










Esercitazione di laboratorio n. 9

Esercizio svolto n. 0: antenne della rete di telefonia mobile

È dato un insieme di n città, disposte su una strada rettilinea, identificate con gli interi da 1 a n , ognuna caratterizzata dal numero di abitanti (migliaia, intero). In ogni città si può installare un'antenna della rete di telefonia mobile alla sola condizione che le città adiacenti (**precedente e successiva, se esistono**) non abbiano l'antenna. Ogni antenna copre solo la popolazione della città dove è posta.

							
antenne possibili	1	2	3	4	5	6	7
città	1	2	3	4	5	6	7
abitanti	14	22	13	25	30	11	90

Con il paradigma della programmazione dinamica bottom-up, determinare il massimo numero di abitanti copribile rispettando la regola di installazione e la corrispondente disposizione delle antenne. Visti i vincoli, è evidente che non si potrà coprire tutta la popolazione di tutte le città

Svolgimento:

si tratta di un problema di ottimizzazione che potrebbe essere risolto identificando tutti i sottoinsiemi di antenne, valutando quelli che soddisfano la regola e, tra questi, quello ottimo. Il modello è quello del powerset.

In alternativa si propone una soluzione basata sul paradigma della **programmazione dinamica**. I dati sono memorizzati in un vettore di interi `val` di $n+1$ celle. La cella di indice 0 corrisponde alla città fittizia che non esiste e che non ha abitanti.

val	0	14	22	13	25	30	11	90
	0	1	2	3	4	5	6	7

Passo 1: applicabilità della programmazione dinamica

Ci si posiziona sulla città di indice k e si guarda all'indietro. Si osservi che è vera la seguente affermazione: la soluzione ottima del problema per la città di indice k corrisponde a uno dei seguenti 2 casi

- nella città k non c'è un'antenna: la soluzione ottima coincide con quella per le prime $k-1$ città
- nella città k c'è un'antenna: la soluzione ottima si ottiene dalla soluzione ottima per le prime $k-2$ città cui si aggiunge l'antenna nella città k .

Supponiamo di memorizzare in un vettore di interi `opt` di $n+1$ celle scandito da un indice k la soluzione ottima che si ottiene considerando le prime k antenne: `opt[0]=0` in quanto non ci sono né città, né abitanti, né antenne; `opt[1]=val[1]` in quanto c'è solamente la città di indice $i=1$ con i suoi abitanti `val[1]`. Per gli altri casi $1 < k \leq n$:



- è possibile piazzare un'antenna nella città k , quindi non è può essere piazzata un'antenna nella città $k-1$ che la precede, bensì nella città ancora prima $k-2$: $opt[k] = opt[k-2] + val[k]$
- non è possibile piazzare un'antenna nella città k , quindi è possibile piazzare un'antenna nella città $k-1$ che la precede: $opt[k] = opt[k-1]$.

Il problema per la città k -esima richiede la soluzione dei sottoproblemi per le città $(k-1)$ -esima o $(k-2)$ -esima. Se $opt[k-1]$ o $opt[k-2]$ non fossero massimi, si potrebbero trovare soluzioni $opt'[k-1] > opt[k-1]$ o $opt'[k-2] > opt[k-2]$ che contraddirebbero l'ipotesi di $opt[k]$ massimo. La programmazione dinamica è quindi applicabile.

Passo 2: soluzione ricorsiva

L'analisi precedente può essere riassunta con la seguente formulazione ricorsiva:

$$opt(k) = \begin{cases} 0 & k = 0 \\ val[1] & k = 1 \\ \max(opt(k-1), val(k) + opt(k-2)) & 1 < k \leq n \end{cases}$$

facilmente codificata in C come:

```
int solveR(int *val, int *opt, int n, int k) {
    if (k==0)
        return 0;
    if (k==1)
        return val[1];
    return max(solveR(val, opt, n, k-1), solveR(val, opt, n, k-2) + val[k]);
}

void solve(int *val, int n) {
    int *opt;
    opt = calloc((n+1), sizeof(int));
    printf("Recursive solution: ");
    printf("maximum population covered %d\n", solveR(val, opt, n, n));
}
```

Questa soluzione porta alla seguente equazione alle ricorrenze:

$$T(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ T(n-1) + T(n-2) + 1 & n > 1 \end{cases}$$

identica a quella vista lezione per i numeri di Fibonacci, dunque la soluzione ricorsiva ha complessità esponenziale.

Passo 3: soluzione con programmazione dinamica bottom-up (calcolo del valore della soluzione ottima)

Ispirandosi alla formulazione ricorsiva della soluzione, la si trasforma in forma iterativa:

- $opt[0]$ e $opt[1]$ sono noti a priori,
- per $2 \leq i \leq n$ $opt[i] = \max(opt[i-1], opt[i-2] + val[i])$.



```
void solveDP(int *val, int n) {
    int i, *opt;
    opt = calloc((n+1), sizeof(int));
    opt[1] = val[1];
    for (i=2; i<=n; i++) {
        if (opt[i-1] > opt[i-2]+val[i])
            opt[i] = opt[i-1];
        else
            opt[i] = opt[i-2] + val[i];
    }
    printf("Dynamic programming solution: ");
    printf("maximum population covered %d\n", opt[n]);
    displaySol(opt, val, n);
}
```

Passo 4: costruzione della soluzione ottima

La funzione `displaySol` costruisce e visualizza la soluzione (città dove si installa un'antenna). Essa utilizza un vettore di interi `sol` di $n+1$ elementi per registrare se l'elemento i -esimo appartiene o meno alla soluzione. La decisione viene presa in base al contenuto del vettore `opt` e viene costruita mediante una scansione da destra verso sinistra, in verso quindi opposto alla scansione con cui `opt` è stato riempito dalla funzione `solveDP`. Il criterio per assegnare 0 o 1 alla cella corrente di `sol` rispecchia quello usato in fase di risoluzione:

- `sol[1]` è assunto valere 1, salvo modificare questa scelta nel corso dell'iterazione successiva
- se `opt[i]==opt[i-1]` è certo che nella città i -esima non è stata piazzata un'antenna, quindi `sol[i]=0`, mentre non si può dire nulla di `sol[i-1]`. L'iterazione quindi prosegue sulla città $(i-1)$ -esima
- se `opt[i] == opt[i-2] + val[i]` è certo che:
 - nella città i -esima è stata piazzata un'antenna, quindi `sol[i]=1`
 - nella città $(i-1)$ -esima non è stata piazzata un'antenna, quindi `sol[i-1]=0`avendo preso una decisione sia per la città i -esima che per la città $(i-1)$ -esima, l'iterazione prosegue sulla città $(i-2)$ -esima.

```
void displaySol(int *opt, int *val, int n){
    int i, j, *sol;
    sol = calloc((n+1), sizeof(int));
    sol[1]=1;
    i=n;
    while (i>=2) {
        printf("i=%d\n", i);
        if (opt[i] == opt[i-1]){
            sol[i] = 0;
            i--;
        }
        else if (opt[i] == opt[i-2] + val[i]) {
            sol[i] = 1;
            sol[i-1] = 0;
            i -=2;
        }
    }
}
```



**POLITECNICO
DI TORINO**

03MNO ALGORITMI E PROGRAMMAZIONE CORSO DI LAUREA IN INGEGNERIA INFORMATICA A.A. 2019/20

```
for (i=1; i<=n; i++)  
    if (sol[i])  
        printf("%d ", val[i]);  
printf("\n");  
}
```