1) % nmax != q->p)
    {
        q->elem[q->u] = e;
        q->u = (q->u+1) % nmax;
    }
    else printf("coda piena\n");
}
```

```
void outcoda (coda *q, tipoelem *e)
{
    if (q->u != q->p)
    {
        *e = q->elem[q->p];
        q->p = (q->p+1) % nmax;
    }
    else printf("coda vuota\n");
}
```

5. Albero Binario – visita in ordine anticipato (preordine)

```
typedef struct EL {
    tipoatomi info;
    struct EL *sin, *des;
};
typedef struct EL nodo;
typedef nodo *albero;
```

```
void preordine (albero alb)
{
    if (alb!=NULL)
    {
        printf("%d", alb->info);
        preordine(alb->sin);
        preordine(alb->des);
    }
}
```

(ricerca binaria su albero binario di ricerca)

```
boolean ricerca (albero alb, tipoatomi e)
{
    if (alb==NULL)
        return FALSE
    else
        if (e == alb->info)
            return TRUE
        else
            if (e < alb->info)
                return ricerca(alb->sin, e);
            else
                return ricerca(alb->des, e);
}
```

6. Alcuni esempi di algoritmi e cenni alla complessità computazionale.

Algoritmi di ricerca in un vettore: ricerca esaustiva

La seguente funzione C ricerca un elemento in un vettore, vale a dire torna il valore vero se l'elemento è presente nel vettore, e un valore falso altrimenti. Per far questo esegue un algoritmo di ricerca esaustiva, cioè scandisce il vettore fino a quando o si è trovato l'elemento, o si è finito di scandire il vettore.

Supponiamo che `vett` sia un tipo array definito come:

```
typedef elem vett[dim]
```

dove `dim` è un numero (non una variabile!), e `elem` è un tipo qualsiasi.

La funzione prende come parametro anche la lunghezza del vettore; in questo modo possiamo descrivere l'algoritmo in modo generale.

```
boolean ricerca (vett z, int n, elem x);
{
    int i; boolean flag;
    flag = FALSE;
    for (i=0; i<n; i++)
        if (z[i] == x) flag = TRUE;
    return flag;
}
```

Si noti che la funzione esegue al più tante iterazioni del ciclo `while` quanti sono gli elementi dell'array. Se questi sono `n`, il tempo occorrente per eseguire il calcolo è quindi proporzionale a `n`. Si dice che questa funzione ha *complessità computazionale lineare*.

D'altra parte, è inutile continuare ad eseguire il ciclo quando si fosse già trovato l'elemento; la seguente funzione introduce una nuova condizione nell'`while`, che permette di uscire anticipatamente dal ciclo quando questo si verifica (le operazioni booleane in C si indicano con i simboli: `!` per il NOT, `&&` per l'AND, `||` per l'OR).

```
boolean ricerca (vett z, int n, elem x);
{
    int i; boolean flag;
    i=0;
    flag = FALSE;
    while (i<n && !flag)
    {
```

```

        if (z[i] == x)    flag =  TRUE;
        i++;
    }
return flag;
}

```

Senza altro questo algoritmo è più vantaggioso del precedente, perché si sono eliminati dei passi inutili. Ma possiamo considerare che ora il tempo di esecuzione della funzione dipende dall'elemento che cerchiamo: se l'elemento non è presente nel vettore, si eseguono n passi; se è presente, si eseguono k passi, dove k è la sua posizione nel vettore. In media, k varrà $n/2$. Possiamo quindi dire che il tempo occorrente è comunque ancora proporzionale a n , e che questa funzione ha ancora *complessità computazionale lineare*.

Ricerca binaria o dicotomica

Un significativo miglioramento dell'efficienza si ha con l'algoritmo di *ricerca binaria* o *dicotomica*. Questo algoritmo presuppone che il vettore sia ordinato (in modo crescente) e si basa sul seguente principio: confronto l'elemento cercato con l'elemento di mezzo del vettore. Se non è uguale, se è minore posso andarlo a cercare nel primo semivettore, se è maggiore nel secondo semivettore. La ricerca nei semivettori adotta lo stesso procedimento, fino a quando si trovi l'elemento o si arrivi a considerare un semivettore di lunghezza unitaria

```

boolean ricerca (vett z, int n, elem x);
{
    int k, m;
    k=0; m=n;
    int i; boolean flag;
    flag = FALSE;
do
    {
        i = (m+n) / 2

