Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Verificare se un vettore è ordinato

Scrivere un algoritmo che, dato in input un array, restituisca

- 1 se tale array è ordinato
- 0 altrimenti

In [3]:
```
// versione 1
#include <stdio.h>
int is_sorted(float vett[], int dim)
{
    int sorted = 1;
    for(int i = 1; i < dim; i++)
       if( vett[i-1] > vett[i])
           sorted = 0;
    return sorted;
}

int main()
{
    float v[] = {1, 4, 6, 8, 2};//{3,1,2,4,6,8,9,-5};
    int n     = 5; //8;
    int ord   = is_sorted(v,n);

    if(ord == 0)
        printf("il vettore non è ordinato\n");
    else
        printf("il vettore è ordinato\n");
}
```

```
il vettore non è ordinato
```

In [4]:
```
// versione 2
#include <stdio.h>
int is_sorted(float vett[], int dim)
{
    int sorted = 1;
    int i      = 1;
    while(i < dim && sorted == 1)
    {
        if( vett[i-1] > vett[i])
           sorted = 0;
        i++;
    }
    return sorted;
}

int main()
{
    float v[] = {1, 4, 6, 8};//{3,1,2,4,6,8,9,-5};
    int n     = 4; //8;
    int ord   = is_sorted(v,n);

    if(ord == 0)
        printf("il vettore non è ordinato\n");
    else
        printf("il vettore è ordinato\n");
}
```

```
il vettore è ordinato
```

Ordinamento per selezione

In [5]:
```
#include <stdio.h>

void stampa_vett(float v[], int n)
{
    printf("(");
    for(int i = 0; i < n; i++)
        printf("%.2f, ", v[i]);
    printf("\b\b)\n");
}

float idx_min_array(float v[], int n, int start)
{
    int idx_min = start;
    float min   = v[start];
    // Loop interno
    for (int i=start+1; i<n; i++)
        if (v[i] < min)
        {
            min     = v[i];
            idx_min = i;
        }
    return idx_min;
}


void selection_sort(float v[], int n)
{
    int idx_min;
    // Loop esterno
    for(int i=0; i<n-1; i++) // n-1 in quanto ultimo elemento sarà già ne
                             // agli spostamenti precedenti
    {
        idx_min    = idx_min_array(v, n, i);
```

```
        float tmp  = v[i];
        v[i]       = v[idx_min];
        v[idx_min] = tmp;
    }

}



int main()
{
    float vett[] = {8,4,2,1,5,7,9,3,6};
    int n        = 9;
    printf("prima: ");
    stampa_vett(vett,n);
    selection_sort(vett,n);
    printf("dopo:  ");
    stampa_vett(vett,n);


    return 0;
}
```

```
prima: (8.00, 4.00, 2.00, 1.00, 5.00, 7.00, 9.00, 3.00, 6.00)
dopo:  (1.00, 2.00, 3.00, 4.00, 5.00, 6.00, 7.00, 8.00, 9.00)
```

Il tempo impiegato dalla funzione `selection_sort(...)` dipende dal tempo t_{iter} impiegato da ogni iterazione del loop esterno.

t_{iter}, a sua volta, non è altro che la somma tra:

- il tempo $t_{min}(k)$ speso per eseguire la funzione `idx_min_array(...)`. Tale funzione non è altro che una ricerca del minimo su un (sotto)array di `v` di lunghezza k, con k che diventa, ad ogni iterazione del loop esterno, più piccolo di 1;
- il tempo t_{swap} speso per fare uno scambio tra due elementi (3 operazioni elementari, indipendente dalla lunghezza dell' array)

Quindi $t_{iter} = t_{min}(k) + t_{swap}$

Inoltre, sappiamo già che $t_{min}(k)$ è **lineare** sulla dimensione dell' input meno 1, cioè $k-1$. Possiamo quindi esprimerlo come $t_{min}(k) = a(k-1) + b$, con a e b coefficienti della parte rispettivamente dipendendente e indipendente dalla dimensione dell' input k.

L'algoritmo di Selection Sort può essere visto come $n-1$ ricerche del minimo, ognuna di queste effettuata un su array di dimensioni sempre più piccola, con relativo scambio.

