2017/2018

Progetto di Algoritmi e Strutture Dati

PISPISA MATTIA

MATRICOLA 132626

INDICE

ENUNCIATO DEL PROBLEMA

Data una stringa di sole lettere minuscole {a, ..., z} (può anche essere vuota) [1], creare un albero ***T(a)*** che rispetti le seguenti condizioni:

- La radice è la prima lettera della stringa
- Il sottoalbero di sinistra contiene le lettere che in ordine lessicografico sono precedenti o uguali al nodo padre
- Il sottoalbero di destra contiene le lettere che in ordine lessicografico sono successive al nodo padre

Successivamente bisogna calcolare la lettera (o una delle lettere) che nella stringa occorre il minor numero di volte[2] e il maggior numero di volte. Il numero di occorrenze di queste lettere le chiameremo rispettivamente ***m*** ed ***M.***

Esempio:

Se la stringa data al programma è "mattia" ***m*** sarà uguale a 0 corrispondente a una qualsiasi delle lettere dell'alfabeto non presenti nella parola (esempio la 'c' e la 'v') mentre ***M*** sarà 2 e corrisponderà al numero di ripetizioni della 't' e della 'a'.

Il problema prevedere di creare un grafo G = (V, E) a partire dalla stringa ***a*** data da linea di comando tale che:

- I vertici rappresentino le foglie dell'albero ***T(a).***
- Un arco tra due foglie sussiste soltanto se esiste un cammino in ***T(a)*** di lunghezza (numero di archi) compresa tra ***m*** e ***M.***

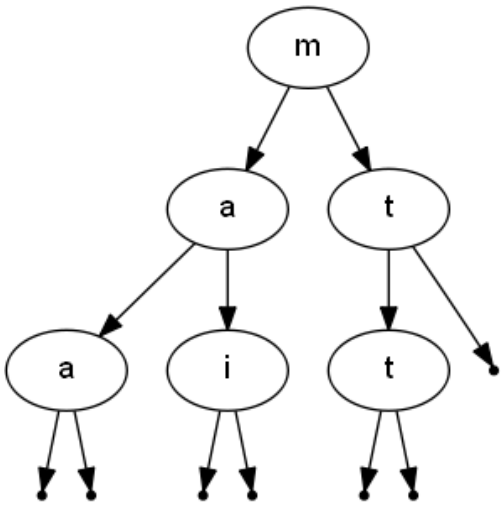
L'output del programma verrà visualizzato sullo standard output e sarà suddiviso in due parti:

- La prima riga conterrà la cardinalità delle foglie e il numero di archi del grafo G separati da uno spazio.
- Le successive righe rappresenteranno il grafo G = (V, E) in formato dot.

[1] Si suppone che l'input venga dato corretto quindi che non vi sia bisogno di fare ulteriori controlli sugli argomenti

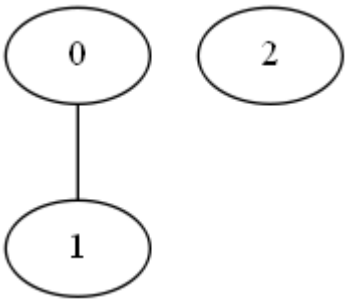
[2] Se una lettera non occorre nella stringa sicuramente questa sarà la lettera con occorrenza minore ossia 0

Prendendo come riferimento la stringa “mattia” l’albero T(mattia) sarà:



Ed il rispettivo Output sarà:

```
3 1
graph G {
a [label = "0"];
i [label = "1"];
t [label = "2"];
a -- i
}
```



(Visualizzazione su graphviz del grafo G)

COMPILAZIONE DEL PROGRAMMA

Il programma è stato sviluppato in C ed è suddiviso in 4 file:

- *Progetto.c* contenente il "main"
- *TreeChar.c* contenente le funzioni che lavoro sull'albero
- *TreeChar.h* il file di intestazione di *treeChar*
- *Makefile* si occupa di gestire la compilazione dei precedenti file

Il Progetto si trova all'interno della cartella "Esercizio".
Invocando il comando make[1] dentro la cartella "Esercizio" viene interpretato il Makefile e compilato il programma.
Compilato il programma sarà sufficiente eseguire l'applicativo "algorithm"

La precedente compilazione del programma è uguale per il software del calcolo dei tempi residente nella cartella "TempiEsecuzione" tranne per il nome dell'applicativo che è "time".

