La tecnica **Greedy** è utile per problemi dove occorre trovare il valore più basso o più alto, il migliore per una certa grandezza. Per risolverli è necessario utilizzare:

- il ***sort*** (se le grandezze sono statiche e un ordinamento iniziale è sufficiente)
- altrimenti le **code con priorità** che consentono di considerare ogni volta l'elemento migliore

esempio: **problema di sequenziamento di lavori**

Non sempre l'ottimo locale risulta l'ottimo assoluto per il problema (es: **problema di cambio di denaro**)

p.31 rappresentazione in pseudocodice degli algoritmi Greedy

inizialmente S (soluzioni) è un insieme vuoto, invece C (candidati) è pieno (diverso da vuoto)

while (S non è completo AND C diverso da vuoto)

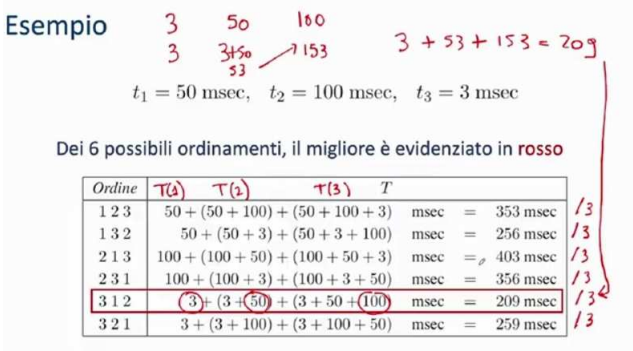
considerare un candidato

ponendolo in X e togliendolo dall'insieme C

se l'insieme delle Soluzioni S unito al candidato X è ammissibile

allora aggiunere il candidato X all'insieme S delle Soluzioni

p.34 **problema di sequenziamento lavori**



può risultare costoso generare tutte le pemutazioni possibili degli N lavori e non è necessario cercarle

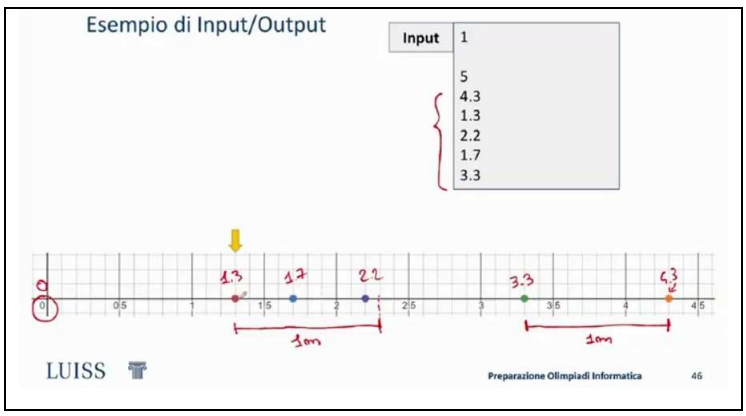
esempio: 3 lavori da 50, 100 e 3 msec

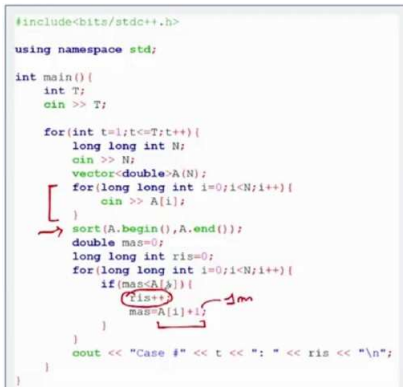
in questo caso con un ***sort*** iniziale la cui complessità è solo O(n log n) la somma viene minimizzata ad ogni passo

p.41 **problema Irrigazioni**

viene naturale considerare l'ordine crescente delle distanze delle piante dall'origine

e utilizzare i singoli tubi da 1 metro per irrigare le piante vicine a partire da quella più vicina ad O