- 1° iterazione: ricerca del minimo su un array di n elementi + scambio
- 2° iterazione: ricerca del minimo su un array di $n-1$ elementi + scambio
- 3° iterazione: ricerca del minimo su un array di $n-2$ elementi + scambio

$\vdots$

- (n-1)° iterazione: ricerca del minimo su un array di 2 elementi + scambio

la complessità complessiva dell'algoritmo sarà data dalla somma de:

- i tempi per le ricerche dei minimi t_{min}.
- il tempo per effettuare tutti gli scambi, che è facilmente calcolabile come $(n-1) \cdot t_{swap}$

Concentriamoci adesso sul tempo complessivo per le ricerche dei minimi:

$$t_{min}(n) + t_{min}(n-1) + t_{min}(n-2) + \cdots + t_{min}(2) = \quad (1)$$

$$= \sum_{k=2}^{n} t_{min}(k) \quad (2)$$

$$= \sum_{k=2}^{n} (a(k-1)+b) = \sum_{k=1}^{n-1} (a \cdot k + b) = a \sum_{k=1}^{n-1} k + (n-1)b$$

La prima sommatoria è la somma dei primi $n-1$ numeri interi, quindi si può usare la formula di Gauss ($\sum_{i=1}^{n} i = n(n+1)/2$):

$$\sum_{k=1}^{n-1} k = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$$

Il tempo complessivo dell'implementazione proposta di Selection Sort è dato da

$$a\frac{n^2-n}{2} + (b + t_{swap})(n-1)$$

Il termine "dominante" di questa espressione è n^2, quindi il tempo necessario al completamento dell'algoritmo varia in maniera *quadratica* al variare della dimensione dell'input.

Ricerca binaria

In [3]:
```
#include <stdio.h>
void stampa_vett_part(int v[], int inizio_idx, int fine_idx)
{
    printf("(");
    if( fine_idx < inizio_idx)
        printf("  ");
    for(int i = inizio_idx; i <= fine_idx; i++)
        printf("%d, ", v[i]);
    printf("\b\b)");
}

void stampa_vett(int v[], int n)
{
    stampa_vett_part(v, 0, n-1);
}

int ricerca_binaria(int v[], int n,  int key)
{
```

```
    int idx_inizio   = 0;
    int idx_fine     = n-1;

    int idx_trovato  = -1;

    int it = 0; // utile solo per visualizzazione
    while(idx_inizio <= idx_fine && idx_trovato == -1)
    {
        int idx_centrale = (idx_inizio + idx_fine) / 2;

        printf("it:%d sottovett. sx: ", it+1);
        stampa_vett_part(v, idx_inizio, idx_centrale-1);
        printf(" centrale: (%d) ", v[idx_centrale]);
        printf("sottovett. dx: ");
        stampa_vett_part(v, idx_centrale+1, idx_fine);
        printf("\n");

        if (v[idx_centrale] == key) // cerco al centro
            idx_trovato = idx_centrale;
        else if(key > v[idx_centrale]) // cerco nel lato destro
            idx_inizio  = idx_centrale + 1;
        else
            idx_fine    = idx_centrale - 1; // cerco nel lato sinistro

        it++; // utile solo per visualizzazione
    }
    return idx_trovato;

}


int main()
{
    int vett[]   = {1,4,6,8,9,10,11};
    int n        = 7;
    int k        = 11;
    printf("vettore: ");
    stampa_vett(vett,n);
    printf("\n");
    int idx_k = ricerca_binaria(vett,n,k);
    if(idx_k == -1)
        printf("il valore %d non si trova nel vettore\n", k);
    else
        printf("il valore %d si trova nel vettore in posizione %d.\n", k,
    return 0;
}
```

```
vettore: (1, 4, 6, 8, 9, 10, 11)
it:1 sottovett. sx: (1, 4, 6) centrale: (8) sottovett. dx: (9, 10, 11)
it:2 sottovett. sx: (9) centrale: (10) sottovett. dx: (11)
it:3 sottovett. sx: () centrale: (11) sottovett. dx: ()
il valore 11 si trova nel vettore in posizione 6.
```

Il corpo del ciclo è composto da poche istruzioni elementari. Definiamo t_{iter} il tempo per eseguire una singola iterazione.

