Documentazione semplice per Walnut

Beatrice Chiara Morgillo

November 2025

Indice

1 Istruzioni d'uso **2**

2 Le basi **3**
2.1 Comandi principali 4
2.1.1 eval 4
2.1.2 def 4
2.1.3 reg 5
2.1.4 macro 6
2.1.5 load 6
2.1.6 exit 6

3 Settimana 1 **7**
3.1 Giorno 1: Progettazione di un algoritmo base 7
3.2 Giorno 2: Testing dell'algoritmo 10
3.2.1 Esempi per cui Greedy e' peggio di una fattorizzazione ottimale 13
3.3 Giorno 3: Comprensione dei dati e dei risultati 15
3.4 Giorno 4: Performance e generatori di parole di Fibonacci e di Thue-Morse 16
3.4.1 Fibonacci 18
3.4.2 Thue-Morse 18
3.4.3 Performance 19
3.5 Giorno 5-7: Dimostrazioni con Walnut 19

4 Settimana 2 **24**
4.1 Giorno 1-3: Chiusura per Thue-Morse e Fibonacci 24

5 Tecniche di ottimizzazione dell'algoritmo Java **30**
5.1 Primi passi di ottimizzazione 30

Capitolo 1

Istruzioni d'uso

Questo file prende spunto soprattutto dalla documentazione (menzionata più avanti in questa pagina), in modo tale da rendere più comprensibile l'utilizzo del software. Per la mia esperienza ho trovato più semplice installare JDK su WSL (Windows Subsystem for Linux) in modo tale d'avviare più rapidamente il software Walnut, attualmente mi sto affidando alla documentazione (presente anche nel Github di Walnut: https://github.com/Walnut-Theorem-Prover/Walnut), altrimenti è presente su arxiv. Mi basta aprire la cartella su VSCode (Windows), avviare il comando `code .` nel terminale WSL per avviare VSCode su WSL e iniziare a usare Walnut, buildo prima il codice digitando nel terminale `./build.sh` e successivamente eseguo Walnut digitando `./walnut.sh`. All'avvio Walnut mostra:

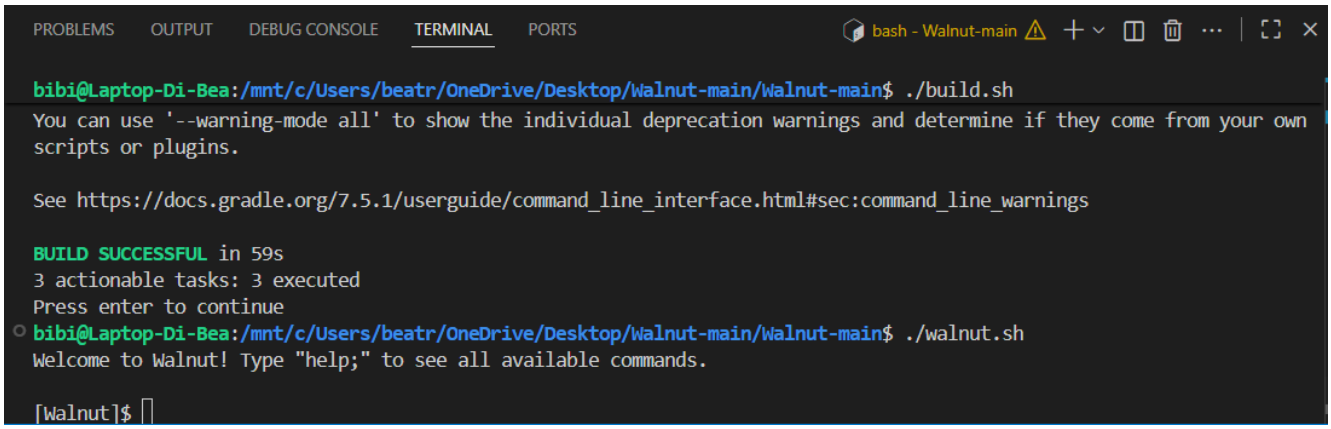

Figura 1.1: Visualizzazione di Walnut da terminale

Capitolo 2

Le basi

Walnut è un software gratuito, scritto in Java, progettato per "decidere" affermazioni logiche del primo ordine. In termini più semplici, puoi usarlo per fornire prove rigorose (o smentite) per affermazioni complesse che riguardano, ad esempio, la combinatoria di parole (sequenze di simboli), la teoria dei numeri e altre aree della matematica discreta.

Un esempio è l'affermazione usata come esempio dal sito stesso:

eval tm_has_overlap: $\exists n \geq 1 \wedge \forall j \ (1 \leq j \leq n \implies T[i+j] = T[i+n+j])$

Dove **eval tm_has_overlap:** sta a indicare il nome che avrà l'affermazione da valutare (in questo caso se la sequenza Thue-Morse presenta degli overlap, come: axaxa, dove a è una singola lettera mentre x può essere qualsiasi parola - possibilmente vuota).

Mentre $\exists n \geq 1 \wedge \forall j \ (1 \leq j \leq n \implies T[i+j] = T[i+n+j])$ è la rappresentazione formale (o matematica) di cosa si vuole decidere, per maggior precisione, Walnut usa E come rappresentazione di "esiste" ($\exists$) e A come rappresentazione di "per ogni" ($\forall$), l'affermazione logica si traduce dunque in: "Esistono i e n (con n $\geq$ 1) E per ogni j (con j $\leq$ n), vale che T[i+j] = T[i+n+j]?"

Il comando da scrivere in Walnut è il seguente:

eval tm_has_overlap "Ei,n n =1 & Aj (j=n) → T[i+j]=T[i+n+j]":

E come termina il comando dà risultati diversi, infatti:

- usando i due punti (:) dà come risultato un report breve dei passaggi principali
- usando il punto e virgola (;) esegue solo il comando, senza dare output specifici, solo true o false
- usando i doppi due punti (::) esegue il comando e fornisce un report completo di tutti i passaggi

Segue il risultato che dà chiedendo un report breve:

```
eval tm_has_overlap "Ei,n n>=1 & Aj (j<=n) => T[i+j]=T[i+n+j]":
n>=1:2 states - 3ms
 j<=n:2 states - 1ms
  T[(i+j)]=T[((i+n)+j)]:12 states - 5ms
   (j<=n=>T[(i+j)]=T[((i+n)+j)]):25 states - 7ms
    (A j (j<=n=>T[(i+j)]=T[((i+n)+j)])):1 states - 12ms
     (n>=1&(A j (j<=n=>T[(i+j)]=T[((i+n)+j)]))):1 states - 0ms
      (E i , n (n>=1&(A j (j<=n=>T[(i+j)]=T[((i+n)+j)])))):1 states - 1ms
Total computation time: 30ms.
_____
FALSE
```

Il risultato della computazione finirà nella cartella Result.

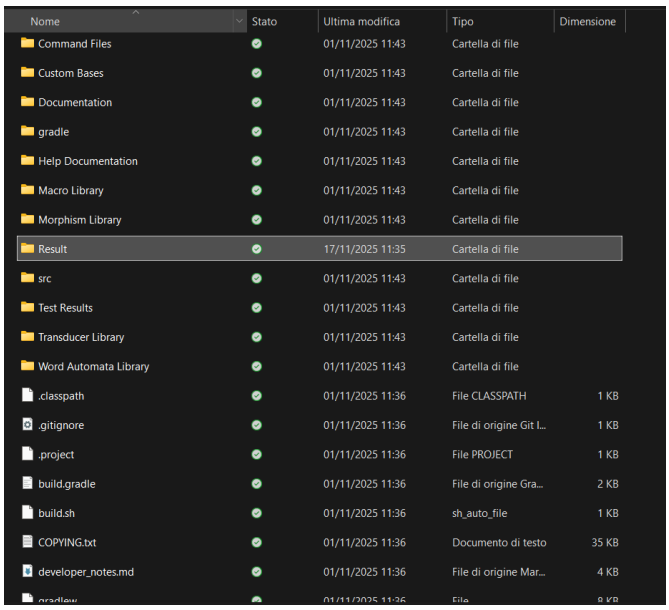

Figura 2.1: Posizionamento della cartella Result

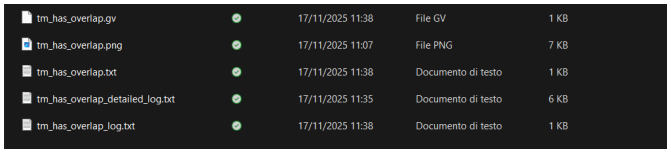

Figura 2.2: Contenuto della cartella Result

In realtà l'immagine png risultante (tm_has_overlap.png) è stata generata con un comando Linux per tradurre il contenuto di tm_has_overlap.gv (file in formato graphviz) in un'immagine:

```
#Innanzitutto si aggiorna o si verifica la presenza di graphviz
sudo apt update
sudo apt install graphviz -y

cd /Path-fino-a-Walnut/Walnut-main
dot -Tpng Result/tm_has_overlap.gv -o Result/tm_has_overlap.png
# Apre la cartella Windows sull'immagine: (apre Explorer nella cartella Result)
explorer.exe "$(wslpath -w Result/tm_has_overlap.png)"
```

Modificando un pochino il file .gv si ha il seguente risultato:

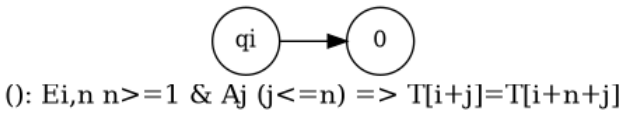

Figura 2.3: Risultato

Dove "qi" rappresenta uno stato fittizio (infatti è solo uno stato iniziale), che va nello stato "0", il quale non è nemmeno uno stato finale, quindi tutta l'affermazione è falsa.

