ELABORATO ALGORITMI E STRUTTURE DATI

Algoritmi di ordinamento tempo lineare e confronto con Insertion Sort

Mario Pace M63000988

Sommario

Codice c++

1.1 Insertion sort

Codice della funzione Insertion Sort:

```
void InsertionSort (float A[], int n)
{
for (int j=1; j<n; j++)
{
float key = A[j];
// Insert A[j] into A[1..j-1]
int i = j-1;
while (i>=0 && A[i]>key)
{
A[i+1] = A[i];
i--;
}
A[i+1] = key;
}
}
```

1.2 Counting Sort

Codice della funzione Counting Sort e della funzione di stampa utilizzata per mostrare a video i risultati:

```
#include<iostream>
#include<stdlib.h>
#include<algorithm>

using namespace std;

void stampa(int *array, int dim) {
   for(int i = 1; i<=dim; i++)
      cout << array[i] << " ";
   cout << endl;
}


void countingSort(int *A, int* B, int dim, int k) {

   int C[k+1];
   for(int i = 0; i<k; i++)
      C[i] = 0;
   for(int i = 1; i <=dim; i++)
      C[A[i]]++;
   for(int i = 1; i< k; i++)
      C[i] = C[i]+C[i-1];
   for(int i = dim; i>=1; i--) {
      B[C[A[i]]] = A[i];
      C[A[i]] = C[A[i]] - 1;
   }

}
```

Codice del main nel quale viene richiamata la funzione:

```
int main() {
   int n,k;
   cout << "Inserisci il numero di elementi da ordinare: \n";
   cin >> n;
   int A[n+1];
   int B[n+1];
   cout << "Inserisci la cardinalita' dell'insieme degli elementi : \n";
   cin >> k;
   cout << "Inserisci gli elementi:" << endl;
   for(int i = 1; i<=n; i++) {
      cin >> A[i];
   }
   cout << "Vettore A (non ordinato) : ";
   stampa(A, n);
   countingSort(A, B, n, k);
   cout << "Vettore B (ordinato) : ";
   stampa(B, n);
}
```

Codice utilizzato per mostrare il tempo di esecuzione della funzione:

```
timeval start, stop;
double elapsedTime;

//da dare prima del codice su cui si vuole fare il test, per registrare il tempo di avvio
gettimeofday(&start, NULL);


  int C[k+1];
   for(int i = 0; i<k; i++)
      C[i] = 0;
   for(int i = 1; i <=n; i++)
      C[A[i]]++;
   for(int i = 1; i< k; i++)
      C[i] = C[i]+C[i-1];
   for(int i = n; i>=1; i--) {
      B[C[A[i]]] = A[i];
      C[A[i]] = C[A[i]] - 1;
  }

//qui si riprende il tempo finale per fare la differenze
gettimeofday(&stop, NULL);

//calcolo delle differenze
elapsedTime = (stop.tv_sec - start.tv_sec) * 1000.0;              // sec to ms
elapsedTime += (stop.tv_usec - start.tv_usec) / 1000.0;           // us to ms

//stampa
cout << "Vettore B (ordinato) : ";
   stampa(B, n);

cout  << " il tempo di esecuzione e' :  " << elapsedTime << " ms.\n";
```

1.3 Radix Sort

Codice della funzione Radix Sort e dell'algoritmo stabile utilizzato per implementarla (in questo caso Counting Sort):

```
void countingSort(int A[], int n, int d)
{
    int B[n];
    int i, C[10] = {0};


    for (i = 0; i < n; i++)
        C[ (A[i]/d)%10 ]++;


    for (i = 1; i < 10; i++)
        C[i] += C[i - 1];


    for (i = n - 1; i >= 0; i--)
    {
        B[C[ (A[i]/d)%10 ] - 1] = A[i];
        C[ (A[i]/d)%10 ]--;
    }


    for (i = 0; i < n; i++)
        A[i] = B[i];
}

void radixsort(int A[], int n, int k)
{

  for (int d = 1; k/d > 0; d *= 10)
        countingSort(A, n, d);
}
```

Codice del main nel quale è richiamata la funzione, compreso del codice necessario per la visualizzazione del tempo di esecuzione:

```
int main()
{
int n,k;
   cout << "Inserisci il numero di elementi da ordinare: \n";
   cin >> n;
   int A[n];

   cout << "Inserisci la cardinalita' dell'insieme degli elementi : \n";
   cin >> k;
   srand(111222333);

   for(int i = 0; i<n; i++) {
      A[i] = rand() % k;
   }


   cout << "Vettore A (non ordinato) : ";
   stampa(A, n);


timeval start, stop;
double elapsedTime;

gettimeofday(&start, NULL);

  radixsort(A, n, k);

gettimeofday(&stop, NULL);

elapsedTime = (stop.tv_sec - start.tv_sec) * 1000.0;              // sec to ms
elapsedTime += (stop.tv_usec - start.tv_usec) / 1000.0;           // us to ms

cout << "Vettore  (ordinato) : ";
   stampa(A, n);

cout  << " /n il tempo di esecuzione e' :  " << elapsedTime << " ms.\n";

    return 0;
}
```

1.4 Bucket Sort

Codice della funzione Bucket Sort:

```
void bucketSort(float A[], int n)
{
    vector<float> b[n];

    for (int i=0; i<n; i++)
    {
       int bi = n*A[i];
       b[bi].push_back(A[i]);
    }


    for (int i=0; i<n; i++)
       //InsertionSort(b[i],b[i].size());
       sort(b[i].begin(), b[i].end());

    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
          A[index++] = b[i][j];
}
```