[1]Se al comando make l'output è "make command not found" molto probabilmente make non è stato installato sul dispositivo. Per installarlo digitare da linea di comando: sudo apt-get install make.

ANALISI DELL'ALGORITMO

In questo paragrafo verrà discussa l'idea utilizzata per risolvere il problema mentre la spiegazione delle singole funzioni verranno date nel capitolo successivo.

Data la stringa *a* viene creato l'albero *T(a)* e un array di interi *Vet(a)*, ogni locazione di *Vet(a)* corrisponderà al numero di occorrenze di una specifica lettera.

Sfruttando *Vet(a)* è stato possibile determinare il valore più grande e quello più piccolo delle occorrenze delle lettere. In questo modo è stato ottenuto ***m*** ed ***M*** (definiti nell'introduzione).

Per determinare le foglie è stato sufficiente scorrere l'albero *T(a)* fino a trovare dei nodi con i figli destro e sinistro a *NIL* e salvarli in memoria per poterli riutilizzare successivamente.

Per determinare l'esistenza di un arco tra i vertici rappresentati le foglie è bastato prendere a due a due le foglie e risalire i nodi parente finché:

1. Non è stato raggiunto il limite m.
2. E' stato superato il limite M.
3. E' stato trovato lo stesso parente.

Nel caso in cui le foglie non fossero alla stessa altezza viene fatta prima risalire quella più in basso (altezza maggiore) per portarla allo stesso livello dell'altra e solo successivamente le due foglie vengono fatte salire insieme.

ANALISI DELLE FUNZIONI

Le funzioni verranno discusso con le immagini relative all'applicativo.

File ***Progetto.c***

- *main (int argc, char **argv);*

  La funzione main riceve gli argomenti da linea di comando, *argc* rappresenta il numero di parametri ricevuti, nel nostro caso sempre 2 (a parte il caso della stringa vuota) e ***argv*, un array di stringhe contenente nella seconda posizione la parola da analizzare.

```
int lenString = 0;
int leaves = 0;
if(argc > 1){
    lenString = strlen(argv[1]);
    leaves = 1;
}
int strNumChar[26] = {0};
int value;
struct charNode *treeChar = NULL;
```

Dato che come input possiamo avere la stringa vuota viene controllato se abbiamo ricevuto in input uno o due parametri. Se ne abbiamo ricevuto uno solo vuol dire che non c'è alcuna stringa da analizzare altrimenti vengono impostate le variabili:

- *lenString* rappresentante la lunghezza della stringa.
- *leaves* corrispondente al numero di foglie nell'albero. Viene impostata ad uno in quanto c'è al minimo una foglia la radice.

Successivamente viene definita la struttura *charNode (pag. 9)* utilizzata per rappresentare la struttura dei nodi.

```
for(int i = 0; i < lenString; i++){
    treeChar = treeCharInsert(treeChar,argv[1][i],0,&leaves);
    value = ((int) argv[1][i])-97;
    strNumChar[value] ++;
}
```

Per creare l'albero viene utilizzato un ciclo for che scandisce tutte le lettere della stringa una per volta e per ognuna di esse viene chiamata la funzione ***treeCharInsert*** che le inserisce nella corretta posizione dentro l'albero. Il parametro *leaves* viene passato per puntatore in modo da poter preservare anche nella funzione "madre" il suo valore.

Ogni lettera viene convertita in un numero compreso da 0 e 25 in modo da avere un indice per l'array *strNumChar* utilizzato per contare le occorrenza di quest'ultime.

```
int maxCharacterRepetitions = 0;
for(int i = 0; i < 26; i++){
    if(maxCharacterRepetitions < strNumChar[i]){
        maxCharacterRepetitions = strNumChar[i];
    }
}

int minCharacterRepetitions = maxCharacterRepetitions;
for(int i = 0; i < 26; i++){
    if(minCharacterRepetitions > strNumChar[i]){
        minCharacterRepetitions = strNumChar[i];
    }
}
```

Per determinare qual è la lettera che occorre più volte nella stringa viene utilizzato un for per scandire il vettore *strNumChar*. Se la posizione corrente ha un valore più alto rispetto alla variabile *maxCharacterRepetitions* allora viene sovrascritto il valore del parametro.
Una volta determinato il numero massimo di occorrenze viene utilizzato quest'ultimo valore per riconfrontare il vettore e determinare il numero più piccolo presente nel vettore.

```
struct charNode *vectorNode = calloc(leaves,sizeof(struct charNode));
for(int i = 0; i < leaves; i++){
    vectorNode[i].data = ' ';
}
createVectorOfLeaves(treeChar,vectorNode);
```

Dato che le foglie dell'albero *T(a)* servono per determinare il numero di archi e quali coppie di vertici ne hanno uno viene allocata la memoria per salvare tutte le foglie di struttura *charNode*.
Successivamente con la funzione ***createVectorOfLeaves*** in ogni locazione viene salvata una foglia.

Calloc è una variante di *malloc*. La funzione *malloc* accetta come argomento il numero di byte di memoria di cui si ha bisogno. Alloca una zona di memoria contigua della dimensione richiesta e restituisce un puntatore all'inizio di tale zona. Il tipo di ritorno è un puntatore ad un tipo qualsiasi.
Calloc a differenza di *malloc* alloca memoria già azzerata.

```
int size = ((leaves - 1)  * (leaves))/2;
char *outputSecondLine = malloc(size * 7);
```

Visto che il numero di archi e la visualizzazione del grafo vengono stampati sullo standard output in due momenti diversi dell'esecuzione ma la funzione che li determina è pressoché la stessa al posto di eseguire il codice due volte si è pensato ti richiamare la funzione una volta sola e di conservare la visualizzazione degli archi in una stringa allocata in modo da contenere un upperbound del numero massimo di archi possibile con n foglie ossia: $(foglie - 1) * (foglie)$ che verrà stampata a video solo successivamente alla visualizzazione del numero delle foglie e degli archi.
Dalle osservazioni (pag. 17) si nota che il numero massimo di archi è 301.

```
printf("%d ",leaves);
int countArch = 0;
int k= 0;
for(int i = 0; i < leaves-1;i++){
    for(int j = i+1; j < leaves;j++){
        k = compareDistanceLeaves(&vectorNode[i],&vectorNode[j],minCharacterRepetitions,maxCharacterRepetitions);
        countArch += k;

        if(k == TRUE){
            sprintf(outputSecondLine + strlen(outputSecondLine),"%c -- %c\n",vectorNode[i].data,vectorNode[j].data);
        }
    }
}
printf("%d\n",countArch);
```

Viene stampato sullo standard input il numero di foglie. Successivamente per ogni coppia di foglie viene determinato con la funzione ***compareDistanceLeaves*** se la loro distanza soddisfa i requisiti per avere un arco. Se esiste viene incrementato il conteggio di questi nella variabile *countArch* e viene salvata la presenza dell'arco (in formato dot) nella stringa *OutputSecondLine* precedentemente allocata.

```
printf("graph G {\n");

for (int i = 0; i < leaves; i++){
    printf("%c [label = \"%d\"];\n",vectorNode[i].data,i);
}

printf("%s",outputSecondLine);

printf("}\n");
```

Il grafo viene stampato in formato dot:

Per ogni foglia viene dato un identificativo corrispondente alla lettera e un nome numerico. Successivamente viene stampata sullo standard output la stringa *outputSecondLine* dove risiedono gli archi in formato dot.

```
free(vectorNode);
free(outputSecondLine);
destroy(treeChar);
```

A fine programma viene liberata la memoria in quanto una zona di memoria allocata dinamicamente se non viene esplicitamente liberata dal programma resterà assegnata al processo fino alla fine.

Questo è molto importante soprattutto per il programma del calcolo dei tempi dato che la funzione main verrà chiamata innumerevoli volte prima che il programma termini. Questo provocherà un allocamento eccessivo di memoria ed un corrispettivo aumento dei tempi.

File ***treeChar.h***

- Questo file rappresenta l'header di treeChar.c.

Per semplicità d'uso vengono definite TRUE e FALSE come 1 e 0.

```c
#include <stdio.h>
#define TRUE 1
#define FALSE 0

struct charNode {
    char data;
    struct charNode *parent;
    struct charNode *left;
    struct charNode *right;
    int height;
    int isLeaf;
};

struct charNode *treeCharInsert(struct charNode *treeChar,char x,int i,int *leaves);

struct charNode *treeCharCreate(char x,int i);

void createVectorOfLeaves(struct charNode *treeChar,struct charNode *vectorNode);

int compareDistanceLeaves(struct charNode *firstLeaf, struct charNode *secondLeaf,int min,int max);

void destroy(struct charNode *treeChar);
```

Viene definita la struttura *charNode* che compone qualsiasi nodo dell'albero *T(a)* ossia:

- *data*, una lettera della stringa.
- *parent*, il puntatore al nodo padre.
- *left* e *right*, i puntatori al figlio destro e sinistro.
- *height*, l'altezza del nodo dalla radice dell'albero.
- *isLeaf*, per determinare se un nodo è una foglia.

Successivamente ci sono tutte le definizioni delle funzioni utilizzate per analizzare l'albero.

File ***treeChar.c***

- treeCharCreate (char x, int i);

  La funzione ha come parametri:
  - *x,* un carattere della stringa ***a.***
  - *i,* un intero rappresentate l'altezza del nodo rispetto alla radice.

  Il valore di ritorno è un puntatore alla struttura *charNode*.

```
struct charNode *treeCharCreate(char x,int i){
    struct charNode *nodeChar = malloc(sizeof(struct charNode));
    nodeChar->data = x;
    nodeChar->parent = NULL;
    nodeChar->left = NULL;
    nodeChar->right = NULL;
    nodeChar->height = i;
    nodeChar->isLeaf = TRUE;

    return nodeChar;
}
```

  Ogni volta che viene chiamata questa funzione alloca la memoria per contenere la struttura di un nodo *charNode* e definisce i parametri di quest'ultimo.

- createVectorOfLeave(struct charNode *treeChar, struct charNode *vectorNode);

  La funzione ha come parametri:
  - *treeChar,* il puntatore alla radice dell'albero T(a).
  - *vectorNode,* il puntatore ad un array di charNode.

```
void createVectorOfLeave(struct charNode *treeChar, struct charNode *vectorNode){

    if(treeChar == NULL)
        return;

    if(treeChar->left == NULL && treeChar->right == NULL){
        int i = 0;
        while(TRUE){
            if(vectorNode[i].data == ' '){
                vectorNode[i] = *treeChar;
                break;
            }
            i++;
        }
        return;
    }

    createVectorOfLeave(treeChar->left,vectorNode);
    createVectorOfLeave(treeChar->right,vectorNode);
}
```

  Questa funzione scorre ricorsivamente l'albero e se trova una foglia la salva nel vettore puntato da *vectorNode.*

- treeCharInsert (struct charNode *treeChar, char x, int i, int *leaves);

  La funzione ha come parametri:
  - *treeChar*, la parte dell'albero T(a) costruito precedentemente richiamando la funzione ***treeCharInsert***
  - *x,* un valido carattere da inserire nel nodo che vogliamo creare
  - *i,* l'altezza del nodo
  - *leaves*, puntatore all'intero che conta il numero di foglie presenti nell'albero.