Non c'è solo dot per converire un file .gv in un'immagine, si può fare anche al sito Graphviz Online

2.1 Comandi principali

2.1.1 eval

Il comando più importante tra tutti, come visto poco fa è utile a decidere e a valutare le affermazioni logiche proposte, ha la seguente sintassi:

```
eval <nome_output> "<predicato>"; #oppure si possono usare : o i :.
```

Il resto è stato spiegato nella sezione sovrastante.

2.1.2 def

Questo comando aiuta a definire un'automa, ha la seguente sintassi:

```
def <nome_automa> "<predicato>";
```

L'automa può essere richiamato anche in predicati futuri, venendo salvato nella cartella "<Path-to-Walnut/Walnut-main/Automata Library>" e risulterà nel file <nome_automa>.txt. L'automa generato verrà poi richiamato con $, un esempio è:

```
eval check_sum "Ea a>=8 & $sum10(b,a)";
```

Darà il seguente risultato:

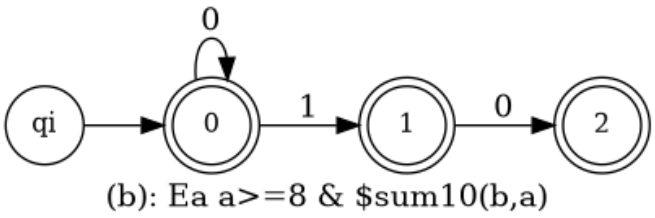

Figura 2.4: Risultato di check_sum

b può assumere i valori di 0,1 e 2, e questo risultato viene visto in binario, quindi 00, 01 e 10, il risultato, quindi, funziona come segue:

- **Caso** $a = 10 \implies b = 0$ **(Input binario:** 0**)**
  L'automa parte dallo stato iniziale q_i. Leggendo il bit 0, percorre la transizione $q_0 \xrightarrow{0} q_0$ (self-loop). Poiché q_0 è uno stato finale (doppio cerchio), la stringa viene accettata. Questo conferma che $b = 0$ è una soluzione valida.
- **Caso** $a = 9 \implies b = 1$ **(Input binario:** 1**)**
  L'automa parte da q_0. Leggendo il bit 1, esegue la transizione $q_0 \xrightarrow{1} q_1$. Lo stato q_1 è uno stato finale, quindi la stringa viene accettata, confermando $b = 1$.
- **Caso** $a = 8 \implies b = 2$ **(Input binario:** 10**)**
  Questo è un percorso a due passi.

  1. Dallo stato q_0, l'automa legge il primo bit (1) ed esegue la transizione $q_0 \xrightarrow{1} q_1$.
  2. Dallo stato q_1, legge il secondo bit (0) ed esegue la transizione $q_1 \xrightarrow{0} q_2$.

  Poiché q_2 è uno stato finale, la sequenza "10" è accettata.

Qualsiasi altra sequenza di bit (ad esempio 11 per il numero 3) non porterebbe a uno stato finale.

2.1.3 reg

A volte è difficile definire un insieme di numeri tramite l'aritmetica (es. le potenze di 2). Il comando reg permette di costruire un automa partendo direttamente da un'espressione regolare.

Ha la seguente sintassi:

```
reg <nome_regex> <alfabeto_di_riferimento> "<regex>";
```

Eseguendolo crea un risultato sempre in "<Path-to-Walnut/Walnut-main/Automata Library>" e risulterà nel file "nome_regex.txt", richiamabile sempre tramite $ nel comando eval. Un esempio è:

```
reg power2 msd_2 "0*10*"; #msd_2 = alfabeto delle cifre piu' significative binarie
eval check "a<20 & $power2(a)";
```

Questo comando calcola tutte le potenze di 2 che ci sono da 0 a 20, le quali sono: a=1, a=2, a=4, a=8 e a=16, in binario sono: a=10000, a=01000, a=00100, a=00010 e a=00001, l'automa avrà il seguente aspetto:

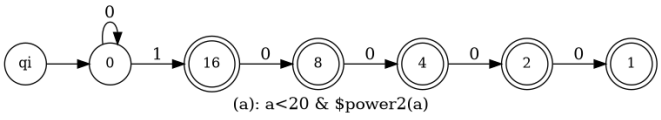

Figura 2.5: Automa per check

2.1.4 macro

Questo comando permette di definire un template di testo per cui fa uso di segnaposto (segnati col simbolo di percentuale %) e all'utilizzo vengono sostituite con i nomi effettivi delle variabili, per richiamare una macro si fa uso dell'#. Riprendendo sum10(a,b) definito prima si può scrivere una macro che dia come risultato tutte le possibili combinazioni (in binario) di a e b, in modo tale che sommati facciano 10 e su Walnut basterà scrivere quanto segue:

```
def sum10 "x+y=10";
macro somma "$sum10(%0,%1)$";
eval prova "#somma(a,b)";
```

Il risultato sarà:

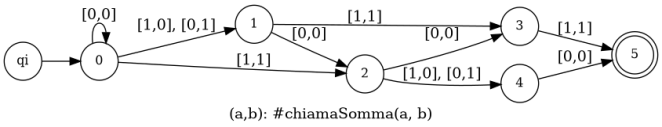

Figura 2.6: Risultato usando la macro per sum10

2.1.5 load

Nel caso in cui ci fossero delle affermazioni lunghe e complicate da scrivere si possono scrivere in un file .txt che verrà poi caricato su Walnut, il file dovrà trovarsi tassativamente nella cartella "<Path-to-Walnut/Walnut-main/Command Files>".

2.1.6 exit

Ovviamente per uscire dal software basta digitare `exit;` e termina l'esecuzione.

Capitolo 3

Settimana 1

3.1 Giorno 1: Progettazione di un algoritmo base

Voglio effettuare lo studio di parole chiuse, lo faccio usando un algoritmo scritto in Java che segue la teoria sulle parole chiuse.

Queste ultime devono rispettare le seguenti clausole:

- se è di lunghezza ≤ 1, allora è chiusa
- contiene un fattore che si ripete sia come prefisso sia come suffisso (bordo)

Un atteggiamento abbastanza ingenuo è come il seguente:

```java
public static boolean isClosed(String w) {
    int n = w.length();

    // Caso base: lunghezza <= 1 sono sempre chiuse]
    if (n <= 1) return true;

    // 1. Trova lenB (lunghezza del bordo pi lungo)
    // Un bordo  un prefisso che  anche suffisso (proprio)
    String border = "";
    int lenB = 0;

    // Cerchiamo il bordo pi lungo partendo da n-1 gi fino a 1
    for (int i = n - 1; i > 0; i--) {
        String pre = w.substring(0, i);
        String suf = w.substring(n - i);
        if (pre.equals(suf)) {
            border = pre;
            lenB = i;
            break; //  stato trovato il pi lungo
        }
    }

    // 2. Se lenB == 0 -> false (la parola  aperta)
    if (lenB == 0) return false;

    // 3. Cerca occorrenze: w.indexOf(bordo, 1)
    // Cerchiamo il bordo partendo dall'indice 1 (quindi saltando il prefisso iniziale, cos
        da non restituire 0, in quanto il bordo  proprio all'inizio della parola)
    int firstInternalIndex = w.indexOf(border, 1);

    // 4. Logica di verifica
    // L'ultima occorrenza valida (il suffisso) inizia all'indice: n - lenB
    // Se troviamo un'occorrenza PRIMA di quella finale, c' un'occorrenza interna.
    if (firstInternalIndex < n - lenB) {
        return false; // Trovato occorrenza interna -> aperta
    }

    // Se arriviamo qui, il bordo appare solo come prefisso e suffisso.
    return true; // -> chiusa
}
```

Per effettuare la fattorizzazione si fa uso di un approccio greedy, per cui s'inizia dalla prima lettera della parola e si calcola la lunghezza massima del fattore chiuso più lungo.

```
44    public static boolean isClosed(String w) {
45       int n = w.length();
46       if (n <= 1) return true;
47
48       String border = "";
49       int lenB = 0;
50
51       for (int i = n - 1; i > 0; i--) {
52          if (w.startsWith(w.substring(n - i))) {
53             border = w.substring(0, i);
54             lenB = i;
55             break;
56          }
57       }
58
59       if (lenB == 0) return false;
60
61       int suffixStartIndex = n - lenB;
62       int foundIndex = w.indexOf(border, 1);
63
64       if (foundIndex < suffixStartIndex) {
65          return false; // Ha occorrenze interne
66       }
67       return true;
68    }
69
70    public static int minimalFactorizationDP(String text) {
71       int n = text.length();
72       // dp[i] = numero minimo di fattori chiusi per il prefisso di lunghezza i
73       int[] dp = new int[n + 1];
74       Arrays.fill(dp, Integer.MAX_VALUE);
75       dp[0] = 0; // Stringa vuota ha 0 fattori
76
77       for (int i = 1; i <= n; i++) {
78          for (int j = 0; j < i; j++) {
79             String sub = text.substring(j, i);
80             if (isClosed(sub)) {
81                if (dp[j] != Integer.MAX_VALUE) {
82                   dp[i] = Math.min(dp[i], dp[j] + 1);
83                }
84             }
85          }
86       }
87       return dp[n];
88    }
89
90    public static int greedyFactorization(String text) {
91       int n = text.length();
92       int count = 0;
93       int idx = 0;
94
95       System.out.print("Greedy Factors: ");
96
97       while (idx < n) {
98          int bestEnd = idx + 1;
99
100         // Cerca il prefisso chiuso pi lungo a partire da idx
101         for (int end = n; end > idx; end--) {
102            String sub = text.substring(idx, end);
103            if (isClosed(sub)) {
104               bestEnd = end;
105               break; // Trovato il pi lungo, esco
106            }
107         }
```

```
108
109             // Stampa il fattore trovato
110             System.out.print("[" + text.substring(idx, bestEnd) + "] ");
111
112             count++;
113             idx = bestEnd;
114         }
115         System.out.println();
116         return count;
117     }
```

3.2 Giorno 2: Testing dell'algoritmo

L'algoritmo aiuta soprattutto a distinguere una fattorizzazione ottimale da una fattorizzazione che calcola il pezzo chiuso più lungo che incontra.