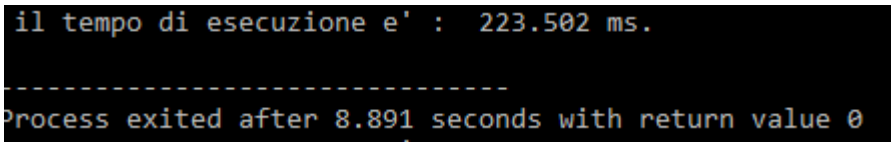
Codice del main nel quale è richiamata la funzione, compreso del codice necessario per la visualizzazione del tempo di esecuzione:

```
int main()
{
int n,k;
   cout << "Inserisci il numero di elementi da ordinare: \n";
   cin >> n;
   float A[n];
   srand(111222333);

   for(int i = 0; i<n; i++) {
      A[i] = float(rand() % 100)/ 100.0;
   }

   cout << "Vettore A (non ordinato) : ";
   stampa(A, n);

timeval start, stop;
double elapsedTime;

gettimeofday(&start, NULL);

  bucketSort(A, n);

gettimeofday(&stop, NULL);

elapsedTime = (stop.tv_sec - start.tv_sec) * 1000.0;              // sec to ms
elapsedTime += (stop.tv_usec - start.tv_usec) / 1000.0;           // us to ms

cout << "Vettore  (ordinato) : ";
   stampa(A, n);

cout  << "\n il tempo di esecuzione e' :  " << elapsedTime << " ms.\n";

   return 0;
}
```

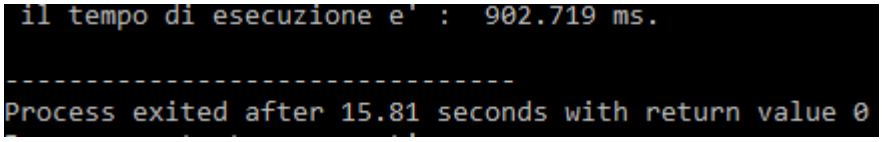
Test e tempo di esecuzione

2.1 Insertion Sort

Si è testato insertion sort su un array di 20000 elementi e il risultato è stato:



Si è poi ripetuto il test su un array di 40000 elementi e il risultato è stato:



Insertion sort ha una complessità TETA(n^2), infatti se raddoppio il numero di elementi N (da 20000 a 40000) il tempo di esecuzione quadruplica.

2.2 Counting Sort

Si è testato counting sort su un array di 100000 elementi con una cardinalità dell'insieme dei valori k=20000 e il risultato è stato:

```
 il tempo di esecuzione e' :  0.998 ms.

--------------------------------
Process exited after 44.11 seconds with return value 0
```

Si è poi ripetuto il test su un array di 200000 elementi con una cardinalità dell'insieme dei valori k=20000 e il risultato è stato:

```
 il tempo di esecuzione e' :  2.001 ms.

--------------------------------
Process exited after 76.79 seconds with return value 0
```

La complessità di counting sort è TETA(n+k), che risulta essere un TETA(n) se k<n come in questo caso, infatti al raddoppiare del numero di elementi dell'array n raddoppia anche il tempo di esecuzione.

2.3 Radix Sort

Si è testato radix sort su un array di 100000 elementi con una cardinalità dell'insieme dei valori k=20000 (quindi d=5) e il risultato è stato:

```
 il tempo di esecuzione e' :  1.994 ms.

--------------------------------
Process exited after 34.03 seconds with return value 0
```

Si è poi ripetuto il test su un array di 200000 elementi con una cardinalità dell'insieme dei valori k=20000 (quindi d=5) e il risultato è stato:

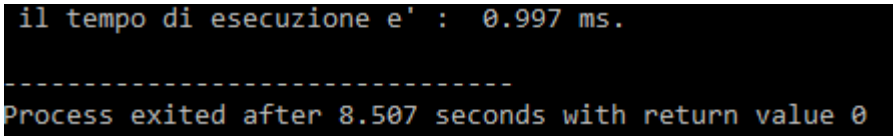
```
 il tempo di esecuzione e' :  3.661 ms.

--------------------------------
Process exited after 63.22 seconds with return value 0
```

La complessità di radix sort è TETA(d(n+k)), che risulta essere un TETA(n) se d è un valore costante e k<n come in questo caso, infatti al raddoppiare del numero di elementi dell'array n raddoppia anche il tempo di esecuzione.

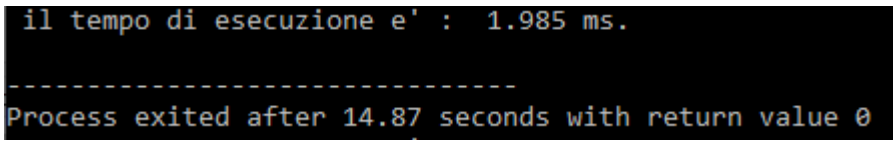
Notiamo inoltre come radix sort impieghi comunque più tempo di counting sort su uno stesso array, poiché la moltiplicazione per d genera dei termini minori che vengono assorbiti asintoticamente da TETA(n) ma che comunque hanno il loro contributo.

2.4 Bucket Sort

Si è testato bucket sort su un array di 20000 elementi e il risultato è stato:



Si è ripetuto il test su un array di 20000 elementi e il risultato è stato:



La complessità di bucket sort è TETA(n) se la distribuzione di probabilità dei valori di ingresso è tale da distribuire uniformemente i valori all'interno delle liste concatenate come avviene in questo caso, infatti al raddoppiare del numero di elementi dell'array n raddoppia anche il tempo di esecuzione.