# Fondamenti di Informatica (Elettronici)

THINK JULIA – Capitolo 9

11 novembre 2020

# Case Study: giocare con le parole[1]

1. Lettura di elenchi di parole
2. Esercizi
3. Ricerca
4. Ciclare con indici
5. Debug
6. Glossario
7. Esercizi

[1] Tratto da https://benlauwens.github.io/ThinkJulia.jl/latest/book.html, disponibile sotto Licenza 'Creative Commons Attribution-NonCommercial 3.0 Unported'.

Questo capitolo presenta il secondo caso di studio, che implica la risoluzione di enigmi di parole cercando parole che hanno determinate proprietà.

Ad esempio, troveremo i palindromi più lunghi in inglese e cercheremo le parole le cui lettere compaiono in ordine alfabetico.

Verrà anche presentato un altro piano di sviluppo del programma: la riduzione a un problema precedentemente risolto.

Section 1

# Lettura di elenchi di parole

# Lettura di elenchi di parole

Per gli esercizi di questo capitolo abbiamo bisogno di un elenco di parole inglesi.

Ci sono molti elenchi di parole disponibili sul Web, ma quello più adatto al nostro scopo è uno degli elenchi di parole raccolti e resi di pubblico dominio da Grady Ward come parte del progetto "Lessico Moby" (vedere https://wikipedia.org/wiki/Moby_Project).

È un elenco di 113.809 parole e frasi ufficiali; ovvero, parole considerate valide nei cruciverba e in altri giochi di parole.

Nella raccolta Moby, il nome del file è 113809of.fic; puoi scaricarne una copia, con il nome `words.txt`, da https://github.com/BenLauwens/ThinkJulia.jl/blob/master/data/words.txt.

# IOStream (Flusso id I/O)

Questo file è in testo normale, quindi puoi aprirlo con un editor di testo, ma puoi anche leggerlo da Julia.

La funzione incorporata `open` prende il nome del file come parametro e restituisce un data stream (flusso di dati) che puoi usare per leggere il file.

```
julia> fin = open("words.txt")
IOStream(<file words.txt>)
```

`fin` è uno stream dal file utilizzato per l'input e quando non è più necessario, deve essere chiuso con `close(fin)`.

Julia fornisce diverse funzioni per la lettura, inclusa `readline`, che legge i caratteri dal file fino a quando arriva a una NEWLINE (`\n`) e restituisce il risultato come una stringa:

```
julia> readline(fin)
"aa"
```

Section 2

# Esercizi

# Exercise 9-1

Scrivete un programma che legga `words.txt` e stampi solo le parole con più di 20 caratteri (senza contare gli spazi).

# Esercizio 9-2

Nel 1939 Ernest Vincent Wright pubblicò un romanzo di 50.000 parole chiamato Gadsby che non contiene la lettera e.

Poiché e è la lettera più comune in inglese, non è facile. In effetti, è difficile costruire un pensiero solitario senza utilizzare quel simbolo più comune.

All'inizio è lento, ma con cautela e ore di allenamento puoi gradualmente acquisire facilità. Va bene, ora mi fermo. (motto di spirito di patata)

Scrivi una funzione chiamata `hasno_e` che restituisca `true` se la parola data non contiene la lettera e.

Modifica il tuo programma dalla sezione precedente per stampare solo le parole che non hanno e e calcola la percentuale delle parole nell'elenco che non hanno e.

# Esercizio 9-3/5

## Esercizio 9-3

Scrivi una funzione denominata `avoids` che accetti una parola e una stringa di lettere proibite e che restituisca true se la parola non utilizza nessuna delle lettere proibite.
Modifica il tuo programma per chiedere all'utente di inserire una stringa di lettere proibite e quindi stampare il numero di parole che non ne contengono.
Riesci a trovare una combinazione di 5 lettere proibite che escluda il minor numero di parole? (come?)

## Esercizio 9-4

Scrivi una funzione denominata `usesonly` che accetti una parola e una stringa di lettere e che restituisca `true` se la parola contiene solo lettere nell'elenco.
Puoi fare una frase usando solo le lettere acefhlo? Altro che "Hoe alfalfa?"

## Esercizio 9-5

# Esercizio 9-6

## Esercizio 9-6

Scrivi una funzione chiamata `isabecedarian` che restituisca `true` se le lettere di una parola appaiono in ordine alfabetico (le doppie lettere sono `ok`).
Quante parole abecedarie ci sono?

Section 3

# Ricerca

# pattern di ricerca

Tutti gli esercizi della sezione precedente hanno qualcosa in comune; possono essere risolti con il pattern di ricerca.

L'esempio più semplice è:

```
function hasno_e(word)
    for letter in word
        if letter == 'e'
            return false
        end
    end
    true
end
```

Il ciclo scorre attraversa i caratteri nelle parole.