L'algoritmo per funzionare ha fatto uso di metodi utili a dividere i lavori: un generatore di parole casuali, un eseguitore della modalità di test (su 5000 campioni generati casualmente), un eseguitore della modalità manuale e una classe che tenga traccia dei risultati:

Listing 3.1: Modifica di tutto il codice in relazione al testing

```
118 import java.io.File;
119 import java.io.FileWriter;
120 import java.io.IOException;
121 import java.io.PrintWriter;
122 import java.time.LocalDateTime;
123 import java.time.format.DateTimeFormatter;
124 import java.util.ArrayList;
125 import java.util.Arrays;
126 import java.util.Collections;
127 import java.util.List;
128 import java.util.Random;
129 import java.util.Scanner;
130 
131 public class Main
132 {
133    // Classe di supporto per restituire sia il numero che la rappresentazione testuale
134    static class Result {
135       int count;
136       String factors;
137 
138       public Result(int count, String factors) {
139          this.count = count;
140          this.factors = factors;
141       }
142    }
143 
144    public static String generateRandomWord(int alphaSize, int minLen, int maxLen) {
145       Random random = new Random();
146       StringBuilder sb = new StringBuilder();
147       int length = random.nextInt(maxLen - minLen + 1) + minLen;
148       char[] fullAlphabet = "abcdefghijklmnopqrstuvwxyz".toCharArray();
149 
150       for (int i = 0; i < length; i++) {
151          int charIndex = random.nextInt(alphaSize);
152          sb.append(fullAlphabet[charIndex]);
153       }
154       return sb.toString();
155    }
156 
157    public static boolean isClosed(String w) {
158       int n = w.length();
159       if (n <= 1) return true;
160 
161       String border = "";
162       int lenB = 0;
163 
164       for (int i = n - 1; i > 0; i--) {
165          if (w.startsWith(w.substring(n - i))) {
166             border = w.substring(0, i);
167             lenB = i;
168             break;
169          }
170       }
171 
172       if (lenB == 0) return false;
173 
174       int suffixStartIndex = n - lenB;
175       int foundIndex = w.indexOf(border, 1);
```

```
176
177         if (foundIndex < suffixStartIndex) {
178             return false; // Ha occorrenze interne
179         }
180         return true;
181     }
182
183     public static Result minimalFactorizationDP(String text) {
184         int n = text.length();
185         int[] dp = new int[n + 1];
186         int[] from = new int[n + 1]; // Array per ricostruire il percorso (backtracking)
187
188         Arrays.fill(dp, Integer.MAX_VALUE);
189         dp[0] = 0; // Stringa vuota ha 0 fattori
190
191         for (int i = 1; i <= n; i++) {
192             for (int j = 0; j < i; j++) {
193                 String sub = text.substring(j, i);
194                 if (isClosed(sub)) {
195                     if (dp[j] != Integer.MAX_VALUE && dp[j] + 1 < dp[i]) {
196                         dp[i] = dp[j] + 1;
197                         from[i] = j; // Ricorda da dove siamo arrivati (da j a i)
198                     }
199                 }
200             }
201         }
202
203         // --- Ricostruzione dei fattori ---
204         List<String> factorList = new ArrayList<>();
205         int curr = n;
206         while (curr > 0) {
207             int prev = from[curr];
208             factorList.add("[" + text.substring(prev, curr) + "]");
209             curr = prev;
210         }
211         Collections.reverse(factorList); // L'abbiamo costruita al contrario, giriamola
212
213         return new Result(dp[n], String.join(" ", factorList));
214     }
215
216     public static Result greedyFactorization(String text) {
217         int n = text.length();
218         int count = 0;
219         int idx = 0;
220         StringBuilder sb = new StringBuilder();
221
222         while (idx < n) {
223             int bestEnd = idx + 1;
224
225             for (int end = n; end > idx; end--) {
226                 String sub = text.substring(idx, end);
227                 if (isClosed(sub)) {
228                     bestEnd = end;
229                     break;
230                 }
231             }
232
233             sb.append("[").append(text.substring(idx, bestEnd)).append("] ");
234             count++;
235             idx = bestEnd;
236         }
237
238         return new Result(count, sb.toString().trim());
239     }
240
241     public static void runTestMode() {
242         System.out.println("\n=== TEST MODE: Greedy vs DP ===");
```

```
243         System.out.println("Generazione di 5.000 parole...");
244 
245         String directoryPath = "/mnt/c/Users/beatr/OneDrive/Desktop/Tirocinio/RisultatiAlgo";
246         String timestamp =
                LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyyMMdd_HHmmss"));
247         String fileName = "risultati_" + timestamp + ".txt";
248         File file = new File(directoryPath, fileName);
249         new File(directoryPath).mkdirs();
250 
251         int totalWordsToGenerate = 5000;
252         int counterExamplesFound = 0;
253         int currentAlphaSize = 2;
254         int thresholdChange = 500;
255 
256         try (PrintWriter writer = new PrintWriter(new FileWriter(file))) {
257             writer.println("=== TEST REPORT ===");
258             writer.println("Data: " + LocalDateTime.now());
259             System.out.println("Salvataggio risultati in: " + file.getAbsolutePath());
260 
261             for (int i = 1; i <= totalWordsToGenerate; i++) {
262 
263                 if (i > thresholdChange && currentAlphaSize < 5) {
264                     currentAlphaSize++;
265                     thresholdChange += 1500;
266                     writer.println(">>> LEVEL UP! Alfabeto: " + currentAlphaSize);
267                 }
268 
269                 String word = generateRandomWord(currentAlphaSize, 20, 50);
270 
271                 Result dpRes = minimalFactorizationDP(word);
272                 Result greedyRes = greedyFactorization(word);
273 
274                 if (greedyRes.count > dpRes.count) {
275                     counterExamplesFound++;
276 
277                     StringBuilder report = new StringBuilder();
278                     report.append("\n[!!!] CONTROESEMPIO #").append(counterExamplesFound)
                            .append(" (Parola ").append(i).append(") [!!!]\n");
279                     report.append("Parola: ").append(word).append("\n");
280                     report.append("Alfabeto: ").append(currentAlphaSize).append("\n");
281 
282                     report.append("DP (Ottimo, ").append(dpRes.count).append("):
                            ").append(dpRes.factors).append("\n");
283                     report.append("Greedy (Errato, ").append(greedyRes.count).append("):
                            ").append(greedyRes.factors).append("\n");
284 
285                     report.append("Differenza: ").append(greedyRes.count -
                            dpRes.count).append("\n");
286                     report.append("--------------------------------------------------");
287 
288                     System.out.println(report.toString());
289                     writer.println(report.toString());
290                     writer.flush();
291                 }
292 
293                 if (i % 500 == 0) System.out.print(".");
294             }
295 
296             String footer = "\n\n=== FINE TEST ===\nControesempi trovati: " +
                    counterExamplesFound;
297             System.out.println(footer);
298             writer.println(footer);
299 
300         } catch (IOException e) {
301             e.printStackTrace();
302         }
303     }
304 
```

```
305
306     public static void runManualMode(Scanner scanner) {
307         System.out.println("\n--- Modalita' Manuale ---");
308
309         while (true) {
310             System.out.print("\nInserisci parola (o 'esci'): ");
311             String input = scanner.nextLine().trim();
312
313             if (input.equalsIgnoreCase("esci")) break;
314             if (input.isEmpty()) continue;
315
316             boolean closed = isClosed(input);
317             Result dpRes = minimalFactorizationDP(input);
318             Result greedyRes = greedyFactorization(input);
319
320             System.out.println("Analisi: " + input);
321             System.out.println(" - Chiusa: " + closed);
322             System.out.println(" - DP (" + dpRes.count + "): " + dpRes.factors);
323             System.out.println(" - Greedy (" + greedyRes.count + "): " + greedyRes.factors);
324
325             if (greedyRes.count > dpRes.count) {
326                 System.out.println(" -> NOTA: Il Greedy ha fallito! Differenza: " +
                        (greedyRes.count - dpRes.count));
327             }
328         }
329     }
330
331     public static void main(String[] args) {
332         Scanner scanner = new Scanner(System.in);
333         while (true) {
334             System.out.println("\n1 - Manuale | 2 - Test Auto | 0 - Esci");
335             String scelta = scanner.nextLine();
336             if (scelta.equals("1")) runManualMode(scanner);
337             else if (scelta.equals("2")) runTestMode();
338             else if (scelta.equals("0")) break;
339         }
340         scanner.close();
341     }
342 }
```

Come si nota, la parte di testing scrive un file in una cartella locale, in modo tale che ogni risultato ottenuto si possa verificare in modo manuale.

