Ordinamento

Moreno Marzolla
Dip. di Scienze dell'Informazione
Università di Bologna

marzolla@cs.unibo.it
http://www.moreno.marzolla.name/

Ordinamento

- Consideriamo un array di n numeri v[1], v[2], ... v[n]
- Vogliamo trovare (indirettamente) una permutazione
  p[1], p[2], ... p[n]
  degli interi 1, ..., n tale che
  v[p[1]] ≤ v[p[2]] ≤ ... ≤ v[p[n]]
- Esempio:
  - v = [7, 32, 88, 21, 92, -4]
  - p = [6, 1, 4, 2, 3, 5]
  - v[p[]] = [-4, 7, 21, 32, 88, 92]

Ordinamento

- Più in generale: è dato un array di n elementi, tali che ciascun elemento sia composto da:
  - una **chiave**, in cui le chiavi sono confrontabili tra loro
  - un **contenuto** arbitrario
- Vogliamo permutare l'array in modo che le chiavi compaiano in ordine non decrescente (oppure non crescente)

Definizioni

- Ordinamento in loco
  - L'algoritmo permuta gli elementi direttamente nell'array originale, senza usare un altro array di appoggio
- Ordinamento stabile
  - L'algoritmo preserva l'ordine con cui elementi con la stessa chiave compaiono nell'array originale

Anno nascita	Nome
1974	Mario Rossi
1953	Luigi Bianchi
1974	Antonio Verdi
1953	Fabio Neri

NON stabile (le frecce si incrociano)

Anno nascita	Nome
1953	Luigi Bianchi
1953	Fabio Neri
1974	Antonio Verdi
1974	Mario Rossi

Nota

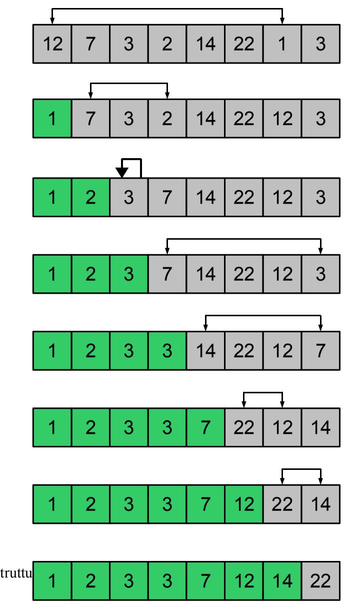
- È possibile rendere ogni algoritmo stabile:
  - Basta usare come chiave di ordinamento la coppia (chiave, posizione nell'array non ordinato)
  - (k1, p1) < (k2, p2) se e solo se:
    - (k1 < k2), oppure
    - (k1 == k2) and (p1 < p2)

Algoritmi di ordinamento "incrementali"

- Partendo da un prefisso A[1..k] ordinato, "estendono" la parte ordinata di un elemento: A[1..k+1]
- Selection sort
  - Cerca il minimo in A[k+1..n] e spostalo in posizione k+1
- Insertion sort
  - Inserisce l'elemento A[k+1] nella posizione corretta all'interno del prefisso già ordinato A[1..k]

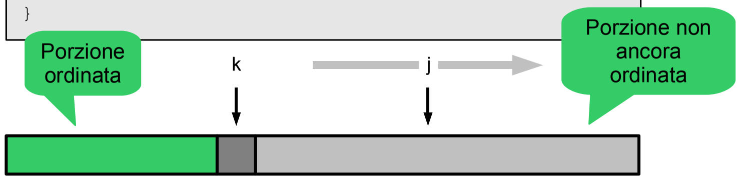
Selection Sort

- Cerco il minimo in A[1]...A[n] e lo scambio con A[1]
- Cerco il minimo in A[2]...A[n] e lo scambio con A[2]
- ...
- Cerco il minimo in A[k]...A[n] e lo scambio con A[k]
- ...



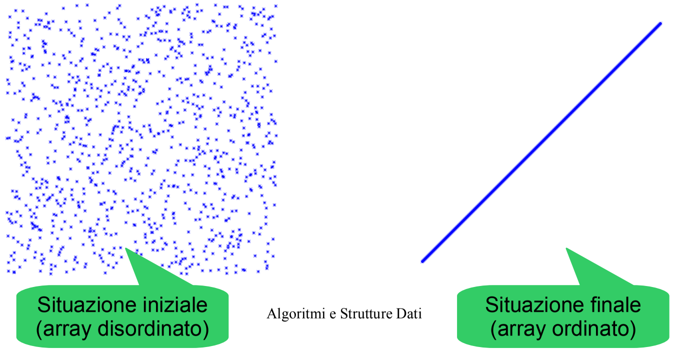
Selection Sort

```
public static void selectionSort(Comparable A[]) {
    for (int k = 0; k < A.length - 1; k++) {
        // cerca il minimo A[m] in A[k..n-1]
        int m = k;
        for (int j = k + 1; j < A.length; j++)
            if (A[j].compareTo(A[m]) < 0)
                m = j;
        // scambia A[k] con A[m]
        if (m != k) {
            Comparable temp = A[m];
            A[m] = A[k];
            A[k] = temp;
        }
    }
}
```

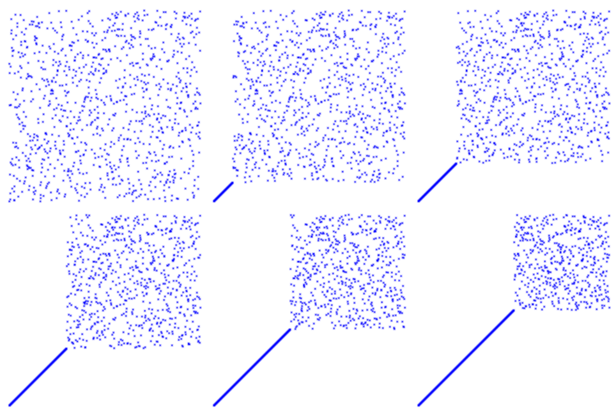


“Visualizzare” il comportamento di un algoritmo di ordinamento

- Consideriamo un vettore A[] contenente tutti e soli gli interi da 1 a N
- Plottiamo i punti di coordinate (i, A[i])



Selection Sort per immagini



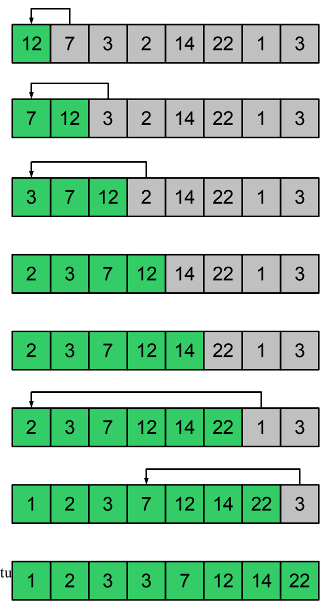
Complessità di Selection Sort

- L'estrazione del k-esimo minimo richiede *(n-k-1)* confronti (k=0,1, … n-2)
- Il costo complessivo è quindi

$$\sum_{k=0}^{n-2}(n-k-1)=\sum_{k=1}^{n-1}k=\Theta(n^2)$$

Insertion Sort

- Idea: al termine del passo k, il vettore ha le prime k componenti ordinate
- Inserisco l'elemento di posizione k+1 nella posizione corretta all'interno dei primi k elementi ordinati



Insertion Sort

```
public static void insertionSort(Comparable A[]) {
    for (int k = 1; k <= A.length - 1; k++) {
        int j;
        Comparable x = A[k];
        // cerca la posizione j in cui inserire A[k]
        for (j = 0; j < k; j++)
            if (A[j].compareTo(x) > 0) break;
        if (j < k) {
            // Sposta A[j..k-1] in A[j+1..k]
            for (int t = k; t > j; t--)
                A[t] = A[t - 1];
            // Inserisci A[k] in posizione j
            A[j] = x;
        }
    }
}
```

Domanda: questa implementazione fornisce un ordinamento stabile?