Se troviamo la lettera `e`, possiamo immediatamente restituire `false`; altrimenti dobbiamo passare alla lettera successiva.

# Funzione avoids

Potresti scrivere questa funzione in modo più conciso usando l'operatore `!in` (∉ TAB), ma ho iniziato con questa versione perché dimostra la logica del pattern di ricerca.

`Avoids` è una versione più generale di `hasno_e` ma ha la stessa struttura:

```
function avoids(word, forbidden)
    for letter in word
        if letter in forbidden
            return false
        end
    end
    true
end
```

Possiamo restituire "`false`" non appena troviamo una lettera proibita; se arriviamo alla fine del ciclo, restituiamo "`true`".

# Funzione usesonly

usesonly è simile tranne che il senso della condizione è invertito:

```
function usesonly(word, available)
    for letter in word
        if letter !in available
            return false
        end
    end
    true
end
```

Invece di un array di lettere proibite, abbiamo un array di lettere disponibili.

Se troviamo nella parola una lettera che non è disponibile, possiamo restituire `false`.

# Funzione usesall 1/2

usesall è simile tranne che invertiamo il ruolo della parola e della stringa di lettere:

```
function usesall(word, required)
    for letter in required
        if letter !in word
            return false
        end
    end
    true
end
```

Invece di attraversare le lettere nelle parole, il ciclo attraversa le lettere richieste.

Se una qualsiasi delle lettere richieste non compare nella parola, possiamo restituire `false`.

# ## Funzione `usesall` 2/2

Se stavi davvero pensando come uno scienziato informatico, avresti riconosciuto che `usesall` era un'istanza di un problema risolto in precedenza e avresti scritto:

```
function usesall(word, required)
    usesonly(required, word)
end
```

Questo è un esempio di un piano di sviluppo del programma chiamato riduzione a un problema risolto in precedenza, il che significa che riconosci il problema su cui stai lavorando come istanza di un problema risolto e applichi una soluzione esistente.

Section 4

# Ciclare con indici

# Funzione isabecedarian 1/4

Ho scritto le funzioni nella sezione precedente con i cicli `for` perché avevo bisogno solo dei caratteri nelle stringhe; Non ho dovuto fare nulla con gli indici.

Per `isabecedarian` dobbiamo confrontare le lettere adiacenti, che è un po 'complicato con un ciclo `for`:

```
function isabecedarian(word)
    i = firstindex(word)
    previous = word[i]
    j = nextind(word, i)
    for c in word[j:end]
        if c < previous
            return false
        end
        previous = c
    end
    true
end
```

# Funzione isabecedarian 2/4

Un'alternativa è usare la ricorsione:

```
function isabecedarian(word)
    if length(word) <= 1
        return true
    end
    i = firstindex(word)
    j = nextind(word, i)
    if word[i] > word[j]
        return false
    end
    isabecedarian(word[j:end])
end
```

# Funzione isabecedarian 3/4

Un'altra opzione è usare un ciclo while:

```
function isabecedarian(word)
    i = firstindex(word)
    j = nextind(word, 1)
    while j <= sizeof(word)
        if word[j] < word[i]
            return false
        end
        i=j
        j = nextind(word, i)
    end
    true
end
```

Il ciclo inizia con `i = 1` e `j = nextind(word, 1)` e termina quando `j > sizeof(word)`.

# Funzione `isabecedarian` 4/4

Ogni volta nel ciclo, confronta il i-esimo carattere (che puoi pensare come il carattere corrente) con il j-esimo carattere (che puoi pensare come il prossimo).

Se il carattere successivo è inferiore (in ordine alfabetico diretto) a quello corrente, allora abbiamo scoperto una rottura nell' ordinamento abecedario e restituiamo `false`. Se arriviamo alla fine del ciclo senza trovare un errore, la parola supera il test.

Per convincerti che il ciclo finisce correttamente, considera un esempio come "flossy".

# Funzione ispalindrome

Ecco una versione di `ispalindrome` che utilizza due indici; si parte dall'inizio e si sale; l'altro inizia alla fine e scende.

```
function ispalindrome(word)
    i = firstindex(word)
    j = lastindex(word)
    while i<j
        if word[i] != word[j]
            return false
        end
        i = nextind(word, i)
        j = prevind(word, j)
    end
    true
end
```

Oppure potremmo ridurre a un problema precedentemente risolto e scrivere:

```
function ispalindrome(word)
    isreverse(word, word)
end
```

Utilizzando `isreverse` da Debugging.

Section 5

# Debug

# Testare i programmi è difficile

Le funzioni in questo capitolo sono relativamente facili da testare perché puoi controllare i risultati manualmente.