3.2.1 Esempi per cui Greedy e' peggio di una fattorizzazione ottimale

In questo caso si prenderanno in considerazione un po' di parole ottenute che sono aperte e che hanno una differenza di almeno 3 fattorizzazioni tra Greedy e DP, seguono:

- abbaabbaaabbbbbbbaaaababababbbaaabaaaabbbab
- babaabbbaabbababaabbbbabaabbabaabaaaabbbbbabaababa
- bbabbccacccccbaabcaacabbc
- ababcbbbcabbbbbacbcccaaabbbabc
- aaaabccabbcbbcbacabbaccacbbbaaabcbcbaabc
- acedcbdedebacdecbcaabeccdaddcddcbbedecddeebccadce

I risultati mostrano le seguenti differenze:

```
Inserisci parola (o 'esci'): abbaabbaaabbbbbbbbaaaababababbbaaabaaaabbbab
Analisi: abbaabbaaabbbbbbbbaaaababababbbaaabaaaabbbab
 — Chiusa: false
 - DP (3): [abbaabb] [aaabbbbbbbbaaaababababbbaaabaaaabb] [bab]
 — Greedy (6): [abbaabbaa] [abbbbbbbbaaaababababb] [aaabaaaab] [bb] [a] [b]
 -> NOTA: Il Greedy ha fallito! Differenza: 3

 ------------------------------------------------------------------------------

Inserisci parola (o 'esci'): babaabbbaabbababaabbbbbabaabbabaabaaaabbbbbabaababa
Analisi: babaabbbaabbababaabbbbbabaabbabaabaaaabbbbbabaababa
 - Chiusa: false
 — DP (3): [bab] [aabbbaabb] [ababaabbbbbabaabbabaabaaaabbbbbabaababa]
 - Greedy (7): [babaabbbaabbababaabbb] [babaabbabaab] [aaaa] [bbbb] [abaaba] [b] [a]
 —> NOTA: Il Greedy ha fallito! Differenza: 4

 ------------------------------------------------------------------------------

Inserisci parola (o 'esci'): bbabbccacccccbaabcaacabbc
Analisi: bbabbccacccccbaabcaacabbc
 — Chiusa: false
 - DP (2): [bb] [abbccacccccbaabcaacabbc]
 — Greedy (7): [bbabb] [ccacc] [ccc] [baab] [caaca] [bb] [c]
 -> NOTA: Il Greedy ha fallito! Differenza: 5

------------------------------------------------------------------------------

Inserisci parola (o 'esci'): ababcbbbcabbbbbacbcccaaabbbabc
Analisi: ababcbbbcabbbbbacbcccaaabbbabc
 - Chiusa: false
 — DP (2): [a] [babcbbbcabbbbbacbcccaaabbbabc]
 - Greedy (8): [abab] [cbbbcabbbbbacb] [ccc] [aaa] [bbb] [a] [b] [c]
 —> NOTA: Il Greedy ha fallito! Differenza: 6

------------------------------------------------------------------------------

Inserisci parola (o 'esci'): aaaabccabbcbbcbacabbaccacbbbaaabcbcbaabc
Analisi: aaaabccabbcbbcbacabbaccacbbbaaabcbcbaabc
 — Chiusa: false
 - DP (3): [aa] [aabccabbcbbcbacabbaccacbbbaa] [abcbcbaabc]
 — Greedy (10): [aaaa] [bccabbc] [bbcbacabb] [accac] [bbb] [aaa] [bcbcb] [aa] [b] [c]
 -> NOTA: Il Greedy ha fallito! Differenza: 7

------------------------------------------------------------------------------

 Inserisci parola (o 'esci'): acedcbdedebacdecbcaabeccdaddcddcbbedecddeebccadce
Analisi: acedcbdedebacdecbcaabeccdaddcddcbbedecddeebccadce
 - Chiusa: false
 — DP (2): [a] [cedcbdedebacdecbcaabeccdaddcddcbbedecddeebccadce]
 - Greedy (10): [acedcbdedebac] [decbcaabeccdaddcddcbbedec] [dd] [ee] [b] [cc] [a] [d] [c] [e]
 —> NOTA: Il Greedy ha fallito! Differenza: 8
```

Si può notare come l'algoritmo greedy sbagli a confronto dell'algoritmo dp perché il primo descrive una parola w in più fattori chiusi per cui w = A*B*C*... , invece l'algoritmo greedy fattorizza w = A*B*...*x*y*z, dove x,y,z sono i fattori alla fine di w, composti da pochi caratteri.

3.3 Giorno 3: Comprensione dei dati e dei risultati

Ho svolto un refactoring del codice Java per rendere altre sezioni delle classi, c'è una classe che si occupa di fattorizzare (Factorizer), una classe che si occupa del risultato (FactorizationResult), una classe sugli utilities per le parole chiuse (ClosedWordUtils) e il main.

```
Tirocinio
├── AlgoritmoJava
│   ├── ClosedWordUtils
│   ├── FactorizationResult
│   ├── Factorizer
│   └── Main
└── RisultatiAlgo
```

Modificando la struttura del programma ho modificato anche il generatore di parole, siccome la fattorizzazione in parole chiuse non è sempre unica ho aggiunto anche una lista che mi calcoli le altre combinazioni possibili. Seguono gli esempi di 2 parole, la prima ha una fattorizzazione in parole chiuse unica, la seconda non ha una fattorizzazione unica:

```
 [!!!]  CONTROESEMPIO #2 (Parola 3) [!!!]
Parola: bbbabbbabbbbaabaabbabaaaabaabbbbbbabaaabbaab
DP (Ottimo, 2): [bbbabbbab] [bbaabaabbabaaaabaabbbbbabaabbaab]
Totale Fattorizzazioni Ottime: 1
Greedy (Errato, 5): [bbbabbbabbba] [ababbabaaaabab] [bbb] [abaab] [baab]
Differenza: 3
--------------------------------------------------------------------------------


[!!!!] CONTROESEMPIO #3 (Parola 4) [!!!!]
Parola: aaababbbaaabaabaaaaabbbabbabbabbbbbabaaaaaabaaabbab
DP (Ottimo, 3): [aaa] [babbbaaabaabaaaaabbbabb] [abbabbbbabaaaaaabaaabbab]
Totale Fattorizzazioni Ottime: 3
Greedy (Errato, 4): [aaababbbaaaba] [abaaaaabbbabbabbabbbbabaaaaa] [abaaab] [bab]
Differenza: 1
--------------------------------------------------------------------------------


 [!!!]  NON-UNICIT TROVATA (Parola 4) [!!!]
Parola: aaababbbaaabaabaaaaabbbabbabbabbbbbabaaaaaabaaabbab
Numero di fattorizzazioni: 3
Differenza: 3
Esempio percorso: [aaa] [babbbaaabaabaaaaabbbabb] [abbabbbbabaaaaaabaaabbab]
--- Elenco Soluzioni Ottimali ---
   -> [aaa] [babbbaaabaabaaaaabbbabb] [abbabbbbabaaaaaabaaabbab]
   -> [aaa] [babbbaaabaabaaaaabbbabbabbabbb] [babaaaaaabaaabbab]
   -> [aaababbbaaab] [aabaaaaabbbabbabbabbbbabaaaaaabaa] [abbab]
--------------------------------------------------------------------------------
```

Col funzionamento del codice si è notata una cosa ben specifica: l'algoritmo greedy sbaglia spesso e volentieri, infatti quasi sempre prende quanti più termini in modo da ottenere una parola chiusa, mostrando come un approccio greedy sia poco ottimale, considerando quante volte fallisce. L'algoritmo ottimale serve a determinare il numero minimo di fattori per il prefisso $w[1 \dots i]$, l'algoritmo considera tutte le possibili posizioni di taglio $j < i$. La struttura logica si basa sulla scomposizione:

$$w[1 \dots i] = \underbrace{w[1 \dots j]}_{\text{prefisso ottimizzato}} \cdot \underbrace{w[j+1 \dots i]}_{\text{fattore chiuso}}$$

Affinché la soluzione per $w[1 \dots i]$ sia globale, è necessario che il prefisso sinistro $w[1 \dots j]$ sia stato a sua volta fattorizzato in modo minimale e che il suffisso destro $w[j+1 \dots i]$ sia una parola chiusa valida.

Questa ricorrenza è tradotta algoritmicamente nell'istruzione:

```
dp[i] = Math.min(dp[i], dp[j] + 1);
```

In questo contesto, `dp[j]` rappresenta la soluzione ottima già calcolata per il prefisso di lunghezza j, mentre il `+1` conteggia il nuovo fattore chiuso $w[j+1 \dots i]$. Iterando su tutti i possibili j, l'algoritmo non effettua una scelta arbitraria, ma esplora le combinazioni per garantire che `dp[i]` converga al minimo assoluto.

3.4 Giorno 4: Performance e generatori di parole di Fibonacci e di Thue-Morse

Ho scritto una nuova classe che mi permetta di generare parole di Fibonacci e di Thue-Morse fino a un limite di 2000 o più caratteri (quando richiamato nel Main - oltre rallenta in modo esagerato il processo di verifica).

Listing 3.2: Classe e i metodi di Generatori

```
27 public class Generatori {
28    public static String fibonacciWord(int n) {
29       if (n == 0) return "b";
30       if (n == 1) return "a";
31
32       String fPrev = "a";
33       String fPrevPrev = "b";
34       String current = "";
35
36       for (int i = 2; i <= n; i++) {
37          current = fPrev + fPrevPrev;
38
39          fPrevPrev = fPrev;
40          fPrev = current;
41       }
42       return current;
43    }
44
45    public static String thueMorseWord(int n) {
46       String current = "a";
47       for (int i = 0; i < n; i++) {
48          StringBuilder next = new StringBuilder();
49          for (char c : current.toCharArray()) {
50             if (c == 'a') next.append("ab");
51             else next.append("ba");
52          }
53          current = next.toString();
54       }
55       return current;
56    }
57 }
```

Come già citato questa classe viene richiamata nel main per effettuare dei test dell'algoritmo dp e dell'algoritmo greedy col seguente metodo:

```
59    public static void generatorMode() {
60       System.out.println("\n=== GENERATOR MODE & REPORT ===");
61
62       String directoryPath = "/mnt/c/Users/beatr/OneDrive/Desktop/Tirocinio/RisultatiAlgo";
63       String timestamp =
             LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyyMMdd_HHmmss"));
64       File file = new File(directoryPath, "report_generatori_" + timestamp + ".txt");
65       new File(directoryPath).mkdirs();
66
67       try (PrintWriter writer = new PrintWriter(new FileWriter(file))) {
68          writer.println("=== REPORT ANALISI GENERATORI ===");
69          writer.println("Data: " + LocalDateTime.now());
70          System.out.println("Log salvato in: " + file.getAbsolutePath());
71
72          String headerFib = "\n=== ANALISI PAROLE DI FIBONACCI ===";
73          System.out.println(headerFib);
74          writer.println(headerFib);
75
76          String tableHeader = String.format("%-5s | %-15s | %-10s | %-10s | %-10s | %-10s |
                %-15s",
77             "n", "Lunghezza", "DP Size", "Greedy", "Time(ms)", "Diff", "Check");
78          System.out.println(tableHeader);
79          writer.println(tableHeader);
80
```

```
81          String separator =
                "-------------------------------------------------------------------------------------------
82          System.out.println(separator);
83          writer.println(separator);
84
85          for (int i = 3; i <= 17; i++) {
86             String fibWord = Generatori.fibonacciWord(i);
87
88             long startT = System.currentTimeMillis();
89             FactorizationResult dp = Factorizer.minimalFactorizationDP(fibWord);
90             FactorizationResult greedy = Factorizer.greedyFactorization(fibWord);
91             long endT = System.currentTimeMillis();
92
93             int diff = greedy.totalFactors - dp.totalFactors;
94             String check = (diff > 0) ? "!!! CONTROESEMPIO" : "OK";
95
96             String row = String.format("%-5d | %-15d | %-10d | %-10d | %-10d | %-10d |
                   %-15s",
97                i, fibWord.length(), dp.totalFactors, greedy.totalFactors, (endT - startT),
                      diff, check);
98
99             System.out.println(row);
100            writer.println(row);
101            writer.flush();
102         }
103         System.out.println(separator);
104         writer.println(separator);
105
106         String headerTM = "\n\n=== ANALISI PAROLE DI THUE-MORSE ===";
107         System.out.println(headerTM);
108         writer.println(headerTM);
109
110         System.out.println(tableHeader);
111         writer.println(tableHeader);
112
113         System.out.println(separator);
114         writer.println(separator);
115
116         for (int i = 1; i <= 11; i++) {
117            String thueWord = Generatori.thueMorseWord(i);
118
119            long startT = System.currentTimeMillis();
120            FactorizationResult dp = Factorizer.minimalFactorizationDP(thueWord);
121            FactorizationResult greedy = Factorizer.greedyFactorization(thueWord);
122            long endT = System.currentTimeMillis();
123
124            int diff = greedy.totalFactors - dp.totalFactors;
125            String check = (diff > 0) ? "!!! CONTROESEMPIO" : "OK";
126
127            String row = String.format("%-5d | %-15d | %-10d | %-10d | %-10d | %-10d |
                   %-15s",
128               i, thueWord.length(), dp.totalFactors, greedy.totalFactors, (endT - startT),
                      diff, check);
129
130            System.out.println(row);
131            writer.println(row);
132            writer.flush();
133         }
134         System.out.println(separator);
135         writer.println(separator);
136
137         System.out.println("\nAnalisi completata.");
138
139      } catch (IOException e) {
140         e.printStackTrace();
141      }
142   }
```

E seguono dei risultati abbastanza strani sulla fattorizzazione in parole chiuse di parole note e non randomiche:

3.4.1 Fibonacci

Con le parole di Fibonacci si ha il seguente scenario:

- alla terza parola di Fibonacci "aba" c'è solo un'unica fattorizzazione sia in modo ingenuo sia in modo ottimale, infatti è una parola chiusa, lo stesso vale per la quarta parola "abaab"
- già alla quinta parola la fattorizzazione tra greedy e algoritmo ottimale presenta una differenza, infatti la parola "abaababa" e la fattorizzazione ottimale sarebbe "abaab" e "aba", invece con l'algoritmo greedy la fattorizzazione è "abaa", "bab" e "a".

I risultati in totale:

=== ANALISI PAROLE DI FIBONACCI ===

n	Lunghezza	DP Size	Greedy	Time(ms)	Diff	Check
3	3	1	1	1	0	OK
4	5	1	1	0	0	OK
5	8	2	3	1	1	!!! CONTROESEMPIO
6	13	2	3	0	1	!!! CONTROESEMPIO
7	21	2	3	1	1	!!! CONTROESEMPIO
8	34	2	3	4	1	!!! CONTROESEMPIO
9	55	2	3	7	1	!!! CONTROESEMPIO
10	89	2	3	20	1	!!! CONTROESEMPIO
11	144	2	3	16	1	!!! CONTROESEMPIO
12	233	2	3	54	1	!!! CONTROESEMPIO
13	377	2	3	188	1	!!! CONTROESEMPIO
14	610	2	3	906	1	!!! CONTROESEMPIO
15	987	2	3	3954	1	!!! CONTROESEMPIO
16	1597	2	3	23856	1	!!! CONTROESEMPIO
17	2584	2	3	144750	1	!!! CONTROESEMPIO

3.4.2 Thue-Morse

Per quanto riguarda le parole Thue-Morse si ottiene il seguente risultato:

- già la prima parola di Thue-Morse viene fattorizzata in modo corretto, infatti "ab" si scompone in "a" e "b" che sono entrambe parole chiuse, lo stesso vale per la seconda parola, infatti "abba" è già fattorizzata per essere una parola chiusa;
- dalla terza parola "abbabaab" presenta una differenza di fattorizzazione tra greedy e ottimale, col primo si ottiene: "abbab", "aa" e b, col secondo si ottiene: "abba" e "baab".

Seguono gli ulteriori risultati:

=== ANALISI PAROLE DI THUE−MORSE ===

n	Lunghezza	DP Size	Greedy	Time(ms)	Diff	Check
1	2	2	2	0	0	OK
2	4	1	1	1	0	OK
3	8	2	3	0	1	!!! CONTROESEMPIO
4	16	3	4	1	1	!!! CONTROESEMPIO
5	32	3	3	0	0	OK
6	64	3	5	0	2	!!! CONTROESEMPIO
7	128	3	6	5	3	!!! CONTROESEMPIO
8	256	3	5	53	2	!!! CONTROESEMPIO
9	512	3	7	439	4	!!! CONTROESEMPIO
10	1024	3	8	5664	5	!!! CONTROESEMPIO
11	2048	3	7	81145	4	!!! CONTROESEMPIO

3.4.3 Performance

Dalle parole di Thue-Morse e dalle parole di Fibonacci si nota come i tempi aumentano vertiginosamente, infatti con parole di lunghezza superiore a 2000 caratteri si raggiunge più di 2 minuti con la diciasettesima parola di Fibonacci e più di 1 minuto con l'undicesima parola di Thue-Morse.

Il tutto non è dovuto dall'algoritmo greedy, anzi è la parte più rapida del controllo, il tutto è dato dall'algoritmo ottimale che impiega tempi più elevati man mano che la parola cresce, infatti seguono i risultati delle performance su parole generate casualmente su lunghezze che vanno da 200 a 2000 caratteri:

```
Parola di lunghezza 200:
Tempo DP: 68 ms (0.068 s)
Tempo Greedy: 2 ms (0.002 s)
------------------------------------------------

Parola di lunghezza 400:
Tempo DP: 350 ms (0.35 s)
Tempo Greedy: 0 ms (0.0 s)
--------------------------------------------------------

Parola di lunghezza 800:
Tempo DP: 2576 ms (2.576 s)
Tempo Greedy: 23 ms (0.023 s)
------------------------------------------------

Parola di lunghezza 1600:
Tempo DP: 39403 ms (39.403 s)
Tempo Greedy: 83 ms (0.083 s)
--------------------------------------------------------

Parola di lunghezza 1000:
Tempo DP: 6645 ms (6.645 s)
Tempo Greedy: 35 ms (0.035 s)
------------------------------------------------

Parola di lunghezza 2000:
Tempo DP: 90221 ms (90.221 s)
Tempo Greedy: 256 ms (0.256 s)
--------------------------------------------------------
```

Si stima che la crescita si avvicini a una complessità temporale di $O(n^3)$.