Insertion Sort

- L'inserimento del k+1-esimo elemento nella posizione corretta rispetto ai primi *k* richiede *k+1* confronti nel caso peggiore
- Il numero di confronti nel caso peggiore risulta essere quindi

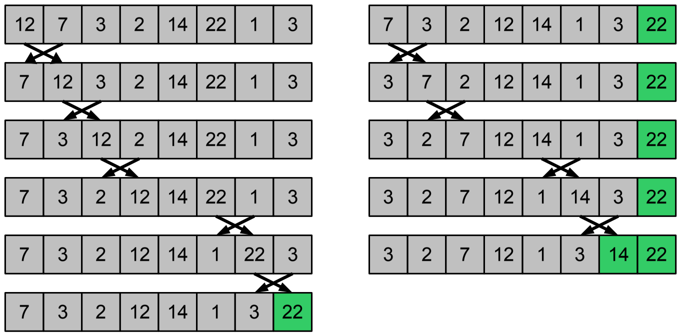
$$\sum_{k=1}^{n-1}(k+1)=\left(\sum_{k=1}^{n-1}k\right)+(n-1)=\Theta(n^2)$$

- Domanda: quante operazioni elementari vengono svolte dall'algoritmo insertion sort nel caso peggiore? e nel caso migliore?

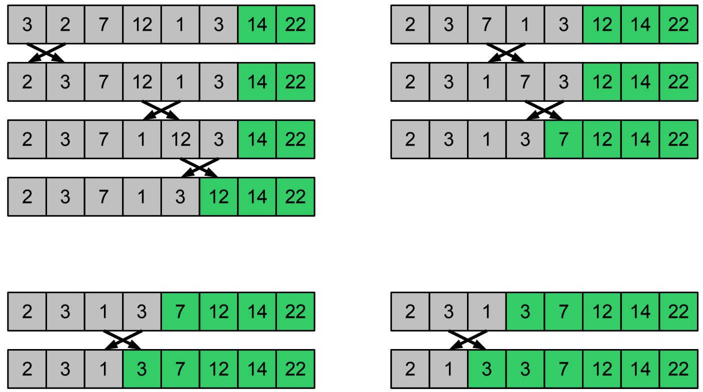
Bubble Sort

- Esegue una serie di scansioni dell'array
  - Ad ogni scansione scambia le coppie di elementi adiacenti che non sono nell'ordine corretto
  - Se al termine di una scansione non è stato effettuato nessuno scambio, l'array è ordinato
- Dopo la prima scansione, l'elemento massimo occupa l'ultima posizione
- Dopo la seconda scansione, il "secondo massimo" occupa la penultima posizione...
- ...dopo la k-esima scansione, i *k* elementi massimi occupano la posizione corretta in fondo all'array

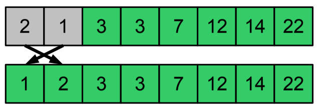
Bubble Sort



Bubble Sort



Bubble Sort



Bubble Sort

```java
public static void bubbleSort(Comparable A[]) {
    for (int i = 1; i < A.length; i++) {
        boolean scambiAvvenuti = false;
        for (int j = 1; j <= A.length - i; j++) {
            // Se A[j-1] > A[j], scambiali
            if (A[j - 1].compareTo(A[j]) > 0) {
                Comparable temp = A[j - 1];
                A[j - 1] = A[j];
                A[j] = temp;
                scambiAvvenuti = true;
            }
        }
        if (!scambiAvvenuti) break;
    }
}
```

Bubble Sort
Invariante di ciclo

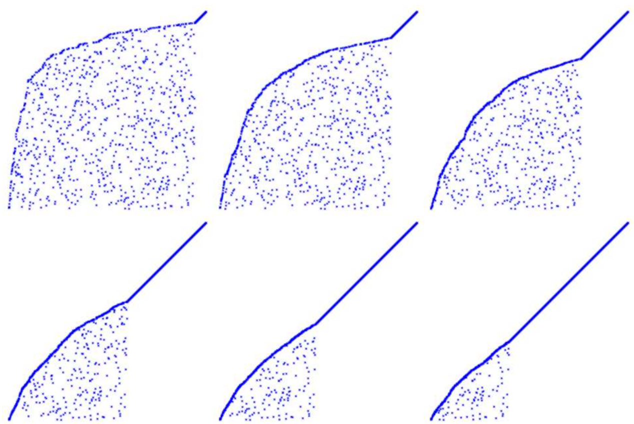
- Dopo l'i-esima iterazione, gli elementi A[n-i]... A[n-1] sono correttamente ordinati e occupano la loro posizione definitiva nell'array ordinato

```
public static void bubbleSort(Comparable A[]) {
    for (int i = 1; i < A.length; i++) {
        boolean scambiAvvenuti = false;
        for (int j = 1; j <= A.length - i; j++) {
            if (A[j - 1].compareTo(A[j]) > 0) {
                Comparable temp = A[j - 1];
                A[j - 1] = A[j];
                A[j] = temp;
                scambiAvvenuti = true;
            }
        }
        if (!scambiAvvenuti) break;
    }
}
```


Bubble Sort

- L'algoritmo Bubble Sort ha costo $O(n^2)$
  - Nel caso ottimo l'algoritmo ha costo $\Theta(n)$: effettua una sola scansione dell'array senza effettuare scambi
- In generale, l'algoritmo ha un comportamento "quasi naturale", nel senso che il tempo di ordinamento tende ad essere legato al grado di "disordine" dell'array
  - La parola chiave è "tende". Infatti, come si comporta l'algoritmo su questo vettore? [2 3 4 5 6 7 8 9 1]

Bubble Sort per immagini



Si può fare di meglio?

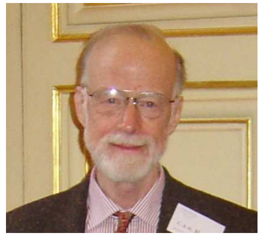
- Gli algoritmi visti fino ad ora hanno costo $O(n^2)$
- È possibile fare di meglio?
  - Quanto meglio?