Bisogna quindi valutare quante volte viene ripetuto il corpo del ciclo.

Caso migliore: l'elemento da cercare si trova proprio al centro dell'array $\Rightarrow$ tempo indipendente dalla dimensione dell'array

Caso peggiore: vengono effettuate tutte le iterazioni possibili (ad esempio, nel caso l'elemento non è presente oppure se l'elemento si trova in uno dei due estremi dell'array). In questi casi, l'array viene ripetutamente spezzato in due parti (una parte sx ed una parte dx). La parte in esame quindi diventa sempre più piccola di un fattore 2 ad ogni iterazione, fino a raggiungere la dimensione di 1 elemento.

- 1° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento l'intero array
- 2° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento una metà ($\frac{n}{2}$) dell'array
- 3° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento una metà della metà ($\frac{n}{4}$) dell'array
- 4° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento $\frac{n}{8}$ dell'array

$\vdots$

- k° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento $\frac{n}{2^{k-1}}$ dell'array

il ciclo si fermerà quando si arriverà al caso di un array di lunghezza 1, i.e. quando $\frac{n}{2^{k-1}} = 1 \Rightarrow 2^{k-1} = n \Rightarrow k-1 = \log_2 n \Rightarrow k = \log_2 n + 1$

l'algoritmo quindi nel caso peggiore terminerà dopo $k = \log_2 n + 1$ iterazioni.

Tempo di esecuzione varia quindi in maniera **logaritmica** al variare della dimensione dell'input!

Esempio: per un array di lunghezza 64, sono necessarie al più appena $\log_2 64 = 6$ iterazioni del ciclo per completare la ricerca.

$\Rightarrow$ in generale più veloce della ricerca lineare!

Inserimento ordinato in array

In [7]:
```
#include <stdio.h>
#define MAXDIM 100
void stampa_vett_part(int v[], int inizio_idx, int fine_idx)
{
    printf("(");
    if( fine_idx < inizio_idx)
        printf("  ");
    for(int i = inizio_idx; i <= fine_idx; i++)
        printf("%d, ", v[i]);
    printf("\b\b)");
}
void stampa_vett(int v[], int n)
{
```

```
    stampa_vett_part(v, 0, n-1);
}

void inserisci_ordinato(int vett[], int dim, int val)
{
    int  i = 0;

    // scorro finchè non trovo un elemento >= val
    while(i < dim && vett[i] < val)
        i++;
    /* una volta trovato (se esiste) sposto a dx di 1
       tutti gli elementi alla sua dx
    */
    for(int j=dim; j > i; j--)
        vett[j] = vett[j-1];
    // inserisco val nella posizione "liberata"
    vett[i] = val;
}


int main()
{
    int n;
    int v[MAXDIM];
    printf("quanti valori vuoi inserire?");
    scanf("%d",&n);
    for(int i = 0; i < n; i++)
    {
        int val;
        printf("Inserisci il prossimo valore:");
        scanf("%d", &val);
        inserisci_ordinato(v,i, val);
    }
    stampa_vett(v, n);
    return 0;
}
```

```
quanti valori vuoi inserire?

Inserisci il prossimo valore:

Inserisci il prossimo valore:

Inserisci il prossimo valore:

(1, 5, 6)
```

L'inserimento ordinato funziona solo se il vettore di partenza è a sua volta ordinato $\Rightarrow$ è necessario utilizzare **sempre** la funzione di inserimento ordinato (fin dal primo elemento) per garantire che quest'ultimo rimanga ordinato.

Domanda: dato che il vettore è ordinato, si può sfruttare la ricerca binaria per cercare la posizione in cui inserire l'elemento?

Risposta: Sì, ma poi comunque, una volta trovata la posizione di inserimento, avrei dovuto spostare gli elementi per far spazio al nuovo elemento, vanificando in parte i benefici della ricerca binaria.

In [ ]: