Coding per la statistica e la Data Science

Università degli Studi di Torino

Gabriele Alfarone, Marianna Carlone

Secondo semestre a.a. 2023/24

Contents

1 Nozioni di base della programmazione

La programmazione è il processo di scrivere istruzioni che un computer può eseguire per risolvere un problema specifico o svolgere una determinata attività. Comprendere le nozioni di base della programmazione e non lo studio a memoria di funzioni, librerie, ecc. è fondamentale per potere sviluppare applicazioni software e risolvere problemi in maniera efficiente.

Algoritmo

Per algoritmo si intende una sequenza di istruzioni ben definite e ordinate che descrivono un metodo per risolvere un problema. Gli algoritmi forniscono concettualmente un guida step-by-step su come eseguire una determinata attività.

Strutture dati

Le strutture dati sono metodi per organizzare e memorizzare dati in modo efficace all'interno di un programma. Le strutture dati possono essere di diversa natura, le più comuni includono vettori, liste, pile, code, grafi. Le variabili invece sono simboli utilizzati per memorizzare dati in memoria durante l'esecuzione di un programma.

Variabili

Le variabili invece sono simboli utilizzati per memorizzare dati in memoria durante l'esecuzione di un programma. Possono contenere valori di diversi tipi, come numeri, stringhe o oggetti complessi. Nel corso si andrà ad approfondire come le variabili interagiscano tra loro in base alla diversa tipologia.

Controllo di flusso

Il controllo di flusso determina l'ordine in cui le istruzioni vengono eseguite in un programma. Le strutture di flusso comuni includono istruzioni condizionali (if-else), cicli (for, while) e istruzioni salto (break, continue). Per quanto riguarda `R`, esso ci offre diversi modi per evitare i cicli. Ad esempio, se si ha un vettore e se ne vuole ottenere la somma degli elementi potrei sia risolvere il problema attraverso un ciclo, come succede in altri linguaggi di programmazione, sia utilizzando una funzione integrata nel sistema. Le funzioni built-in di R sono vettoriali

Funzioni

Le funzioni sono blocchi di codice autonomi che eseguono una determinata operazione. Le funzioni consentono di organizzare il codice in unità riutilizzabili e modulari, migliorando la leggibilità e la manutenibilità del codice. Non sempre il codice "fila liscio", quindi la gestione degli errori e il processo di identificazione sono strettamente necessari. Le tecniche comuni includono la validazione dell'input, la gestione delle eccezioni e il debug del codice.

Pensiero Algoritmico

Il pensiero algoritmico è la capacità di risolvere problemi in modo logico e sistematico, identificando gli algoritmi appropriati e applicando tecniche di risoluzione dei problemi. Il pensiero algoritmico è una competenza fondamentale per diventare un programmatore efficace.

1.1 Esempi di algoritmi

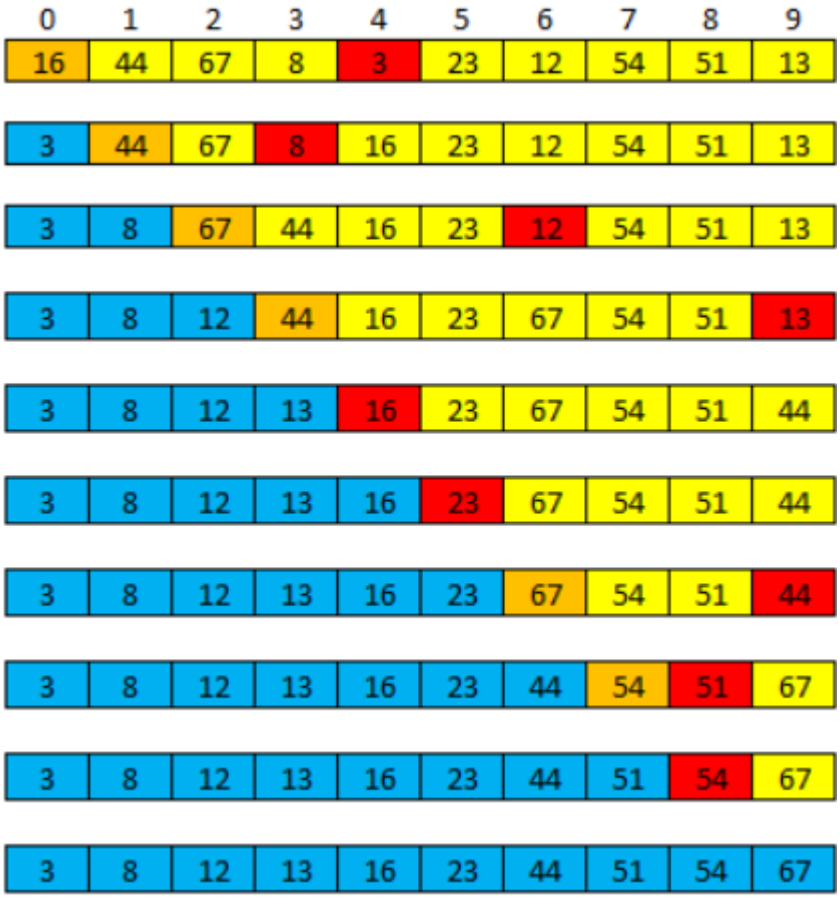
In questa sezione esploreremo diversi esempi di algoritmi utilizzati per risolvere una vasta gamma di problemi computazionali, come:

- ordinamento di una lista di numeri: algoritmo di ordinamento a selezione
- ricerca di un elemento in un array: ricerca sequenziale o ricerca binaria
- calcolo del fattoriale di un numero: ricorsione
- risoluzione di un labirinto: algoritmo di ricerca
- compressione di dati: algoritmo di compressione basato su frequenze

1.1.1 Algoritmo di ordinamento a selezione Implementato (vedi Appunti-finali.Rmd)

L'algoritmo di ordinamento a selezione identifica di volta in volta il numero minore nella sequenza di partenza e lo sposta nella sequenza ordinata; di fatto la sequenza viene suddivisa in due parti: la sottosequenza ordinata, che occupa le prime posizioni dell'array, e la sottosequenza da ordinare, che costituisce la parte restante dell'array. Dovendo ordinare un array A di lunghezza n, si fa scorrere l'indice i da 1 a $n-1$ ripetendo i seguenti passi:

- si cerca il più piccolo elemento della sottosequenza $A[i \dots n]$;
- si scambia questo elemento con l'elemento i-esimo.



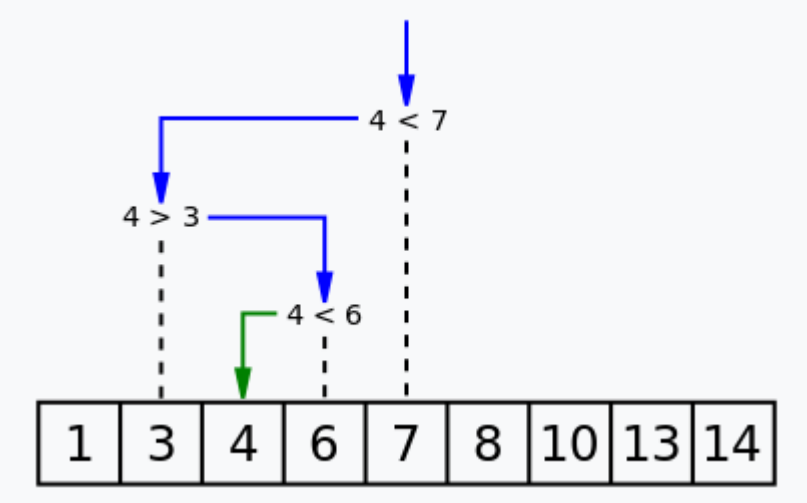
1.1.2 Ricerca sequenziale Implementato (vedi Appunti-finali.Rmd)

In informatica la ricerca sequenziale (o ricerca lineare) è un algoritmo utilizzabile per trovare un elemento in un insieme non ordinato. L'algoritmo controlla in sequenza gli elementi dell'insieme, arrestandosi quando ne trova uno che soddisfa il criterio di ricerca; non potendosi avvalere di alcun ordinamento tra gli elementi, l'algoritmo può concludere con certezza che l'insieme non contiene alcun elemento corrispondente solo dopo averli verificati tutti, richiedendo pertanto un numero di controlli, nel caso peggiore, pari alla cardinalità dell'intero insieme.

1.1.3 Ricerca binaria Implementato (vedi Appunti-finali.Rmd)

L'algoritmo cerca un elemento all'interno di un array che deve necessariamente essere ordinato in ordine crescente, effettuando mediamente meno confronti rispetto ad una ricerca sequenziale, e quindi più rapidamente rispetto a quest'ultima perché, sfruttando l'ordinamento, dimezza l'intervallo di ricerca ad ogni passaggio. La ricerca binaria non usa mai più di $log_2(N)$ confronti per ricercare una valore. L'algoritmo è simile al metodo usato per poter ricevere ambo i lati, ovvero trovare una parola sul dizionario: sapendo che il vocabolario è ordinato secondo l'alfabeto, l'idea è quella di iniziare la ricerca non dal primo elemento, ma da quello centrale, cioè a metà del dizionario. Si confronta questo elemento con quello cercato:

- se corrisponde, la ricerca termina indicando che l'elemento è stato trovato;
- se è superiore, la ricerca viene ripetuta sugli elementi precedenti (ovvero sulla prima metà del dizionario), scartando quelli successivi;
- se invece è inferiore, la ricerca viene ripetuta sugli elementi successivi (ovvero sulla seconda metà del dizionario), scartando quelli precedenti.

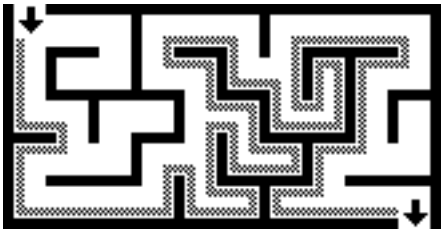


1.1.4 Ricorsione

La programmazione ricorsiva si basa sull'idea che per molti problemi la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema. La soluzione di un problema viene individuata supponendo di saperlo risolvere su casi più semplici.

1.1.5 Algoritmo di ricerca - Labirinto

Il wall follower, la regola più nota per attraversare i labirinti, è anche noto come regola della mano sinistra o della mano destra. Se il labirinto è semplicemente collegato, cioè tutte le sue pareti sono collegate insieme o al confine esterno del labirinto, mantenendo una mano in contatto con una parete del labirinto si garantisce che il risolutore raggiungerà un'uscita diversa, se ce n'è una.



1.1.6 Algoritmo di compressione

Il problema che ci si pone è quello di comprimere una sequenza di dati. Una possibile soluzione potrebbe essere quella di assegnare codici binari più corti alle sequenze di simboli più frequenti. Ad esempio, consideriamo la frequenza "*aaaabbc*":

- calcolare la frequenza di ogni simbolo; $a = 4$, $b = 2$, $c = 1$
- assegnare codici binari
- codice della sequenza: 0000101011

1.1.7 Altri esempi di algoritmi

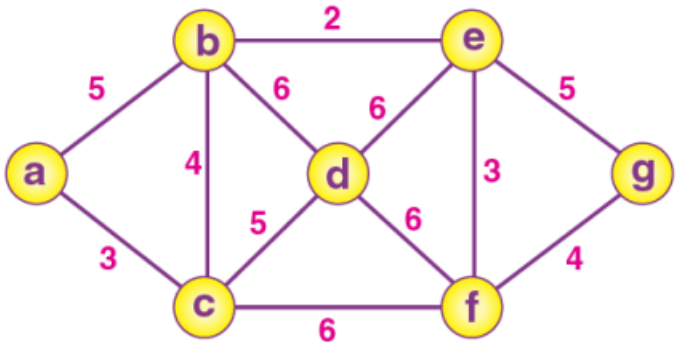
In questa sezione, introdurremo una serie di algoritmi per comprendere come vengono utilizzati per risolvere una varietà di problemi e ottimizzare i processi.

- Algoritmo **BubbleSort**: algoritmo di ordinamento a bolle. Utilizzato per ordinare una lista di elementi confrontando coppie di elementi adiacenti e scambiandoli se sono nell'ordine sbagliato. Il numero di iterazioni necessarie ad ordinare l'array dipende da quanto risulta "disordinato" l'array. Di solito nel calcolo della complessità si considera il *worst case*.
- Algoritmo di **permutazione**: utilizzato per generare tutte le possibili permutazioni di un insieme di elementi. Inventato nel 1700 in Inghilterra per produrre una combinazione diversa di suoni di campane ("tintinnalogia"), poi ripreso e migliorato da un informatico negli anni '70.
- Algoritmi di **ricerca degli elementi comuni**: utilizzati per trovare gli elementi comuni tra due o più insiemi di dati.
- Algoritmi di **calcolo del massimo**: utilizzato per trovare il massimo elemento in una lista di valori.
- Algoritmo di **ordinamento rapido** (Quicksort): un algoritmo di ordinamento efficiente che utilizza il concetto di partizionamento per ordinare una lista di elementi.
- Algoritmo di **visita in ampiezza** (Breadth-First Search, BFS): utilizzato per esplorare o attraversare tutti i nodi di un grafo in ampiezza, visitando tutti i nodi adiacenti prima di passare ai nodi successivi.
- Algoritmo di **visita in profondità** (Depth-First Search, DFS): utilizzato per esplorare o attraversare tutti i nodi di un grafo in profondità, continuando a scendere lungo un ramo finché non si raggiunge la fine, quindi tornando indietro e continuando con gli altri rami.

1.1.8 Scenario per pratica algoritmica: i grafi

Creare un algoritmo per ognuno di questi problemi:

- Ricerca del percorso più breve (Shortest Path):
- Ricerca del percorso minimo (Minimum Soanning Tree):
- Ricerca del cammino più lungo (Longest Path):
- Ricerca del flusso massimo (Maximum Flow):



1.1.9 Algoritmo di Kruskal (1956)

Utilizzato per trovare un albero di copertura minimo in un grafo con pesi sugli archi.
un algoritmo che restituisca in output il percorso da compiere per coprire tutti i nodi di un grafo con il peso totale minimo

1. Ordina gli archi del grafo in modo non decrescente rispetto al peso.
2. Inizializza un insieme vuoto di archi T.
3. Per ogni arco (u, v) nell'ordine: se aggiungere (u, v) a T non crea cicli, aggiungi (u, v) a T.
4. Restituisci l'insieme di archi T come albero di copertura minimo.

1.1.10 Algoritmo di Dijkstra (1956)

Ricerca del percorso più breve da un nodo di partenza a tutti gli altri nodi in un grafo con pesi sugli archi non negativi.

1. Inizializza un array $d[]$ di dimensione V dove $d[v]$ rappresenta la distanza più breve dal nodo di partenza s al nodo v, inizializzando tutte le distanze a infinito tranne $d[s]$ a 0.
2. Inizializza un set S dei nodi non ancora visitati.
3. Finché S non è vuoto:
   - Scegli il nodo u in S con la distanza minima stimata, $d[u]$.
   - Per ogni arco (u, v) adiacente a u:
     - se $d[u] + w(u, v) < d[v]$, aggiorna $d[v]$ a $d[u] + w(u, v)$.
     - Rimuovi u da S.

Conclusioni

Ci si rende conto che per implementare un algoritmo, c'è bisogno in primis di memorizzare dati, calcolarne altri, verificare se vengono rispettate determinate condizioni spesso all'interno di iterazioni. In linea generale quindi si parla di variabili, cicli, operazioni e strutture di controllo condizionale.

2 Variabili e Operazioni

La programmazione è l'arte e la scienza di creare istruzioni che un computer può eseguire per risolvere problemi e compiere azioni specifiche. Una delle prime cose che si impara quando si inizia a programmare è la dichiarazione e l'assegnazione di variabili. Le variabili sono come contenitori per i dati che si utilizzano nei programmi.

2.1 Assegnazione di variabili

Dopo aver dichiarato una variabile, è possibile assegnarle un valore utilizzando l'operatore di assegnazione `<-` o `=`. Ecco un esempio di come si potrebbe assegnare valori a variabili precedentemente dichiarate:

- assegna il testo "Ciao, mondo!" alla variabile *testo* - testo = "Ciao, mondo!";
- assegna il valore booleano vero alla variabile condizione - *condizione* <- TRUE;

2.2 Tipi di dati fondamentali

Esistono diverse tipologie di dati in base al linguaggio di programmazione utilizzato. I fondamentali risultano:

- *Numeri interi*: rappresentano numeri interi, positivi o negativi, senza parte frazionaria.
- *Numeri in virgola mobile*: sono utilizzati per rappresentare numeri reali che possono avere una parte frazionaria.
- *Caratteri o stringhe*: sono utilizzati per rappresentare sequenze di caratteri.
- *Booleani*: i valori booleani rappresentano verità o falsità.
- *Array*: sono strutture dati che contengono una raccolta di elementi dello stesso tipo.
- *Null/Nil*: è utilizzato per indicare l'assenza di un valore.

Le tipologie di dati più frequenti in `R` risultano essere:

- Numeri: interi (nell'environment sono seguiti dalla lettera "L") e numeri decimali.
- Stringhe di caratteri.
- Vettori: sequenze ordinate di elementi dello stesso tipo.
- Liste: collezioni ordinate di oggetti di diversi tipi.

2.3 Operazioni su variabili numeriche

Dichiarare due variabili numeriche a e b. Calcolare la somma, la differenza, il prodotto e il quoziente tra a e b.

```
a <- 3
b <- 5

a + b # Somma
```

```
## [1] 8
```

```r
a - b # Differenza
```

```
## [1] -2
```

```r
a * b # Prodotto
```

```
## [1] 15
```

```r
a / b # Divisione
```

```
## [1] 0.6
```

Calcolare il quadrato di un numero.

```r
a ^ 2
```

```
## [1] 9
```

```r
a ** 2
```

```
## [1] 9
```

```r
a^2 == a**2
```

```
## [1] TRUE
```

Tra i numeri interi è definita la funzione **modulo**, che dà come risultato il resto della divisione euclidea del primo numero per il secondo. Cioè dati $a, b \in \mathbb{Z}$, con $b \neq 0$, allora a modulo b dà come risultato il resto della divisione euclidea $\frac{a}{b}$.

```r
b %% a
```

```
## [1] 2
```

Per generare una sequenza di numeri casuali possiamo utilizzare il comando `sample()`.

```r
casuale <- sample(1:100, 1)
casuale
```

```
## [1] 74
```

Arrotondare un numero decimale.

```r
decimale <- 3.14159
round(decimale, digits = 2)
```

```
## [1] 3.14
```

`digits` indica il numero di cifre che si vogliono dopo la virgola.

2.4 Operazioni sulle stringhe

Dichiarare una variabile stringa `nome` e assegnarle un nome. Ottenere la lunghezza della stringa. Ottenere la stringa in maiuscolo.

```
nome <- "Mario"
nchar(nome)
```

```
## [1] 5
```

```
toupper(nome)
```

```
## [1] "MARIO"
```

Per ottenere la lunghezza di una stringa non è possibile utilizzare la funzione `length` poiché restituisce i numeri di elementi presenti in un oggetto. In questo caso restituirebbe 1.

Concatenare due stringhe.

```
stringa_1 <- "Hello"
stringa_2 <- "World"
paste(stringa_1, stringa_2)
```

```
## [1] "Hello World"
```

```
c(stringa_1, stringa_2)
```

```
## [1] "Hello" "World"
```

```
paste0(stringa_1, stringa_2)
```

```
## [1] "HelloWorld"
```

La funzione `paste` concatena due stringhe e crea a sua volta un'unica stringa, a differenza della funzione `c()`. Di default, `paste()` utilizza come separatore lo spazio, ma è possibile modificarlo attraverso il parametro `sep`. Invece la funzione `paste0()` concatena stringhe senza separatori. Può risultare utile in determinati casi estrarre una sottostringa e questo risulta possibile attraverso la funzione `substr()`.

```
frase <- "Questo è un esempio di sottostringa"
substr(frase, start = 11, stop = 17)
```

```
## [1] "n esemp"
```

Contare le occorrenze di una lettera in una stringa.

```
parola <- "banana"
lettera <- "a"
sum(strsplit(parola, "")[[1]]==lettera)
```

```
## [1] 3
```

```
typeof(strsplit(parola, "")[[1]])
```

```
## [1] "character"
```

La funzione `strsplit()` crea una **lista** di oggetti, in questo caso un vettore di caratteri. Nel caso specifico la stringa viene separata con il separatore `""`. Ciò permette di ottenere un vettore di caratteri con elementi le lettere che compongono la parola.

2.5 Ricerca e sostituzione di sottostringhe

`R` offre funzioni per la ricerca e la sostituzione di sottostringhe all'interno di una stringa. Per la ricerca di sottostringhe, possiamo utilizzare le funzioni `grep()` e `grepl()`. Sia `grep()` che `grepl()` sono funzioni utilizzate per cercare pattern in un vettore di caratteri (o in una lista di caratteri), ma differiscono nel modo in cui restituiscono i risultati.

grep():

- La funzione `grep()` restituisce gli indici degli elementi nel vettore che corrispondono al pattern specificato.
- Se non trova corrispondenze, restituisce un vettore vuoto.
- Se trova una corrispondenza, restituisce l'indice del primo elemento corrispondente.

grepl():

- La funzione `grepl()` restituisce un vettore logico che indica se c'è corrispondenza per ogni elemento del vettore di input.
- Per ogni elemento nel vettore, restituisce `TRUE` se il pattern è trovato in quell'elemento, altrimenti `FALSE`.
- Può essere usata per controllare se c'è almeno una corrispondenza per ogni elemento del vettore.

```
vettore <- c("ciao", "mondo", "hello", "world")
grep("o", vettore)
```

```
## [1] 1 2 3 4
```

```
testo <- "Questo è un esempio di testo"
sottostringa <- "esempio"
grepl(sottostringa, testo)
```

```
## [1] TRUE
```

Per la sostituzione di sottostringhe si può utilizzare la funzione `gsub()`.

```
nuovo_testo <- gsub("esempio", "frammento", testo)
nuovo_testo
```

```
## [1] "Questo è un frammento di testo"
```

2.6 Esercizi aggiuntivi

Esercizi aperti sulle stringhe

Scrivi un codice che generi una stringa di caratteri formata da tutte le lettere dell'alfabeto minuscole, partendo da "a" fino a "z".

```
paste(letters, collapse = "")
```

```
## [1] "abcdefghijklmnopqrstuvwxyz"
```

Scrivi un codice che, data una stringa, stampi i primi 10 caratteri.

```
stringa <- "scrivo a caso una stringa"
substr(stringa, start = 1, stop = 10)
```

```
## [1] "scrivo a c"
```

Data una stringa, stampa una nuova stringa contenente solo caratteri unici presenti nella stringa in input, mantenendo l'ordine originale dei caratteri.

```
paste(unique(strsplit(stringa, "")[[1]]), collapse = "")
```

```
## [1] "scrivo auntg"
```

Da notare come al momento del `paste()` lo spazio venga considerato come un carattere. Infatti:

```
unique(strsplit(stringa, "")[[1]])
```

```
##  [1] "s" "c" "r" "i" "v" "o" " " "a" "u" "n" "t" "g"
```

Esercizi sui vettori

Creare un vettore `numeri` contenente i primi 5 numeri interi dispari. Aggiungere un numero intero pari al vettore. Stampare il vettore risultante.

```
numeri <- c(1,3,5,7,9)
numeri
```

```
## [1] 1 3 5 7 9
```

```
seq(1,9,2)
```

```
## [1] 1 3 5 7 9
```

```
numeri <- c(numeri, 2)
numeri
```

```
## [1] 1 3 5 7 9 2
```

Calcolare la somma degli elementi di un vettore.

```
sum(numeri)
```

```
## [1] 27
```

Ordinare un vettore in ordine crescente.

```
vettore <- c(5,3,8,1,9)
sort(vettore)
```

```
## [1] 1 3 5 8 9
```

Calcolare la media dei valori unici di un vettore.

```
vettore <- c(1,2,3,3,4,5,5)
mean(unique(vettore))
```

```
## [1] 3
```

Creare un vettore di numeri da -100 a 100 ogni 0.25 steps.

```
v <- seq(-100, 100, 0.25)
```

Calcolare la radice quadrata degli elementi del vettore (`sqrt()`) avendo cura di filtrare i valori negativi prima di eseguire il calcolo.

```
m <- sqrt(v[v >= 0])
```

Selezionare i primi 50 elementi del vettore risultato (gli elementi con indici da 1 a 50).

```
k <- m[1:50]
```

Eliminare le cifre decimali, ossia convertire i numeri in interi.

```
i <- as.integer(k)
```

Eliminare dal vettore di interi gli zeri, creando un nuovo vettore di numeri positivi e visualizzando il risultato.

```
i[i != 0]
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
## [39] 3 3 3 3 3 3 3 3
```

Esercizi sulle liste

Creare una lista dati contenente un vettore di nomi e un vettore di età. Aggiungere un nuovo nome e l'età corrispondente alla lista. Stampare la lista risultante.

```r
nomi <- c("Silvia", "Marianna", "Gabriele", "Aldo")
eta <- c(33, 24, 12, 8)
anagrafica <- list(nomi = nomi, eta = eta)
anagrafica
```

```
## $nomi
## [1] "Silvia"   "Marianna" "Gabriele" "Aldo"
##
## $eta
## [1] 33 24 12  8
```

```r
anagrafica$nomi <- c(nomi, "Giulia")
anagrafica$eta <- c(eta, 82)
anagrafica
```

```
## $nomi
## [1] "Silvia"   "Marianna" "Gabriele" "Aldo"     "Giulia"
##
## $eta
## [1] 33 24 12  8 82
```

Creare una lista `l3` che contenga 3 elementi:

- un vettore di numeri
- una stringa
- una lista contenete un singolo numero

```r
l3 <- list(c(1,2,3), "ciao", list(3.14))
l3
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "ciao"
##
## [[3]]
## [[3]][[1]]
## [1] 3.14
```

Unire due liste.

```r
lista1 <- list(a=1, b=2)
lista2 <- list(c=3, d=4)
lista_unione <-c(lista1, lista2)
lista_unione
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
##
## $d
## [1] 4
```

Rimuovere un elemento dalla lista.

```
lista <- list(a = 1, b = 2, c = 3)
lista <- lista[-which(names(lista) == "b")]
lista
```

```
## $a
## [1] 1
##
## $c
## [1] 3
```

Crea una nuova lista contenente solo gli elementi unici presenti nella lista di input.

```
r <- list(a = "Mamma mia che bello", b = "Oggi gioco con i LEGO")
r
```

```
## $a
## [1] "Mamma mia che bello"
##
## $b
## [1] "Oggi gioco con i LEGO"
```

```
r$a <- paste(unique(strsplit(r$a,"")[[1]]), collapse="")
r$b <- paste(unique(strsplit(r$b,"")[[1]]), collapse="")
r
```

```
## $a
## [1] "Mam icheblo"
##
## $b
## [1] "Ogi ocnLEG"
```

Data una lista di liste di numeri come input, crea una nuova lista di liste in cui gli elementi di ciascuna lista sono ordinati in ordine crescente.

```
ord <- list(a = c(2,12,35,4,89,7,0),b = c(3,8,2,7,1,6,24,0))
ord
```

```
## $a
## [1]  2 12 35  4 89  7  0
##
## $b
## [1]  3  8  2  7  1  6 24  0
```

```
ord$a <- sort(ord$a)
ord$b <- sort(ord$b)
ord
```

```
## $a
## [1]  0  2  4  7 12 35 89
##
## $b
## [1]  0  1  2  3  6  7  8 24
```

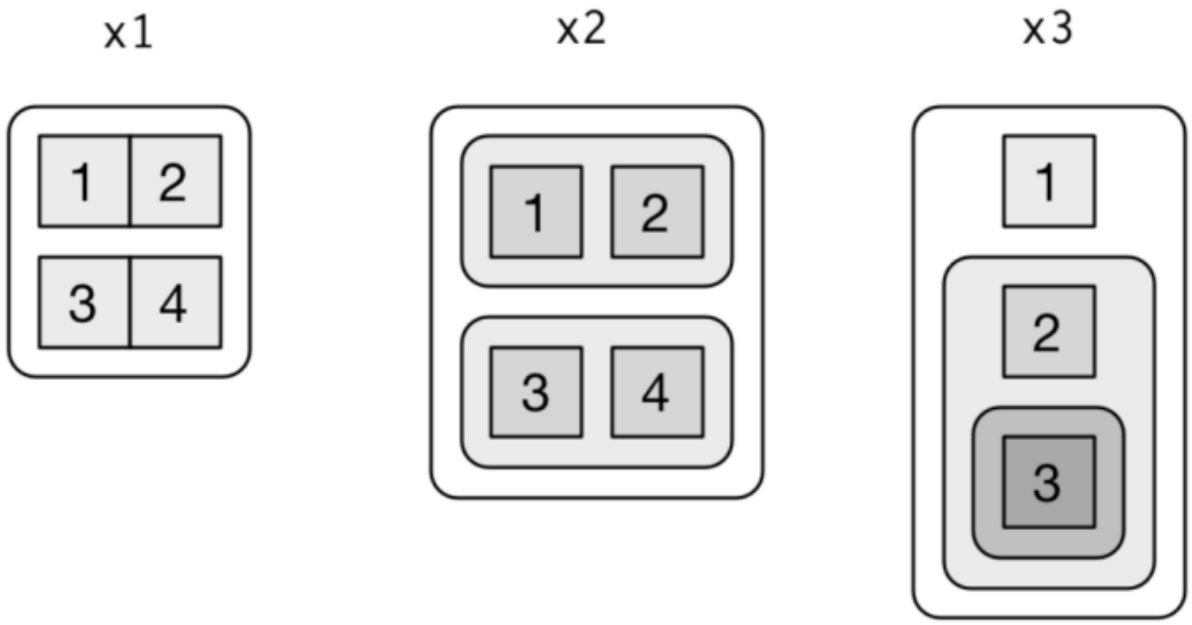
Tecnicamente non ha senso fare la `sort` perché gli elementi sono disgiunti, ovvero non sono elementi di un vettore, ma di una lista.

2.7 Approfondimenti

In questa sezione, si andrà ad esplorare più approfonditamente le liste e i vettori in `R`, con l'obiettivo di ampliare la nostra comprensione su queste strutture dati e sulle loro applicazioni nella programmazione statistica e nell'analisi dei dati.

2.7.1 Approfondimento sulle liste

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```



Creare la lista `x4` contenente un vettore di stringhe, una lista di stringhe e un vettore numerico.

```
x4 <- list(c("hello", "world"), list("ciao", "mondo"), c(3, 6, 9))
x4
```

```
## [[1]]
## [1] "hello" "world"
##
## [[2]]
## [[2]][[1]]
## [1] "ciao"
##
## [[2]][[2]]
## [1] "mondo"
##
##
## [[3]]
## [1] 3 6 9
```

Creare la lista `x5` contenente una lista di vettori.

```
x5 <- list(list(c(2, 4, 6), c(1, 3, 5)))
x5
```

```
## [[1]]
## [[1]][[1]]
## [1] 2 4 6
##
## [[1]][[2]]
## [1] 1 3 5
```

Creare la lista `x6` contenente una lista di tre liste la cui prima contiene a sua volta tre tipi di dati diversi.

```
x6 <- list(list(list(1L, "ciao", 3.14), list(4, 5, 6), list(7, 8, 9)))
x6
```

```
## [[1]]
## [[1]][[1]]
## [[1]][[1]][[1]]
## [1] 1
##
## [[1]][[1]][[2]]
## [1] "ciao"
##
## [[1]][[1]][[3]]
## [1] 3.14
##
##
## [[1]][[2]]
## [[1]][[2]][[1]]
## [1] 4
##
```

```
## [[1]][[2]][[2]]
## [1] 5
##
## [[1]][[2]][[3]]
## [1] 6
##
##
## [[1]][[3]]
## [[1]][[3]][[1]]
## [1] 7
##
## [[1]][[3]][[2]]
## [1] 8
##
## [[1]][[3]][[3]]
## [1] 9
```

Ci sono tre modi per suddividere una lista:

- [ ] estrae una sotto-lista. Il risultato sarà sempre una lista.
- [[ ]] estrae un singolo componente da una lista, quindi rimuove un livello di gerarchia dalla lista.
- $ è un'abbreviazione per estrarre

2.7.2 Approfondimento sui vettori

Esempi di vettori in R:

- creazione di un vettore logico

```
vettore_logico <- c(TRUE, FALSE, T)
vettore_logico
```

```
## [1]  TRUE FALSE  TRUE
```

- creazione di un vettore di interi

```
vettore_intero <- c(1L, 2L, 3L, 4L, 5L)
vettore_intero
```

```
## [1] 1 2 3 4 5
```

- creazione di un vettore di numeri decimali

```
vettore_double <- c(1.5, 2.7, 3.9)
vettore_double
```

```
## [1] 1.5 2.7 3.9
```

- creazione di un vettore di caratteri

```r
vettore_carattere <- c("cane", "gatto", "topo")
```

Esistono due tipi di **coercizione**:

1. **coercizione esplicita**: tramite funzioni come `as.logical()`, `as.integer()`, `as.double()` o `as.character()`. Controllare se è possibile correggere il tipo del vettore iniziale.
2. **coercizione implicita**: avviene quando si utilizza un vettore in un contesto che si aspetta un certo tipo di valori. Un esempio è l'utilizzo di un vettore logico in un contesto numerico: `TRUE` è convertito in 1 e `FALSE` è convertito in 0, la somma di un vettore logico è il numero di `TRUE` e la media di un vettore logico è la proporzione di `TRUE`.

```r
v1 <- c(TRUE, 1L)
typeof(v1)
```

```
## [1] "integer"
```

```r
v1
```

```
## [1] 1 1
```

```r
v2 <- c(1L, 1.5)
typeof(v2)
```

```
## [1] "double"
```

```r
v2
```

```
## [1] 1.0 1.5
```

```r
v3 <- c(1.5, "a")
typeof(v3)
```

```
## [1] "character"
```

```r
v3
```

```
## [1] "1.5" "a"
```

Vettore atomico

Un vettore atomico in `R` non può contenere una combinazione di tipi diversi. Il tipo di dato è una caratteristica dell'intero vettore e non dei singoli elementi al suo interno. Se si desidera mescolare più tipi di dati, è necessario utilizzare una lista anziché un vettore atomico.

Funzioni di verifica dei dati di vettori

Operazioni con i vettori

In `R`, le operazioni matematiche di base lavorano con i vettori. Ciò significa che non si dovrebbe mai aver bisogno di eseguire un'iterazione esplicita quando si eseguono semplici calcoli matematici. Cosa succede se si sommano due vettori di lunghezza diversa?

Tipo di vettore	lgl	int	dbl	chr	list
is.logical()	✓				
is.integer()		✓			
is.double()			✓		
is.numeric()	✓	✓	✓		
is.character()				✓	
is.atomic()	✓	✓	✓	✓	
is.list()					✓
is.vector()	✓	✓	✓	✓	✓

```
1:10 + 1:2 #Residuo viene utilizzato spesso nella programmazione più avanzata
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

R espande il vettore più corto a quello più lungo.

```
1:10 + 1:3
```

```
## Warning in 1:10 + 1:3: la lunghezza più lunga dell'oggetto non è un multiplo
## della lunghezza più corta dell'oggetto
```

```
##  [1]  2  4  6  5  7  9  8 10 12 11
```

Tutti i tipi di vettore possono essere nominati. Si possono nominare durante la "creazione" specificando in `c()...`:

```
c(x = 1, y = 2, z = 4)
```

```
## x y z
## 1 2 4
```

... oppure dopo averlo fatto, con `setNames()`:

```
vet<-setNames(1:3, c("a", "b", "c"))
vet
```

```
## a b c
## 1 2 3
```

```
vet<-setNames(1:5, c("a", "b", "c"))
vet
```

```
##    a    b    c <NA> <NA>
##    1    2    3    4    5
```

```
vet["a"]<-5
vet
```

```
##    a    b    c <NA> <NA>
##    5    2    3    4    5
```

'E importante assegnare sempre un nome al vettore quando si utilizza la funzione `setNames()`, altrimenti verrebbe restituito il seguente "errore":

```
vett <- rep(1, 5)
setNames(vett, c("a", "b", "c"))
```

```
##    a    b    c <NA> <NA>
##    1    1    1    1    1
```

```
setNames(vett, c("a", "b", "c", "d", "e"))
```

```
## a b c d e
## 1 1 1 1 1
```

```
vett["a"]
```

```
## [1] NA
```

```
vett["a"] = 10
vett
```

```
##                 a
##  1  1  1  1  1 10
```

```
vett["a"]
```

```
##  a
## 10
```

I numeri interi devono essere tutti positivi, tutti negativi o zero.

- Il sottoinsieme con i numeri interi positivi mantiene gli elementi in quelle posizioni:

  ```
  x <- c("one", "two", "three", "four", "five")
  x[c(3, 2, 5)]
  ```

  ```
  ## [1] "three" "two"   "five"
  ```

  ```
  x[seq(from=1, to=5 ,by=1)]
  ```

  ```
  ## [1] "one"   "two"   "three" "four"  "five"
  ```

  ```
  x[seq(from=1, to=8 ,by=1)]
  ```

  ```
  ## [1] "one"   "two"   "three" "four"  "five"  NA      NA      NA
  ```

```
x[c(0,2)]
```

```
## [1] "two"
```

- Ripetendo una posizione, si può effettivamente fare un output più lungo dell'input:

```
x[c(1, 1, 5, 5, 5, 2)]
```

```
## [1] "one"  "one"  "five" "five" "five" "two"
```

```
seq <- seq(1,5,2)
x[c(seq,seq)]
```

```
## [1] "one"   "three" "five"  "one"   "three" "five"
```

- I valori negativi fanno rimuovere gli elementi nelle posizioni specificate:

```
x[c(-1, -3, -5)]
```

```
## [1] "two"  "four"
```

```
x[c(1,2,-3)]
```

```
## Error in x[c(1, 2, -3)]: solo gli 0 si possono usare contemporaneamente con subscript negativi
```

```
x[c(0, -3, -5)]
```

```
## [1] "one"  "two"  "four"
```

Il sottoinsieme con un vettore logico mantiene tutti i valori corrispondenti ad un valore `TRUE`. Questo è più spesso utile in combinazione con le funzioni di confronto. Esempi:

```
x <- c(10, 3, NA, 5, 8, 1, NA)
is.na(x)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

```
x[!is.na(x)] #operazione per trattare i missing values
```

```
## [1] 10  3  5  8  1
```

```
x[x %% 2 == 0] #tutti i valori pari sfrutta l'operazione modulo
```

```
## [1] 10 NA  8 NA
```

Se si ha un vettore nominato, si può sotto-selezionare con un vettore di stringhe:

```r
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
```

```
## xyz def
##   5   2
```

```r
x[c(3,2)] #funziona anche se non utilizziamo un vettore di stringhe
```

```
## xyz def
##   5   2
```

3 Strutture di controllo condizionale (if-else)

In R, le strutture di controllo condizionale sono utilizzate per eseguire determinate istruzioni solo se una condizione specifica è verificata.

```r
x <- 10
if(x > 5){
  print("x è maggiore di 5")
} else {
  print("x è minore di 5")
}
```

```
## [1] "x è maggiore di 5"
```

Gli operatori logici sono utilizzati per combinare condizioni logiche e ottenere un risultato logico. I principali operatori logici sono:

- **AND** (&): restituisce `TRUE` solo se entrambe le condizioni sono vere.

```r
x <- 10
if(x > 5 & x < 10){
  print("x è maggiore di 5 e minore di 10")
} else {
  print("condizione non verificata") #se x è diverso da 6,7,8,9
}
```

```
## [1] "condizione non verificata"
```

```r
x <- 7
if(x > 5 & x < 10){
  print("x è maggiore di 5 e minore di 10")
} else {
  print("condizione non verificata")
}
```

```
## [1] "x è maggiore di 5 e minore di 10"
```

- Lo stesso ragionamento può essere esteso in due variabili:

```r
x <- 12
y <- 67
if (x > 10 & y < 20){
  print("qui stampa quando x è maggiore di 10 e y è minore di 20")
} else {
  cat("qui stampa quando x è minore o uguale a 10 oppure quando y è maggiore
      o uguale a 20")
}
```

```
## qui stampa quando x è minore o uguale a 10 oppure quando y è maggiore
##       o uguale a 20
```

```r
x <- 12
y <- 18
if (x > 10 & y < 20){
  print("qui stampa quando x è maggiore di 10 e y è minore di 20")
} else {
  cat("qui stampa quando x è minore o uguale a 10 oppure quando y è maggiore
      o uguale a 20")
}
```

```
## [1] "qui stampa quando x è maggiore di 10 e y è minore di 20"
```

- **OR** (|): restituisce `TRUE` se almeno una delle condizioni è vera.

```r
x <- 10
if(x > 5 | x < 10){
  print("x è maggiore di 5 o minore di 10")
} else {
  print("condizione non verificata")
}
```

```
## [1] "x è maggiore di 5 o minore di 10"
```

```r
x <- 8
if(x < 5 | x > 10){
  print("x è minore di 5 o maggiore di 10")
} else {
  print("condizione non verificata")
}
```

```
## [1] "condizione non verificata"
```

- Esempio estendibile al caso in due variabili:

```r
x <- 25
y <- 67
if (x>10 | y<20){
  print("qui stampa quando x è maggiore di 10 o y è minore di 20")
} else {
  print("qui stampa quando nessuna delle due è verificata")
}
```

```
## [1] "qui stampa quando x è maggiore di 10 o y è minore di 20"
```

```
x <- 8
y <- 37
if (x>10 | y<20){
  print("qui stampa quando x è maggiore di 10 o y è minore di 20")
} else {
  print("qui stampa quando nessuna delle due è verificata")
}
```

```
## [1] "qui stampa quando nessuna delle due è verificata"
```

- **Not** (!): inverte il valore di verità di una condizione.

```
a <- -5

if (!a > 0){
  print("a non è positivo")
}
```

```
## [1] "a non è positivo"
```

Esercitazioni pratiche con istruzioni condizionali (else-if)

```
voto <- 75

if(voto >= 90){
  print("A")
} else if(voto >=80){
  print("B")
} else if(voto >= 70){
  print("C")
} else if(voto >= 60){
  print("D")
} else {
  print("F")
}
```

```
## [1] "C"
```

Esercizi finali

- Scrivi un programma in `R` che controlli se un numero è positivo, negativo o zero.

```
x <- runif(1, -10, 10)
x
```

```
## [1] 8.004494
```

```r
if(x > 0){
  print("Positivo")
} else if(x == 0){
  print("Zero")
} else {
  print("Negativo")
}
```

```
## [1] "Positivo"
```

- Scrivi un programma in R che trovi il massimo tra tre numeri dati.

```r
#Prima opzione
x <- c(1, 2, 4)
if(x[1] >= x[2] & x[1] >= x[3]){
  print(x[1])
} else if(x[2] >= x[1] & x[2] >= x[3]){
  print(x[2])
} else if (x[3] >= x[2] & x[3] >= x[1]) {
  print(x[3])
}
```

```
## [1] 4
```

```r
#Seconda opzione (professore)
x <- 20
y <- 5
z <- 13
if (x > y){
  if(x > z){
    print("il massimo è x")
  } else {
    print("il massimo è z")
  }
} else {
  if(y > z){
    print("il massimo è y")
  } else {
    print("il massimo è z")
  }
}
```

```
## [1] "il massimo è x"
```

- Scrivi un programma in R che verifichi se un anno è bisestile o meno.

```r
anno <- 2024

if((anno %% 4 == 0 & anno %% 100 != 0) | (anno %% 400 == 0)){
  print(paste("l'anno", anno, "è bisestile"))
} else {
  print(paste("l'anno", anno, "non è bisestile"))
}
```

```
## [1] "l'anno 2024 è bisestile"
```

- Scrivi un programma in `R` che determini se un numero intero è pari o dispari.

```
x <- 6

if(x %% 2 == 0){
  print("Pari")
} else {
  print("Dispari")
}
```

```
## [1] "Pari"
```

- Scrivi un programma in R che ordini tre numeri dati in ordine crescente utilizzando solo le istruzioni condizionali `if-else`.

```
x <- runif(3, -10, 10)
x
```

```
## [1] -9.990004 -7.123479 -7.197063
```

```
if(x[1] <= x[2] & x[1] <= x[3]){
  if(x[2] <= x[3]){
    print(c(x[1], x[2], x[3]))
  } else {
    print(c(x[1], x[3], x[2]))
  }
} else if(x[2] <= x[1] & x[2] <= x[3]){
  if(x[1] <= x[3]){
    print(c(x[2], x[1], x[3]))
  } else {
    print(c(x[2], x[3], x[1]))
  }
} else {
  if(x[1] <= x[2]){
    print(c(x[3], x[1], x[2]))
  } else {
    print(c(x[3], x[2], x[1]))
  }
}
```

```
## [1] -9.990004 -7.197063 -7.123479
```

4 Cicli

Il ciclo `while` viene utilizzato per eseguire un blocco di codice ripetutamente finché una condizione specificata è vera.

```r
x <- 1

while(x <= 5){
  print(paste("Iterazione:", x))
  x <- x + 1 #modifica della variabile
}
```

```
## [1] "Iterazione: 1"
## [1] "Iterazione: 2"
## [1] "Iterazione: 3"
## [1] "Iterazione: 4"
## [1] "Iterazione: 5"
```

Per ogni ciclo `while` esiste il rispettivo ciclo `for`. Il ciclo `for` viene utilizzato per eseguire un blocco di codice un numero specificato di volte. 'E molto utile quando si è in presenza di una collezione di elementi non soltanto numerici ma anche in presenza di stringhe.

```r
for(i in 1:5){
    print(paste("Iterazione: ", i))
}
```

```
## [1] "Iterazione:  1"
## [1] "Iterazione:  2"
## [1] "Iterazione:  3"
## [1] "Iterazione:  4"
## [1] "Iterazione:  5"
```

I cicli sono spesso utilizzati per eseguire operazioni sui vettori o sulle liste.

```r
vettore <- c("a", "b", "c", "d", "e")
for(elem in vettore){
  print(elem)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

```r
lista <- list("a", "b", "c", "d", "e")
for(elem in lista){
  print(elem)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

Alcuni esempi pratici sull'utilizzo dei cicli in R.

```
#Somma dei primi 10 numeri interi positivi
somma <- 0
for(i in 1:10){
  somma <- somma + i
}
print(somma)
```

```
## [1] 55
```

```
#Stampa solo i numeri pari in una sequenza da 1 a 20
for(i in 1:20){
  if(i %% 2 == 0){
    print(i)
  }
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

4.1 Esercizi finali

- Scrivi un programma in R che utilizzi un ciclo `while` per stampare i numeri da 1 a 10.

```
x <- 1
while(x <= 10){
  print(x)
  x <- x + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

- Scrivi un programma in R che utilizzi un ciclo `for` per calcolare la somma dei quadrati dei numeri da 1 a 5.

```r
somma <- 0

for(i in 1:5){
  somma <- somma + i^2
}
somma
```

```
## [1] 55
```

- Scrivi un programma in `R` che utilizzi un ciclo `while` per stampare i primi 5 numeri della sequenza di Fibonacci.

```r
#Prima opzione
v <- c(0, 1)
i <- 2
while(i <= 5){
  somma <- v[i] + v[i-1]
  v[i+1] <- somma
  i <- i + 1
}
v
```

```
## [1] 0 1 1 2 3 5
```

```r
#Seconda opzione
a <- 1
b <- 1
count <- 0
fib_seq <- c() #vettore vuoto
while (count < 6) {
  fib_seq <- c(fib_seq, a)
  next_num <- a + b
  a <- b
  b <- next_num
  count <- count + 1
}
print(fib_seq)
```

```
## [1] 1 1 2 3 5 8
```

- Scrivi un programma in `R` che utilizzi un ciclo `for` per trovare il massimo di un vettore di numeri.

```r
vettore <- c(3, 7, 1, 9, 4, 6)
massimo <- vettore[1]  # Assume il primo elemento come massimo iniziale
for (num in vettore){
  if (num > massimo){
    massimo <- num
  }
}
print(massimo)
```

```
## [1] 9
```

- Scrivi un programma con un ciclo `for` che riordina gli elementi del vettore in ordine crescente.

```
x <- c(2,3,5,1,4)
for(i in 1:(length(x)-1)){    #Bubble sort
  for(j in (i+1):length(x)){
    if(x[i] > x[j]){
      x[c(i,j)] = x[c(j,i)]
print(x) #stampa solo per far vedere come funziona
    } else {
      x[c(i,j)] = x[c(i,j)]
    }
  }
}
```

```
## [1] 1 3 5 2 4
## [1] 1 2 5 3 4
## [1] 1 2 3 5 4
## [1] 1 2 3 4 5
```

- Scrivi un programma con un ciclo `for` o `while` in `R` che date due stringhe, restituisca la concatenazione delle stesse.

```
stringa1 <- "Ciao"
stringa2 <- "Mondo!"
risultato <- ""
lunghezza1 <- nchar(stringa1)
lunghezza2 <- nchar(stringa2)

i <- 1
while (i <= lunghezza1) {
  risultato <- paste(risultato, substr(stringa1, i, i), sep = "")
  i <- i + 1
}

i <- 1
while (i <= lunghezza2) {
  risultato <- paste(risultato, substr(stringa2, i, i), sep = "")
  i <- i + 1
}
print(risultato)
```

```
## [1] "CiaoMondo!"
```

- Scrivi un programma in `R` che data una stringa, restituisca la sua lunghezza, contando all'interno di un loop `for` o `while`.

```
#Prima opzione
stringa <- "Questa è una stringa per Questa prova"
lunghezza <- 0
```

```
for(i in 1:nchar(stringa)){
  lunghezza = lunghezza + 1
}
lunghezza
```

```
## [1] 37
```

```
#Seconda opzione (senza utilizzare nchar())
stringa <- "Ciao mondo!"
lunghezza <- 0
for (carattere in strsplit(stringa, "")[[1]]) {
  lunghezza <- lunghezza + 1
}
print(lunghezza)
```

```
## [1] 11
```

```
print(lunghezza == nchar(stringa))
```

```
## [1] TRUE
```

- Scrivi un programma in `R` che data una stringa, restituisca la stessa stringa al contrario, usando `for/while`.

```
reverse <- ""
for(i in nchar(stringa):1){
  reverse <- paste(reverse, substr(stringa, i, i), sep = "")
}

reverse
```

```
## [1] "!odnom oaiC"
```

- Scrivi un programma in `R` che data una frase, la frammenti in diverse parole, inserendole in un vettore. Dopodichè, cicli sul vettore e conti quante volte compare una certa parola all'interno della frase.

```
#Conta quante volte si ripete ogni parola nella frase
stringa <- "che bello bello è il mondo nuovo"
v <- strsplit(stringa, " ")[[1]]
v
```

```
## [1] "che"   "bello" "bello" "è"     "il"    "mondo" "nuovo"
```

```
cont <- 0
for (parola in unique(v)){
  cont <- sum(v == parola)
  cat("La parola", parola, "compare", cont, "volte nella frase.\n")
}
```

```
## La parola che compare 1 volte nella frase.
## La parola bello compare 2 volte nella frase.
## La parola è compare 1 volte nella frase.
## La parola il compare 1 volte nella frase.
## La parola mondo compare 1 volte nella frase.
## La parola nuovo compare 1 volte nella frase.
```

```
#Conta quante volte si ripete una determinata parola nella frase
stringa <- "La mela rossa Ã¨ dentro una scatola rossa"
split <- strsplit(stringa, " ")
parole <- split[[1]]
parola_da_considerare = "rossa"
contatore <- 0
for (p in parole){
  if (parola_da_considerare == p)
    contatore <- contatore + 1
}

print(contatore)
```

```
## [1] 2
```

5 Funzioni

Le funzioni in `R` sono blocchi di codice che eseguono un'azione specifica quando vengono chiamate. Possono essere definite dall'utente o essere predefinite. Le funzioni vengono definite utilizzando la sintassi `function()`. Gli argomenti vengono specificati all'interno delle parentesi tonde. Il corpo della funzione è racchiuso tra parentesi graffe. Il valore di ritorno è specificato con `return()`.

```
mia_funzione <- function(x, y){
  risultato <- x + y
  return(risultato)
}
```

Si possono chiamare le funzioni definite passando loro gli argomenti necessari. Il valore restituito può essere assegnato ad una variabile per un utilizzo successivo.

5.1 Esercizi

- Scrivi una funzione che calcoli il quadrato di un numero.

```
quadrato <- function(numero){
  risultato <- x^2
  return(risultato)
}

quadrato(3)
```

```
## [1]  1  4  9 16 25
```

- Definisci una funzione che accetti un lista di numeri e restituisca la loro somma.

```
x <- list(a = c("gatto"), b = c(3, 5, 3, 7), c = c(4, 1, 8))
y <- list(a = c(3, 5, 3, 7), b = c(4, 1, 8))
somma <- function(lista){
  if(is.list(lista) & !any(sapply(lista, is.character))){
    return(sum(unlist(lista)))
  } else {
    return("Errore")
  }
}

somma(x)
```

```
## [1] "Errore"
```

```
somma(y)
```

```
## [1] 31
```

- Crea una funzione che prenda in input una stringa e restituisca la sua lunghezza, senza considerare gli spazi vuoti.

```
lunghezza <- function(x){
  if(is.character(x)){
    return(nchar(gsub(" ", "", x)))
  }
}

stringa <- "Marianna non ha la cazzimma"
lunghezza(stringa)
```

```
## [1] 23
```

- Scrivi una funzione che calcoli la deviazione standard di un vettore di numeri.

```
dev.st <- function(x){
  return(sd(x))
}

v <- runif(100)
dev.st(v)
```

```
## [1] 0.2890146
```

- Creare una funzione che restituisca il massimo tra due numeri.

```
massimo <- function(x){
  return(x[which.max(x)])
}

massimo(v)
```

```
## [1] 0.9886502
```

- Definisci una funzione che prenda in input una lista di parole e restituisca la parola più lunga.

```
#Prima opzione (struttura ad insieme)
lunga <- function(x){
  if(is.list(x) == T){
    parola_più_lunga <- ""
    max_l <- 0
    for(elem in x){
      if(nchar(elem) > max_l){
        max_l <- nchar(elem)
        parola_più_lunga <- elem
      }
    }
  return(parola_più_lunga)
  }
}

l_parole <- list("Non", "mi", "va", "di", "fare", "niente")
lunga(l_parole)
```

```
## [1] "niente"
```

```
#Seconda opzione (usando gli indici)
lunga <- function(x){
  if(is.list(x) == T){
    parola_più_lunga <- ""
    max_l <- 0
    for(i in length(x)){
      if(nchar(x[[i]]) > max_l){
        max_l <- nchar(x[[i]])
        parola_più_lunga <- x[[i]]
      }
    }
    return(parola_più_lunga)
  }
}

l_parole <- list("Non", "mi", "va", "di", "fare", "niente")
lunga(l_parole)
```

```
## [1] "niente"
```

5.2 Concetti avanzati sulle funzioni

- Funzioni **ricorsive**: le funzioni che si richiamano all'interno del proprio corpo. 'E importante includere un caso base che interrompa la ricorsione. Un esempio è il calcolo del fattoriale:

```r
fattoriale <- function(n){
  if(n <= 1){
    return(1)
  } else {
    return(n * fattoriale(n - 1))
  }
}
fattoriale(5)
```

```
## [1] 120
```

- Funzioni **anonime**: funzioni senza un nome definito, create utilizzando la sintassi `function()` direttamente all'interno di un'espressione. Sono utili per operazioni brevi o quando non è necessario definire una funzione separata.

```r
doppio <- function(x){
  return(x*2)
}

quadrato <- function(x) x^2

doppio(5)
```

```
## [1] 10
```

```r
quadrato(3)
```

```
## [1] 9
```

- Funzioni **ad alto ordine**: funzioni che possono prendere in input altre funzioni come argomenti e/o restituirle come output;

```r
applica_funzione <- function(funzione, dati){
  risultato <- funzione(dati)
  return(risultato)
}

applica_funzione(sum, c(1:5))
```

```
## [1] 15
```

- **Ambiente delle funzioni**: le funzioni in `R` operano all'interno di un ambiente. Le variabili locali vengono definite e valutate all'interno di questo ambiente. Le variabili definite all'interno di una funzione non sono accessibili al di fuori di essa. Questo contribuisce alla modularità e all'incapsulamento del codice.
- **Passaggio per riferimento vs. passaggio per valore**: riferimento agli oggetti vs. copia degli oggetti stessi. Nel passaggio per riferimento, si passa un riferimento all'oggetto in memoria e le modifiche all'oggetto influenzano tutte le referenze ad esso. Nel passaggio per valore, invece, viene passata una copia dell'oggetto, per cui le modifiche all'oggetto interno della funzione non influenzano l'oggetto originale. In `R` non esiste il passaggio per riferimento, ma esistono delle soluzioni "di ripiego":

```
x <- 100
miaf <- function(x){
  x <<- 0
  return(x)
}
miaf(x)
```

```
## [1] 0
```

```
x
```

```
## [1] 0
```

Facendo l'assegnazione alla variabile `x` con la simbologia `<<-` si esce dall'ambiente locale della funzione e si va a cercare una variabile globale nominata `x` nell'ambiente esterno (globale) della funzione.

Si ripetono alcuni esercizi precedenti, ma utilizzando tecnica ricorsiva, funzioni anonime o funzioni ad alto ordine.

- Creare una funzione *ad alto ordine* che restituisca il massimo tra due numeri.

```
massimo <- function(funzione, x){
  return(x[funzione(x)])
}

v <- runif(100)
massimo(which.max, v)
```

```
## [1] 0.9996757
```

6 Matrici e array

In `R` i vettori, le matrici e gli array sono strutture dati fondamentali per gestire dati multi-dimensionali.

- **Vettore**: una collezione di elementi dello stesso tipo.

```
vettore <- c(1:5)
vettore
```

```
## [1] 1 2 3 4 5
```

- **Matrice**: una struttura bidimensionale di elementi dello stesso tipo.

```
matrice <- matrix(1:9, nrow = 3, ncol = 3)
matrice
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

- **Array**: una struttura multidimensionale di elementi dello stesso tipo (*differenza rispetto ai data frame*).

```
array <- array(1:8, dim = c(2, 2, 2))
array
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

6.1 Matrici

Una matrice è un array bidimensionale di elementi omogenei (dello stesso tipo). Differisce dai vettori in quanto questi ultimi sono limitati ad una sola dimensione, mentre le matrici possiedono n righe e m colonne. Si può facilmente creare una matrice a partire da un vettore `v` con la funzione `matrix(v, nrow, ncol, byrow = F)`:

- `v`: vettore dei dati,
- `nrow, ncol`: numero desiderato di righe e colonne,
- `byrow`: opzione per riempire la matrice riga per riga (`TRUE`) oppure colonna per colonna (`FALSE`, di default).

Il numero di elementi nel vettore `v` deve essere un multiplo o sotto-multiplo del numero di righe e colonne. In caso contrario, `R` sfrutterà il meccanismo del riciclo visto in precedenza con i vettori. Si può omettere la dicitura `nrow =` e `ncol =` sfruttando semplicemente l'ordine dei parametri, facendo attenzione a non invertirli. La funzione `matrix()` si aspetta di trovare prima il numero di righe (subito dopo il vettore dei dati) e poi il numero di colonne.

```
matrix(1:6, 2, 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
matrix(1:6, 2, 3, byrow = T)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```r
matrix(1:10, nrow = 2)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

```r
matrix(1:10, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```r
matrix(1:10, 2, 3)
```

```
## Warning in matrix(1:10, 2, 3): la lunghezza [10] dei dati non è un sotto
## multiplo o un multiplo del numero di colonne [3]
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
#Meccanismo di riciclo
matrix(1:6, 2, 6)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    1    3    5
## [2,]    2    4    6    2    4    6
```

Quale istruzione usare per creare questa matrice?

```
     [,1] [,2] [,3]
[1,]    1    8   15
[2,]    2    9   16
[3,]    3   10   17
[4,]    4   11   18
[5,]    5   12   19
[6,]    6   13   20
[7,]    7   14   21
```

```r
#Risposta
matrix(1:21, 7, 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    8   15
## [2,]    2    9   16
## [3,]    3   10   17
## [4,]    4   11   18
## [5,]    5   12   19
## [6,]    6   13   20
## [7,]    7   14   21
```

6.1.1 Richiamo degli elementi di una matrice

Per richiamare determinati valori all'interno della matrice si utilizza l'operatore `[]`:

```
m <- matrix(1:21, 7, 3)
m[1, 1]
```

```
## [1] 1
```

```
m[1, ]
```

```
## [1]  1  8 15
```

```
m[, 1]
```

```
## [1] 1 2 3 4 5 6 7
```

```
m[1:3, 1]
```

```
## [1] 1 2 3
```

```
m[1:2, 1:3]
```

```
##      [,1] [,2] [,3]
## [1,]    1    8   15
## [2,]    2    9   16
```

```
m[, 3:1]
```

```
##      [,1] [,2] [,3]
## [1,]   15    8    1
## [2,]   16    9    2
## [3,]   17   10    3
## [4,]   18   11    4
## [5,]   19   12    5
## [6,]   20   13    6
## [7,]   21   14    7
```

6.1.2 Etichette della matrice

Con l'argomento opzionale `dimnames` è possibile fornire una lista di etichette personalizzate per i nomi delle righe e delle colonne. In alternativa, si può creare una matrice senza etichette e aggiungerle in un secondo momento assegnandole direttamente con `rownames()` e `colnames()`. Queste funzioni ritornano infatti l'insieme di etichette della matrice passata in input; sovrascrivendo il vettore ritornato si cambiano i nomi delle righe/colonne.

```
righe <- c("A", "B")
colonne <- c("V", "W", "X", "Y", "Z")
etichette <- list(righe, colonne)
matrix(1:10, 2, 5, byrow = T, dimnames = etichette)
```

```
##   V W X Y  Z
## A 1 2 3 4  5
## B 6 7 8 9 10
```

La funzione `length()` ritorna il numero di celle della matrice passata in input, che equivale al numero di righe moltiplicato per il numero di colonne. Le funzioni `nrow()` e `ncol()` restituiscono rispettivamente il numero di righe e il numero di colonne di una matrice. Con la funzione `dim()` è possibile ricavare direttamente le dimensioni della matrice, ottenendo un vettore del tipo `(numero di righe, numero di colonne)`.

```
M <- matrix(1:4, 2, 2)
N <- matrix(c(1, 0, 0, 1), 2, 2)
M * N
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    4
```

```
M * N == N
```

```
##      [,1]  [,2]
## [1,] TRUE  TRUE
## [2,] TRUE FALSE
```

6.1.3 Operazioni tra matrici

In `R` è possibile effettuare diverse operazioni sulle matrici, tra cui:

- somma e differenza,
- prodotto matriciale, con l'operatore `%*%`,
- calcolo dell'inversa, tramite la funzione `solve()`,
- calcolo della trasposta, tramite la funzione `t()`,
- restituzione della diagonale, tramite della funzione `diag()`.

```
M <- matrix(1:4, 2, 2)
N <- matrix(c(1, 0, 0, 1), 2, 2)
M * N
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    4
```

```
M * N == M
```

```
##       [,1]  [,2]
## [1,]  TRUE FALSE
## [2,] FALSE  TRUE
```

6.2 Array

Gli array sono una generalizzazione dei vettori e delle matrici, ma possono gestire più di 2 dimensioni. Analogamente a quanto visto per le matrici, si può creare un array con la funzione `array(v, dim)`, dove `v` è il vettore dei dati e `dim` è il vettore contenente le dimensioni dell'array. L'argomento opzionale `dimnames`, come per le matrici, permette di specificare il nome delle diverse dimensioni.

```
vector1 <- c(5, 9, 3)
vector2 <- c(10:15)
dim1 <- c("A1", "A2", "A3")
dim2 <- c("B1", "B2", "B3")
dim3 <- c("C1", "C2")
array(c(vector1, vector2), dim = c(3, 3, 2), dimnames = list(dim1, dim2, dim3))
```

```
## , , C1
##
##    B1 B2 B3
## A1  5 10 13
## A2  9 11 14
## A3  3 12 15
##
## , , C2
##
##    B1 B2 B3
## A1  5 10 13
## A2  9 11 14
## A3  3 12 15
```

6.2.1 Esercizi

- Scrivi un codice `R` per creare un vettore con i numeri da 1 a 10.

```
c(1:10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

- Crea una matrice `2 x 2` con i numeri da 1 a 4 e calcola la somma delle righe.

```
m <- matrix(1:4, 2, 2)
m
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
rowSums(matrix(1:4, 2, 2))
```

```
## [1] 4 6
```

- Genera un array con dimensioni `2 x 2 x 3` e calcola la media tra tutti gli elementi.

```r
array1 <- array(1:12, dim = c(2, 2, 3))
array1
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 3
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
```

```r
mean(array1)
```

```
## [1] 6.5
```

6.2.2 Esercizi finali

- Crea una matrice `3 x 3` con valori casuali compresi tra 1 e 10.

```r
vettore <- sample(1:10, 9)
matrice <- matrix(vettore, 3, 3)
matrice
```

```
##      [,1] [,2] [,3]
## [1,]    8    9   10
## [2,]    3    7    5
## [3,]    1    6    4
```

- Calcola la somma degli elementi di una matrice `4 x 4` data.

```r
vettore <- sample(1:40, 16)
matrice <- matrix(vettore, 4, 4)
matrice
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   25   12    6
## [2,]    9   30   35   37
## [3,]   22    4    5   16
## [4,]   29   24   27    7
```

```r
sum(matrice)
```

```
## [1] 298
```

- Trova il massimo valore di ogni colonna di una matrice `5 x 3`.

```r
vettore <- sample(1:50, 15)
matrice <- matrix(vettore, 5, 3)
matrice
```

```
##      [,1] [,2] [,3]
## [1,]   44   21   10
## [2,]    3   33   29
## [3,]   39   45   49
## [4,]    7   43   42
## [5,]   23    6    9
```

```r
apply(matrice, 2, max)
```

```
## [1] 44 45 49
```

- Crea un array tridimensionale `2 x 2 x 3` con numeri interi casuali compresi tra 1 e 20.

```r
vettore <- sample(1:20, 12)
array2 <- array(vettore, dim = c(2, 2, 3))
array2
```

```
## , , 1
##
##      [,1] [,2]
## [1,]   10    9
## [2,]    7   17
##
## , , 2
##
##      [,1] [,2]
## [1,]    2   20
## [2,]   12   11
##
## , , 3
##
##      [,1] [,2]
## [1,]   18   15
## [2,]    3    6
```

- Calcola la media degli elementi di ogni strato dell'array creato nell'esercizio 4.

```r
apply(array2, 3, mean)
```

```
## [1] 10.75 11.25 10.50
```

- Trova il valore minimo di ogni riga dell'array creato nell'esercizio 4.

```
for(i in 1:dim(array2)[3]) {
  print(apply(array2[, , i],1, min))
}
```

```
## [1] 9 7
## [1]  2 11
## [1] 15  3
```

- Crea una matrice `2 x 2` con i valori da 1 a 4 e calcola la sua trasposta.

```
matrice <- matrix(1:4, 2, 2, byrow = T)
matrice
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```
t(matrice)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

- Crea una matrice `3 x 3` con i valori da 1 a 9 e calcola la sua inversa.

```
matrice <- matrix(1:9, 3, 3)
matrice
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
solve(matrice)
```

```
## Error in solve.default(matrice): Routine Lapack dgesv: il sistema è esattamente singolare: U[3,3] = 0
```

Viene restituito un errore poiché la matrice è singolare, quindi non è possibile calcolarne l'inversa.

- Crea un array tridimensionale `3 x 2 x 2` con i valori da 1 a 12 e calcola la somma degli elementi lungo la terza dimensione.

```
array3 <- array(1:12, dim = c(3, 2, 2))
array3
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
```

```
apply(array3, 3, sum)
```

```
## [1] 21 57
```

7 Dataframes

I dataframes sono concettualmente simili alle matrici, in cui però le colonne possono contenere tipi di dati diversi. In questo senso sono equiparabili alle tabelle di un foglio di calcolo. Si può creare un dataframe a partire dai vettori colonna con la funzione `data.frame()`, di default `R` utilizzerà come nome delle colonne i nomi dei vettori passati in input. Analogamente alle altre strutture dati è possibile accedere alle informazioni tramite indici (utili per selezionare le righe), oppure tramite il nome della colonna; anche per i dataframes è disponibile la sintassi ridotta che fa uso del `$`. Le funzioni `names()`, `dim()`, `ncol()` e `nrow()` sono definite anche per i dataframes. Combinando le funzioni built-in e la selezione condizionale si possono effettuare alcune analisi basilari, ad esempio l'età media di chi possiede un reddito alto.

```
nomi_italiani <- c("Matteo", "Andrea", "Antonio", "Marianna", "Gabriele",
                   "Adele", "Letizia", "Giovanni", "Luca", "Rossella", "Aldo", "Silvia")
nomi <- sample(nomi_italiani, 100, replace = T)
eta <- sample(18:80, 100, replace = T)
redditi <- sample(c("basso", "medio", "alto"), 100, replace = T)
df <- data.frame(nome = nomi,
                 eta = eta,
                 reddito = redditi)
head(df)
```

```
##       nome eta reddito
## 1    Adele  40   medio
## 2   Andrea  48    alto
## 3   Silvia  22   medio
## 4 Rossella  67   medio
## 5 Gabriele  60    alto
## 6   Andrea  79    alto
```

```
names(df)
```

```
## [1] "nome"    "eta"     "reddito"
```

```r
dim(df)
```

```
## [1] 100   3
```

```r
nrow(df)
```

```
## [1] 100
```

```r
ncol(df)
```

```
## [1] 3
```

```r
df$eta
```

```
##   [1] 40 48 22 67 60 79 58 63 59 19 67 73 39 21 49 63 79 76 50 28 52 46 53 51 75
##  [26] 52 21 33 23 39 28 62 70 28 46 59 66 47 62 30 20 52 51 25 71 63 75 48 33 51
##  [51] 75 39 50 37 46 63 61 77 75 36 77 57 80 46 66 38 71 25 27 66 22 68 50 38 61
##  [76] 53 78 56 30 56 19 72 79 53 67 28 24 57 18 29 76 35 69 38 78 58 61 69 18 74
```

```r
df[3, 2]
```

```
## [1] 22
```

```r
df$eta[4]
```

```
## [1] 67
```

```r
df[2, "eta"]
```

```
## [1] 48
```

```r
df[df$reddito=="alto", ]
```

```
##        nome eta reddito
## 2    Andrea  48    alto
## 5  Gabriele  60    alto
## 6    Andrea  79    alto
## 7    Silvia  58    alto
## 10   Andrea  19    alto
## 12 Marianna  73    alto
## 18   Matteo  76    alto
## 26   Matteo  52    alto
## 36 Gabriele  59    alto
## 38    Adele  47    alto
## 45  Letizia  71    alto
## 52   Silvia  39    alto
## 56     Luca  63    alto
## 63  Antonio  80    alto
```

```
## 67    Adele  71    alto
## 70 Gabriele  66    alto
## 76     Aldo  53    alto
## 77 Marianna  78    alto
## 79     Luca  30    alto
## 83 Gabriele  79    alto
## 85   Matteo  67    alto
## 89  Letizia  18    alto
## 91    Adele  76    alto
## 95 Marianna  78    alto
## 97  Letizia  61    alto
## 98     Luca  69    alto
## 99 Rossella  18    alto
```

```
etamediacond <- function(df, reddito){
  x <- c()
  for (i in 1:length(unique(reddito))) {
    x <- c(x,(mean(df[df$reddito == reddito[i], 2])))
  }
  return(x)
}

etamediacond(df, df$reddito)
```

```
## [1] 47.02564 58.81481 47.02564
```

`R` fornisce già alcuni pacchetti contenenti dataset pronti all'uso: è il caso del package `data`. Per una breve descrizione sui diversi dataset disponibili nel pacchetto bisogna lanciare `data()`. Per importare uno specifico dataset si usa il comando `data("nome dataset")`. Si utilizza `head(nome dataset)` per avere un'anteprima del dataset stesso senza visualizzarlo interamente sullo schermo. Di default saranno visualizzate le prime 6 righe: per cambiare questo valore impostare l'opzione `n` della funzione.

7.1 Esercizi

Importare il dataset `airquality` e visualizzare i primi 8 elementi.

```
data("airquality")
head(airquality, 8)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

Filtrare i dati creando un dataframe solo per il mese di luglio.

```r
luglio <- airquality[airquality$Month == 7, ]
```

Calcolare e stampare la concentrazione media di ozono per il mese di luglio, ricordandosi di filtrare gli eventuali valori mancanti (`NA`).

```r
mean(luglio$Ozone, na.rm = T)
```

```
## [1] 59.11538
```

8 Factor e tabelle di frequenza

8.1 Factor

La funzione `factor()` in `R` è utilizzata per convertire un vettore di dati in un fattore. Un fattore è un tipo di dato in `R` utilizzato per rappresentare variabili categoriali con un numero finito di livelli o labels. La sintassi è la seguente:

```r
factor(x, levels, labels)
```

Esempio:

```r
vettore <- c("A", "B", "A", "C", "B")
fattore <- factor(vettore)
fattore
```

```
## [1] A B A C B
## Levels: A B C
```

Si supponga di avere un vettore di dati che contenga i risultati di un esame rappresentati da valori numerici da 1 a 5 e si voglia convertire questi valori in un fattore con livelli significativi.

```r
vettore <- c(3, 2, 5, 1, 4, 3, 1, 2, 5)
livelli <- c(1, 2, 3, 4, 5)
etichette <- c("Insufficiente", "Sufficiente", "Buono", "Ottimo", "Eccellente")
fattore <- factor(vettore, levels = livelli, labels = etichette)
fattore
```

```
## [1] Buono         Sufficiente   Eccellente    Insufficiente Ottimo
## [6] Buono         Insufficiente Sufficiente   Eccellente
## Levels: Insufficiente Sufficiente Buono Ottimo Eccellente
```

8.2 Tabelle di frequenza

La funzione `table()` in `R` è utilizzata per creare tabelle di frequenza che mostrano la distribuzione delle frequenze di variabili categoriche. Esempio:

```r
vettore <- c("A", "B", "A", "C", "B", "A", "A", "B", "C")
table(vettore)
```

```
## vettore
## A B C
## 4 3 2
```

```r
vettore
```

```
## [1] "A" "B" "A" "C" "B" "A" "A" "B" "C"
```

8.3 Esercizi

Utilizzando la funzione `table()`, crea una tabella di frequenza per un vettore che rappresenta il risultato di un lancio di un dado.

```r
results <- sample(1:6, 10, replace = T)
table(results)
```

```
## results
## 1 2 4 5
## 2 4 3 1
```

```r
results
```

```
##  [1] 4 2 4 1 2 2 4 1 2 5
```

Analisi di frequenza su un dataset simulato che contiene informazioni su sesso, età e livello di istruzione di un gruppo di individui.

```r
bwages <- read.csv2("bwages.csv", header = T, sep = ",")
str(bwages)
```

```
## 'data.frame':    1472 obs. of  4 variables:
##  $ wage : chr  "7.7802076852" "4.81850475584" "10.563645423" "7.0424294557" ...
##  $ educ : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ exper: int  23 15 31 32 9 15 26 23 13 22 ...
##  $ sex  : int  1 1 0 0 0 0 0 0 1 0 ...
```

```r
bwages$sex <- as.factor(bwages$sex)
bwages$educ <- as.factor(bwages$educ)
str(bwages)
```

```
## 'data.frame':    1472 obs. of  4 variables:
##  $ wage : chr  "7.7802076852" "4.81850475584" "10.563645423" "7.0424294557" ...
##  $ educ : Factor w/ 5 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ exper: int  23 15 31 32 9 15 26 23 13 22 ...
##  $ sex  : Factor w/ 2 levels "0","1": 2 2 1 1 1 1 1 1 2 1 ...
```

```
table(bwages$sex)
```

```
##
##   0   1
## 697 775
```

```
table(bwages$educ)
```

```
##
##   1   2   3   4   5
##  99 265 420 356 332
```

```
table(bwages$educ, bwages$sex)
```

```
##
##       0   1
##   1  42  57
##   2 120 145
##   3 186 234
##   4 181 175
##   5 168 164
```

9 Data import/export in R

In `R`, l'I/O da file è fondamentale per leggere e scrivere dati da e verso file.

La funzione `read.table()` è comunemente utilizzata per leggere dati tabellari da file di testo.

```
dati <- read.table("dati.txt", header = TRUE)
```

Altre funzioni, come `read.csv()` e `read.delim()` sono specializzate nella lettura di file CSV e di testo delimitato.

```
dati <- read.csv("dati.csv")
```

La funzione `readLines()` consente di leggere le righe di un file di testo.

```
righe <- readLines("testo.txt")
```

La funzione `scan()` è utilizzata per leggere dati da file di testo o da input utente.

```
dati <- scan("dati.txt", what = numeric(), sep = ",")
```

La funzione `write.csv()` è utilizzata per scrivere dati `R` in un file di testo. La sintassi di base è la seguente:

```
write.csv(dataframe, "nomefile.csv", row.names = FALSE)
```

dove `dataframe` è il dataframe da esportare e `"nomefile.csv"` è il nome del file CSV di destinazione. L'opzione `row.names = FALSE` specifica che non si desidera riportare l'intestazione delle righe (i nomi delle colonne) nel file csv.

```r
#Caricare il dataset mtcars.
data("mtcars")

#Filtrare le auto con un numero di cilindri maggiore di 4.
mtcars <- mtcars[mtcars$cyl > 4, ]

#Selezionare solo le colonne mpg, cyl, hp e gear
mtcars <- mtcars[, c(1, 2, 4, 10)]

#Rinominare la colonna mpg in kml (km per litro) e trasformare i dati.
names(mtcars)[names(mtcars) == "mpg"] <- "kml"
mtcars$kml <- mtcars$kml * 0.425144

#Salvare il risultato in un file CSV chiamato mtcars2.csv
write.csv(mtcars, "mtcars2.csv", row.names = TRUE)
```

10 Tidyverse e Dplyr

`Tidyverse` è un insieme di pacchetti in `R` progettati per facilitare l'analisi dei dati. Uno dei principali pacchetti di `Tidyverse` è `dplyr`, che fornisce una grammatica coerente per manipolare i dati. `dplyr` ha diverse funzionalità:

- filtraggio dei dati

```r
filter(data, condizone)
```

- selezione di colonne,

```r
select(data, col1, col2)
```

- aggiunta nuove colonne

```r
mutate(data, nuovacolonna = espressione)
```

- raggruppamento e sommarizzazione dei dati

```r
group_by(data, colonna) %>%
    summarise(media = mean(colonna))
```

- unione dei dati

```r
left_join(data1, data2, by = "colonna")
```

- ordinamento dei dati

```r
arrange(data, colonna)
```

10.1 Esempio sul dataset mtcars

Filtraggio dei dati per auto con `mpg` maggiore di 20.

```
data(mtcars)
mtcars_filtered <- mtcars %>%
  filter(mpg > 20)
head(mtcars_filtered)
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
```

Selezione delle colonne `mpg` e `cyl`.

```
mtcars_selected <- mtcars %>%
  select(mpg, cyl)
head(mtcars_selected)
```

```
##                   mpg cyl
## Mazda RX4         21.0   6
## Mazda RX4 Wag     21.0   6
## Datsun 710        22.8   4
## Hornet 4 Drive    21.4   6
## Hornet Sportabout 18.7   8
## Valiant           18.1   6
```

Aggiunta di una nuova colonna `hp_per_cyl` che rappresenta la potenza per cilindro.

```
mtcars_new_column <- mtcars %>%
  mutate(hp_per_cyl = hp/cyl)
head(mtcars_new_column)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb hp_per_cyl
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4   18.33333
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4   18.33333
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1   23.25000
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1   18.33333
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2   21.87500
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1   17.50000
```

Raggruppamento per numero di cilindri e calcolo della media di `mpg`.

```
mtcars_grouped <- mtcars %>%
  group_by(cyl) %>%
  summarise(mean_mpg = mean(mpg))
head(mtcars_grouped)
```

```
## # A tibble: 3 x 2
##     cyl mean_mpg
##   <dbl>    <dbl>
## 1     4     26.7
## 2     6     19.7
## 3     8     15.1
```

Unione dei dati in base alla colonna `cyl`.

```
categoria_cilindrata <- data.frame(cyl = c(4, 6, 8),
                                   Livello_cilindrata = c("Bassa cilindrata",
                                   "Media cilindrata", "Alta cilindrata"))
mtcars_merged <- mtcars %>%
  left_join(categoria_cilindrata, by = "cyl")
head(mtcars_merged)
```

```
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb Livello_cilindrata
## 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4   Media cilindrata
## 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4   Media cilindrata
## 3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1   Bassa cilindrata
## 4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1   Media cilindrata
## 5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2    Alta cilindrata
## 6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1   Media cilindrata
```

Ordinamento dei dati in base alla colonna `mpg`.

```
mtcars_ordered <- mtcars %>%
  arrange(mpg)
head(mtcars_ordered)
```

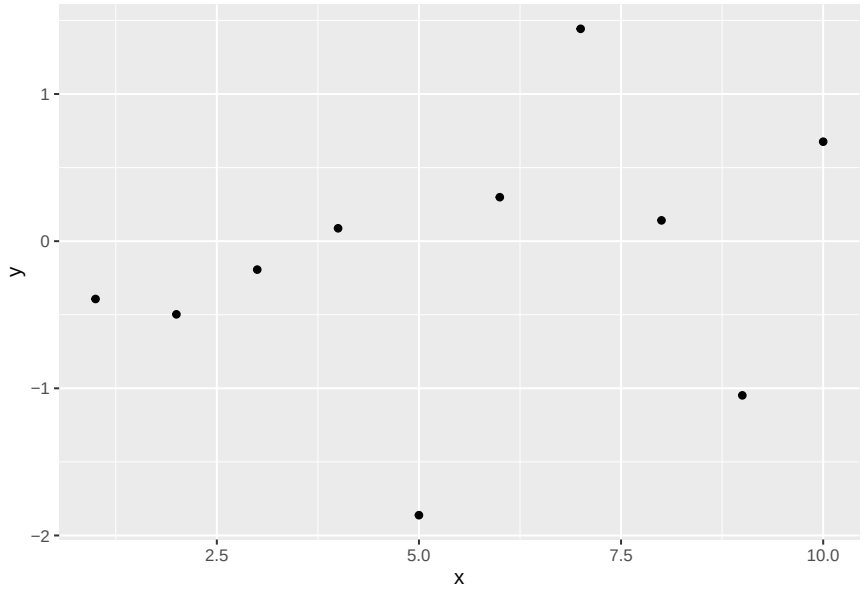
```
##                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Maserati Bora       15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
```

11 Visualizzazione dei dati in R

Il pacchetto `ggplot2` è una libreria di `R` per la visualizzazione dei dati che offre un'interfaccia coerente e potente per creare grafici, tra cui scatter plot, line plot, istogrammi e altri. La creazione di un grafico di base comporta la definizione di un dataset e l'aggiunta di uno o più layer grafici.

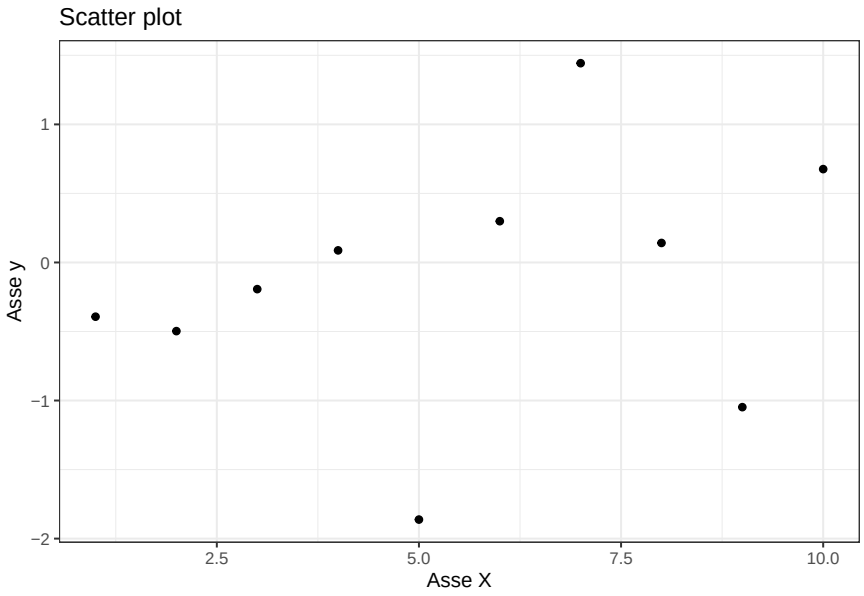
```
#Caricamento del pacchetto ggplot2
library(ggplot2)

#Creazione di un dataset di esempio
dati <- data.frame(x = 1:10,
                   y = rnorm(10))
```

```r
#Creazione di uno scatter plot
ggplot(dati, aes(x, y),) +
  geom_point()
```



ggplot2 offre molte opzioni per personalizzare grafici, tra cui la modifica dei colori, dei titoli, delle etichette degli assi e l'aggiunta di elementi grafici come linee, rettangoli e testo.

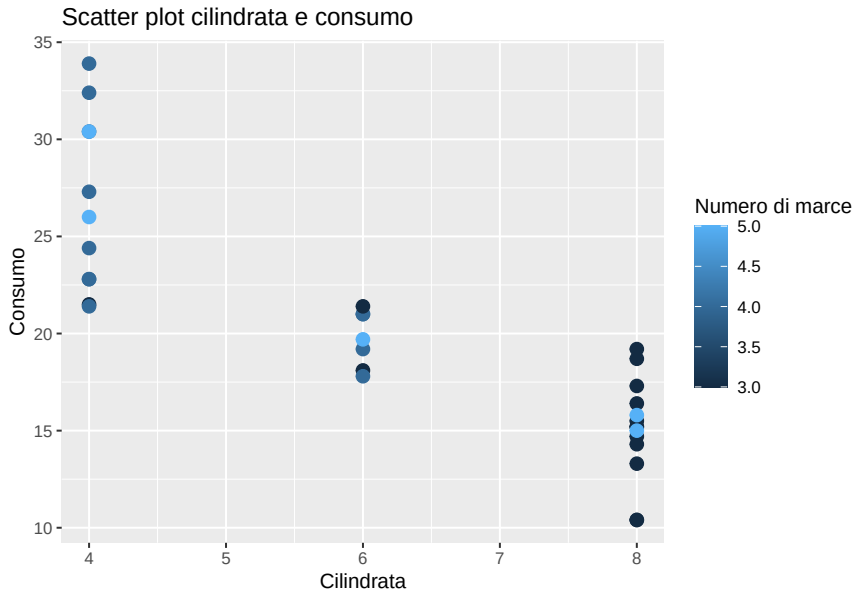
```r
ggplot(dati, aes(x, y),) +
  labs(title = "Scatter plot", x = "Asse X", y = "Asse y") +
  geom_point() +
  theme_bw()
```



Tipi di geometria in `ggplot2`:

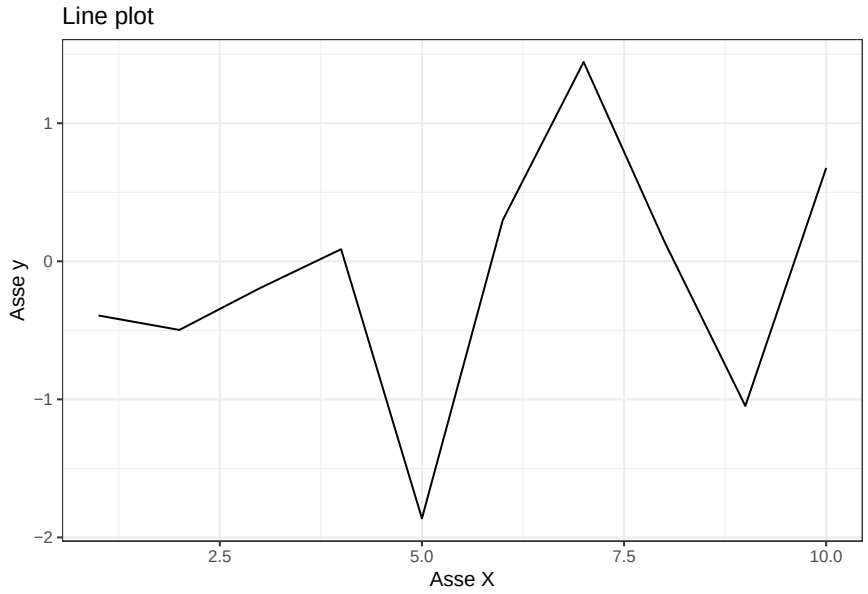
- `geom_point()` crea un grafico a dispersione (scatter plot) mostrando i punti dati

```
ggplot(mtcars, aes(cyl, mpg)) +
  geom_point(size = 3, aes(color = gear)) +
  labs(title = "Scatter plot cilindrata e consumo",
       x = "Cilindrata", y = "Consumo", color = "Numero di marce")
```



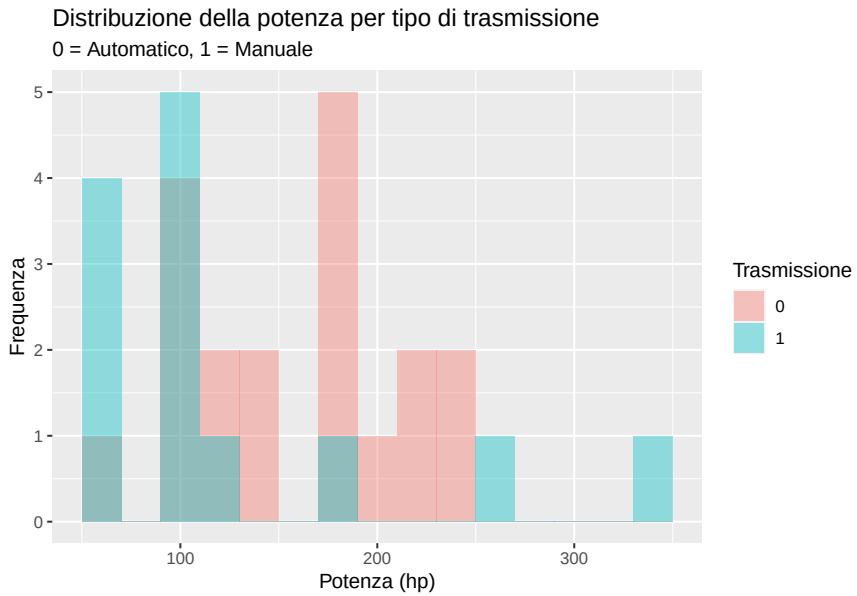
- `geom_line()` crea un grafico a linee collegando i punti dati in ordine

```
ggplot(dati, aes(x, y),) +
  labs(title = "Line plot", x = "Asse X", y = "Asse y") +
  geom_line() +
  theme_bw()
```



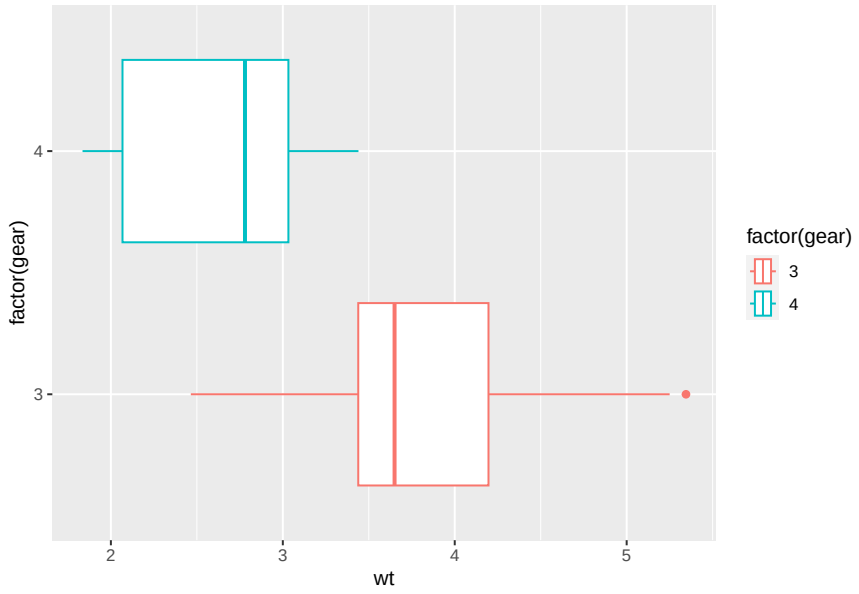
- `geom_bar()` crea un grafico a barre rappresentando la frequenza o il valore di una variabile
- `geom_histogram()` crea un istogramma mostrando la distribuzione di una variabile continua

```
data(mtcars)

ggplot(mtcars, aes(x = hp, fill = factor(am))) +
  geom_histogram(binwidth = 20, position = "identity", alpha = 0.4) +
  labs(title = "Distribuzione della potenza per tipo di trasmissione",
       x = "Potenza (hp)", y = "Frequenza", fill = "Trasmissione",
       subtitle = "0 = Automatico, 1 = Manuale")
```



- `geom_boxplot()` crea un boxplot visualizzando la distribuzione di una variabile attraverso i quartili

```r
ggplot(subset(mtcars, gear == c(3, 4)), aes(wt, factor(gear))) +
  geom_boxplot(aes(color = factor(gear)))
```



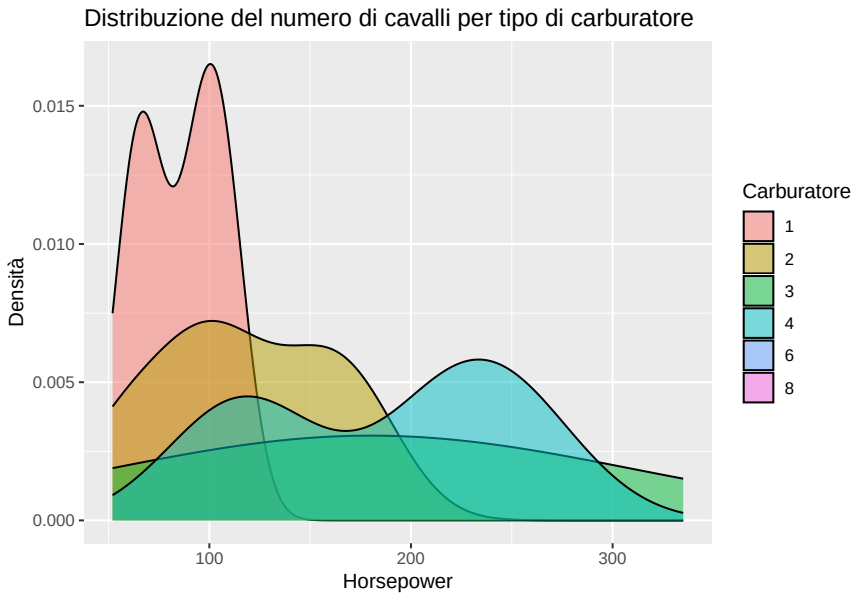
- `geom_density()` crea un grafico di densità mostrando la distribuzione di una variabile continua
- `geom_text()` aggiunge testo ai grafici, come etichette o annotazioni
- `geom_path()` traccia un percorso tra i punti dati, utile per dati sequenziali
- `geom_smooth()` aggiunge una curva di regressione o una linea liscia ai dati
- `geom_area()` riempie l'area sotto una curva, solitamente usato con linee o curve

```r
ggplot(mtcars, aes(x = hp, fill = factor(carb))) +
  geom_density(alpha = 0.5) +
  labs(title = "Distribuzione del numero di cavalli per tipo di carburatore",
       x = "Horsepower", y = "Densità",
       fill = "Carburatore")
```

```
## Warning: Groups with fewer than two data points have been dropped.
## Groups with fewer than two data points have been dropped.
```

```
## Warning in max(ids, na.rm = TRUE): nessun argomento non-mancante al massimo; si
## restituisce -Inf
```

```
## Warning in max(ids, na.rm = TRUE): nessun argomento non-mancante al massimo; si
## restituisce -Inf
```



- `geom_polygon()` crea un poligono collegando i punti dati, usato per rappresentare aree
- `geom_jitter()` aggiunge jitter ai dati, utile per rendere più chiari i dati sovrapposti
- `geom_tile()` crea un grafico a matrice di tessere, utile per rappresentare dati categorici
- `geom_violin()` crea un grafico a violino mostrando la distribuzione di una variabile
- `geom_errorbar()` aggiunge barre di errore ai grafici, utile per rappresentare la variazione
- `geom_abline()` aggiunge una linea di regressione o una linea con un'equazione specifica
- `geom_hline()` aggiunge una linea orizzontale a un grafico
- `geom_vline()` aggiunge una linea verticale a un grafico
- `geom_label()` aggiunge etichette ai punti sui grafici

12 Funzioni apply

Le funzioni di applicazione in `R` sono utilizzate per applicare una funzione ad una struttura dati, come una lista, un vettore, una matrice o un array. Esistono diverse funzioni di applicazione in `R`, tra cui:

- `lapply()`: applica una funzione ad ogni elemento di una lista
- `sapply()`: simile a `lapply()`, ma semplifica i risultati se possibile.
- `vapply()`: simile a `sapply()`, ma richiede un tipo di output specificato
- `mapply()`: applica una funzione a più argomenti.
- `apply()`: applica una funzione a matrici o array multidimensionali.

12.1 Esempi di utilizzo

Applica la funzione `sqrt()` ad ogni elemento di una lista.

```
lista <- list(a = 1:5, b = 6:10)
lapply(lista, sqrt)
```

```
## $a
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
##
## $b
## [1] 2.449490 2.645751 2.828427 3.000000 3.162278
```

Applica la funzione `mean()` ad ogni elemento di una lista.

```
sapply(lista, mean)
```

```
## a b
## 3 8
```

La differenza tra `sapply()` e `lapply()` è che la prima funzione restituisce in output un vettore nominato, mentre la seconda restituisce una lista.

Applica la funzione `mean()` ad ogni elemento di una lista e restituisce un vettore con tipo specificato.

```
vapply(lista, mean, FUN.VALUE = numeric(1))
```

```
## a b
## 3 8
```

Applica la funzione `paste()` a due vettori.

```
v1 <- c("a", "b", "c")
v2 <- 1:3
mapply(paste, v1, v2)
```

```
##     a     b     c
## "a 1" "b 2" "c 3"
```

Applica la funzione `mean()` alle *righe* di una matrice.

```
matrice <- matrix(1:9, nrow = 3)
apply(matrice, 1, mean)
```

```
## [1] 4 5 6
```

Applica la funzione `mean()` alle *colonne* di una matrice.

```
matrice <- matrix(1:9, nrow = 3)
apply(matrice, 2, mean)
```

```
## [1] 2 5 8
```

12.2 Esercizi di implementazione in R

Scrivi una funzione che cerchi in una matrice un numero e applichi una funzione su tale numero.

```r
applica_se_trovi <- function(matrix, number, fun){
  if(is.matrix(matrix) == F){
    stop("The first parameter is not a matrix")
  } else {
    for(i in 1:nrow(matrix)){
      for(j in 1:ncol(matrix)){
        if(matrix[i, j] == number){
          matrix[i, j] <- fun(number)
          return(matrix)
        }
      }
    }
  }
}

set.seed(1)
matrice <- matrix(round(runif(9, 1, 100), 0), nrow = 3)
applica_se_trovi(matrice, 27, sqrt)
```

```
##           [,1] [,2] [,3]
## [1,]  5.196152   91   95
## [2,] 38.000000   21   66
## [3,] 58.000000   90   63
```

```r
#alternativa
applica_se_trovi_bis <- function(m, n, func){
  if (!is.matrix(m)){
    stop("La funzione si applica solamente su matrici!")
  }

  nr <- nrow(m)
  nc <- ncol(m)

  i <- 1
  while(i <= nr){
    j <- 1
    while(j <= nc){
      val <- m[i, j]
      if(val == n){
        m[i, j] <- func(val)
      }
      j <- j + 1
    }
    i <- i + 1
  }

  return (m)
}
applica_se_trovi_bis(matrice, 27, sqrt)
```

```
##           [,1] [,2] [,3]
## [1,]  5.196152   91   95
```

```
## [2,] 38.000000   21   66
## [3,] 58.000000   90   63
```

Da notare che la matrice data in output è di tipo double, non integer come quella passata in input, perché R coercizza in modo implicito l'intero dato quando viene applicato una funzione che trasforma anche un solo elemento di esso (questo ragionamento funziona se genero la matrice con la funzione `sample`).

Scrivi un funzione che cerchi in una matrice un numero e applichi una funzione su tutta la riga in cui è presente il valore.

```
applica_se <- function(matrix, number, fun){
  if(is.matrix(matrix) == F){
    stop("The first parameter is not a matrix")
  } else {
    for(i in 1:nrow(matrix)){
      for(j in 1:ncol(matrix)){
        if(matrix[i, j] == number){
          return(fun(matrix[i, ]))
        }
      }
    }
  }
}

applica_se(matrice, 27, mean)
```

```
## [1] 71
```

```
#alternativa

applica_se_bis <- function(m, n, margin, func){
  if (!is.matrix(m)){
    stop("La funzione si applica solamente su matrici!")
  }

  if (margin != 1 && margin != 2){
    stop("Il parametro margin deve valere 1 o 2")
  }

  nr <- nrow(m)
  nc <- ncol(m)

  i <- 1
  while(i <= nr){
    j <- 1
    while(j <= nc){
      val <- m[i, j]
      if(val == n){
        if(margin == 1){ # applica su riga
           m[i, ] <- func(m[i, ])
        } else {
           m[, j] <- func(m[, j])
        }
```

```
      }
      j <- j + 1
    }
    i <- i + 1
  }

  return (m)
}

applica_se_bis(matrice, 27, 1, mean)
```

```
##      [,1] [,2] [,3]
## [1,]   71   71   71
## [2,]   38   21   66
## [3,]   58   90   63
```

Creare una versione della funzione `applica_se` che usi un parametro `margin` per operare su righe o colonne.

```
matrice <- matrix(round(runif(9, 1, 100), 0), nrow = 3)
applica_se_adv <- function(matrix, number, margin, fun){
  if(is.matrix(matrix) == F){
    stop("The first parameter is not a matrix")
  } else {
    for(i in 1:nrow(matrix)){
      for(j in 1:ncol(matrix)){
        if(matrix[i, j] == number){
          if(margin == 1){
            return(fun(matrix[i, ]))
          } else if (margin == 2){
            return(fun(matrix[, j]))
          } else {
            stop("Margin must be 1 or 2")
          }
        }
      }
    }
  }
}

applica_se_adv(matrice, 27, 2, mean)
```

Creare una versione di entrambe le funzioni con un parametro ulteriore `mul` che se `TRUE` continui la ricerca una volta trovato un primo match.

```
applica_se_adv <- function(matrix, number, margin, fun, mul = FALSE){
  if(is.matrix(matrix) == F){
    stop("The first parameter is not a matrix")
  } else {
    for(i in 1:nrow(matrix)){
      for(j in 1:ncol(matrix)){
        if(matrix[i, j] == number){
          if(margin == 1){
```

```
            return(fun(matrix[i, ]))
          } else if (margin == 2){
            return(fun(matrix[, j]))
          } else {
            stop("Margin should be 1 or 2")
          }
        }
      }
    }
  }
}

applica_se_adv(matrice, 27, 2, mean)
```

13 Esercizi di implementazione di funzioni

13.1 Vettori

1) Scrivi una funzione che calcoli la media di un vettore di numeri utilizzando un ciclo.

```
my_mean <- function(v){
  if(!is.numeric(v)){
    stop("L'input deve essere numerico!")
  }
  s <- 0
  for (elem in v) {
    s <- s + elem
  }
  return (s / length(v))
}

crea_vettore_num_casuale <- function(lun, min, max){
  return(runif(lun, min, max))
}
#TESTING
test_vec <- function(perfun, rfun, iter){
  i <- 1
  while(i <= iter){
  v <- crea_vettore_num_casuale(10, 0, 100)
  mia <- perfun(v)
  r <- rfun(v)
  if (all.equal(mia, r)){
    print(paste("iterazione", i , "OK"))
  }
  else {
    print(paste("Iterazione", i , "ERROR"))
  }
  i <- i+1
}
}
```

```
test_vec(my_mean, mean,8)
```

```
## [1] "iterazione 1 OK"
## [1] "iterazione 2 OK"
## [1] "iterazione 3 OK"
## [1] "iterazione 4 OK"
## [1] "iterazione 5 OK"
## [1] "iterazione 6 OK"
## [1] "iterazione 7 OK"
## [1] "iterazione 8 OK"
```

2) Implementa un funziona che calcoli la mediana di un vettore di numeri. Si può usare la funzione *bulit-in* in `sort()`

```
my_median <- function(v){
  if(!is.numeric(v)){
    stop("L'input deve essere numerico!")
  }
  v <- sort(v)
  if(length(v) %% 2 == 0){
    return(mean(c(v[length(v)/2], v[length(v)/2+1])))
  } else {
    return(v[(length(v)+1)/2])
  }
}

#TESTING
test_vec(my_median, median, 8)
```

```
## [1] "iterazione 1 OK"
## [1] "iterazione 2 OK"
## [1] "iterazione 3 OK"
## [1] "iterazione 4 OK"
## [1] "iterazione 5 OK"
## [1] "iterazione 6 OK"
## [1] "iterazione 7 OK"
## [1] "iterazione 8 OK"
```

3) Crea una funzione che restituisca un vettore contente solo gli elementi unici presenti nel vettore di input. Non usare una funzione *built-in* `unique()`. Si può usare l'operatore `%in%`. Usare funzione `vector()` per creare un vettore vuoto.

```
my_unique <- function(v){
  if(is.list(v)){
    stop("L'input non è di tipo atomic()")
  }
  j <- vector()
  i <- 1
  while(i <= length(v)){
    if (!v[i] %in% j){
      j <- c(j, v[i])
```

```
    }
   i <- i+1
  }
 return(j)
}

test <- c(1, 1, 3, 5, 6)

all.equal(unique(test), my_unique(test))
```

```
## [1] TRUE
```

```
test <- c(letters[1:7], letters[1:12]) #character

all.equal(unique(test), my_unique(test))
```

```
## [1] TRUE
```

```
#ALTERNATIVA CON FOR (meno efficiente)

my_unique <- function(x){
  if(!is.numeric(x)){
    stop("The vector in input must be numeric")
  }
  uniq <- vector()
  for(elem in x){
    if(elem %in% uniq){
      next
    } else {
      uniq <- c(uniq, elem)
    }
  }
  return(uniq)
}

my_unique(v) == unique(v)
```

```
##   [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [31] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [46] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [61] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [76] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [91] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```
length(my_unique(v)) == length(unique(v))
```

```
## [1] TRUE
```

```r
#ALTERNATIVA CON WHILE

my_unique_2 <- function(vec){
  if(is.list(vec)){
    stop("L'input non è di tipo atomico")
  }
  n <- length(vec)
  v <- vector()
  for (i in 1:n) {
    if(!(vec[i] %in% v))
    v <- c(v, vec[i])
  }
  return(v)
}
my_unique_2(test)
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

13.2 Matrici

1) Scrivi una funzione che calcoli la somma degli elementi di una matrice.

```r
my_matrix_sum <- function(m){
  if(!is.matrix(m)){
    stop("L'input non è di tipo matrix()")
  }
  sum <- 0
  for (i in 1:nrow(m)) {
    for (j in 1:ncol(m)) {
      sum <- sum + m[i,j]
    }
  }
  return(sum)
}

crea_matrice_num_casuale_normali <- function(n, media, devst, row, col){
  return(matrix(data = rnorm(n, mean = media, sd = devst ),
                nrow = row,
                ncol = col ))
}

test_matrix <- function(perfun, rfun, iter){
i <- 1
while(i <= iter){
  v <- crea_matrice_num_casuale_normali(10, 0, 1, 5 ,2)
  mia <- perfun(v)
  r <- rfun(v)
  if (all.equal(round(mia), round(r))){
    print(paste("iterazione", i , "OK"))
  }
  else {
    print(paste("Iterazione", i , "ERROR"))
```

```
  }
  i <- i+1
}
}

test_matrix(my_matrix_sum, sum, 10)
```

```
## [1] "iterazione 1 OK"
## [1] "iterazione 2 OK"
## [1] "iterazione 3 OK"
## [1] "iterazione 4 OK"
## [1] "iterazione 5 OK"
## [1] "iterazione 6 OK"
## [1] "iterazione 7 OK"
## [1] "iterazione 8 OK"
## [1] "iterazione 9 OK"
## [1] "iterazione 10 OK"
```

2) Implementa una funzione che calcoli la trasposta di una matrice.

```
my_matrix_transpose <- function(m){
  if(!is.matrix(m)){
    stop("L'input non è di tipo matrix()")
  }
  trasp <- vector()
    for (i in 1:nrow(m)) {
      trasp <- c(trasp,m[i,])
    }
  return(matrix(trasp, nrow = ncol(m), ncol = nrow(m)))
}

#ALTERNATIVA

my_matrix_transpose <- function(matrix){
  if(!is.matrix(matrix)){
    stop("The input must be a matrix")
  }
  transpose <- matrix(rep(0, prod(dim(matrix))), nrow = ncol(matrix), ncol = nrow(matrix))
  for(i in 1:nrow(matrix)){
    transpose[, i] <- matrix[i, ]
  }
  return(transpose)
}

test_matrix(my_matrix_transpose,t,10)
```

```
## [1] "iterazione 1 OK"
## [1] "iterazione 2 OK"
## [1] "iterazione 3 OK"
## [1] "iterazione 4 OK"
## [1] "iterazione 5 OK"
## [1] "iterazione 6 OK"
```

```
## [1] "iterazione 7 OK"
## [1] "iterazione 8 OK"
## [1] "iterazione 9 OK"
## [1] "iterazione 10 OK"
```

13.3 Array

1) Scrivi una funzione che calcoli la somma degli elementi di un array tridimensionale.

```
my_array_sum <- function(x){
  if(!is.array(x) | length(dim(x)) != 3){
    stop("The input must be an array of three dimensions")
  }
  sum <- 0
  for(i in 1:nrow(x)){
    for(j in 1:ncol(x)){
      for(h in 1:dim(x)[3]){
        sum <- sum + x[i, j, h]
      }
    }
  }
  return(sum)
}

crea_array_num_casuali <-function(n, i, j){
  array(sample(n), dim = c(i, j, 3))
}

test_array<-function(perfun, rfun, iter){
i <- 1
while(i <= iter){
  v <- crea_array_num_casuali(30, 2, 5)
  mia <- perfun(v)
  r <- rfun(v)
  if (all.equal(round(mia),round(r))){
    print(paste("iterazione", i , "OK"))
  }
  else {
    print(paste("Iterazione", i , "ERROR"))
  }
  i <- i+1
}
}

test_array(my_array_sum, sum, 10)
```

```
## [1] "iterazione 1 OK"
## [1] "iterazione 2 OK"
## [1] "iterazione 3 OK"
## [1] "iterazione 4 OK"
## [1] "iterazione 5 OK"
## [1] "iterazione 6 OK"
## [1] "iterazione 7 OK"
```

```
## [1] "iterazione 8 OK"
## [1] "iterazione 9 OK"
## [1] "iterazione 10 OK"
```

2) Implementa una funzione che calcoli la media degli elementi di un array multidimensionale.

```
my_array_mean <- function(arr){
  if(!is.array(arr)){
    stop("L'input non è di tipo array()")
  }
  v <- 0
    for (i in 1:dim(arr)[1]) {
      for (j in 1:dim(arr)[2]) {
        for(s in 1:dim(arr)[3]){
          v <- v + arr[i,j,s]
        }
      }
    }
   return(v/length(arr))
}

#ALTERNATIVA CON UTILIZZO DI ALTRE FUNZIONI

my_array_mean <- function(x){
  if(!is.array(x)){
    stop("The input must be an array")
  }
  somma <- my_array_sum(x)
  m_mean <- somma/prod(dim(x))
  return(m_mean)
}

#TESTING

test_array(my_array_mean, mean, 10)
```

```
## [1] "iterazione 1 OK"
## [1] "iterazione 2 OK"
## [1] "iterazione 3 OK"
## [1] "iterazione 4 OK"
## [1] "iterazione 5 OK"
## [1] "iterazione 6 OK"
## [1] "iterazione 7 OK"
## [1] "iterazione 8 OK"
## [1] "iterazione 9 OK"
## [1] "iterazione 10 OK"
```

3) Implementa una funzione che restituisca `TRUE` se la media degli elementi di un array multidimensionale supera un valore di soglia s dato in input.

```
my_array_sup <- function(arr, sup){
  if(!is.array(arr)){
```

```
    stop("L'input non è di tipo array()")
  }
  v <- 0
    for (i in 1:dim(arr)[1]) {
      for (j in 1:dim(arr)[2]) {
        for(s in 1:dim(arr)[3]){
          v <- v + arr[i,j,s]
        }
      }
    }
  if(v/length(arr) > sup){
       return("TRUE")
  } else{
    FALSE
  }
}

#ALTERNATIVA CON UTILIZZO DI ALTRE FUNZIONI

my_array_sup <- function(x, s){
  if(!is.array(x)){
    stop("The input must be an array")
  }
  media <- my_array_mean(x)
  if(media > s){
    return(TRUE)
  } else {
    return(FALSE)
  }
}

set.seed(123)
test <- crea_array_num_casuali(40,2,3)

my_array_sup(test, 2)
```

```
## [1] TRUE
```

```
my_array_sup(test, 50)
```

```
## [1] FALSE
```

```
my_array_mean(test)
```

```
## [1] 21.77778
```

```
mean(test)
```

```
## [1] 21.77778
```

```r
# Esercizio "bonus" calcola media di valori oltre una certa soglia
my_array_mean_if <- function(arr, sup){
  if(!is.array(arr)){
    stop("L'input non è di tipo array()")
  }
  v <- 0
  cicli <- 0
    for (i in 1:dim(arr)[1]) {
      for (j in 1:dim(arr)[2]) {
        for(s in 1:dim(arr)[3]){
          if (arr[i,j,s] > sup){
            v <- v + arr[i,j,s]
            cicli <- cicli + 1
          }
        }
      }
    }
  if(cicli == 0){
    return(0)
  } else{
    return(v/cicli)
  }
}

#TESTING

test <- crea_array_num_casuali(40,2,3)
round(my_array_mean_if(test, 2)) == round(mean(test[test > 2]))
```

```
## [1] TRUE
```

13.4 Liste

1) Scrivi una funzione che restituisca il numero di elementi presenti in una lista.

