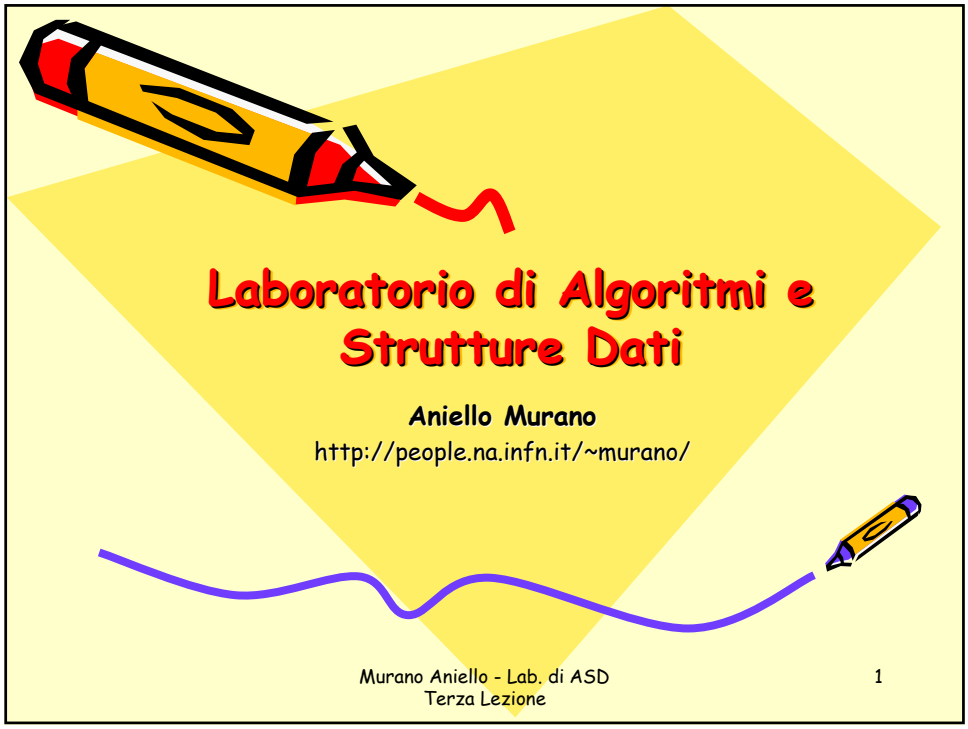
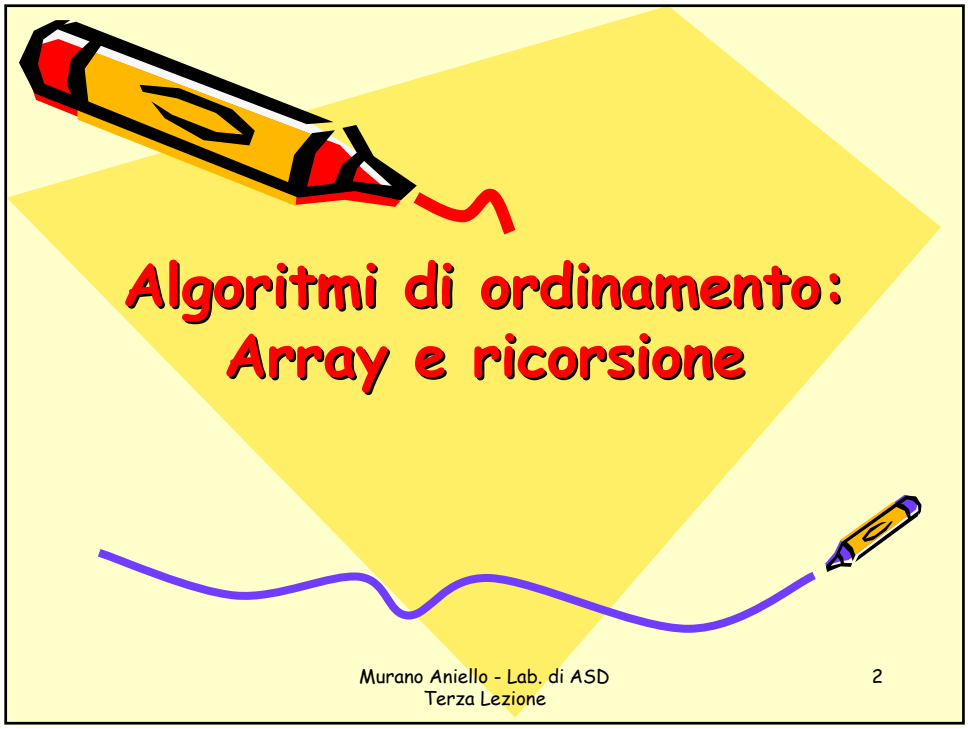


