ALGORITMI E STRUTTURE DATI

Progetto di Laboratorio a.a 2021/2022

di Chiara Tumminelli

INDICE

1. ESERCIZIO 1 (C)

2. ESERCIZIO 2 (C)

1. ESERCIZIO 1 (C)

1.1 TESTO

Implementare una libreria che offra due algoritmi di ordinamento: Quick Sort e Binary Insertion Sort. Con Binary Insertion Sort ci riferiamo a una versione dell'algoritmo Insertion Sort in cui la posizione all'interno della sezione ordinata del vettore in cui inserire l'elemento corrente è determinata tramite ricerca binaria. Nell'implementazione del Quick Sort, la scelta del pivot dovrà essere studiata e discussa nella relazione.

Il codice che implementa Quick Sort e Binary Insertion Sort deve essere generico. Inoltre, la libreria deve permettere di specificare (cioè deve accettare in input) il criterio secondo cui ordinare i dati.

1.2 SCELTE IMPLEMENTATIVE

Per la lettura del file “records.csv” si è deciso di definire il suo filepath nel seguente modo:

```
#define RECORDS_PATH “../resources/records.csv”
```

Si è assunto che ogni linea del file “records.csv” non sforasse di 1000 caratteri:

```
#define LINE_LENGTH 1000
```

Si è definita la lunghezza del file “records.csv”, che è stata modificata più volte per effettuare i test presenti nei paragrafi successivi:

```
#define CSV_LENGTH 20000000
```

Il programma chiede all’utente se vuole ordinare il csv mediante il QuickSort oppure mediante il Binary Insertion Sort. Nel caso l’utente risponda che vuole eseguire il QuickSort, il programma chiede all’utente quale versione del QuickSort vuole eseguire:

```
Which algorithm do you want to use?

        Press 1 for quickSort.
        Press 2 for binary insertion sort.

        Your choice:
```

```
Which pivot do you want to use?

        Press 1 for first element.
        Press 2 for last element.
        Press 3 for random element.
        Press 4 for median element.
```

L’ordinamento dei record genera tre file testuali, uno per ogni tipo di dato. I loro filepath sono i seguenti:

```
#define CHAR_ORD_RECORDS_PATH “../resources/char_ord_records.txt”
#define INT_ORD_RECORDS_PATH “../resources/int_ord_records.txt”
#define FLOAT_ORD_RECORDS_PATH “../resources/float_ord_records.txt”
```

1.3 QUICKSORT

1.3.1 Tempi di esecuzione del QuickSort utilizzando come Pivot il primo elemento

CSV contenente **200 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	0.110000	0.111059
INT	0.079000	0.084716
FLOAT	0.094000	0.086066

CSV contenente **2 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	2.078000	2.040049
INT	1.344000	1.425446
FLOAT	1.359000	1.425126

CSV contenente **20 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	30.015000	29.470484
INT	20.109000	21.474490
FLOAT	32.234000	23.966710

1.3.2 Tempi di esecuzione del QuickSort utilizzando come Pivot l'ultimo elemento

CSV contenente **200 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	0.641000	0.435784
INT	0.141000	0.087766
FLOAT	0.172000	0.091402

CSV contenente **2 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	83.329000	58.218797
INT	1.938000	1.464480
FLOAT	1.921000	1.395916

CSV contenente **20 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	> 10 min	> 10 min
INT	22.187000	14.968658
FLOAT	27.141000	19.410314

1.3.3 Tempi di esecuzione del QuickSort utilizzando come Pivot un elemento random

CSV contenente **200 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	0.203000	0.118246
INT	0.156000	0.091931
FLOAT	0.157000	0.095334

CSV contenente **2 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	3.047000	2.058375
INT	2.047000	1.478331
FLOAT	2.328000	1.476706

CSV contenente **20 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	42.843000	30.575006
INT	32.641000	22.101545
FLOAT	31.405000	24.561113

1.3.4 Tempi di esecuzione del QuickSort utilizzando come Pivot la mediana

CSV contenente **200 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	0.234000	0.123935
INT	0.141000	0.097685
FLOAT	0.203000	0.097254

CSV contenente **2 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	3.516000	2.280540
INT	2.250000	1.637648
FLOAT	2.125000	1.655431

CSV contenente **20 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	47.422000	33.386737
INT	33.141000	23.553022
FLOAT	40.939000	23.498981

1.3.5 Valutazioni finali sulla complessità del QuickSort

Il QuickSort lavora partizionando l'array da ordinare, per poi ordinarne ricorsivamente ogni partizione. La complessità dell'algoritmo, quindi, dipende dalla scelta del pivot: se il pivot viene scelto male, quindi se uno dei due sub-array ha lunghezza zero, i confronti sono $O(n^2)$; invece, se il pivot viene scelto bene, ovvero in modo da avere due sub-array di egual dimensione, i confronti sono O(nlogn).

