

Algoritmi di ordinamento - 2 -

Mergesort, divide et impera

Mergesort

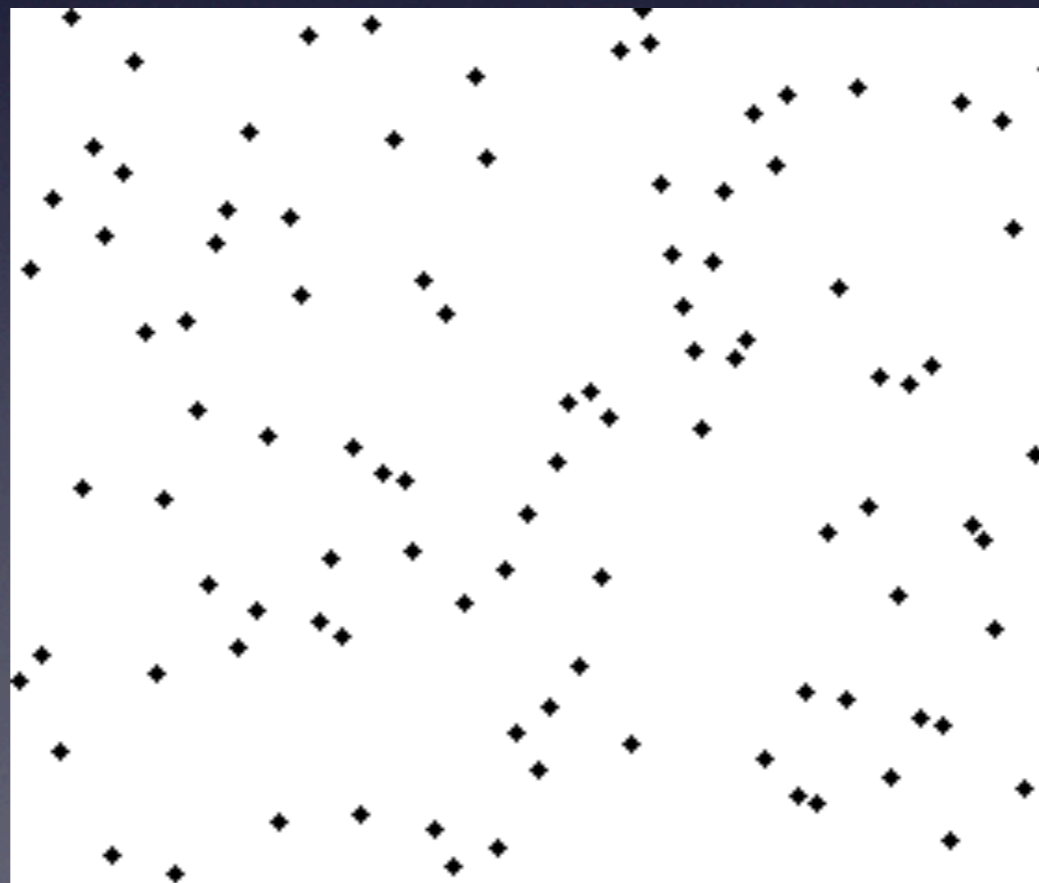
- Inventato da von Neumann nel 1945
- Esempio del paradigma algoritmico del *divide et impera*
- Richiede spazio ausiliario ($O(N)$)
 - Heapsort ha stessa complessità computazionale ma è $O(1)$ per lo spazio ausiliario
- E' implementato come algoritmo standard nelle librerie di alcuni linguaggi (Perl, Java)
- E' facile implementare una versione stabile

Mergesort

- Si divide il vettore dei dati in due parti ordinate separatamente, quindi si fondono le parti per ottenere un vettore ordinato globalmente
- il problema è la fusione

Mergesort

- Si divide il vettore dei dati in due parti ordinate separatamente, quindi si fondono le parti per ottenere un vettore ordinato globalmente
- il problema è la fusione



Algoritmo: fusione

- Si deve ordinare un vettore partizionato in due metà ordinate al loro interno
- il semi-vettore sinistro è copiato su un vettore appoggio
- si seleziona il minimo tra appoggio e semi-vettore destro e si copia sul vettore complessivo
- si termina quando tutti gli elementi dell'appoggio sono stati copiati

Fusione: complessità

- I semi-vettori sono ordinati: selezione e copia sono a costo costante
- E' $O(N)$: ad ogni selezione il vettore complessivo cresce di 1
- c'è un costo iniziale di copia del vettore appoggio ($O(N)$)

Fusione: implementazione

- Si usano due indici: l e r , che indicano il numero di elementi copiati dal vett. appoggio e dal semi-vettore destro
- si copia il minimo dei due vettori in $l+r$ nel vettore complessivo
- se r raggiunge il massimo prima di l allora si continua copiando gli elementi ancora da selezionare dall'appoggio
- se l raggiunge il massimo prima di r allora si può finire, gli elementi del semi-vettore destro sono già nella posizione giusta

```

void merge(struct data *V, int N, int N1)
{
    int l,r;
    struct data *tmp;
    int count;

    tmp=(struct data *)malloc(N1*sizeof(struct data));
    for (count=0; count<N1; count++)
        tmp[count]=V[count];

    l=r=0; // notare l'uso dell'invariante
    while( l<N1 && r<(N-N1) ) {
        if( tmp[l]<V[N1+r] ) {
            V[l+r]=tmp[l];
            l++;
        } else {
            V[l+r]=V[N1+r];
            r++;
        }
    }
    while ( l<N1 ) {
        V[l+r]=tmp[l];
        l++;
    }
    free(tmp);
}

```


Mergesort: costo

- $\Gamma_{MergeS}(N) = c_1 + 2 \cdot \Gamma_{MergeS}(N/2) + c_2 \cdot N$ se $N > 2$,
 c_3 se $N = 2$
- $c_2 \cdot N$: costo di merge
- c_1 : costo per decidere se proseguire nel partizionamento

Divide et impera: costo

- $C_N = 2 \cdot C_{N/2} + N$ per $N \geq 2$.
considerando per semplicità $N=2^n$
- $C_{2^n} = 2 \cdot C_{2^{n-1}} + 2^n$
- $\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1$
- $\quad = \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1$
- ...
- $\quad = n$
- $\quad = 2^n \cdot n = N \cdot \ln_2 N$

Mergesort: implementazione

```
void _mergesort(struct data *V, int N, struct data *tmp)
{
    if ( N>2 ) {
        _mergesort(V, N/2, tmp);
        _mergesort(&V[N/2], N-N/2, &tmp[N/2]);
        merge(V, N, N/2);
    } else {
        if ( V[0]<V[1] )
            swap( V, 0, 1 );
    }
}
```

```
void mergesort(struct data *V, int N)
{
    struct data *tmp;
    tmp=(struct data *)malloc(N*sizeof(struct data));
    _mergesort(V, N, tmp);
    free(tmp);
}
```

Esempio

