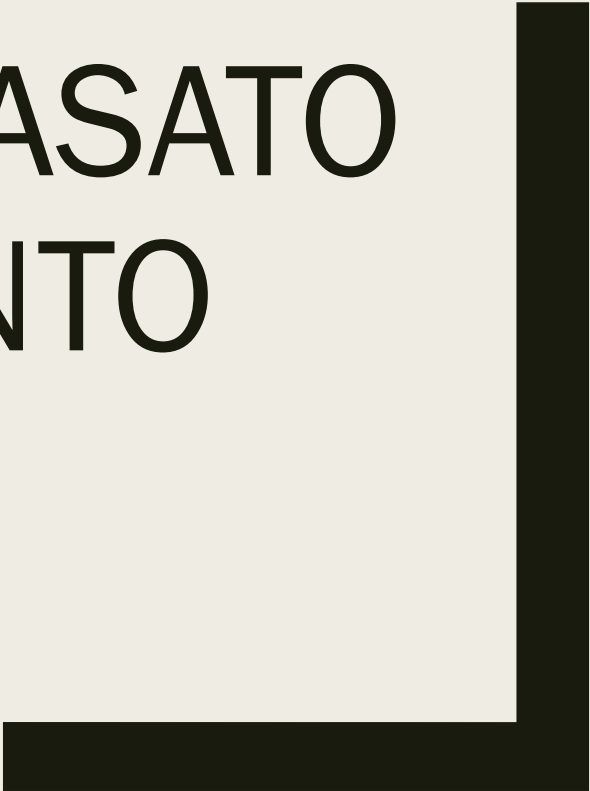




SORTING NON BASATO SUL CONFRONTO

ASD 2016-17
Fabrizio d'Amore



promemoria (sorting basato sul confronto)

- Lower bound $\Omega(n \log n)$
- Upper bound $O(n \log n)$
- Diversi algoritmi ottimali
 - *non tutti in-place*
- alcuni sub-ottimali, ma di buone prestazioni in pratica

<i>sort</i>	<i>w,c,</i>	<i>avg</i>	<i>b.c.</i>	<i>in-place?</i>
merge	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
heap	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	sì
quick	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no (ric.)
insertion	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	sì
selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	sì
bubble	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	sì

sorting non basato sul confronto



- si tratta di algoritmi che per raggiungere l'obiettivo dell'ordinamento non confrontano fra loro i valori da ordinare
 - *ce ne sono di varia tipologia*
- lower bound utili in pratica non sono noti
- in condizioni "speciali" hanno prestazioni estremamente buone
- spesso (ma non necessariamente) sostituiscono ai confronti l'impiego di strutture dati accessorie, finendo con il lavorare non in-place
- vedremo: counting, bucket, radix

counting sort

- pensato per ordinare valori interi, riconducibili a numeri interi
- conta le molteplicità dei valori in input
 - *memorizzandole su array di appoggio*
- costruisce l'output a partire da tali molteplicità
- conveniente quando i valori da ordinare appartengono a un insieme di cardinalità contenuta
- costo in presenza di HP semplificative aggiuntive: $\Theta(n)$
 - *altrimenti costo esponenziale: dominato da max-min (valori in input)*

```
countingSort(int* a, int n) {  
    int max=0, min=0;  
    for(int i = 1; i < n; i++)  
        if(a[i] > a[max]) max = i;  
        else if(a[i] < min) min = i;  
    int *tmp = malloc(max-min+1);  
    for(int i = 1; i < n; i++)  
        tmp[a[i]-min]++;  
    int h = 0;  
    for(int i = 0; i < max-min; i++)  
        while(tmp[i]--) a[h++] = i+min;  
}
```

bucket sort

- non dissimile dal counting sort, lo generalizza
- per semplicità di descrizione si assume che l'input sia costituito da reali in $[0,1]$
- il bucket sort ha un buon comportamento quando l'input è "ben distribuito"
- suddivide l'insieme dei possibili valori in input in n intervalli di eguale dimensione (buckets) e quindi distribuisce gli n valori in input fra i buckets
 - *se input è ben distribuito ci aspettiamo che siano pochi i valori assegnati allo stesso bucket*
 - *usiamo un array B di n puntatori ai diversi bucket*
- ordina (ricorsivamente?) ciascuno dei bucket e quindi visita ordinatamente i bucket per ottenere l'input ordinato
- costo atteso se input ben distribuito: $\Theta(n)$
 - *costo atteso ancora lineare se la somma dei quadrati del numero di elementi in ogni bucket è $\Theta(n)$*
 - *altrimenti costo può salire (dominato dal costo di sort)*

```
BUCKET-SORT(array A, int n)
  for(i = 1; i <= n; i++)
    insert A[i] nella lista B[floor(n*A[i])]
  for(i = 0; i <= n-1; i++)
    sort B[i] // come???
  concatenate B[0] ... B[n-1]
```

radix sort

- considera gli elementi da ordinare come composti da d cifre binarie
- ordina rispetto la i -esima cifra
 - $i = 1, 2, \dots, d$
- per ordinare rispetto la cifra i può usare il passo di partizionamento impiegato dal quick-sort
 - *o altro algoritmo stabile*
- costo $\Theta(dn)$

```
RADIX-SORT(array A, int d)
  for  $i \leftarrow 1$  to  $d$ 
    do use a stable sort to sort array A on digit  $i$ 
```

```
// es. Java semplificato
static void rs(int[] a, int start, int endp1, int mask) {
  if((endp1 - start <= 1) || (mask == 0)) return;
  int j = start, k = endp1;
  while(j < k) {
    while((j < k) && ((a[j] & mask) == 0x0)) j++;
    while((j < k) && ((a[k-1] & mask) != 0x0)) k--;
    if(j < k) swap(a, j, k-1);
  }
  rs(a, start, j, mask >>> 1);
  rs(a, j, endp1, mask >>> 1);
}
```