Si è deciso di implementare il QuickSort testando l'utilizzo di quattro pivot: il primo elemento, l'ultimo elemento, l'elemento randomico e l'elemento centrale. In particolare, è emerso che l'elemento centrale sarebbe la scelta migliore in quanto, dividendo l'array in due parti uguali ad ogni passo ed esaminando tutti gli n elementi, porta il tempo di esecuzione a O(nlogn). I pivot peggiori, invece, risultano il primo e l'ultimo elemento in quanto, per input originariamente ordinati, ogni chiamata ricorsiva ridurrebbe di solo un'unità la dimensione dell'array da ordinare, rendendo necessarie n chiamate ricorsive per effettuare l'ordinamento e portando il tempo di esecuzione a $O(n^2)$, ovvero il peggiore runtime. Una soluzione a questo problema potrebbe essere quello di scegliere come pivot un elemento casuale. In questo caso, la probabilità di selezionare proprio la mediana è ad ogni passo uguale a 1/n, quindi è inversamente proporzionale alla lunghezza dell'array, ma la bontà del QuickSort risiede proprio nell'avere un comportamento ottimo anche nei casi sfavorevoli. Infatti, scegliendo un pivot randomico nel caso di campioni numerosi, anche se la sequenza fosse già ordinata risulterebbe altamente improbabile che il pivot scelto fosse il primo o l'ultimo elemento.

In conclusione, l'unico punto debole del QuickSort risiede nella degenerazione del caso peggiore. L'algoritmo, infatti, preferisce sequenza disordinate e dà luogo a comportamenti poco intelligenti nel caso di sequenze già ordinate. Può sembrare sorprendente, infatti, che uno dei sistemi migliori per effettuare tale scelta sia quello di affidarsi al caso.

1.4 BINARY INSERTION SORT

1.4.1 Tempi di esecuzione del Binary Insertion Sort

CSV contenente **200 000** valori

	Intel i5-8250U	**Intel i7-8550U**
CHAR	26.078000	15.778322
INT	26.641000	15.652791
FLOAT	26.499000	15.656468

CSV contenente **2 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	> 10 min	> 10 min
INT	> 10 min	> 10 min
FLOAT	> 10 min	> 10 min

CSV contenente **20 000 000** valori

	Intel i5-8250U	Intel i7-8550U
CHAR	> 10 min	> 10 min
INT	> 10 min	> 10 min
FLOAT	> 10 min	> 10 min

1.4.2 Valutazioni finali sulla complessità del Binary Insertion Sort

Il Binary Insertion Sort è un algoritmo di ordinamento simile all'Insertion Sort, ma invece di utilizzare la ricerca lineare per trovare la posizione in cui deve essere inserito un elemento utilizza la ricerca binaria. In questo modo, il valore comparativo dell'inserimento di un singolo elemento si riduce da O(n) a O(logn). Si tratta di un algoritmo adattivo, ovvero funziona più velocemente quando l'input è già parzialmente ordinato, ed è più efficiente quando l'input possiede un numero basso di elementi, come possiamo notare dai test effettuati nei paragrafi precedenti.

Il caso peggiore vale $O(n^2)$ e si verifica quando l'input è invertito, quindi inizialmente ordinato in ordine decrescente. Il caso migliore, invece, si verifica quando l'elemento è già nella sua posizione ordinata; in questo caso, non dobbiamo spostare nessuno degli elementi e possiamo inserire l'elemento in O(1). Per l'i-esimo elemento, quindi, si eseguono (log i) operazioni, pertanto la complessità vale O(nlogn). Nel caso medio, quindi assumendo che gli elementi dell'input siano confusi, occorre effettuare O(i/2) passaggi per l'inserimento dell'i-esimo elemento, quindi la complessità temporale media è $O(n^2)$.