Algoritmi “divide et impera”

- Idea generale
  - Divide: Scomporre il problema in sottoproblemi dello stesso tipo (cioè sottoproblemi di ordinamento)
  - Risolvere ricorsivamente i sottoproblemi
  - Impera: Combinare le soluzioni parziali per ottenere la soluzione al problema di partenza
- Vedremo due algoritmi di ordinamento di tipo divide et impera
  - Quick Sort
  - Merge Sort

Quick Sort

- Inventato nel 1962 da Sir Charles Anthony Richard Hoare
  - All'epoca *exchange student* presso la Moscow State University
  - Vincitore del *Turing Award* (l'equivalente del Nobel per l'informatica) nel 1980 per il suo contributo nel campo dei linguaggi di programmazione
  - Hoare, C. A. R. *"Quicksort."* Computer Journal 5 (1): 10-15. (1962).



C. A. R. Hoare (1934—)
http://en.wikipedia.org/wiki/C._A._R._Hoare

Quick Sort

- Algoritmo ricorsivo “divide et impera”
  - Scegli un elemento x del vettore v, e partiziona il vettore in due parti considerando gli elementi ≤x e quelli >x
  - Ordina ricorsivamente le due parti
  - Restituisci il risultato concatenando le due parti ordinate
- R. Sedgewick, “*Implementing Quicksort Programs*”, Communications of the ACM, 21(10):847-857, 1978 http://portal.acm.org/citation.cfm?id=359631

Quick Sort

- Input: Array A[1..n], indici i,f tali che $1 \leq i < f \leq n$
- Divide-et-impera
  - Scegli un numero m nell'intervallo [i, i+1, ... f]
  - Divide: permuta l'array A[i..f] in due sottoarray A[i..m-1] e A[m+1..f] (eventualmente vuoti) in modo che:

$$\forall j \in [i \ldots m-1]: A[j] \leq A[m]$$
$$\forall k \in [m+1 \ldots f]: A[m] < A[k]$$

    - A[m] prende il nome di pivot
  - Impera: ordina i due sottoarray A[i..m-1] e A[m+1..f] richiamando ricorsivamente quicksort
  - Combina: non fa nulla; i due sottoarray ordinati e l'elemento A[m] sono già ordinati

Quick Sort

```
public static void quickSort(Comparable A[]) {
    quickSortRec(A, 0, A.length - 1);
}

public static void quickSortRec(Comparable[] A, int i, int f) {
    if (i >= f) return;
    int m = partition(A, i, f);
    quickSortRec(A, i, m - 1);
    quickSortRec(A, m+1, f);
}
```

Ricordarsi che in Java gli array sono indicizzati a partire da 0, non da 1

Quick Sort: partition() Idea di base

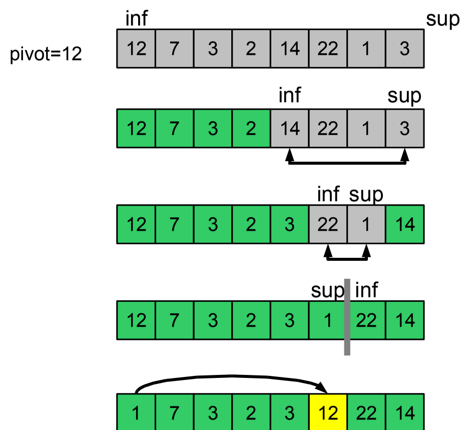
- Manteniamo due indici, inf e sup, che vengono fatti scorrere dalle estremità del vettore verso il centro
  - Il sotto-vettore A[i..inf-1] è composto da elementi ≤ pivot
  - Il sotto-vettore A[sup+1..f] è composto da elementi > pivot
- Quando entrambi (inf e sup) non possono essere fatti avanzare verso il centro, si scambia A[inf] e A[sup]

Quick Sort: partition()

```
private static int partition(Comparable A[], int i, int f) {
    int inf = i, sup = f + 1;
    Comparable temp, x = A[i];
    while (true) {
        do {
            inf++;
        } while (inf <= f && A[inf].compareTo(x) <= 0);
        do {
            sup--;
        } while (A[sup].compareTo(x) > 0);
        if (inf < sup) {
            temp = A[inf];
            A[inf] = A[sup];
            A[sup] = temp;
        } else
            break;
    }
    temp = A[i];
    A[i] = A[sup];
    A[sup] = temp;
    return sup;
}
```

Scelta deterministica del pivot

Esempio di partizionamento

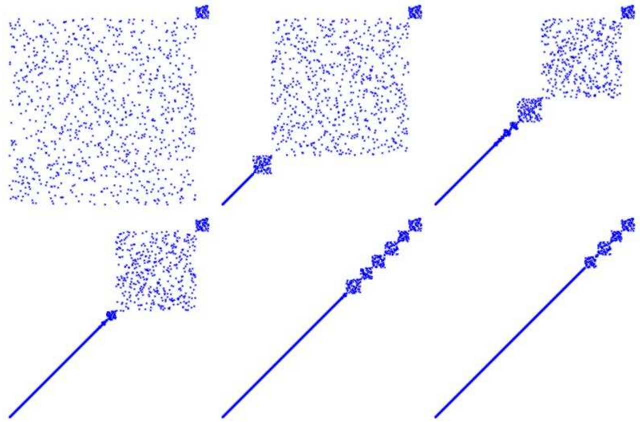


Esercizio
(problema 4.7 p. 116 del libro di testo)

- Il problema della bandiera nazionale. Supponiamo di avere un array A[1..n] di elementi che possono assumere solo tre valori: bianco, verde e rosso. Ordinare l'array in modo che tutti gli elementi verdi siano a sinistra, quelli bianchi al centro e quelli rossi a destra.
- L'algoritmo DEVE richiedere tempo $O(n)$ e memoria aggiuntiva $O(1)$. Può confrontare ed eventualmente scambiare tra loro elementi, e NON DEVE fare uso di ulteriori array di appoggio, né usare contatori per tenere traccia del numero di elementi di un certo colore
- L'algoritmo DEVE richiedere una singola scansione dell'array.

