Sorch

Un algoritmo di ricerca che riduce il disordine del campione.

Progetto finale per il corso di Algoritmi e Strutture di Dati.

Giles Sebastian

Matricola: 1071522

E-mail: s1071522@studenti.univpm.it

Maggio 2017

Sorgenti di questo documento, della funzione e del programma di benchmarking sono visibili su https://github.com/sebgiles/sorch

1 L'idea

Ordinare un vettore per poi sfruttare la ricerca dicotomica, oppure usare una semplice ricerca lineare?

La scelta dipende dalla dimensione della popolazione e dal numero di ricerche che si prevede di fare. L'algoritmo "sorch" (*sort-search*) fornisce un buon compromesso tra il pesante costo iniziale per l'ordinamento e la lentezza di una ricerca lineare.

L'idea di fondo è quella di ottenere una progressiva riduzione dei tempi di ricerca all'aumentare del numero di ricerche effettuate, contribuendo ognuna di esse a dare uno pseudo ordinamento al vettore.

2 L'algoritmo

La prima ricerca è necessariamente di impronta lineare essendo il vettore disordinato. La ricerca lineare classica compie n confronti solo nel caso peggiore, questo algoritmo invece impone di percorrere l'intero vettore in modo da partizionarlo (come per il primo passo di quicksort).

Compiere simultaneamente ricerca e partizionamento in un unico ciclo permette di sfruttare i confronti con il pivot per ridurre (in media dimezzare) il numero di confronti di uguaglianza richiesti per la ricerca.

Alla fine della prima ricerca il vettore sarà partizionato rispetto ad un pivot.

Alla chiamata successiva, l'azione di ricerca e partizionamento viene ripetuta sul sottoinsieme opportunamente a sinistra o destra del pivot in base al confronto dell'elemento cercato con il pivot stesso. Il risultato del partizionamento è un secondo pivot che divide ulteriormente il vettore.

Questa procedura si presta alla **ricorsione**.

È necessario un meccanismo che tenga traccia dei partizionamenti svolti, al fine di trarne vantaggio nella fase di *divide* delle ricerche successive, basata sul confronto dell'elemento cercato con i pivot. La struttura di dati più naturale per questo scopo è un albero binario di ricerca.

La radice di questo albero (*albero degli indici*) deve essere inizializzata con il **puntatore al primo elemento** del vettore da "indicizzare" e la sua **lunghezza**, il puntatore al pivot viene inizializzato a NULL, per indicare che la partizione non è ancora stata fatta.

Il *caso base*, e quindi la chiusura della ricorsione, può avere 3 casi:

1. Se il puntatore al pivot è NULL **avviene un partizionamento con simultanea ricerca.** Alla fine di ogni partizionamento la posizione del pivot viene memorizzata sul nodo e vengono aggiunte due foglie, figlie di questo nodo, rispettivamente per le partizioni sinistra e destra.

   Ogni foglia appena creata viene inizializzata nello stesso modo della radice dell'albero, considerando il puntatore al primo elemento della corrispondente partizione e la sua lunghezza.

   Se nel blocco appena partizionato non ci fossero elementi minori (risp. maggiori) del pivot, durante la creazione della foglia sinistra (risp. destra) viene riconosciuta la lunghezza 0 della partizione e viene assegnato un puntatore NULL. Questo avviene in particolare quando tutti gli elementi adiacenti nel vettore sono diventati dei pivot e cioè il vettore è completamente indicizzato (e ordinato) su un intervallo.

   Se l'elemento cercato fosse stato trovato ne viene ritornato il puntatore, altrimenti NULL.

2. **Viene ritornato immediatamente NULL.** Se l'albero è NULL quanto avviene nel caso 1 garantisce che il vettore sia completamente ordinato in un intorno del punto in cui ci si aspetta di trovarlo, ma l'elemento non c'è.

3. **Viene ritornato immediatamente il puntatore al pivot** qualora questo coincida con l'elemento cercato.

3 Alcune misurazioni

I grafici in appendice sono stati disegnati usando Matlab. I dati sono stati ottenuti con un programma in C che ha eseguito ricerche ripetute con i tre algoritmi su diversi campioni casuali misurandone il tempo impiegato.

In ordinata compare il tempo accumulato per compiere il numero di ricerche in ascissa. Per il metodo dicotomico questo include il tempo richiesto per l'ordinamento iniziale.

Tutte le curve sono state ottenute facendo la media dei risultati per 10 popolazioni diverse, ognuna costitutita dai primi n numeri naturali, ogni volta disordinati casualmente.

Le query di ricerca sono generate casualmente in maniera uniforme.