        {
            if (z[i] == x)    flag =  TRUE;
            if (z[i] < x)
                k = i;
            else
                m = i;
        }
        while (k<m && ! flag)
            return flag;
    }
}

```

Questa funzione esegue un numero di iterazioni del ciclo al più uguale al logaritmo di n in base 2. Infatti il vettore viene diviso per 2 p volte, fino a quando non diventa $n/2^p = 1$, cioè $n = 2^p$, e quindi $p = \log_2 n$.

Il tempo occorrente per il calcolo è quindi al più proporzionale al logaritmo di n , e si dice che questa funzione ha *complessità computazionale logaritmica*. Per capire quanto sia più vantaggioso questo algoritmo di ricerca rispetto alla ricerca esaustiva basta fare un semplice esercizio in cui si consideri come cresce il tempo di esecuzione al crescere della dimensione del vettore, per valori della dimensione del vettore pari a 10, 100, 1000, 10000.

D'altra parte, questo algoritmo assume che gli elementi del vettore siano ordinati, quindi assume anche che il tipo *elem* degli elementi sia un tipo su cui è definito un ordinamento totale e una operazione di confronto (l'operazione <).

Lo strumento della complessità computazionale, di cui qui non diamo una definizione formale, è quindi utile a classificare gli algoritmi per la loro potenziale efficienza nel trattare problemi di grandi dimensioni. Un algoritmo che avesse una complessità computazionale esponenziale, cioè il cui tempo di esecuzione cresce come 2^n potrebbe essere utile solo per piccole dimensioni del problema, e completamente inutile da un punto di vista pratico per dimensioni appena maggiori, quand'anche si avesse a disposizione un calcolatore molto potente, cioè molto veloce nell'eseguire le istruzioni. Per capire quanto questo sia vero si consideri come esercizio quanto vale il tempo di 2^n microsecondi, per $n = 10, 100, 1000, 10000$.

Si noti infine che alla minore complessità computazionale della ricerca binaria fa riscontro una "maggiore complicazione" dell'algoritmo.

7. Algoritmi di ordinamento

Come ulteriore esempio di algoritmi diversi, con diversa complessità computazionale, che calcolano la stessa funzione, mostriamo alcuni algoritmi di ordinamento di un vettore.

Questi algoritmi condividono la definizione del tipo vettore:

```
typedef tipoelem vett[dim];
```

e la definizione di una procedura di scambio del valore di due variabili. Si noti come lo scambio richieda una variabile di appoggio, e come le variabili da scambiare siano passate alla procedura per indirizzo.

Procedura di scambio di due elementi di un vettore

```
void scambia (tipoelem *x, *y)
{
    tipoelem aux;
    aux=*x;
    *x=*y;
    *y=aux;
}
```

Notare, nelle procedure che seguono, che il vettore da ordinare non è, apparentemente, passato per indirizzo; si deve invece ricordare che i vettori sono trattati in C come puntatori al loro primo elemento, quindi sono sempre passati per indirizzo.

Selection_sort (ordinamento per selezione)

La procedura di ordinamento per selezione esegue, per ogni passo del ciclo esterno, un ciclo interno che sceglie l'elemento minimo e lo porta nella prima posizione, usando la procedura di scambio data sopra definita. Il prossimo passo del ciclo esterno si può limitare ad ordinare il resto del vettore, ripetendo lo stesso algoritmo. Ad ogni passo vengono quindi considerati vettori sempre più piccoli, fino ad arrivare a un vettore di lunghezza unitaria che è ordinato per definizione.

```
void selection_sort (vett z, int n);
{
```

```

int i,j,imin;
for (i=0; i<n-1; i++)
{
    imin=i;
    for (j=i+1; j<n; j++)
        if (z[j]<z[imin])
            imin = j;
    scambia(&z[i], &z[imin]);
}
}

```

Per valutare la complessità di questo algoritmo, notiamo che il ciclo interno compie $n-2$ passi la prima volta, $n-3$ la seconda, $n-4$ la terza, e così via. Quindi complessivamente il numero di passi è pari a $\sum_{i=1, n-2} i = (n-2)*(n-1)/2 = n^2/2 - 3*n/2 - 1$:

Possiamo quindi dire, trascurando i termini di ordine inferiore, che il tempo occorrente per l'ordinamento è proporzionale a n^2 , e che questa funzione ha *complessità computazionale quadratica*.

Bubblesort (ordinamento a bolle)

Questo algoritmo opera scandendo il vettore a partire dal fondo; man mano che trova due elementi successivi in ordine scorretto, li scambia. In questo modo l'elemento minimo "galleggia" via via verso la prima posizione del vettore, dopo di che si può passare ad ordinare il resto dell'array. Il nome dell'algoritmo è ripreso dal fatto che gli elementi di valore minore tendono a risalire come le bolle in una bicchiere d'acqua. Un vantaggio di questo algoritmo è che, mentre si scandisce il vettore dal basso verso l'alto, si può controllare se il vettore risulta già ordinato (se non si sono fatti scambi, vuol dire che il vettore è già ordinato). In questo caso, posso abbandonare la procedura.