3.5 Giorno 5-7: Dimostrazioni con Walnut

Per i vari giorni ho impostato meglio lo studio di Walnut per processare le varie dimostrazioni, ho scritto alcune dimostrazioni rapide per diventare familiare a Walnut e alla scrittura di alcuni predicati.

Per svolgere ho scritto i predicati:

```
def IsBorder(k, n) "(k>0) & (k<n) & (A i (i<k) => T[i]=T[n-k+i])";

IsBorder.txt:

msd_2 msd_2

0 0
0 0 -> 0
0 1 -> 1

1 0
0 0 -> 2
1 0 -> 3
0 1 -> 4
```

```
2 0
0 0 -> 1
1 0 -> 5
0 1 -> 4

3 0
0 1 -> 6

4 0
0 0 -> 7
1 0 -> 6
0 1 -> 1
1 1 -> 5

5 1
0 0 -> 5
1 1 -> 5

6 1
0 0 -> 6

7 0
0 0 -> 4
0 1 -> 1
1 1 -> 5
```

`def OccursInside "(E j (j > 0) & (j < n−k) & (A i (i < k) => T[i] = T[j+i]))":`

OccursInside.txt:

```
msd_2 msd_2

0 0
0 0 -> 0
0 1 -> 1

1 0
0 0 -> 2
1 0 -> 3
0 1 -> 4
1 1 -> 5

2 1
0 0 -> 4
1 0 -> 6
0 1 -> 4
1 1 -> 4

3 0
0 1 -> 7

4 1
0 0 -> 4
1 0 -> 4
0 1 -> 4
1 1 -> 4

5 0
0 0 -> 8
0 1 -> 8

6 0
```

```
0 0 -> 9
0 1 -> 4
1 1 -> 10

7 0
0 0 -> 7
0 1 -> 8

8 1
0 0 -> 8
0 1 -> 8

9 1
0 0 -> 9
0 1 -> 4
1 1 -> 10

10 0
0 0 -> 10
0 1 -> 4
1 1 -> 10
```

(continua il 20-01) Queste sono versioni per le parole di Thue-Morse, ma ci sono anche per le parole di Fibonacci.

```
def IsBorder_Fib "?msd_fib (k>0) & (k<n) & (A i (i<k) => F[i] = F[n-k+i])":

IsBorder_Fib.txt:

msd_fib msd_fib

0 0
0 0 -> 0
0 1 -> 1

1 0
0 0 -> 2
1 0 -> 3

2 0
0 0 -> 4
1 0 -> 5
0 1 -> 1
1 1 -> 5

3 0
0 0 -> 6
0 1 -> 7

4 0
0 0 -> 2
1 0 -> 3
0 1 -> 1
1 1 -> 5

5 1
0 0 -> 8

6 0
0 1 -> 5

7 0
0 0 -> 9
```

1 0 −> 3

8 1
0 0 −> 8
1 1 −> 5

9 0
1 0 −> 5

def OccursInside_Fib "?msd_fib (E j (j > 0) & (j < n−k) & (A i (i < k) => F[i] = F[j+i]))":

`OccursInside_Fib.txt:`

msd_fib msd_fib

0 0
0 0 −> 0
0 1 −> 1

1 0
0 0 −> 2
1 0 −> 3

2 1
0 0 −> 4
1 0 −> 5
0 1 −> 6
1 1 −> 7

3 0
0 0 −> 8
0 1 −> 9

4 1
0 0 −> 4
1 0 −> 10
0 1 −> 6
1 1 −> 7

5 0
0 0 −> 11
0 1 −> 6

6 1
0 0 −> 4
1 0 −> 10

7 1
0 0 −> 4

8 0
0 1 −> 12

9 0
0 0 −> 13
1 0 −> 3

10 1
0 0 −> 4
0 1 −> 6

11 0

```
0 0 -> 4
1 0 -> 14
0 1 -> 6
1 1 -> 15

12 0
0 0 -> 16

13 1
0 0 -> 4
1 0 -> 17
0 1 -> 6
1 1 -> 7

14 0
0 0 -> 11
0 1 -> 18

15 0
0 0 -> 11

16 0
0 0 -> 16
0 1 -> 6
1 1 -> 12

17 0
0 0 -> 16
0 1 -> 6

18 0
0 0 -> 4
1 0 -> 14
```

Capitolo 4

Settimana 2

4.1 Giorno 1-3: Chiusura per Thue-Morse e Fibonacci

In questo giorno ho stabilito gli automi per verificare la chiusura delle parole di Thue-Morse e per le parole di Fibonacci, dopodiché implementando un'automa che mi permetta di verificare i fattori chiusi in una parola, seguono le definizioni su Walnut:

```
[Walnut]$ def IsBorder_TM "(k>0) & (k<n) & (A i (i<k) => T[i]=T[n-k+i])":

isBorder_TM.txt:

msd_2 msd_2

0 0
0 0 -> 0
0 1 -> 1

1 0
0 0 -> 2
1 0 -> 3
0 1 -> 4

2 0
0 0 -> 1
1 0 -> 5
0 1 -> 4

3 0
0 1 -> 6

4 0
0 0 -> 7
1 0 -> 6
0 1 -> 1
1 1 -> 5

5 1
0 0 -> 5
1 1 -> 5

6 1
0 0 -> 6

7 0
0 0 -> 4
0 1 -> 1
1 1 -> 5
```

```
[Walnut]$ def occursInside_TM "E j (j>0) & (j<n-k) & (A i (i<k) => T[i]=T[j+i])":

occursInside.txt:

msd_2 msd_2

0 0
0 0 -> 0
0 1 -> 1

1 0
0 0 -> 2
1 0 -> 3
0 1 -> 4
1 1 -> 5

2 1
0 0 -> 4
1 0 -> 6
0 1 -> 4
1 1 -> 4

3 0
0 1 -> 7

4 1
0 0 -> 4
1 0 -> 4
0 1 -> 4
1 1 -> 4

5 0
0 0 -> 8
0 1 -> 8

6 0
0 0 -> 9
0 1 -> 4
1 1 -> 10

7 0
0 0 -> 7
0 1 -> 8

8 1
0 0 -> 8
0 1 -> 8

9 1
0 0 -> 9
0 1 -> 4
1 1 -> 10

10 0
0 0 -> 10
0 1 -> 4
1 1 -> 10

[Walnut]$ def isClosed_TM "n<2 | E k ($isBorder_TM(n,k) & ~$occursInside_TM(n,k))";

isClosed_TM.txt;
```

```
msd_2

0 1
0 -> 0
1 -> 1

1 1
0 -> 2

2 0
0 -> 3
1 -> 4

3 1
0 -> 5
1 -> 6

4 1
0 -> 4

5 0
0 -> 5
1 -> 6

6 1
0 -> 6
1 -> 6
```

La composizione è stata analoga per le parole di Fibonacci, ed è la seguente:

```
[Walnut]$ def IsBorder_Fib "?msd_fib (k>0) & (k<n) & (A i (i<k) => F[i]=F[n-k+i])":

isBorder_Fib.txt:

msd_fib msd_fib

0 0
0 0 -> 0
0 1 -> 1

1 0
0 0 -> 2
1 0 -> 3

2 0
0 0 -> 4
1 0 -> 5
0 1 -> 1
1 1 -> 5

3 0
0 0 -> 6
0 1 -> 7

4 0
0 0 -> 2
1 0 -> 3
0 1 -> 1
1 1 -> 5

5 1
0 0 -> 8
```

6 0
0 1 −> 5

7 0
0 0 −> 9
1 0 −> 3

8 1
0 0 −> 8
1 1 −> 5

9 0
1 0 −> 5

[Walnut]$ def occursInside_Fib "?msd_fib E j (j>0) & (j<n−k) & (A i (i<k) => F[i]=F[j+i])":

occursInside_Fib.txt:

msd_fib msd_fib

0 0
0 0 −> 0
0 1 −> 1

1 0
0 0 −> 2
1 0 −> 3

2 1
0 0 −> 4
1 0 −> 5
0 1 −> 6
1 1 −> 7

3 0
0 0 −> 8
0 1 −> 9

4 1
0 0 −> 4
1 0 −> 10
0 1 −> 6
1 1 −> 7

5 0
0 0 −> 11
0 1 −> 6

6 1
0 0 −> 4
1 0 −> 10

7 1
0 0 −> 4

8 0
0 1 −> 12

9 0
0 0 −> 13
1 0 −> 3

```
10 1
0 0 -> 4
0 1 -> 6

11 0
0 0 -> 4
1 0 -> 14
0 1 -> 6
1 1 -> 15

12 0
0 0 -> 16

13 1
0 0 -> 4
1 0 -> 17
0 1 -> 6
1 1 -> 7

14 0
0 0 -> 11
0 1 -> 18

15 0
0 0 -> 11

16 0
0 0 -> 16
0 1 -> 6
1 1 -> 12

17 0
0 0 -> 16
0 1 -> 6

18 0
0 0 -> 4
1 0 -> 14
```

[Walnut]$ def isClosed_Fib "n<2 | (E k $isBorder_Fib(k,n) & ~$occursInside_Fib(k,n))";

isClosed_Fib.txt:

```
msd_2

0 1
0 -> 0
1 -> 1

1 1
0 -> 2

2 0
0 -> 3
1 -> 4

3 1
0 -> 5
1 -> 6

4 0
```

```
0 -> 7

5 1
1 -> 8

6 1
0 -> 9

7 0
0 -> 9
1 -> 4

8 1
0 -> 5

9 1
0 -> 9
1 -> 6
```

Avendo definito la chiusura per le parole di Fibonacci e per le parole di Thue-Morse per valutare solo parole che iniziano da un indice 0, poi mi sono occupata di scriverlo per dei fattori interni e che quindi non iniziano da 0, ma da una posizione qualsiasi (sono stati scritti sia per parole di Fibonacci sia per parole di Thue-Morse).

```
[Walnut]$ def internalFactorBorder_TM "(k>0) & (k<len) & (A i (i<k) => T[s+i]=T[s+len-k+i])":

[Walnut]$ def internalFactorOccursInside_TM "E j (j>0) & (j<len-k) & (A i (i<k) =>
    T[s+i]=T[s+j+i])":

[Walnut]$ def isClosedFactor_TM "len < 2 | (Ek $internalFactorBorder_TM(k, len, s) &
    ~$internalFactorOccursInside_TM(k, len, s))";

[Walnut]$ def internalFactorBorder_Fib "?fib (k>0) & (k<len) & (A i (i<k) => F[s+i]=F[s+len-k+i])":

[Walnut]$ def internalFactorOccursInside_Fib "?fib E j (j>0) & (j<len-k) & (A i (i<k) =>
    F[s+i]=F[s+j+i])":

[Walnut]$ def isClosedFactor_Fib "len<2 | (E k $internalFactorBorder_Fib(k,len,s) &
    ~$internalFactorOccursInside_Fib(k,len,s))";
```

Infine mi sono occupata di scrivere una logica per delle parole generiche e per farlo ho usato le macro, che servono per evitare ripetizioni e che si basa su un sistema numerico e su delle variabili (%0 e %1 - si noti che non si possono usare macro e/o funzioni nestate).

```
[Walnut]$ macro bordoGenerale "?%0 (k>0) & (k<len) & (At i (i<k) => %1[start+i] = %1[start+len-k+i])";

[Walnut]$ macro occursInsideGenerale "?%0 Ex j (j>0) & (j<len-k) & (At i (i<k) => %1[start+i] =
    %1[start+j+i])";

[Walnut]$ macro isClosedGenerale "?%0 (len>1) & (E k (k>0) & (k<len) & (A i (i<k) =>
    %1[start+i]=%1[start+len-k+i]) & ~(E j (j>0) & (j<len-k) & (A i (i<k) =>
    %1[start+i]=%1[start+j+i])))";

[Walnut]$ def isClosed_W "#isClosedGenerale(msd_2,W)";
```

Capitolo 5

Tecniche di ottimizzazione dell'algoritmo Java

5.1 Primi passi di ottimizzazione

L'algoritmo DP iniziale impiegava molto tempo data la sua complessità temporale che raggiungeva O(n$\hat{5}$), considerando il costante utilizzo del metodo substring della classe String di Java, che ha complessità O(n), il metodo presenta molteplici for innestati tra di loro, pertanto il tempo aumentava vertiginosamente, anche il metodo isClosed era di classe O(n):

```
8  import java.util.ArrayList;
9  import java.util.Arrays;
10 import java.util.LinkedList;
11 import java.util.List;
12
13 public class Factorizer {
14
15    public static FactorizationResult minimalFactorizationDP(String text) {
16       int n = text.length();
17       int[] dp = new int[n + 1];
18       long[] ways = new long[n + 1];
19
20
21       List<Integer>[] predecessors = new ArrayList[n + 1];
22       for (int k = 0; k <= n; k++) {
23          predecessors[k] = new ArrayList<>();
24       }
25
26       Arrays.fill(dp, Integer.MAX_VALUE);
27       dp[0] = 0;
28       ways[0] = 1;
29
30       for (int i = 1; i <= n; i++) {
31          for (int j = 0; j < i; j++) {
32             String sub = text.substring(j, i);
33             if (ClosedWordUtils.isClosed(sub)) {
34                if (dp[j] != Integer.MAX_VALUE) {
35                   int candidateLen = dp[j] + 1;
36
37
38                   if (candidateLen < dp[i]) {
39                      dp[i] = candidateLen;
40                      ways[i] = ways[j];
41
42
43                      predecessors[i].clear();
44                      predecessors[i].add(j);
45                   }
46
47                   else if (candidateLen == dp[i]) {
48                      ways[i] += ways[j];
```

```
                    predecessors[i].add(j);
                }
            }
        }
    }


    List<String> allSolutions = new ArrayList<>();



    if (dp[n] != Integer.MAX_VALUE) {
        collectPaths(n, predecessors, text, new LinkedList<>(), allSolutions, 10);
    }

    String example = allSolutions.isEmpty() ? "" : allSolutions.get(0);

    return new FactorizationResult(dp[n], example, ways[n], allSolutions);
}


private static void collectPaths(int currentIdx, List<Integer>[] preds, String text,
                         LinkedList<String> currentPath, List<String> results, int
                             limit) {

    if (results.size() >= limit) return;


    if (currentIdx == 0) {
        results.add(String.join(" ", currentPath));
        return;
    }


    for (int prevIdx : preds[currentIdx]) {
        String factor = "[" + text.substring(prevIdx, currentIdx) + "]";
        currentPath.addFirst(factor);

        collectPaths(prevIdx, preds, text, currentPath, results, limit);

        currentPath.removeFirst();
    }
}

public static FactorizationResult greedyFactorization(String text) {
    int n = text.length();
    int count = 0;
    int idx = 0;
    StringBuilder sb = new StringBuilder();

    while (idx < n) {
        int bestEnd = idx + 1;
        for (int end = n; end > idx; end--) {
            String sub = text.substring(idx, end);
            if (ClosedWordUtils.isClosed(sub)) {
                bestEnd = end;
                break;
            }
        }
        sb.append("[").append(text.substring(idx, bestEnd)).append("] ");
        count++;
        idx = bestEnd;
    }

    List<String> singleSol = new ArrayList<>();
```

```
        singleSol.add(sb.toString().trim());

        return new FactorizationResult(count, sb.toString().trim(), 1, singleSol);
    }
}

//classe ClosedWordUtils

    public static boolean isClosed(String w) {
        int n = w.length();
        if (n <= 1) return true;

        String border = "";
        int lenB = 0;


        for (int i = n - 1; i > 0; i--) {
            if (w.startsWith(w.substring(n - i))) {
                border = w.substring(0, i);
                lenB = i;
                break;
            }
        }

        if (lenB == 0) return false;


        int suffixStartIndex = n - lenB;
        int foundIndex = w.indexOf(border, 1);

        if (foundIndex < suffixStartIndex) {
            return false;
        }
        return true;
    }
```

L'intento in un primo momento è stato quello di ottimizzare il metodo isClosed e per farlo mi sono affidata a un algoritmo di ricerca di sequenze, l'algoritmo Knuth-Morris-Pratt, di conseguenza ho dovuto definire un array per tenere traccia del bordo della parola:

```
public class KMPUtils {

    // Calcolo l'array dei bordi e lo faccio in un tempo di O(n), in quanto solo il for
        incide sul tempo;
    // infatti il while pu essere eseguito al massimo n volte in totale, poich ogni volta
        che si entra nel while
    // si riduce j, che pu essere incrementato al massimo n volte dal ciclo for principale.
    // Rendendo il costo del while ammortizzato.
    // Di conseguenza ho n operazioni totali effettuate dal for, n operazioni totali
        effettuate dal while e
    // ho n+n=2n operazioni totali, di conseguenza la complessit temporale  di O(n) e una
        complessit
    // spaziale di O(n) per l'array dei bordi.

    public static int[] borderArray(char[] text, int start, int end) {
        int m = end - start + 1;
        int[] border = new int[m];
        border[0] = 0; // il bordo del primo carattere  sempre 0
        int j = 0; // lunghezza del bordo corrente

        for (int i = 1; i < m; i++) {
            while (j > 0 && text[start + i] != text[start + j]) {
                j = border[j - 1]; // fallback al bordo precedente
            }
            // Se il carattere corrente (i)  uguale al carattere che mi aspetto dal prefisso
                (j),
            // significa che il bordo continua.
```

```
172            // Esempio: in "aaaaa", la quinta 'a' estende il bordo.
173            //In "aaaab", la 'b' rompe la sequenza e non entro qui.
174            if (text[start + i] == text[start + j]) {
175               j++;
176            }
177            border[i] = j;
178         }
179         return border;
180      }
181
182
183      /*
184      Calcolo la chiusura della parola, ogni operazione viene svolta in tempo costante,
           eccezion fatta per il for,
185      che viene richiamato al massimo n-1 volte (n volte), di conseguenza questo metodo ha
           complessit di O(n),
186      richiamando anche borderArray solo una volta significa che ho comunque n+n=2n operazioni
           svolte, pertanto
187      il risultato di O(n) come complessit
188      */
189      public static boolean isClosedOpt(char[] text, int start, int end) {
190         int n = end - start + 1;
191
192         //la parola ha lunghezza di al massimo 1, quindi  chiusa, anche la parola vuota
              chiusa
193         if(n<=1) return true;
194
195         int[] b = borderArray(text, start, end);
196
197         int lenB = b[n-1];
198
199         if(lenB==0) return false;
200
201         for (int i = 0; i < n - 1; i++) {
202            if (b[i] == lenB) {
203               return false; // Occorrenza interna trovata
204            }
205         }
206
207         return true; //nessuna occorrenza trovata => parola chiusa
208      }
209
210   }
```