Questo algoritmo verrà utilizzato nell'algoritmo di selezione del k-esimo

Quick Sort per immagini



Quick Sort: Analisi del costo

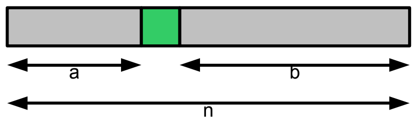
- Costo di partition(): $\Theta(f-i)$
- Costo Quick Sort: Dipende dal partizionamento
- Partizionamento peggiore
  - Dato un problema di dimensione n, viene sempre diviso in due sottoproblemi di dimensione 0 e n-1
  - $T(n) = T(n-1)+T(0)+\Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
- **Domanda**: Quando si verifica il caso pessimo?
- Partizionamento migliore
  - Data un problema di dimensione n, viene sempre diviso in due sottoproblemi di dimensione n/2
  - $T(n) = 2T(n/2)+\Theta(n) = \Theta(n \log n)$ (caso 2 Master Theorem)

QuickSort: Analisi nel caso medio

- In generale, possiamo scrivere la relazione di ricorrenza per T(n)—che esprime il numero di contronti richiesti—come segue:

$$T(n) = T(a) + T(b) + n-1$$

con $(a+b)=(n-1)$



- Il problema è che a e b cambiano (potenzialmente) ad ogni iterazione

QuickSort: Analisi nel caso medio

- Assumendo che tutti i partizionamenti siano equiprobabili, possiamo scrivere:

$$T(n)=\sum_{a=0}^{n-1}\frac{1}{n}\left(n-1+T(a)+T(n-a-1)\right)$$

- Osserviamo che i termini T(a) e T(n-a-1) danno luogo alla stessa sommatoria, da cui possiamo semplificare

$$T(n)=n-1+\frac{2}{n}\sum_{a=0}^{n-1}T(a)$$

QuickSort: Analisi nel caso medio

- Teorema: la relazione di ricorrenza

$$T(n)=n-1+\frac{2}{n}\sum_{a=0}^{n-1}T(a)$$

ha soluzione T(n) ≤ 2n log n

- Dimostrazione: verifichiamo per sostituzione che la soluzione T(n) verifica la relazione T(n) ≤ α n log n
  - Verificheremo che si avrà α=2

QuickSort: Analisi nel caso medio

$$T(n)=n-1+\frac{2}{n}\sum_{i=0}^{n-1}T(i)$$

$$\leq n-1+\frac{2}{n}\sum_{i=0}^{n-1}\alpha\, i\log i$$

$$=n-1+\frac{2\alpha}{n}\sum_{i=2}^{n-1}i\log i$$

$$\leq n-1+\frac{2\alpha}{n}\int_2^n x\log x\,dx$$

continua...

QuickSort: Analisi nel caso medio

$$\int f(x)g'(x)dx=f(x)g(x)-\int f'(x)g(x)dx$$

$$T(n)\leq n-1+\frac{2\alpha}{n}\int_2^n x\log x\,dx$$

$$=n-1+\frac{2\alpha}{n}\left(\frac{n^2\log n}{2}-\frac{n^2}{4}-2\log 2+1\right)$$

$$=n-1+\alpha n\log n-\alpha\frac{n}{2}-O(1)$$

$$\leq \alpha n\log n$$

- L'ultima disuguaglianza vale per α≥2 (e per valori sufficientemente grandi di *n*), da cui la tesi è dimostrata

Quick Sort: Versione randomizzata

- La scelta del pivot nell'operazione partition() è cruciale per evitare che si presenti il caso pessimo
- Abbiamo visto una implementazione in cui il pivot è sempre il primo elemento del (sotto-)vettore
  - In questa situazione è abbastanza facile costruire esempi in cui si verifica il caso pessimo
- Possiamo ridurre la probabilità che si verifichi il caso pessimo mediante randomizzazione
  - Scegliamo in maniera pseudocasuale il pivot tra tutti gli elementi del (sotto-)vettore

Quick Sort: partition()
versione randomizzata

```
private static int partition(Comparable A[], int i, int f) {
    int inf = i, sup = f + 1,
        pos = i + (int) Math.floor((f-i+1) * Math.random());
    Comparable temp, x = A[pos];
    A[pos] = A[i];
    A[i] = x;
    while (true) {
        do {
            inf++;
        } while (inf <= f && A[inf].compareTo(x) <= 0);
        do {
            sup--;
        } while (A[sup].compareTo(x) > 0);
        if (inf < sup) {
            temp = A[inf];
            A[inf] = A[sup];
            A[sup] = temp;
        } else
            break;
    }
    temp = A[i];
    A[i] = A[sup];
    A[sup] = temp;
    return sup;
}
```

Scelta pseudocasuale del pivot

Merge Sort

- Inventato da John von Neumann nel 1945
- Algoritmo *divide et impera*
- Idea:
  - Dividere A[] in due meta' A1[] e A2[] (senza permutare) di dimensioni uguali;
  - Applicare ricorsivamente Merge Sort a A1[] e A2[]
  - Fondere (*merge*) gli array ordinati A1[] e A2[] per ottenere l'array A[] ordinato

John von Neumann (1903—1957)
http://en.wikipedia.org/wiki/John_von_Neumann

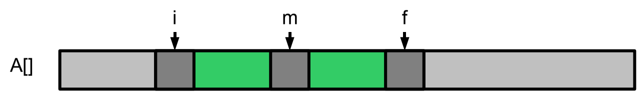
Merge Sort vs Quick Sort

- Quick Sort:
  - partizionamento complesso, merge banale (di fatto nessuna operazione di merge è richiesta)
- Merge Sort:
  - partizionamento banale, operazione merge complessa

Merge Sort

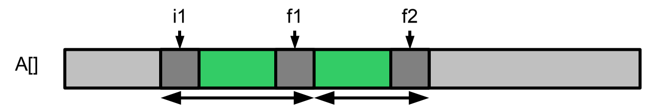
```
public static void mergeSort(Comparable A[]) {
   mergeSortRec(A, 0, A.length - 1);
}

private static void mergeSortRec(Comparable A[], int i, int f) {
   if (i >= f) return;
   int m = (i + f) / 2;
   mergeSortRec(A, i, m);
   mergeSortRec(A, m + 1, f);
   merge(A, i, m, f);
}
```

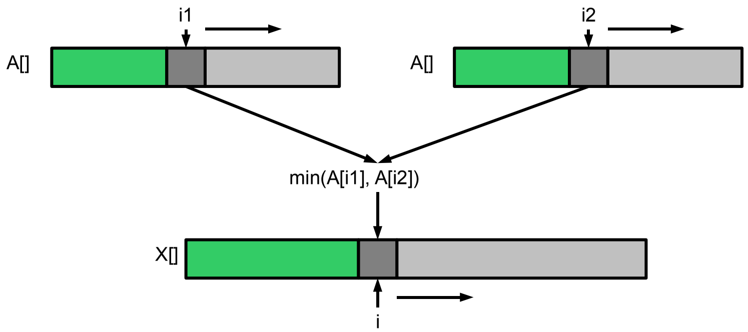


Operazione merge()