  Il valore di ritorno è la struttura dell'albero ***T(a)*** modificata con un nodo in più.

```
struct charNode *treeCharInsert(struct charNode *treeChar,char x,int i,int *leaves){
    if(treeChar == NULL){
        return treeCharCreate(x,i);
    }

    if(x <= treeChar->data){
        treeChar->left = treeCharInsert(treeChar->left,x,i+1,leaves);
        treeChar->left->parent = treeChar;
        if(treeChar->isLeaf == TRUE){
            treeChar->isLeaf = FALSE;
        }else if (treeChar->left->isLeaf == TRUE){
            *leaves += 1;
        }
    }else{
        treeChar->right = treeCharInsert(treeChar->right,x,i+1,leaves);
        treeChar->right->parent = treeChar;
        if(treeChar->isLeaf == TRUE){
            treeChar->isLeaf = FALSE;
        }else if (treeChar->right->isLeaf == TRUE){
            *leaves += 1;
        }
    }


    return treeChar;
}
```

Quando questa funzione viene richiamta scorre l'albero scendendo sul figlio destro o sinistro in base all'ordine lessicografico del carattere da inserire rispetto a quello del nodo in esame. Questo viene ripetuto fino a raggiungere il figlio di una foglia ossia un puntatore NULL. A questo punto viene richiamata la funzione ***treeCharCreate*** per creare il nodo.
In questa funzione vengono anche contate le foglie nell'albero in modo da non dover successivamente ripercorrere l'albero per contarle.
L'idea per contare il numero di foglie nell'albero è la seguente:
Se aggiungo un nuovo nodo sotto ad un altro che aveva i puntatori destro e sinistro a NIL allora il numero di foglie è lo stesso. Se aggiungo un nodo come figlio destro ad uno nodo che già precedentemente aveva il figlio sinistro diverso da NIL allora ho creato un nuovo percorso e quindi ci sarà una nuova foglia in ***T(a)***. Si applica lo stesso ragionamento per l'inserimento di un nodo nel figlio sinistro ad un nodo padre con il figlio destro diverso da NIL.

- compareDistanceLeaves (struct charNode *firstLeave, struct charNode *secondLeave,int min,int max);

  La funzione ha come parametri:

  - due nodi *charNode.*
  - ***m***, ***M.***

  Il valore di ritorno è 1 se esiste un arco tra i due nodi altrimenti 0

```
int compareDistanceLeaves(struct charNode *firstLeaf, struct charNode *secondLeaf,int min,int max){
    int count = 0;
    if(firstLeaf->height > secondLeaf->height){
        while( firstLeaf->height != secondLeaf->height && count <= max){
            firstLeaf = firstLeaf->parent;
            count++;
        }
    }else if( firstLeaf->height < secondLeaf->height){
        while(firstLeaf->height != secondLeaf->height  && count <= max){
            secondLeaf = secondLeaf->parent;
            count++;
        }
    }

    while (count <= max && firstLeaf->data != secondLeaf->data){
        count +=2;
        firstLeaf = firstLeaf->parent;
        secondLeaf = secondLeaf->parent;
    }

    if( count >= min && count <= max && firstLeaf->data == secondLeaf->data){
        return TRUE;
    }else{
        return FALSE;
    }
}
```

Questa funzione confronta l'altezza dei due nodi. Se uno ha altezza maggiore dell'altro viene spostato il puntatore al nodo padre fino a che non si ottiene un altezza uguale all'altro nodo. A questo punto se il cammino ha ancora il peso dentro l'intervallo ***[m,M]*** vengono confrontati i caratteri dei nodi padre dei due nodi fintanto che o non si arriva alla radice (visto che questa è unica per forza è uguale per i due nodi) o si è superato il limite superiore dell'intervallo (***M***).

Se tutte le condizioni per avere un arco tra due vertici(le foglie di ***T(a)***) sono state rispettate viene ritornato 1 (TRUE) altrimenti 0 (FALSE).

- destroy (struct charNode *treeChar);

  La funzione ha come parametro un nodo dell'albero.

```
void destroy(struct charNode *treeChar){

    if(treeChar == NULL)
        return;

    struct charNode *l = treeChar->left;
    struct charNode *r = treeChar->right;
    free(treeChar);
    destroy(l);
    destroy(r);
}
```

Cancella ricorsivamente tutte le foglie dell'albero in modo da liberare la memoria.