In sintesi, il Binary Insertion Sort è particolarmente efficiente quando l'input è composto da pochi valori e quando l'input è già parzialmente ordinato. Tuttavia, quando la dimensione dell'array è grande, conviene ricorrere al QuickSort. Un punto di forza del Binary Insertion Sort consiste nel fatto che fa meno confronti; pertanto, è efficiente utilizzarlo quando il costo del confronto è elevato.

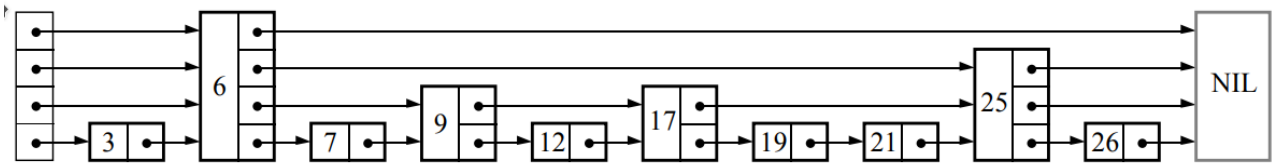
2. ESERCIZIO 2 (C)

2.1 TESTO

Realizzare una struttura dati chiamata skip_list. La skip_list è un tipo di lista concatenata che memorizza una lista ordinata di elementi.

Al contrario delle liste concatenate classiche, la skip_list è una struttura dati probabilistica che permette di realizzare l'operazione di ricerca con complessità O(log n) in termini di tempo. Anche le operazioni di inserimento e cancellazione di elementi possono essere realizzate in tempo O(log n). Per questa ragione, la skip_list è una delle strutture dati che vengono spesso utilizzate per indicizzare dati.

Ogni nodo di una lista concatenata contiene un puntatore all'elemento successivo nella lista. Dobbiamo quindi scorrere la lista sequenzialmente per trovare un elemento nella lista. La skip_list velocizza l'operazione di ricerca creando delle "vie espresse" che permettono di saltare parte della lista durante l'operazione di ricerca. Questo è possibile perché ogni nodo della skip_list contiene non solo un singolo puntatore al prossimo elemento della lista, ma un array di puntatori che ci permettono di saltare a diversi punti seguenti nella lista. Un esempio di questo schema è rappresentato nella seguente figura:



Si implementi quindi una libreria che realizza la struttura dati skip_list. L'implementazione deve essere generica per quanto riguarda il tipo dei dati memorizzati nella struttura.

La libreria deve anche includere delle funzioni che permettono di creare una skip_list vuota e cancellare completamente una skip_list esistente. Quest'ultima operazione, in particolare, deve liberare correttamente tutta la memoria allocata per la skip_list.

2.2 SCELTE IMPLEMENTATIVE

Per la lettura del file “dictionary.txt” si è deciso di definire il suo filepath nel seguente modo:

```
#define DICTIONARY “../resources/dictionary.txt”
```

Per la lettura del file “correctme.txt” si è deciso di definire il suo filepath nel seguente modo:

```
#define CORRECTME “../resources/correctme.txt”
```

Si è definita la lunghezza del file “dictionary.txt” nel seguente modo:

```
#define DICT_LEN 661562
```

Si è definita la costante MAX_HEIGHT per definire il massimo numero di puntatori che possono esserci in un singolo nodo della skip_list. Essa é poi stata modificata più volte per effettuare i test presenti nel paragrafo successivo:

```
#define MAX_HEIGHT 12
```

2.3 SKIPLIST

2.3.1 Tempi di esecuzione dell'inserimento

MAX_HEIGHT	Intel i5-8250U	Intel i7-8550U
5	445.247000	350.306503
7	70.420000	49.040562
10	8.529000	5.574792
12	3.888000	2.095269

2.3.2 Valutazioni finali sulla complessità dell'algoritmo

Per quanto riguarda l'inserimento delle parole all'interno del dizionario, quindi il loro caricamento in memoria, la complessità temporale risulta inversamente proporzionale alla MAX_HEIGHT, che rappresenta il numero massimo di puntatori presenti in un singolo nodo della SkipList. Infatti, maggiore è il numero di puntatori di ciascun nodo, minore sarà il tempo necessario per effettuare le operazioni di ordinamento necessarie alla SkipList.