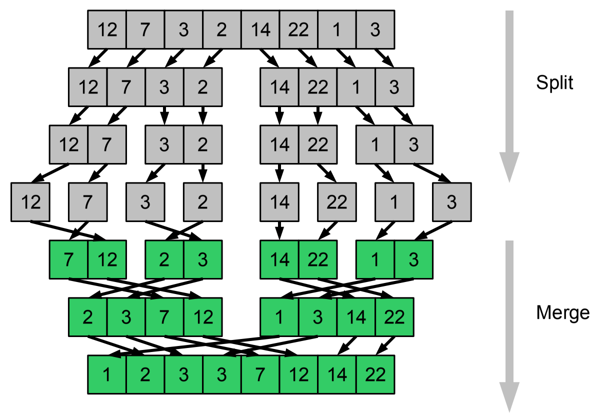
```
private static void merge(Comparable A[], int i1, int f1, int f2)
{
   Comparable[] X = new Comparable[f2 - i1 + 1];
   int i = 0, i2 = f1 + 1, k = i1;
   while (i1 <= f1 && i2 <= f2) {
      if (A[i1].compareTo(A[i2]) < 0)
         X[i++] = A[i1++];
      else
         X[i++] = A[i2++];
   }
   if (i1 <= f1)
      for (int j = i1; j <= f1; j++, i++) X[i] = A[j];
   else
      for (int j = i2; j <= f2; j++, i++) X[i] = A[j];
   for (int t = 0; k <= f2; k++, t++) A[k] = X[t];
}
```



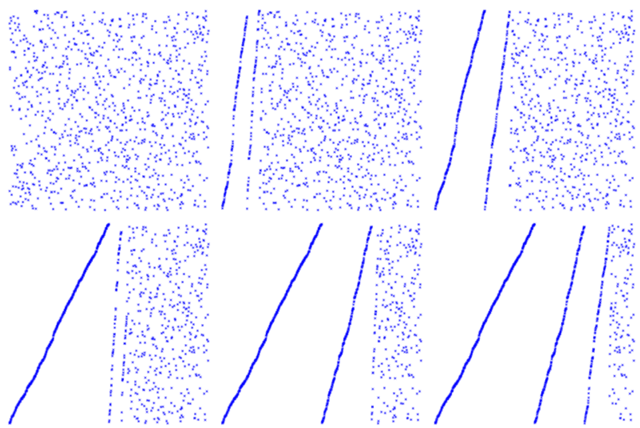
Operazione merge()



Merge Sort: esempio



Merge Sort per immagini



Merge Sort: complessità

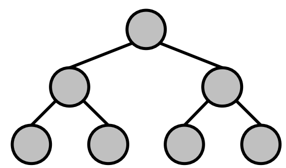
- $T(n) = 2T(n/2) + \Theta(n)$
- In base al Master Theorem (caso 2), si ha
  $T(n) = \Theta(n \log n)$
- La complessità di Merge Sort non dipende dalla configurazione iniziale dell'array da ordinare
  - Quindi il limite di cui sopra vale nei casi ottimo/pessimo/medio
- Svantaggi rispetto a Quick Sort: Merge Sort richiede ulteriore spazio (non ordina in-place)
  - Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola, "*Practical in-place mergesort*", http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8523

Heapsort

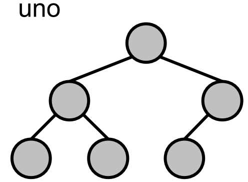
- L'idea
  - Utilizzare una struttura dati—detta heap—per ordinare un array
  - Costo computazionale: O(n log n)
  - Ordinamento sul posto
- Inoltre
  - Il concetto di heap può essere utilizzato per implementare code con priorità

Alberi binari

- Albero binario perfetto
  - Tutte le foglie hanno la stessa altezza h
  - Nodi interni hanno grado 2
- Un albero perfetto
  - Ha altezza $h \approx \log N$
  - $N = \#nodi = 2^{h+1}-1$

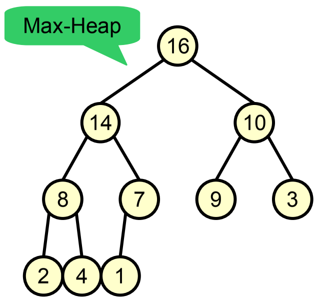


- Albero binario completo
  - Tutte le foglie hanno profondità h o h-1
  - Tutti i nodi a livello h sono "accatastati" a sinistra
  - Tutti i nodi interni hanno grado 2, eccetto al più uno



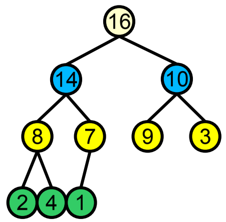
Alberi binari heap

- Un albero binario completo è un albero max-heap sse
  - Ad ogni nodo i viene associato un valore A[i]
  - A[Parent(i)] ≥ A[i]
- Un albero binario completo è un albero min-heap sse
  - Ad ogni nodo i viene associato un valore A[i]
  - A[Parent(i)] ≤ A[i]
- Ovviamente, le definizioni e gli algoritmi di max-heap sono simmetrici rispetto a min-heap



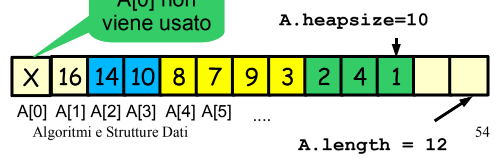
Array heap

- E' possibile rappresentare un albero binario heap tramite un array heap (oltre che tramite puntatori)
- Cosa contiene?
  - Array A, di lunghezza A.length
  - Dimensione A.heapsize ≤ A.length
- Come è organizzato?
  - A[1] contiene la radice
  - Parent(i) = Math.floor(i/2)
  - Left(i) = 2*i
  - Right(i) = 2*i+1



A[0] non viene usato

Domanda: Gli elementi dell'albero heap compaiono nel vettore nello stesso ordine della visita ...



Operazioni su array heap

- findMax(): Individua il valore massimo contenuto in uno uno heap
  - Il massimo è sempre la radice, ossia A[1]
  - L'operazione ha costo O(1)
- fixHeap(): Ripristinare la proprietà di max-heap
  - Supponiamo di rimpiazzare la radice A[1] di un max-heap con un valore qualsiasi
  - Vogliamo fare in modo che A[] diventi nuovamente uno heap
- heapify(): Costruire uno heap a partire da un array privo di alcun ordine
- deleteMax(): rimuovi l'elemento massimo da un max-heap A[]

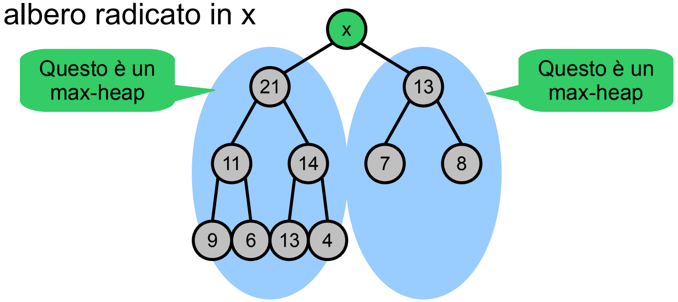
Operazione heapify()

- Parametri:
  - S[] è un array (arbitrario); assumiamo che lo heap abbia n elementi S[1], ... S[n] (S[0] non viene usato)
  - i è l'indice dell'elemento che diventerà la radice dello heap (i≥1)
  - n indica l'indice dell'ultimo elemento dello heap