```

void bubblesort (vett z, int n);
{
    int i,j;
    boolean fine;
    i = 0
    do
    {
        i++;
        fine = TRUE;
        for (j=n-1; j>=i; j--)
            if (z[j]<z[j-1])
                {
                    scambia(&z[j], &z[j-1]);
                    fine = FALSE;
                }
    }
    while (!fine && i!=n-1)
}

```

Si può vedere facilmente che le argomentazioni per il calcolo della complessità di questo algoritmo, sono le stesse dell'algoritmo di ordinamento per selezione, con la variante che esistono casi in cui l'algoritmo può terminare prematuramente, quando (porzioni de) il vettore risulti già ordinato. Possiamo però considerare che probabilisticamente questo sia un evento raro, e che questa funzione presenta mediamente una *complessità computazionale quadratica*.

Quicksort (ordinamento veloce)

Questo algoritmo si basa su un principio diverso, che mira, in analogia con l'algoritmo di ricerca binaria, ad abbattere la complessità computazionale suddividendo il vettore in due parti, in modo da ripetere l'ordinamento di sottovettori solo un numero di volte pari al logaritmo della dimensione.

L'algoritmo fissa dapprima un elemento detto *pivot*; attraverso una scansione simultanea dal primo elemento in poi e dall'ultimo all'indietro, e scambiando gli elementi eventualmente trovati non in ordine in questa scansione, suddivide il vettore nel sottovettore contenente tutti gli elementi minori del pivot, e in quello contenente gli elementi maggiori. In particolare, nella funzione che segue, viene scelto come pivot il primo elemento del vettore; la scansione viene effettuata mediante due indici, che si muovono dagli estremi del vettore in senso contrario fino a toccarsi.

Una volta compiuta la suddivisione nei due sottovettori, si passa ad applicare lo stesso algoritmo ai due sottovettori, fino a quando i sottovettori non diventino di dimensione unitaria. Per poter far questo, occorre avere a disposizione una lista in cui memorizzo le coppie di indice minimo e massimo dei sottovettori da ordinare. Quindi la funzione ripetutamente applicherà il procedimento visto sopra, a partire dall'intero vettore; questo procedimento inserirà nella lista gli indici di due sottovettori, a meno che i sottovettori non risultino di un elemento. Nei passi successivi si estrarranno dalla lista gli indici di uno dei vettori ancora da ordinare e si applicherà a questo il procedimento di suddivisione, finché la lista non contenga più coppie di indici.

Nella funzione che segue si è appunto supposto di avere realizzato una lista e di avere a disposizione delle funzioni che operano sulla lista (inserzione, estrazione, test che la lista sia vuota). Una tale struttura si può agevolmente realizzare con un vettore.

```
void quicksort (vett z, int n)
{
    int sin,des, i,j;
    tipoelem x;
    tipolista lis;          /*suppongo l'esistenza di un tipo lista
                             su due valori interi, con funzioni ins e estr.
                             Nella lista vengono memorizzate le coppie di indice
                             minimo e massimo dei sotto-vettori ancora da ordinare*/
    ins (&lis, 0, n-1);
    while (!listavuota(lis))
    {
        estrai (&lis, &sin,&des);
        i=sin;
        j=des+1;
        x=z[sin];
        while(i<j)
        {
            do j--;
            while(z[j]>x);
            do i++;
            while(z[i]<=x && i!=j);
            if (i<j)
                scambia(&z[i], &z[j]);
        }
        if (sin!=j)
            scambia(&z[sin], &z[j]);
        if (sin<j-1)
            ins (&lis, sin,j-1);
        if (des>i)
            ins (&lis, i,des);
    }
}
```

}
}

Per valutare la complessità di questo algoritmo, supponiamo innanzitutto che il vettore sia di dimensioni tali che la sua suddivisione successiva in sottovettori sia uniforme: in questo caso avremo che la prima suddivisione produce 2 vettori di $n/2$ elementi ciascuno, la seconda produce 4 vettori di 4 elementi ciascuno, ... la p -esima 2^p sottovettori di $n/2^p = 1$ elementi, quindi $p = \log_2 n$. La scansione di ciascuno dei sottovettori è lineare, quindi la prima suddivisione del vettore intero viene fatta in n passi, la seconda prevede due scansioni di $n/2$ passi ciascuna, in totale n passi, la terza quattro scansioni di $n/4$ passi, in totale ancora n passi, etc.... Il totale di passi fatti dall'algoritmo ammonterà pertanto a $n * \log_2 n$ e il tempo occorrente per l'ordinamento sarà proporzionale a $n \log n$. Quindi questa funzione ha *complessità computazionale semilogaritmica*, che è comunque molto migliore di quella quadratica degli altri algoritmi. Questo risultato, che giustifica il nome dell'algoritmo, si è ancora una volta ottenuto al prezzo di una "maggiore complicazione" dell'algoritmo.