In azzurro, arancione e giallo si hanno rispettivamente le curve per ricerca lineare, ricerca dicotomica e sorch.

- **Figura 1** $n = 10^5$, 10^2 query prese uniformemente su tutto il campione.

  È visibile la superiorità della ricerca lineare per le prime ricerche.

- **Figura 2** $n = 10^5$, 10^4 query prese uniformemente su tutto il campione.

  È visibile il vantaggio a lungo termine ottenuto dall'ordinamento.

- **Figura 3** $n = 10^5$, 10^2 query prese uniformemente sull'intervallo $[4.5 * 10^4, 5.5 * 10^4]$

  L'algoritmo sorch è molto performante quando le query sono vicine tra loro rispetto alla dimensione del campione, questo è dovuto ad una fitta indicizzazione sull'intervallo interessato.

- **Figura 4** $n = 10^3$, 10^2 query prese uniformemente su tutto il campione.

  Quando il campione è ristretto l'ordinamento costa poco e viene ripagato subito. L'algoritmo sorch rimane indietro a causa dell'overhead per la gestione dell'albero degli indici.

4 Codice C

```c
typedef struct node node;
typedef node* indextree;

struct node{
  TInfo* pivot;
  TInfo* data;
  int size;
  indextree left;
  indextree right;
};

// crea un nuovo nodo per l'albero degli indici
// ritorna il puntatore al nodo
indextree newindextree(TInfo a[], int n){
  if (n < 1) return NULL; //indica una foglia senza dati
  node* new;
  new = (node *) malloc(sizeof(node));
  new->pivot = NULL;
  new->data = a;
  new->size = n;
  new->left = NULL;
  new->right = NULL;
  return new;
}
```

```
26 // Ricerca l'elemento x nel vettore indicizzato dall'albero t.
27 // RITORNA: il puntatore all'elemento sul vettore (NULL se assente).
28 // POST: - L'albero degli indici ha un nuovo pivot.
29 //       - Il vettore viene partizionato.
30 int* sorch (indextree t, TInfo x){
31
32   if (t == NULL) return NULL; // foglia senza dati ferma la ricorsione
33
34   // siamo su una foglia (il blocco deve ancora essere partizionato)
35   if (t->pivot == NULL){
36
37     TInfo pivot = t->data[0]; //scegliamo il primo elemento come pivot
38
39     // sapere se l'elemento cercato sia maggiore o minore del pivot ci
40     // permette di ridurre i confronti necessari alla ricerca lineare
41     int smaller = less(x , pivot);
42
43     TInfo* found = NULL; //puntatore all'eventuale elemento trovato
44
45     // il seguente ciclo    identico a quello per il partizionamento del
46     // quicksort, con l'aggiunta di confronti per la ricerca di x
47     int k = 1;
48     for(int i = 1; i < t->size; i++){
49       if (less(t->data[i], pivot)){
50         swap(&t->data[i], &t->data[k++]);
51         if(smaller)
52           if(equal(t->data[k-1], x))
53             found = &t->data[k-1];
54       }
55       if (!smaller)
56         if(equal(t->data[i], x))
57           found = &t->data[i];
58     }
59     swap(&t->data[0], &t->data[k-1]); // posiziona il pivot come separatore
60     t->pivot = &t->data[k-1]; // memorizza il puntatore al pivot sul nodo
61
62     //assegnazione delle partizioni a due nuove foglie
63     t->left = newindextree(t->data, k-1);
64     t->right = newindextree(&t->data[k], t->size - k);
65
66     // devono essere aggiunti due controlli aggiuntivi:
67     // 1 - in caso l'elemento cercato coincida con il pivot
68     if(equal(pivot, x))
69       found = t->pivot;
70     // 2 - in caso l'elemento cercato fosse stato scambiato col pivot
71     else if (found == t->pivot)
72       found = &t->data[0];
73
74     return found;
75   }
76
77   //ferma la ricorsione se l'elemento cercato coincide col pivot
78   if (equal(x, *(t->pivot)))
79     return t->pivot;
80
81   //divide et impera
82   if (less(x, *(t->pivot)))
83     return sorch(t->left , x);
84   else
85     return sorch(t->right , x);
86 }
```