Laboratorio di Algoritmi e Strutture Dati

Aniello Murano
http://people.na.infn.it/~murano/



Algoritmi di ordinamento: Array e ricorsione

Indice

- Insertion Sort
- Quicksort
- Heapsort

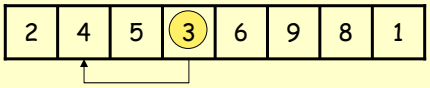
Insertion Sort

- L'insertion Sort è uno algoritmo di ordinamento molto efficiente per ordinare un piccolo numero di elementi
- L'idea di ordinamento è simile al modo che un giocatore di bridge potrebbe usare per ordinare le carte nella propria mano.
- Si inizia con la mano vuota e le carte capovolte sul tavolo
- Poi si prende una carta alla volta dal tavolo e si inserisce nella giusta posizione
- Per trovare la giusta posizione per una carta, la confrontiamo con le altre carte nella mano, da destra verso sinistra. Ogni carta più grande verrà spostata verso destra in modo da fare posto alla carta da inserire.

Funzione Insertion Sort

```
void insertion(int interi[20],int tot)
{
 int  temp;       /* Indice temporaneo per scambiare elementi */
 int  prossimo;
 int  attuale;

 for (prossimo=1;prossimo<tot;prossimo++)
    {
     temp=interi[prossimo];
     attuale=prossimo-1;
     while ((attuale>=0) && (interi[attuale]>temp))
         {
        interi[attuale+1]=interi[attuale];
        attuale=attuale-1;
       }
     interi[attuale+1]=temp;
    }
}
```



Insertion Sort

```
# include <stdio.h>
# define MAX 20
int  i,j;              /* Indici di scorrimento dell'array */
void insertion(int interi[MAX],int tot);
main()
{
int  interi[MAX]   /* Array che contiene i valori da ordinare */
int  tot;          /* Numero totale di elementi nell'array */
 printf("\nQuanti elementi deve contenere l'array: ");
 scanf("%d",&tot);
 while (tot>20)
   {printf("\n max 20 elementi: ");
   scanf("%d",&tot);}
 for (i=0;i<tot;i++)
    { printf("\nInserire il %d° elemento: ",i+1);
     scanf("%d",&interi[i]);}
   insertion(interi,tot);
   printf("\nArray Ordinato:");
   for (i=0;i<tot;i++)
     printf(" %d",interi[i]);
}
```

Ingegneria del software...

- **Autore**: Nome, Cognome, matricola …
- **Titolo**: nome della funzione (o modulo) implementata.
- **Scopo:** obiettivi dell'algoritmo implementato(sintetico)
- **Specifiche:** Nomi di funzioni, array, variabili importanti
- **Descrizione:** Informazioni sull'algoritmo implementato.
- **Lista dei Parametri:** Parametri input e output
- **Complessità di Tempo e Di Spazio**
- **Altri parametri eventualmente vuoti:**
  - **Indicatori di Errore:**
  - **Routine Ausiliarie**
  - **Indicazioni sull'utilizzo**
- **Implementazione**