CALCOLO DEI TEMPI

TEMPI TEORICI

n → numeri di nodi nell'albero/numero di caratteri nella stringa.

f → numero di foglie nell'albero $\begin{cases} \Omega(1) & l'albero \text{ è } sbilanciato \\ O\left(\frac{n}{2}\right) = O(n) & l'albero \text{ è } bilanciato \end{cases}$

ma il numero massimo di foglie possibili in *T(a)* sono 26 quindi:

f → $O(26) = \boldsymbol{\Theta(1)}$

- *treeCharCreate* → $\boldsymbol{\Theta(1)}$
- *treeCharInsert* → $\begin{cases} \boldsymbol{\Omega(logn)} & l'albero \text{ è } bilanciato \\ \boldsymbol{O(n)} & l'albero \text{ è } sbilanciato \end{cases}$
- *createVectorOfLeave* → $\Theta(n) + O(f * f) = \boldsymbol{\Theta(n)}$

  Visito al più una volta tutti i nodi dell'albero e per ogni foglia inserisco il nodo in una locazione vuota di *vectorNode* di lunghezza massima f.

- *compareDistanceLeaves* → $\begin{cases} \boldsymbol{\Omega(1)} & nodi\ figli\ dello\ stesso\ padre \\ \boldsymbol{O(M)} & massima\ distanza\ accettabile\ per\ un\ arco \end{cases}$

- **Costruzione dell'albero** (fa uso di *treeCharInsert)* →

$$\Theta(n) * treeCharInsert = \begin{cases} \boldsymbol{\Omega(n * logn)} & l'albero \text{ è } bilanciato \\ \boldsymbol{O(n^2)} & l'albero \text{ è } sbilanciato \end{cases}$$

- **Ricerca massima/minima occorenza** → $\Theta(26) = \boldsymbol{\Theta(1)}$

- **Definizione degli archi** (fa uso di compareDistanceLeaves) →

$$\frac{f*(f+1)}{2} * compareDistanceLeaves = \begin{cases} \Omega(f^2) \\ O(f^2 * M) \end{cases} = \begin{cases} \boldsymbol{\Omega(1)} \\ \boldsymbol{O(M)} \end{cases}$$

La complessità computazionale teorica sarà:

- Nel caso migliore → $\Omega(n * logn + 1 + n + 1) = \boldsymbol{\Omega(n * logn)}$
- Nel caso peggiore → $O(n^2 + 1 + n + M) = \boldsymbol{O(n^2)}$

Conclusione:
Secondo l'analisi teorica il costo computazionale ricade tutto sulla costruzione dell'albero sia nel caso migliore che nel caso peggiore.

DIMOSTRAZIONI DELLA COMPLESSITA'

Dato che *treeCharInsert* e *createVectorOfLeaves* sono funzioni ricorsive dimostro che la loro complessità nel caso peggiore è effettivamente quella riportata nei tempi teorici.

- ***treeCharInsert***

$$T(n) = \begin{cases} \theta(1) & se\ n = 0 \\ \theta(1) + T(n-1) & se\ n > 0 \end{cases}$$

Questo è il caso in cui l'albero è tutto sbilanciato a destra o a sinistra e il nodo è da inserire proprio nel ramo sbilanciato.

$$T(n) = \sum_{i=0}^{n} \theta(1)$$

θ(1) è una costante K quindi:

$$\sum_{i=0}^{n} K = K * \sum_{i=1}^{n} 1 \ = Kn\ \epsilon\ \theta(n)$$

- ***createVectorOfLeaves***

$$T(n) = \begin{cases} \theta(1) & se\ n = 0 \\ O(f) + T(m) + T(n-m-1) & se\ n > 0 \end{cases}$$

O(f): in ogni ricorsione eseguo un ciclo while che scorre al massimo *f* locazioni, *f* è al massimo 26, considero *O(f) = ϑ(1).*

$$T(n) = \begin{cases} \theta(1) & se\ n = 0 \\ \theta(1) + T(m) + T(n-m-1) & se\ n > 0 \end{cases}$$

Dimostrazione per induzione, nel caso peggiore T(n) = θ(n).