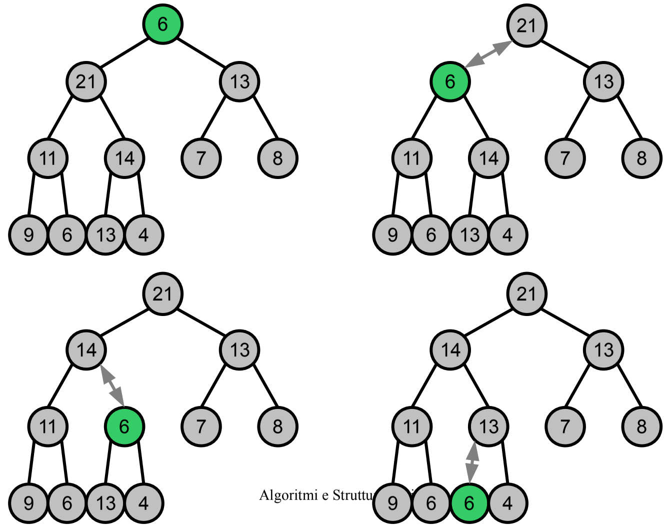
```
private static void heapify(Comparable S[], int n, int i) {
    if (i > n) return;
    heapify(S, n, 2 * i); // crea heap radicato in S[2*i]
    heapify(S, n, 2 * i + 1); // crea heap radicato in S[2*i+1]
    fixHeap(S, n, i);
}
// per trasformare un array S in uno heap:
// heapify(S, S.length, 1 );
```

Operazione fixHeap()

- Supponiamo di avere trasformato in max-heap i sottoalberi destro e sinistro di un nodo x
- L'operazione fixHeap() trasforma in max-heap l'intero albero radicato in x



Operazione fixHeap()



Operazione fixHeap()

- Ripristina la proprietà di ordinamento di uno max-heap rispetto ad un nodo radice di indice i.
- Si confronta ricorsivamente S[i] con il massimo tra i suoi figli e si opera uno scambio ogni volta che la proprietà di ordinamento non è verificata.

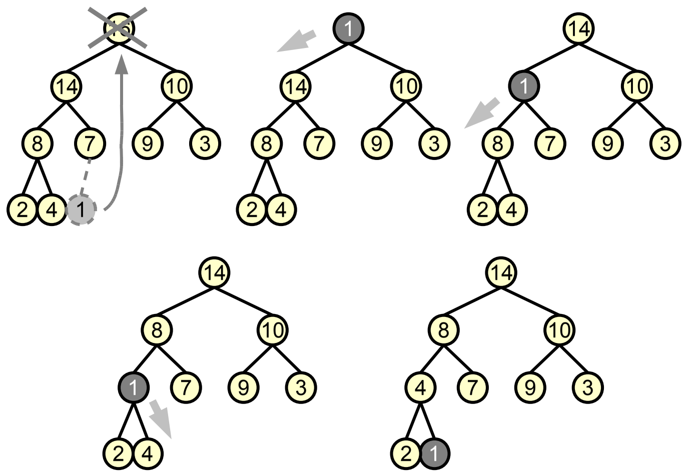
```
private static void fixHeap(Comparable S[], int c, int i) {
    int max = 2 * i; // figlio sinistro
    if (2 * i > c) return;
    if (2 * i + 1 <= c && S[2 * i].compareTo(S[2 * i + 1]) < 0)
        max = 2 * i + 1; // figlio destro
    if (S[i].compareTo(S[max]) < 0) {
        Comparable temp = S[max];
        S[max] = S[i];
        S[i] = temp;
        fixHeap(S, c, max);
    }
}
```

c è l'indice dell'ultimo elemento dello heap

operazione deleteMax()

- Scopo: rimuove la radice (cioè il valore massimo) dallo heap, mantenendo la proprietà di max-heap
- Idea
  - al posto del vecchio valore A[1] metto il valore presente nell'ultima posizione dell'array heap
  - applico fixHeap() per ripristinare la proprietà di heap

Esempio



Costo computazionale

- fixHeap()
  - Nel caso pessimo, il numero di scambi è uguale alla profondità dello heap
  - Cioè O(log n)
- heapify()
  - T(n) = 2T(n/2) + O(log n)
  - da cui T(n) = O(n) (caso (1) del Master Theorem)
- findMax()
  - O(1)
- deleteMax()
  - la stessa di fixHeap(), ossia O(log n)

Heap Sort

- Idea:
  1. Costruire un max-heap a partire dal vettore A[] originale, mediante l'operazione heapify()
  2. Estrarre il massimo (findMax() + deleteMax())
     - Lo heap si contrae di un elemento
  3. Inserire il massimo in ultima posizione di A[]
  4. Ripetere il punto 2. finché lo heap diventa vuoto

Heap Sort

```
public static void heapSort(Comparable S[]) {
  heapify(S, S.length - 1, 1);
  for (int c = (S.length - 1); c > 0; c--) {
    Comparable k = findMax(S);
    deleteMax(S, c);
    S[c] = k;
  }
}
```

O(n)

O(1)

O(log n)

Ricordare che gli elementi da ordinare stanno in S[1], ... S[n]

- Costo computazionale:
  - O(n) per heapify() iniziale
  - Ciascuna iterazione del ciclo 'for' costa O(log c)
- Totale:

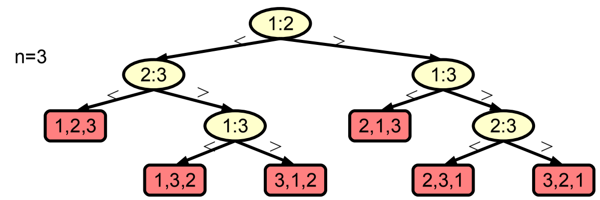
$$T(n)=O(n)+O\left(\sum_{c=n}^{1}\log c\right)=O(n\log n)$$

Algoritmi di ordinamento: sommario

- Abbiamo visto diversi algoritmi di ordinamento:
  - Selection Sort: ottimo/medio/pessimo $\Theta(n^2)$
  - Insertion Sort: ottimo/medio/pessimo $\Theta(n^2)$ (Domanda: come modificarlo per richiederlo $\Theta(n)$ nel caso ottimo?)
  - Merge Sort: ottimo/medio/pessimo $\Theta(n \log n)$
  - Heap Sort: $O(n \log n)$
  - Quick Sort: ottimo/medio $O(n \log n)$, pessimo $O(n^2)$
- Nota:
  - Tutti questi algoritmi sono basati su confronti
    - le decisioni sull'ordinamento vengono prese in base al confronto (<,=,>) fra due valori
- Domanda
  - È possibile fare meglio di $O(n \log n)$?