Anche così, è tra difficile e impossibile scegliere un insieme di parole che testano tutti i possibili errori.

Prendendo `hasno_e` come esempio, ci sono due casi ovvi da controllare: le parole che hanno una e dovrebbero restituire `false` e le parole che non hanno una e dovrebbero restituire `true`.

Non dovresti avere problemi a trovare uno di ciascuno.

# Serie di casi di test

All'interno di ogni caso, ci sono alcuni sotto-casi meno ovvi.

- Tra le parole che hanno una "e", dovresti testare le parole con una "e" all'inizio, alla fine, e da qualche parte nel mezzo.

# Serie di casi di test

All'interno di ogni caso, ci sono alcuni sotto-casi meno ovvi.

- Tra le parole che hanno una "e", dovresti testare le parole con una "e" all'inizio, alla fine, e da qualche parte nel mezzo.
- Dovresti testare parole lunghe, parole brevi e parole molto brevi, come la stringa vuota.

# Serie di casi di test

All'interno di ogni caso, ci sono alcuni sotto-casi meno ovvi.

- Tra le parole che hanno una "e", dovresti testare le parole con una "e" all'inizio, alla fine, e da qualche parte nel mezzo.
- Dovresti testare parole lunghe, parole brevi e parole molto brevi, come la stringa vuota.
- La stringa vuota è un esempio di un caso speciale, che è uno dei casi non ovvi in cui spesso si nascondono errori.

# Serie di casi di test

All'interno di ogni caso, ci sono alcuni sotto-casi meno ovvi.

- Tra le parole che hanno una "e", dovresti testare le parole con una "e" all'inizio, alla fine, e da qualche parte nel mezzo.
- Dovresti testare parole lunghe, parole brevi e parole molto brevi, come la stringa vuota.
- La stringa vuota è un esempio di un caso speciale, che è uno dei casi non ovvi in cui spesso si nascondono errori.
- Oltre ai casi di test che generi, puoi anche testare il tuo programma con un elenco di parole come words.txt.

# Serie di casi di test

All'interno di ogni caso, ci sono alcuni sotto-casi meno ovvi.

- Tra le parole che hanno una "e", dovresti testare le parole con una "e" all'inizio, alla fine, e da qualche parte nel mezzo.
- Dovresti testare parole lunghe, parole brevi e parole molto brevi, come la stringa vuota.
- La stringa vuota è un esempio di un caso speciale, che è uno dei casi non ovvi in cui spesso si nascondono errori.
- Oltre ai casi di test che generi, puoi anche testare il tuo programma con un elenco di parole come words.txt.
- Analizzando l'output, potresti essere in grado di rilevare gli errori, ma fai attenzione: potresti rilevare un tipo di errore (parole che non dovrebbero essere incluse, ma lo sono) e non un altro (parole che dovrebbero essere incluse, ma non lo sono).

# Serie di casi di test

All'interno di ogni caso, ci sono alcuni sotto-casi meno ovvi.

- Tra le parole che hanno una "e", dovresti testare le parole con una "e" all'inizio, alla fine, e da qualche parte nel mezzo.
- Dovresti testare parole lunghe, parole brevi e parole molto brevi, come la stringa vuota.
- La stringa vuota è un esempio di un caso speciale, che è uno dei casi non ovvi in cui spesso si nascondono errori.
- Oltre ai casi di test che generi, puoi anche testare il tuo programma con un elenco di parole come words.txt.
- Analizzando l'output, potresti essere in grado di rilevare gli errori, ma fai attenzione: potresti rilevare un tipo di errore (parole che non dovrebbero essere incluse, ma lo sono) e non un altro (parole che dovrebbero essere incluse, ma non lo sono).
- In generale, i test possono aiutarti a trovare bug, ma non è facile generare una buona serie di casi di test e, anche se lo fai, non puoi essere sicuro che il tuo programma sia corretto.

# Assenza di bug?

Secondo un leggendario scienziato informatico:

> *Il test del programma può essere utilizzato per mostrare la presenza di bug, ma mai per mostrare la loro assenza!*
>
> *Edsger W. Dijkstra*

Section 6

# Glossario

# Glossario

stream da file Un valore che rappresenta un file aperto.

# Glossario

**stream da file** Un valore che rappresenta un file aperto.

**riduzione a un problema precedentemente risolto** Un modo per risolvere un problema esprimendolo come un'istanza di un problema risolto in precedenza.

# Glossario

**stream da file** Un valore che rappresenta un file aperto.

**riduzione a un problema precedentemente risolto** Un modo per risolvere un problema esprimendolo come un'istanza di un problema risolto in precedenza.

**caso speciale** Uno scenario di test atipico o non ovvio (e con meno probabilità di essere gestito correttamente).

# aaaa

Section 7

# Esercizi

# aaaa

# aaaa