Dopodiché ho provato a capire come modificare il controllo e la fattorizzazione delle parole, in un primo momento raggiungeva complessità $O(n\hat{3})$, soprattutto con le modifiche apportate al metodo isClosed:

```
211   import java.util.ArrayList;
212   import java.util.Arrays;
213   import java.util.LinkedList;
214   import java.util.List;
215
216   public class Factorizer {
217
218      /*
219      Il seguente metodo  progettato per trovare un numero minimo di fattorizzazioni di una
           stringa data
220      per trovare sottostringhe chiuse utilizzando un approccio di programmazione dinamica.
221      Trasforma la stringa di input in un array di caratteri e inizializza diverse strutture
           dati per tenere traccia
222      in modo tale da avere una complessit temporale di O(n^3) (o di O(n^2) nel migliore dei
           casi) e
223      spaziale di O(n).
224      Restituisce un oggetto FactorizationResult che contiene il numero minimo di
           fattorizzazioni.
225      */
226
```

```java
public static FactorizationResult minimalFactorizationDP(String textString) {

    char[] text = textString.toCharArray();
    int n = text.length;

    int[] dp = new int[n + 1];
    // dp[i] rappresenta il costo minimo per fattorizzare il prefisso di lunghezza i
    long[] ways = new long[n + 1];
    // ways[i] rappresenta il numero di modi per ottenere il costo minimo
    // per fattorizzare il prefisso di lunghezza i
    int[] parent = new int[n + 1];
    // parent[i] rappresenta l'indice di partenza dell'ultima sottostringa chiusa



    List<Integer>[] buckets = new ArrayList[n + 1];
    // buckets[cost] contiene gli indici i tali che dp[i] == cost
    for (int k = 0; k <= n; k++) buckets[k] = new ArrayList<>();

    Arrays.fill(dp, Integer.MAX_VALUE);

    dp[0] = 0;
    ways[0] = 1;
    buckets[0].add(0);

    int minCostFound = 0;
    int j;
    int newCost;
    boolean foundOptimal;

    for (int i = 1; i <= n; i++) {
    // calcoliamo dp[i], ways[i] e parent[i],  il primo ciclo for che scorre tutti i
        prefissi della stringa
    // e ha complessit O(n)
       foundOptimal = false;

       for (int cost = minCostFound; cost < i; cost++) {
       // calcoliamo dp[i] scorrendo i bucket dei costi crescenti
       // e ha complessit O(n) nel caso peggiore, altrimenti  costante

          if (buckets[cost].isEmpty()) continue;

          for (int indexInBucket = buckets[cost].size() - 1; indexInBucket >= 0;
              indexInBucket--) {
          // scorre tutti gli indici j tali che dp[j] == cost
          // e ha complessit O(n) nel caso peggiore
             j = buckets[cost].get(indexInBucket);

             // controlliamo se la sottostringa text[j..i-1]  chiusa e nonostante abbia
                 complessit O(n),
             // l'uso di bucket e la rottura anticipata del ciclo sui costi
             // riducono notevolmente il numero di chiamate a isClosedOpt,
             // migliorando le prestazioni complessive.
             if (KMPUtils.isClosedOpt(text, j, i - 1)) {
                newCost = cost + 1;
                if (newCost < dp[i]) {
                   dp[i] = newCost;
                   ways[i] = ways[j];
                   parent[i] = j;
                   foundOptimal = true;
                }
                else if (newCost == dp[i]) {
                   ways[i] += ways[j];
                }
             }
          }

```

```
291
292             if (foundOptimal) {
293                 break;
294                 //interrompiamo il ciclo sui costi, in modo tale da ridurre la complessit
                        temporale
295             }
296         }
297
298         if (dp[i] <= n) {
299             buckets[dp[i]].add(i);
300         }
301     }
302
303     return new FactorizationResult(dp[n], "", ways[n], new ArrayList<>());
304 }
305 }
```

Ma comunque continuava a essere troppo lento, infatti eseguendolo su parole con lunghezza di circa 10000 caratteri ottenevo risultati elevati, ad esempio il tempo impiegato per fattorizzare una parola così lunga era intorno ai 20 minuti:

```
TEST: Random (Len=10.000, Alpha=2)
[1] Factorizer... Done in 1263418 ms (circa 21 minuti) | Fattori: 5
```

Mi sono dedicata a un'ulteriore modifica e ho iniziato a calcolare l'array di bordi non più dall'inizio alla fine, ma da posizioni specifiche che vengono incrementate man mano, in modo tale da ammortizzare l'approccio KMP, ragion per cui il risultato della complessità temporale scende da $O(n\hat{3})$ a $O(n\hat{2})$:

```
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.util.LinkedList;
6
7  public class FactorizerOpt {
8
9      /*
10     Questo metodo implementa un algoritmo di fattorizzazione ottimale
11     utilizzando la programmazione dinamica e il concetto di prefissi
12     ripetuti (border array) per ridurre il numero di sottostringhe da considerare.
13     Sfrutta degli indici per iniziare e aggiornare la tabella DP in modo efficiente, in modo
           tale
14     da ridurre la complessit temporale passando da una complessit cubica a una complessit
           quadratica.
15     */
16
17     public static FactorizationResult minimalFactorizationDP(String textString) {
18         if (textString == null || textString.isEmpty()) {
19             return new FactorizationResult(0, "", 1, new ArrayList<>());
20         }
21
22         char[] text = textString.toCharArray();
23         int n = text.length;
24
25         int[] dp = new int[n + 1];
26         long[] ways = new long[n + 1];
27         int[] parent = new int[n + 1];
28
29         Arrays.fill(dp, Integer.MAX_VALUE);
30         Arrays.fill(parent, -1);
31         dp[0] = 0;
32         ways[0] = 1;
33
34         /*
35         Variabili usate per la costruzione del border array e il conteggio dei prefissi
36         */
37
38         int[] border = new int[n];
39         int[] count = new int[n + 1];
```

```
40
41
42          for (int j = 0; j < n; j++) {
43
44
45              if (dp[j] == Integer.MAX_VALUE) continue;
46
47              // Reset border e count per la nuova posizione j
48              Arrays.fill(count, 0, n - j + 1, 0);
49
50              border[0] = 0;
51              count[0] = 1;
52
53              // Considera il singolo carattere come fattore e ha complessit costante
54              updateDP(dp, ways, parent, j, j + 1, dp[j] + 1);
55
56              int currentMaxLen = n - j;
57
58              for (int len = 2; len <= currentMaxLen; len++) {
59                  int i = j + len - 1;
60
61                  int k = border[len - 2];
62
63                  while (k > 0 && text[j + k] != text[i]) {
64                      k = border[k - 1];
65                  }
66
67                  if (text[j + k] == text[i]) {
68                      k++;
69                  }
70
71                  border[len - 1] = k;
72                  count[k]++;
73
74                  if (k > 0 && count[k] == 1) {
75                      updateDP(dp, ways, parent, j, i + 1, dp[j] + 1);
76                  }
77              }
78          }
79
80          String solution = reconstructPath(textString, parent, n);
81          return new FactorizationResult(dp[n], solution, ways[n], new ArrayList<>());
82      }
83
84      /*
85      Questo metodo aggiorna la tabella DP, il conteggio dei modi e il genitore
86      per una data sottostringa in modo efficiente.
87      */
88      private static void updateDP(int[] dp, long[] ways, int[] parent, int start, int target,
            int newCost) {
89          if (newCost < dp[target]) {
90              dp[target] = newCost;
91              ways[target] = ways[start];
92              parent[target] = start;
93          } else if (newCost == dp[target]) {
94              ways[target] += ways[start];
95          }
96      }
97
98      private static String reconstructPath(String text, int[] parent, int n) {
99          if (n == 0 || dpIsInvalid(parent, n)) return "";
100         LinkedList<String> factors = new LinkedList<>();
101         int current = n;
102         while (current > 0) {
103             int start = parent[current];
104             if (start == -1) break;
105             factors.addFirst("[" + text.substring(start, current) + "]");
```

```
106             current = start;
107         }
108         return String.join(" ", factors);
109     }
110 
111     private static boolean dpIsInvalid(int[] parent, int n) {
112         return parent[n] == -1;
113     }
114 }
```

E attualmente l'algoritmo fattorizza parole con 10000 o più caratteri in un tempo di millisecondi:

```
>>> TEST: Random 10.000 char (Alpha=2)
Esecuzione FactorizerOpt... Done!
  Tempo: 363 ms (0.36 sec)
  Fattori: 5
  Soluzione: [aaabbbbabbaaa] [baab]
      [aababababbabaababababbaabaababbababaababababbbbaabababbbaaababababbaabaababaaaabbaa...
_____________________________________________________________________________

>>> TEST: Random 20.000 char (Alpha=2)
Esecuzione FactorizerOpt... Done!
  Tempo: 1,536 ms (1.54 sec)
  Fattori: 5
  Soluzione: [bababbababababaaabababbbbbabbbbabab...
_____________________________________________________________________________

>>> TEST: Random 50.000 char (Alpha=2)
Esecuzione FactorizerOpt... Done!
  Tempo: 9,039 ms (9.04 sec)
  Fattori: 4
  Soluzione: [baabbbbaabaabaaaaabbaabaabbbaaababbabbbaba...
_____________________________________________________________________________
```

È capace di svolgere e fare il controllo su tutti i fattori anche per parole molto lunghe impiegando un tempo bassissimo. In aggiunta usando solo degli array ha complessità spaziale lineare, ragion per cui non occupa troppo spazio e si possono studiare risultati anche per parole infinitamente più lunghe.