Limite inferiore alla complessità del problema dell'ordinamento

- Assunzioni
  - Consideriamo un qualunque algoritmo X basato su confronti
  - Assumiamo che tutti i valori siano distinti
- L'algoritmo X
  - può essere rappresentato tramite un albero di decisione, un albero binario che rappresenta i confronti fra gli elementi



Limite inferiore alla complessità del problema dell'ordinamento

- Idea
  - Ogni algoritmo basato su confronti può essere sempre descritto tramite un albero di decisione
  - Ogni albero di decisione può essere interpretato come un algoritmo di ordinamento
- Proprietà
  - Cammino radice-foglia in un albero di decisione: *sequenza di confronti eseguiti dall'algoritmo corrispondente*
  - Altezza dell'albero di decisione: *# confronti eseguiti dall'algoritmo corrispondente nel caso pessimo*
  - Altezza media dell'albero di decisione: *# confronti eseguiti dall'algoritmo corrispondente nel caso medio*

Limite inferiore alla complessità del problema dell'ordinamento

- Lemma 1
  - Un albero di decisione per l'ordinamento di n elementi contiene almeno n! foglie
- Dimostrazione
  - Ogni foglia corrisponde ad una possibile soluzione del problema dell'ordinamento
  - Una soluzione del problema dell'ordinamento consiste in una permutazione dei valori di input
  - Ci sono n! possibili permutazioni

Limite inferiore alla complessità del problema dell'ordinamento

- Lemma 2
  - Sia T un albero binario in cui ogni nodo interno ha esattamente 2 figli e sia k il numero delle sue foglie. L'altezza dell'albero è almeno *$\log_2 k$*
- Dimostrazione (per induzione strutturale)
  - Considriamo un albero con un solo nodo:
    $h(1) = 0 \geq \log_2 1 = 0$
  - Passo induttivo

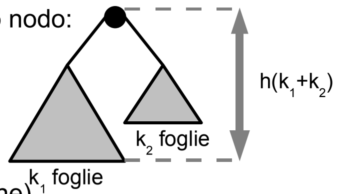
$h(k_1+k_2) = 1 + \max\{ h(k_1), h(k_2) \}$

$\geq 1 + h(k_1)$

$\geq 1 + \log_2 k_1$ (per induzione)

$= \log_2 2 + \log_2 k_1 = \log_2 (2k_1) \geq \log_2 (k_1 + k_2)$

Supponiamo $k_1 > k_2$



Limite inferiore alla complessità del problema dell'ordinamento

- Teorema
  - Il numero di confronti necessari per ordinare n elementi nel caso peggiore è *$\Omega(n \log n)$*
  - **Domanda:** Dimostrazione
  - **Suggerimenti:**
    - Ogni algoritmo basato su confronti richiede tempo proporzionale all'altezza dell'albero di decisione
    - L'albero di decisione ha n! foglie
    - Un albero di decisione con n! foglie ha altezza $\Omega(\log n!)$
    - Utilizzare l'approssimazione di Stirling del fattoriale:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

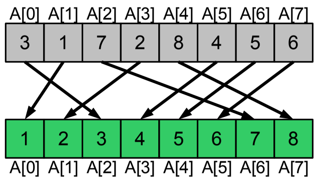
Ordinare in tempo lineare

Tecniche lineari di ordinamento

- Una considerazione
  - Il limite inferiore sull'ordinamento si applica solo agli algoritmi basati su confronti
- Altri approcci
  - Counting Sort
  - Bucket Sort
  - Radix Sort

Counting Sort
caso banale

- Supponiamo di avere un vettore A[0, n-1] di n interi, in cui gli elementi sono tutti e soli i valori da 1 a n. Ogni valore compare esattamente una volta
- Quale è la posizione “corretta” (nell'array ordinato) dell'elemento A[i] nell'array originale?
  - Ovviamente (A[i]-1)



Counting Sort
caso meno banale

- I valori di A[0..n-1] appartengono all'intervallo [0, k-1] (ciascun valore può comparire zero o più volte)
  - Costruisco un array Y[0, k-1]; Y[i] conta il numero di volte in cui il valore i compare in A[]
  - Ricolloco i valori così ottenuti in A

```java
public static void countingSort(int[] A, int k) {
    int[] Y = new int[k];
    int j = 0;
    for (int i = 0; i < k; i++) Y[i] = 0;
    for (int i = 0; i < A.length; i++) Y[A[i]]++;
    for (int i = 0; i < k; i++) {
        while (Y[i] > 0) {
            A[j] = i;
            j++;
            Y[i]--;
        }
    }
}
```


Couting Sort: Costo

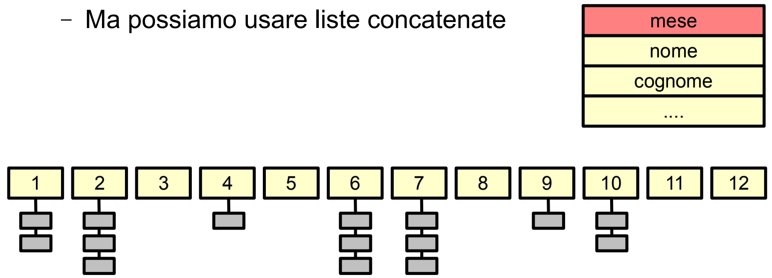
- O(max{n,k}) = O(n+k)
- Se k=Θ(n), allora il costo è O(n)

“Pigeonhole Sort” (Bucket Sort)

Torre colombaia
http://www.prolocosalento.it/allistefelline/main.shtml?A=f_alliste

Bucket Sort

- Bucket Sort
  - Cosa succede se i valori da ordinare non sono numeri interi, ma record associati ad una chiave?
  - Non possiamo usare counting
  - Ma possiamo usare liste concatenate



Bucket Sort

- Ordina n record con chiavi intere in [1,k]

```
Algoritmo bucketSort(array X[1..n], intero k)
   Sia Y un array di dimensione k
   for i := 1 to k do
       Y[i]:=lista vuota
   endfor
   for i := 1 to n do
       Appendi X[i] alla lista Y[chiave(X[i])];
   endfor
   for i := 1 to k do
       copia ordinatamente in X gli elementi di Y[i]
   endfor
```

- Costo: O(n+k)

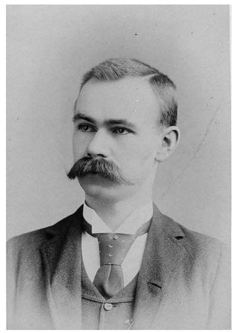
Radix Sort

- Bucket Sort è Interessante, ma a volte il valore k è troppo grande
- Esempio
  - Supponiamo di voler ordinare n numeri con 4 cifre decimali
  - Questo richiederebbe *n+10000* operazion; se *n log n* < *n+10000*, questo non sarebbe conveniente
- Idea
  - Ogni cifra decimale è un candidato ideale per Bucket Sort
  - Se Bucket Sort è stabile, possiamo ordinare a partire dalle cifre meno significative

Radix Sort

- Le origini dell'algoritmo risalgono al 1887 (Herman Hollerith e le macchine tabulatrici)

Ordinatrice di schede IBM 082 (13 slots, ogni scheda ha 12 righe di fori + 1 slot per schede scartate)