Documentazione per Insertion Sort

- **Scopo** : Ordinamento di un array di numeri interi dato in ingresso
- **Specifiche:**
  - ✓ array di interi "interi[MAX]"
  - ✓ void insertion(int interi[MAX], int tot);
- **Descrizione** : L'insertion sort è un algoritmo molto efficiente per ordinare pochi numeri. Questo algoritmo è simile al modo che si potrebbe usare per ordinare un mazzo di carte…
- **Lista dei Parametri**
- *Input:*
  - ✓ interi[]: vettore contenente gli elementi da ordinare
  - ✓ tot numero degli elementi contenuti nel vettore
- *Output:* array interi[] ordinato in ordine crescente.

Documentazione per Insertion Sort

- **Complessità di Tempo e Di Spazio:**
  - *Complessità di tempo:* L'algoritmo inserisce il componente **interi[i]** nel vettore già ordinato di componenti **interi[0]...interi[i-1]** spostando di una posizione tutti i componenti che seguono quello da inserire. Ad ogni passo la procedura compie nel caso peggiore (vettore ordinato al contrario), N-1 confronti, essendo N la lunghezza del vettore corrente e i-1 spostamenti. Le operazioni nel caso peggiore sono dunque (1+2+...+N-1), cioè, nel caso peggiore la complessità asintotica di tempo è $O(n^2)$. Nel caso migliore (vettore ordinato) bastano N-1 confronti.
  - *Complessità di spazio:* La struttura dati utilizzata per implementare l'algoritmo è un ARRAY monodimensionale, contenente i valori da ordinare, di conseguenza la complessità di spazio è O(n).
- Esempi di esecuzione:
  - Dati in ingresso i numeri: 20 11 45, si ottiene in uscita: 11 20 45

    Si veda il documento "Documentazione InsertionSort.pdf"

Quicksort

- Il Quicksort è un algoritmo di ordinamento molto efficiente.
- Il suo tempo asintotico medio è O(n*logn)
- È un metodo di ordinamento ricorsivo: si sceglie un elemento del vettore (pivot) e si partiziona il vettore in due parti. La prima partizione contiene tutti gli elementi minori o uguali al pivot, mentre la seconda gli elementi maggiori. A questo punto basta riapplicare il metodo (reinvocare ricorsivamente la procedura che lo realizza) ai due sotto-vettori costituiti dalle partizioni per ordinarli. (Si veda per una simulazione la lezione del Prof. Benerecetti "QuickSort.pdf")

Implementazione di Quicksort 1/3

```
# include <stdio.h>
# define MAX 20
int  i,j, tot
void quicksort(int array[MAX], int begin, int end);
int partition(int array[MAX], int begin, int end);
main()
{    <<Inserimento elementi nell'array  >>
     quicksort(interi, 0, tot-1);
     <<Visualizzazione array ordinato >>
}
```


Implementazione di Quicksort 2/3

```
void quicksort(int array[MAX], int begin, int end)
{
   int q;
   if (begin<end)
     { q=partition(array, begin, end);
       quicksort(array, begin, q);
       quicksort(array, q+1, end);
     }
}
```

Implementazione di Quicksort 3/3

```
int partition(int array[], int begin, int end)
{
     int interi[MAX], pivot,l,r,tmp;
     pivot = array[begin];
     l = begin - 1;
     r = end+1;
     while(1)
     {
      do r=r-1;
      while(array[r]>pivot);
      do l=l+1;
      while(array[l]<pivot);
      if (l<r) {
         tmp=array[l];
         array[l]=array[r];
         array[r]=tmp;}
      else return r;
     }
}
```