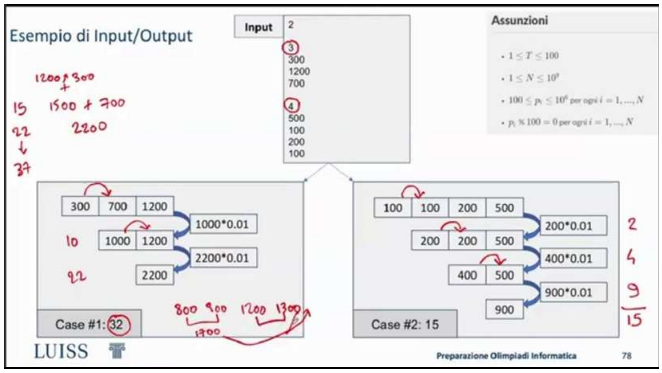


```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int T;
    cin >> T;
    for(int t=1;t<=T;t++){
        long long int N;
        cin >> N;
        vector<double>A(N);
        for(long long int i=0;i<N;i++){
            cin >> A[i];
        }
        sort(A.begin(),A.end());
        double mas=0;
        long long int ris=0;
        for(long long int i=0;i<N;i++){
            if(mas<A[i]){
                ris++;
                mas=A[i]+1;
            }
        }
        cout << "Case #" << t << ": " << ris << "\n";
    }
}
```

p. 48 problema **turni di guardia**

la prof. segnala che il doppio ciclo della soluzione proposta potrebbe non essere efficiente

p. 68 problema **Bonifici**



Cambia lo scenario ma questo problema si riconduce ad una soluzione simile a quella vista per somme costose.

Conviene usare una **coda con priorità** che però estrae (e toglie) sempre il massimo valore.

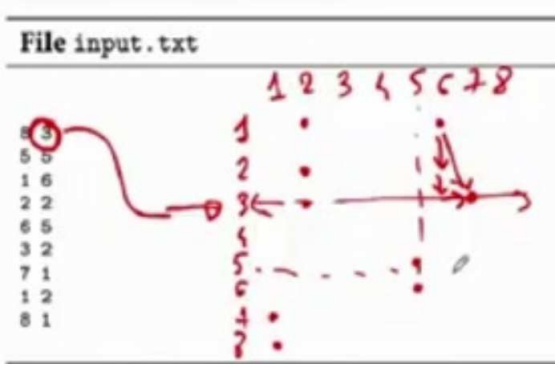
Per ottenere i valori in ordine crescente, basta renderli tutti negativi prima di caricarli nella **coda con priorità**

- 1200 < -700 < -300 quindi estrarrà prima -300

La struttura **coda con priorità** gestisce gli ordinamenti in modo economico e trasparente per l'utente ed è ottima per problemi che richiedono ordinamenti continui dei dati che cambiano dinamicamente.

p. 72 problema **Sbarramento tattico**

le armate devono essere distribuite su colonne diverse della riga r (es: r=3) con il minimo di spostamenti



Va solo calcolato il costo minimo per spostare tutte le armate alla riga r, non occorre FARE gli spostamenti, ma solo immaginare che esista una sequenza di passi per spostare le armate;

occorre solo calcolare le distanze; considerare le armate in un certo ordine e spostare una armata nella colonna vicina libera; quante armate ci sono sulla stessa colonna?

bisogna spostare una armata trovando la prima colonna libera

soluzione della professoressa

```c
/* Innanzitutto calcola il costo per spostare tutti
nella riga R, poi aggiunge (greedy) il costo per
spostare l'i-esima truppa dal basso in posizione i */

#include <stdio.h>

#define MAXN 512

int Count[MAXN];
int N, R;

int main() {
    FILE *fin, *fout;
    int i, j, Row, Col, SOL;

    fin = fopen("input.txt", "r");
    fout = fopen("output.txt", "w");

    R--;
    SOL = 0;
    for (i = 0; i < N; i++) {
        Row--, Col--;
        SOL += abs( Row - R );
        Count[Col]++;
    }
    for (i = j = 0; i < N && j < N; ) {
        if (Count[i] == 0) i++;
        else {
            SOL += abs( i - j );
            Count[i]--;
            j++;
        }
    }
    fprintf(fout, "%d\n", SOL);
    return 0;
}
```