- Tolgo la complessità in T(n)

$$T(n) = \begin{cases} a & se\ n = 0 \\ b + T(m) + T(n-m-1) & se\ n > 0 \end{cases}$$

- T(n) = O(n)

Tesi: T(n) = O(n)
Tesi: $\exists c > 0, \exists\ \overline{n},\ \forall\ n\ \geq \overline{n} \quad T(n)\ \leq c * n$

Base: n = 0

$$T(0) = a \leq c * 0$$ Non va bene

Base: n = 1

$$T(1) = b + T(0) + T(0) = b + 2a \leq_{ts} c * 1$$

$$c \geq b + 2a$$

Passo:

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

m < n per ipotesi induttiva $\leq c * m$

n-m-1 < n per ipotesi induttiva $\leq c * (n - m - 1)$

$$T(n) = b + T(m) + T(n - m - 1) \leq b + cm + c(n - m - 1)$$

$$T(n) \leq b + cn - c \leq_{ts} cn$$

$$c \geq b$$

Conclusione: $T(n) = O(n)$

TEMPI EFFETTIVI

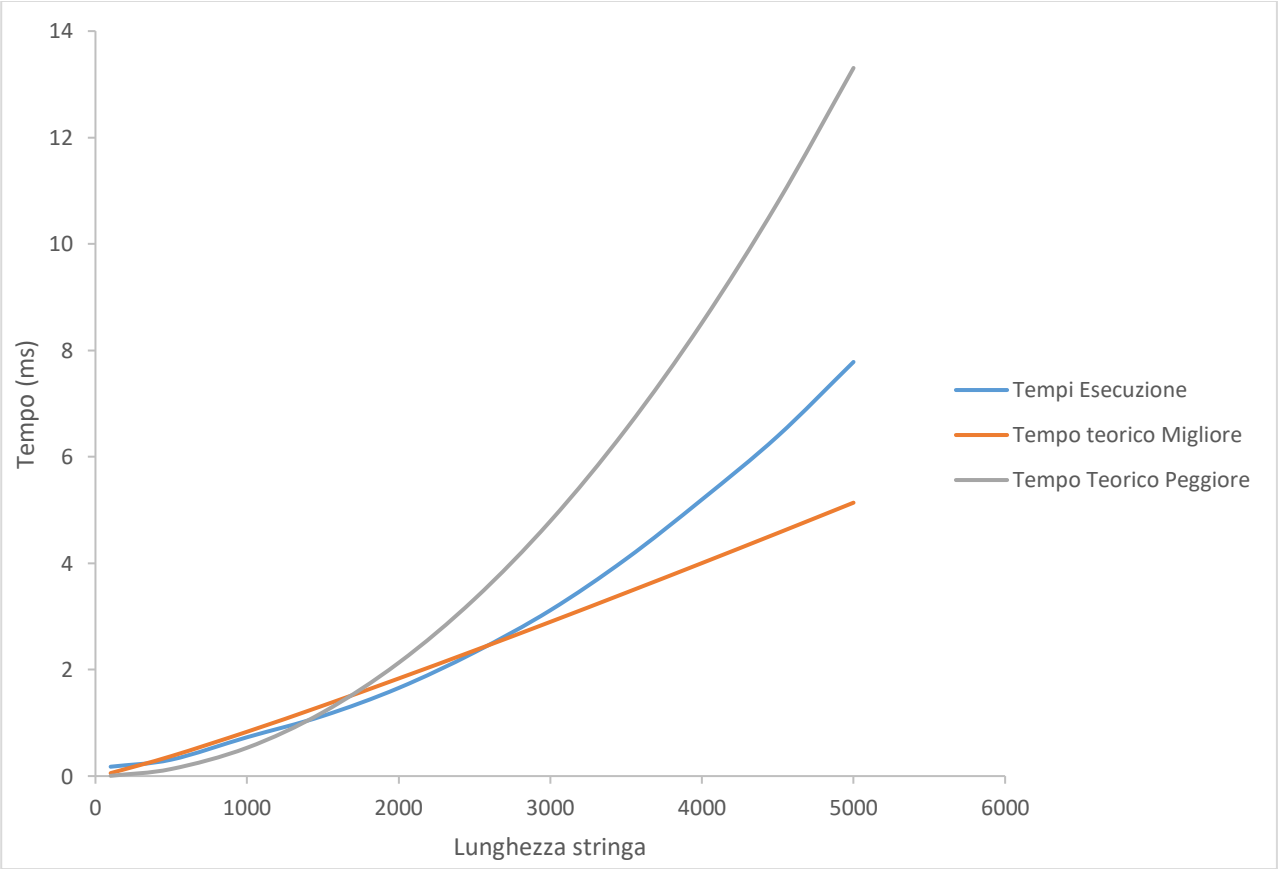
Tramite il programma per l'analisi dei tempi (directory "Tempi Esecuzione") basato sugli algoritmi presentati al laboratorio di Algoritmi e Strutture Dati e mantenendo i parametri simili a quelli riportati sulle dispense sono stati analizzati gli effettivi tempi di esecuzione del codice sulle diverse lunghezze della stringa.