Complessità del Quicksort

- Il caso migliore si ha quando la scelta del pivot crea, ad ogni livello della ricorsione, due partizioni di *uguale dimensione*. In tal caso si converge in k = logN passi. Al passo k si opera su M=2^k vettori di lunghezza L=N/(2^k)
- Il numero di confronti ad ogni livello e' L*M, pertanto, il numero globale di confronti e' **1*N+2*N/2 + 4*N/4 +... + N*1 = NlogN**
- Il caso più sfortunato si ha quando ad ogni passo si sceglie un pivot tale che un sotto-vettore ha lunghezza 1. In tal caso si converge in k=N-1 passi. Pertanto, il numero globale di confronti e': **N-1+N-2+ ... +2+1 = N(N/2)**

Heapsort

- L'Heapsort è un algoritmo di ordinamento molto efficiente:
- Come l'insertion Sort e il Quicksort, l'Heapsort ordina sul posto
- Meglio dell'Insertion Sort e del Quicksort, il running time dell'Heapsort è O(nlogn) nel caso peggiore
- L'algoritmo di Heapsort basa la sua potenza sull'utilizzo di una struttura dati chiamata **Heap**, che gestisce intelligentemente le informazioni durante l'esecuzione dell'algoritmo di ordinamento.

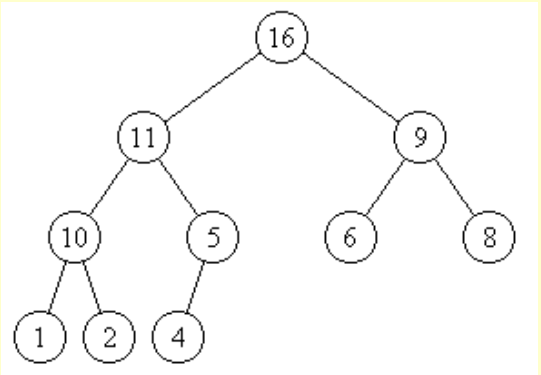
Heap

- La struttura dati Heap (binaria) è un array che può essere visto come un albero binario completo
- Proprietà fondamentale degli Heap è che il valore associato al nodo padre è sempre maggiore o uguale a quello associato ai nodi figli
- Un Array A per rappresentare un Heap ha bisogno di due attributi:
  - Lunghezza dell'array
  - Elementi dell'Heap memorizzati nell'array

Organizzazione dell'array

- La radice dell'Heap è sempre memorizzata nel primo elemento dell'array.
- Dato un nodo i, il nodo padre, figlio sinistro e destro di i possono essere calcolati velocemente nel modo seguente:
  - int left(int i) { return 2*i+1;}
  - int right(int i) { return 2*i+2;}
  - int parent (int i) {return (i-1)/2;}
- N.B. Nel C il primo elemento di un vettore A è A[0]. Nel libro di testo "Algoritmi e Strutture Dati", il primo elemento di un vettore A è invece A[1] e i valori precedenti sono dati dalle seguenti pseudo-funzioni:
  - Parent (i) return$\lfloor i/2 \rfloor$
  - Left (i) return 2i
  - Right(i) return 2i+1

Esempio di Heap



Heap con 10 vertici

0	1	2	3	4	5	6	7	8	9
16	**11**	**9**	**10**	**5**	**6**	**8**	**1**	**2**	**4**

Heapify

- Una subroutine molto importante per la manipolazione degli Heap è Heapfy. Questa routine ha il compito di assicurare il rispetto della proprietà fondamentale degli Heap. Cioè, che il valore di ogni nodo non è inferiore di quello dei propri figli.
- Di seguito mostriamo una funzione ricorsiva Heapify che ha il compito di far scendere il valore di un nodo che viola la proprietà di Heap lungo i suoi sottoalberi.