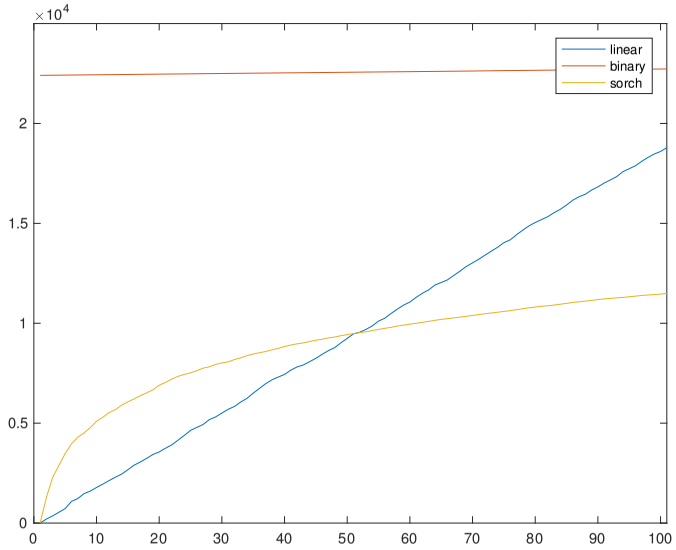

Figura 1

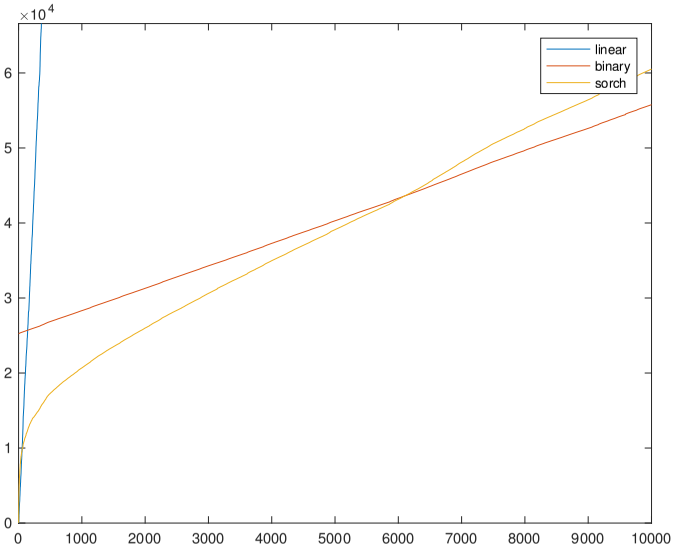

Figura 2

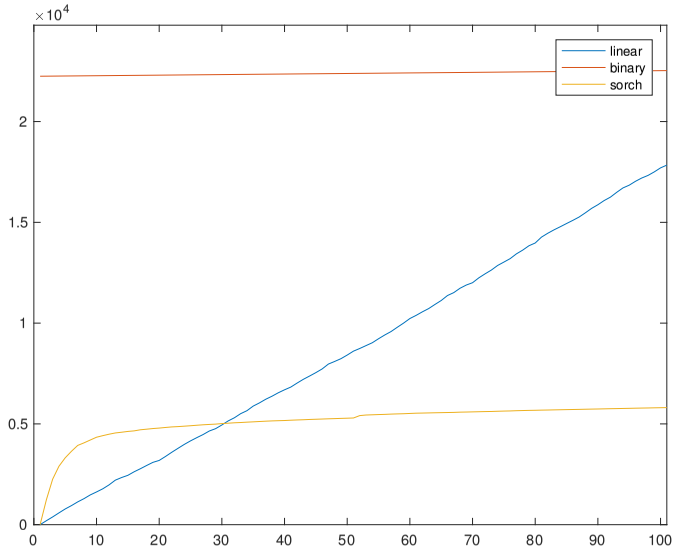

Figura 3

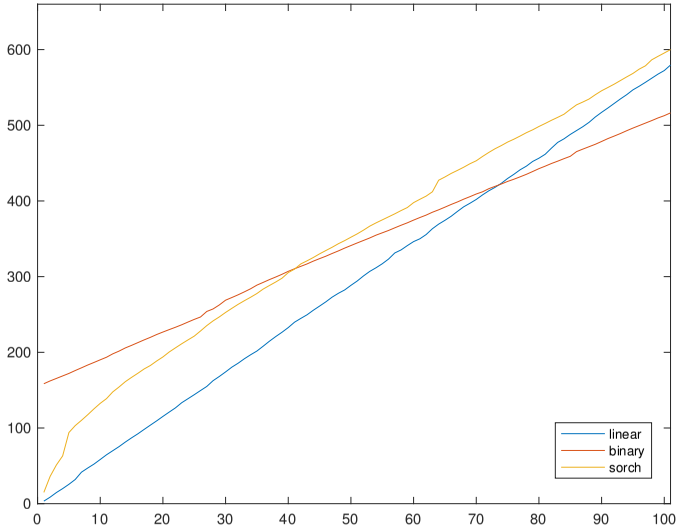

Figura 4