Di seguito vengono riportati i tempi analizzati.

Lunghezza della Stringa	Tempi in Secondi (s)
100	0.000176
500	0.000309
1000	0.000730
1500	0.001130
2000	0.001656
2500	0.002325
3000	0.003116
3500	0.004081
4000	0.005197
4500	0.006389
5000	0.007783

Avendo un analisi teorica ed effettiva dei tempi si può ora andare a stimare visivamente di quanto quelli teorici si scostano da quelli pratici.

A seguire viene delineato un grafico con i tempi analizzati.



OSSERVAZIONI

- Dal grafico riportato precedentemente è stato verificato quello che mi aspettavo ossia i valori ottenuti con l'analisi del tempo sono nell'intervallo tra i tempi teorici migliori e quelli peggiori.
  Si può notare come con stringhe di lunghezza inferiore a 2500 caratteri l'andamento è simile a quello n*logn mentre all'aumentare della dimensione i tempi calcolati si avvicinano ad un andamento quadratico.

- Il numero massimo di foglie nell'albero è 26.
  L'alfabeto da cui attingiamo per creare i nodi dell'albero e composto da 26 caratteri diversi, ogni qualvolta troviamo un carattere uguale ad uno già precedentemente inserito questo verrà sempre giustapposto nel suo sottoalbero destro. Se per ipotesi avessimo due foglie uguali f1 e f2 (in ordine prima è stata inserita f1 e poi f2) deduremmo che f2 non è scesa ricorsivamente nel sottalbero di f1, questo porta ad un assurdo.

- Massimo numero di archi generabile.
  Il caso limite analizzato è la stringa:
  "aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyz".
  Con questa stringa si ottiene un grafo con 26 vertici e 301 archi.
  Questa stringa ha una foglia per ogni lettera dell'alfabeto, la definizione per l'esistenza di un arco è di avere un cammino tra due foglie compreso tra *[m,M]*. Dato che la lettera "y" è ripetuta più della cardinalità dell'alfabeto *M* sarà più grande di ogni cammino tra due foglie.
  Perché ripetere propio la "y"?
  Per avere una foglia ogni lettera (eccetto la "z") deve essere ripetuta due volte, la "z" essendo l'ultima lettera dell'alfabeto è gia una foglia senza doverla ripetere quindi, la lunghezza del cammino dalla "y" foglia alla prima "y" inserita nell'albero è di *M-1* e l'unico cammino di lunghezza M è quello con la "z".
  Se si generasse questa tipologia di stringa ripetendo una lettera qualsiasi diversa dalla "y" es la "c" si otterrebbe che il cammino dalla "c" ad un'altra foglia è di minimo M+1 archi. Questo tipo di input avrà un numero massimo di archi pari a 300.