Implementazione di Heapify

```
void Heapify(int A[MAX], int i)
{
    int l,r,largest;
    l = left(i);
    r = right(i);
    if (l < HeapSize && A[l] > A[i])
          largest = l;
    else largest = i;
    if (r < HeapSize && A[r] > A[largest])
          largest = r;

    if (largest != i) {
          swap(A, i, largest);
          Heapify(A, largest);
    }
}
```

Costruire un Heap

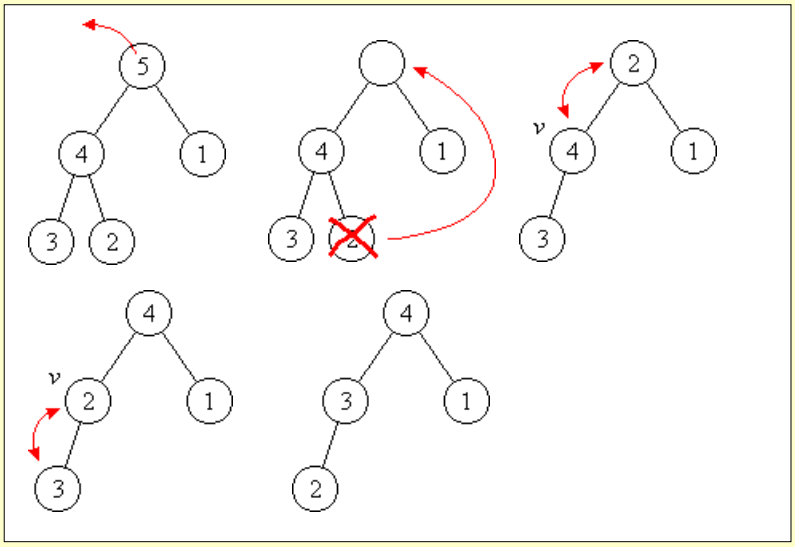
- La seguente procedura serve a costruire un Heap da un array:

```
void BuildHeap(int A[MAX])
{
int i;
   HeapSize = ArraySize;
   for (i=ArraySize/2; i>=0; i--)
     Heapify(A, i);
}
```


Funzione HeapSort

```
void HeapSort(int A[MAX])
{
int i;
   BuildHeap(A);
   for (i=ArraySize-1; i>=1; i--) {
         swap(A, 0, i);
         HeapSize--;
         Heapify(A, 0);
   }
}
```

Simulazione



Algoritmo di HeapSort

```
#include <stdlib.h>
#define MAX 20
int ArraySize, HeapSize, tot;

int left(int i)   { return 2*i+1;}
int right(int i) { return 2*i+2;}
int p(int i)      {return (i-1)/2;}

void swap(int A[MAX], int i, int j)
    {int tmp = A[i];
    A[i] = A[j];
    A[j] =tmp;}

void Heapify(int A[MAX], int i);
void BuildHeap(int A[MAX]);
void HeapSort(int A[MAX]);
```

Main di HeapSort

```
main()
{
int A[MAX], k;
 printf("\nQuanti elementi deve contenere l'array: ");
 scanf("%d",&tot);
 while (tot>MAX)
   {printf("\n max 20 elementi: ");   scanf("%d",&tot);}
 for (k=0;k<tot;k++) {
    printf("\nInserire il %d° elemento: ",k+1);
    scanf("%d",&A[k]); }
 HeapSize=ArraySize=tot;
 HeapSort(A);
 printf("\nArray Ordinato:");
  for (k=0;k<tot;k++)
    printf(" %d",A[k]);
}
```


Complessità

- Il running time di Heapify è $O(h)$ dove h è l'altezza dell'Heap. Siccome l'heap è un albero binario completo, il running time è $O(\log n)$. Più in dettaglio la sua complessità è la soluzione della ricorrenza $T(n) \leq T(2n/3) + \Theta(1)$ utilizzando il master method (caso 2).
- BuildHeap fa $O(n)$ chiamate a Heapify. Per cui il running time di Buildheap è sicuramente $O(n \log n)$. Si noti che le chiamate a Heapify avvengono su nodi ad altezza variabile minore di h. Da un'analisi dettagliata, risulta che il running time di Heapify è $O(n)$.
- Heapsort fa $O(n)$ chiamate a Heapify. Dunque il running time di Heapsort è $O(n \log n)$