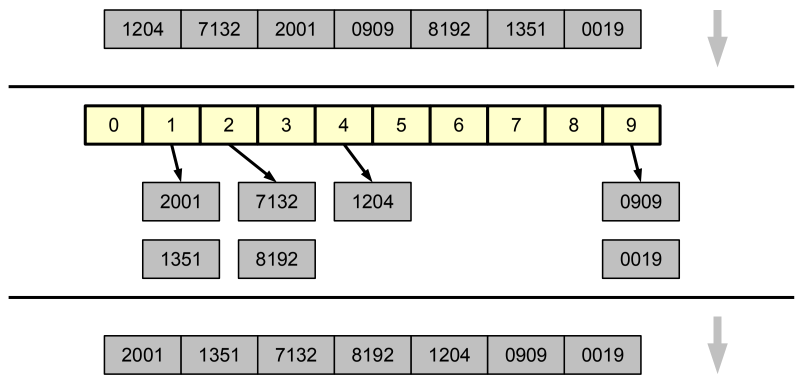
Herman Hollerith (1860—1929)
http://en.wikipedia.org/wiki/Herman_Hollerith

Radix Sort

- Idea:
  - Prima ordino in base alla cifra delle unità
  - Poi ordino in base alla cifra delle decine
  - Poi ordino in base alla cifra delle centinaia
  - ...
- Importante: ad ogni passo è indispensabile usare un algoritmo di ordinamento stabile

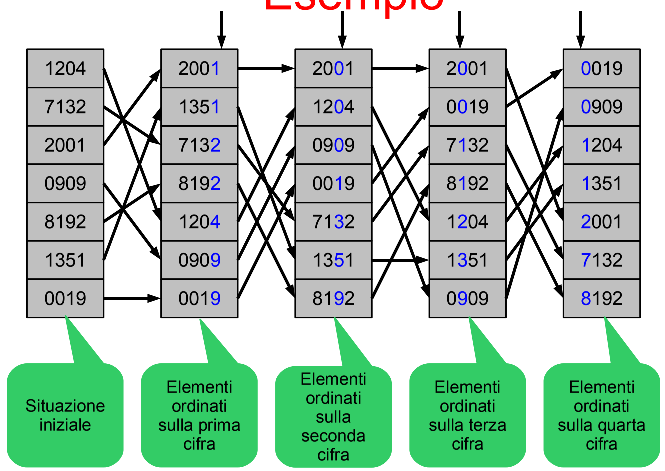
Esempio

Array di partenza



Array ordinato in base alla prima cifra a destra (unità)

Esempio



Radix Sort

- Assume che gli elementi dell'array A abbiano tutti valore nell'intervallo [0, k–1]
- L'ordinamento avviene applicando l'algoritmo Bucket Sort sulle cifre che compongono la rappresentazione in base b degli elementi di A

```
public static void radixSort(int[] A, int k, int b) {
   int t = 0;
   while (t <= Math.ceil(Math.log(k) / Math.log(b))){
      sortByDigit(A, b, t);
      t++;
   }
}
```

Ordinamento (stabile) rispetto alla cifra t (t=0 è quella meno significativa)

Numero di cifre in base b che compongono l'intero k

sortByDigit(A, b, t)

- Una versione specializzata di Bucket Sort per ordinare numeri interi in base alla t-esima cifra (da sinistra) in base b

```java
public static void sortByDigit(int[] A, int b, int t) {
    List[] Y = new List[b];
    int temp, c, j;
    for (int i = 0; i < b; i++) Y[i] = new LinkedList();
    for (int i = 0; i < A.length; i++) {
        temp = A[i] % ((int) (Math.pow(b, t + 1)));
        c = (int) Math.floor(temp / (Math.pow(b, t)));
        Y[c].add(new Integer(A[i]));
    }
    j=0;
    for (int i = 0; i < b; i++) {
        while (Y[i].size() > 0) {
            A[j] = ((Integer) Y[i].get(0)).intValue();
            j++;
        }
    }
}
```


Radix Sort

- Teorema
  - Dati n numeri di d cifre, dove ogni cifra può avere b valori distinti, Radix Sort ordina correttamente i numeri in tempo O(d(n+b))
- Dimostrazione (correttezza):
  - Per induzione: dopo *i* chiamate a sortByDigit, i numeri sono ordinati in base alle prime *i* cifre meno significative.
  - **Domanda**: dimostrare.
- Dimostrazione (complessità):
  - d chiamate a sortByDigit, ogni chiamata ha costo O(n+b)

Radix Sort

- Teorema
  - Usando come base (numero di cifre) un valore $b=\Theta(n)$, l'algoritmo Radix Sort ordina n numeri interi in [0, k-1] in tempo

$$O\left(n\left(1+\frac{\log k}{\log n}\right)\right)$$

  - **Domanda**: Dimostrare
  - Esempio:
    - 1.000.000 di numeri a 32 bit, base $b=2^{16}$, due passate in tempo lineare sono sufficienti
    - Attenzione: memoria aggiuntiva O(b+n)

Ordinamento—Riassunto

Algoritmo	Stabile?	In loco?	Caso Ottimo	Caso Pessimo	Caso Medio
Insertion Sort	Si	Si	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	Si	Si	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	Si	No	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quick Sort	No	Si	$\Theta(n \log n)$	$O(n^2)$	$\Theta(n \log n)$
Heap Sort	No	Si	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Couting Sort	N.A.	No	O(n+k)	O(n+k)	O(n+k)
Bucket Sort	Si	No	O(n+k)	O(n+k)	O(n+k)
Radix Sort	Si	No	O(d(n+b))	O(d(n+b))	O(d(n+b))

Ordinamento—Riassunto

- Insertion Sort / Selection Sort
  - $\Theta(n^2)$, stabile, in loco, iterativo.
- Merge Sort
  - $\Theta(n \log n)$, stabile, richiede $O(n)$ spazio aggiuntivo, ricorsivo (richiede $O(\log n)$ spazio nello stack).
- Heap Sort
  - $O(n \log n)$, non stabile, sul posto, iterativo.
- Quick Sort
  - $\Theta(n \log n)$ in media, $\Theta(n^2)$ nel caso peggiore, non stabile, ricorsivo (richiede $O(\log n)$ spazio nello stack).

Ordinamento—Riassunto

- Counting Sort
  - $O(n+k)$, richiede $O(k)$ memoria aggiuntiva, iterativo. Conveniente quando $k=O(n)$
- Bucket Sort
  - $O(n+k)$, stabile, richiede $O(n+k)$ memoria aggiuntiva, iterativo. Conveniente quando $k=O(n)$
- Radix Sort
  - $O(d(n+b))$, richiede $O(n+b)$ memoria aggiuntiva. Conveniente quando $b=O(n)$.

Ordinamento—Conclusioni

- Divide-et-impera
  - Merge Sort: “divide” semplice, “combina” complesso
  - Quick Sort: “divide” complesso, “combina” nullo
- Utilizzo di strutture dati efficienti
  - Heap Sort basato su Heap
- Randomizzazione
  - La tecnica di randomizzazione ci permette di “evitare” il caso pessimo
- Dipendenza dal modello
  - Cambiando l'insieme di assunzioni, è possibile ottenere algoritmi più efficienti