```r
pts <- list(cars[,1], cars[,2], letters[1:15])

my_list_length <- function(lis){
  if(!is.list(lis)){
    stop("The input must be a list")
  }
  count <- 0
  for(elem in lis){
    count <- count + 1
  }
  return(count)
}

#ALTERNATIVA CHE NON CONSIDERA GLI NA

my_list_lenght <- function(l){
  if (!is.list(l)){
```

```
    stop("L'input non è di tipo list()")
  }
  v <- vector()
  for (s in 1:length(pts)) {
    v <- c(v, !is.na(l[[s]]))
  }
  return(sum(v))
}


my_list_lenght(pts)
```

```
## [1] 115
```

2) Implementa una funzione che inverte l'ordine degli elementi in una lista.

```
my_list_reverse <- function(lis){
  if(!is.list(lis)){
    stop("The input must be a list")
  }
  reverse <- list()
  for(i in 1:length(lis)){
    reverse[[i]] <- lis[[length(lis) - i + 1]]
  }
  return(reverse)
}

#ALTERNATIVA ORDINE DECRESCENTE

my_list_reverse <- function(l){
  if (!is.list(l)){
    stop("L'input non è di tipo list()")
  }
  for (s in 1:length(l)) {
   l[[s]] <- sort(l[[s]], TRUE)
  }
  return(l)
}

my_list_reverse(pts)
```

```
## [[1]]
##  [1] 25 24 24 24 24 23 22 20 20 20 20 20 19 19 19 18 18 18 18 17 17 17 16 16 15
## [26] 15 15 14 14 14 14 13 13 13 13 12 12 12 12 11 11 10 10 10  9  8  7  7  4  4
##
## [[2]]
##  [1] 120  93  92  85  84  80  76  70  68  66  64  60  56  56  54  54  52  50  48
## [20]  46  46  42  40  40  36  36  34  34  34  32  32  32  28  28  26  26  26  26
## [39]  24  22  20  20  18  17  16  14  10  10   4   2
##
## [[3]]
##  [1] "o" "n" "m" "l" "k" "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```

13.5 Dataframes

1) Scrivi una funzione che calcoli la media delle colonne numeriche in un dataframe.

```
my_df_mean <- function(df){
  if(!is.data.frame(df)){
    stop("L'input non è di tipo data.frame() !")
  }
  v <- vector()
  c <- character()
  df <- na.omit(df)
  for (j in 1:ncol(df)) {
    if(is.numeric(df[,j])){
      c <- c(c, names(df)[j])
      v <- c(v, mean(df[,j]))
    }
  }
  return(setNames(v, c))
}

#ALTERNATIVA

my_df_mean <- function(df){
  if(!is.data.frame(df)){
    stop("The input must be a dataframe")
  }
  mean_values <- vector()
  for(i in 1:ncol(df)){
    if(is.numeric(df[, i])){
      mean_values <- c(mean_values, sum(df[, i], na.rm = T)/nrow(df))
    }
  }
  result_df <- data.frame(Column_Index = which(sapply(df, is.numeric)),
                          Mean = mean_values)
  return(result_df)
}

my_df_mean(iris)
```

```
##              Column_Index     Mean
## Sepal.Length            1 5.843333
## Sepal.Width             2 3.057333
## Petal.Length            3 3.758000
## Petal.Width             4 1.199333
```

2) Implementa una funzione che filtri le righe di un dataframe in base a una condizione specificata.

```
my_df_filter <- function (df, cond, value, var){
  if(!is.data.frame(df)){
    stop("L'input non è di tipo dataframe")
  }
  filtro <- switch(cond,
    "==" = df[df[[var]] == value, ],
```

```
    "!=" = df[df[[var]] != value, ],  #non funziona con il $
    ">"  = df[df[[var]] > value, ],
    "<"  = df[df[[var]] < value, ],
    ">=" = df[df[[var]] >= value, ],
    "<="= df[df[[var]] <= value, ])
return(filtro)
}

#ALTERNATIVA

my_df_filter_2 <- function(df, condition){
  if(!is.data.frame(df)){
    stop("The input must be a dataframe")
  }
  df_subset <- data.frame()
  for(i in 1:nrow(df)){
    if(condition[i]){
      df_subset <- rbind(df_subset, df[i, ])
    }
  }
  return(df_subset)
}

crea_dataframe <- function(num_rows, num_numeric, num_factor) {
  numeric_names <- paste0("num", 1:num_numeric)
  factor_names <- paste0("factor", 1:num_factor)
  df <- data.frame(matrix(ncol = num_numeric + num_factor, nrow = num_rows))
  names(df)[1:num_numeric] <- numeric_names
  names(df)[(num_numeric + 1):(num_numeric + num_factor)] <- factor_names
  for(i in 1:num_numeric){
    df[, i] <- runif(num_rows)
  }
  for(i in 1:num_factor){
    df[, num_numeric + i] <- sample(letters, num_rows, replace = TRUE)
  }

  return(df)
}

set.seed(123)
df <- crea_dataframe(num_rows = 10, num_numeric = 3, num_factor = 2)

my_df_filter(mtcars, "==", 1 , "vs")
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
```

```
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
my_df_filter_2(df, df$factor2 == "i")
```

```
##        num1      num2      num3 factor1 factor2
## 4 0.8830174 0.5726334 0.9942698       n       i
## 5 0.9404673 0.1029247 0.6557058       q       i
```

14 Esercizi di implementazione di algoritmi di ordinamento e test di efficienza

Bubble Sort: implementa l'algoritmo di ordinamento a bolle di un vettore (gli elementi più grandi vengono spostati a destra lungo il vettore). Si implementa confrontando coppie di elementi adiacenti, scambiandoli se non sono in ordine, fino a che non si arriva a completare una scansione del vettore senza alcuno scambio. Contare il numero di operazioni/istruzioni totali.

```
# Implementazione del bubble sort
bubble_sort <- function(v){
  scansiona_ancora = TRUE
  while(scansiona_ancora == TRUE){
    scansiona_ancora = FALSE
    for (i in 1:(length(v)-1)){
      if(v[i] > v[i+1]){
        variabile_appoggio <- v[i]
        v[i] <- v[i+1]
        v[i+1] <- variabile_appoggio
        scansiona_ancora = TRUE
      }
    }
  }
  return (v)
}

# Implementazione del bubble sort
bubble_sort_eff <- function(v){
  conta_op <- 0
  scansiona_ancora = TRUE
  conta_op <- conta_op + 1 # assegn scansiona_ancora
  while(scansiona_ancora == TRUE){
    conta_op <- conta_op + 1 # condizione del while
    scansiona_ancora = FALSE
    conta_op <- conta_op + 1 # assegn scansiona_ancora
    for (i in 1:(length(v)-1)){
      conta_op <- conta_op + 1
      if(v[i] > v[i+1]){
        conta_op <- conta_op + 1 # controllo della if
```

```
        variabile_appoggio <- v[i]
        conta_op <- conta_op + 1 # assegn variabile appoggio
        v[i] <- v[i+1]
        conta_op <- conta_op + 1 # primo scambio
        v[i+1] <- variabile_appoggio
        conta_op <- conta_op + 1 # secondo scambio
        scansiona_ancora = TRUE
        conta_op <- conta_op + 1 # assegn scansiona_ancora
      }
    }
  }
  print(paste("Totale operazioni:", conta_op))
  return (v)
}

v <- runif(20, 0, 100) # Avg case
#v <- 1:20 # Best case
#v <- 200:1 # Worst case

bubble_sort_eff(v)
```

```
## [1] "Totale operazioni: 856"

##  [1]  0.06247733  9.48406609 12.75316502 20.65313896 27.43836446 37.44627759
##  [7] 38.39696378 43.98316876 44.85163414 56.09479838 62.92211316 66.51151946
## [13] 71.01824014 75.33078643 75.44751586 79.43423211 81.00643530 81.23895095
## [19] 81.46400389 89.50453592
```

Implementa l'algoritmo di ordinamento rapido: dato un vettore, si sceglie un elemento casuale (pivot) e si effettua un partizionamento in due vettori (sx e dx): quello di sx ha elementi minori del pivot (quello di dx maggiori) Ricorsivamente, si ordinano i vettori sx e dx fino al caso limite (vettori con un valore solo, che viene semplicemente restituito. Contare il numero di operazioni/istruzioni totali.

```
# Implementazione del quick sort
quick_sort <- function(v){
  if (length(v) <= 1){
    return (v)
  }

  pivot <- v[1]
  sx <- v[v < pivot]
  dx <- v[v > pivot]

  minori <- quick_sort(sx)
  maggiori <- quick_sort(dx)

  risultato <- c(minori, pivot, maggiori)
  return (risultato)
}

# Implementazione del quick sort con calcolo eff.
quick_sort_eff <- function(v){
  cont_quick_sort <- 0
```

```
  cont_quick_sort <- cont_quick_sort + 1
  if (length(v) <= 1){
    return (v)
  }

  pivot <- v[1]
  cont_quick_sort <- cont_quick_sort + 1
  sx <- v[v < pivot]
  cont_quick_sort <- cont_quick_sort + length(v)
  dx <- v[v > pivot]
  cont_quick_sort <- cont_quick_sort + length(v)

  minori <- quick_sort(sx)
  maggiori <- quick_sort(dx)

  risultato <- c(minori, pivot, maggiori)
  cont_quick_sort <- cont_quick_sort + 1
  print(paste("Totale operazioni:", cont_quick_sort))
  return (risultato)
}


# Testing
bubble_sort(v) == sort(v)
```

```
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [16] TRUE TRUE TRUE TRUE TRUE
```

```
quick_sort(v) == sort(v)
```

```
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [16] TRUE TRUE TRUE TRUE TRUE
```

Confrontare l'efficienza dei due algoritmi con una funzione di test (su n vettori generati sinteticamente).

```
# Efficienza
#v <- runif(20, 0, 100) # Avg case
#v <- 1:20 # Best case
v <- 20:1 # Worst case
quick_sort_eff(v)
```

```
## [1] "Totale operazioni: 43"
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
bubble_sort_eff(v)
```

```
## [1] "Totale operazioni: 1371"
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

15 Esercitazione 1 - Space Exploration

15.1 Importare i dati

4) Importare il dataset, impostando correttamente separatore campi, separatore decimale e presenza dell'intestazione.

```
launches <- read.csv("launches.csv", header = TRUE, dec = ",", na.strings = "",
                     quote = "")
launches <- arrange(launches, launch_date)
```

Quale struttura dati ci si aspetta di trovare dopo aver importato i dati?

```
class(launches)
```

```
## [1] "data.frame"
```

5) Esplorare i dati caricati nel workspace attraverso l'interfaccia di `R` (riquadro "Environment" in alto a destra)

```
glimpse(launches)
```

```
## Rows: 5,756
## Columns: 11
## $ tag         <chr> "1957 ALP", "1957 BET", "1957-F01", "1958 ALP", "1958-F01"~
## $ JD          <chr> "2436116.31", "2436145.6", "2436179.2", "2436235.66", "243~
## $ launch_date <chr> "1957-10-04", "1957-11-03", "1957-12-06", "1958-02-01", "1~
## $ launch_year <int> 1957, 1957, 1957, 1958, 1958, 1958, 1958, 1958, 1958, 1958~
## $ type        <chr> "Sputnik 8K71PS", "Sputnik 8K71PS", "Vanguard", "Jupiter C~
## $ variant     <chr> NA, NA, NA, "JunoI", NA, "JunoI", NA, "JunoI", NA, NA, NA,~
## $ mission     <chr> "PS-1", "PS-2", "Vanguard Test Satellite", "Explorer 1", "~
## $ agency      <chr> NA, NA, "US", "US", "US", "US", "US", "US", NA, "US", NA, ~
## $ state_code  <chr> "SU", "SU", "US", "US", "US", "US", "US", "US", "SU", "US"~
## $ category    <chr> "O", "O", "F", "O", "F", "F", "O", "O", "F", "F", "O", "F"~
## $ agency_type <chr> "state", "state", "state", "state", "state", "state", "sta~
```

15.2 Albori dell'esplorazione spaziale

6) Sapendo già di avere i dati ordinati per data, ricavare la prima riga della tabella.

```
launches[1,]
```

```
##        tag         JD launch_date launch_year           type variant mission
## 1 1957 ALP 2436116.31  1957-10-04        1957 Sputnik 8K71PS    <NA>    PS-1
##   agency state_code category agency_type
## 1   <NA>         SU        O       state
```

7) Dalla prima riga estrarre, in altrettante variabili, i valori associati ai campi `type`, `mission` e `launch_date`.

```r
type_1 <- launches[1, "type"]
mission_1 <- launches[1, "mission"]
launch_date_1 <- launches[1, "launch_date"]
```

8) Stampare due righe di testo, utilizzando opportunamente `print()`, `paste()` e le variabili memorizzate, in modo tale da ottenere il seguente risultato (senza virgolette):

```r
print(paste("Prima missione spaziale:", mission_1, type_1, "il giorno", launch_date_1))
```

```
## [1] "Prima missione spaziale: PS-1 Sputnik 8K71PS il giorno 1957-10-04"
```

```r
cat("Prima missione spaziale:", mission_1, type_1, "\nil giorno", launch_date_1)
```

```
## Prima missione spaziale: PS-1 Sputnik 8K71PS
## il giorno 1957-10-04
```

Qual è l'ordinamento applicato di default dalla funzione `sort`? Crescente o decrescente?

L'ordinamento di default della funzione `sort` è crescente. Ipotizzando di avere un vettore numerico v, le seguenti istruzioni sono equivalenti?

```r
v <- sample(1:10, 5)
min(v)
```

```
## [1] 3
```

```r
sort(v)[1]
```

```
## [1] 3
```

Si la soluzione è la medesima.

Estrarre, in una nuova variabile `dates`, i valori della colonna `launch_date`

```r
dates <- launches[, "launch_date"]
```

10) Ricavare e stampare a schermo il terzo elemento di dates

```r
dates[3]
```

```
## [1] "1957-12-06"
```

11) Ripetere l'operazione con gli elementi che vanno dal terzo all'ottavo

```r
dates[3:8]
```

```
## [1] "1957-12-06" "1958-02-01" "1958-02-05" "1958-03-05" "1958-03-17"
## [6] "1958-03-26"
```

12) Ripetere l'operazione selezionando solamente i seguenti indici:

```
dates[c(876, 909, 932, 659, 978, 1021, 1065, 1164, 1227, 1322, 1393)]
```

```
##  [1] "1968-10-31" "1969-01-23" "1969-03-24" "1967-03-22" "1969-08-09"
##  [6] "1969-12-23" "1970-05-20" "1971-03-03" "1971-08-19" "1972-05-25"
## [11] "1972-12-28"
```

Dati due numeri interi n e m, le seguenti istruzioni sono equivalenti?

```
seq(1, 4, 1)
```

```
## [1] 1 2 3 4
```

```
1:4
```

```
## [1] 1 2 3 4
```

Si.

13) Convertire, con l'apposita funzione e impostando correttamente il formato, le date da stringhe a oggetti di tipo `Date`, salvare quindi il risultato in una nuova variabile.

```
correct_dates <- as.Date(dates, format = "%Y-%m-%d")
head(correct_dates)
```

```
## [1] "1957-10-04" "1957-11-03" "1957-12-06" "1958-02-01" "1958-02-05"
## [6] "1958-03-05"
```

14) Ricavare e stampare a schermo il tipo dei dati prima e dopo la conversione; confrontare il risultato e assicurarsi che siano diversi.

```
typeof(correct_dates) == typeof(dates)
```

```
## [1] FALSE
```

Cosa è successo ai campi che non avevano la data del dataframe originale?

```
which(is.na(launches$launch_date))
```

```
##  [1] 5714 5715 5716 5717 5718 5719 5720 5721 5722 5723 5724 5725 5726 5727 5728
## [16] 5729 5730 5731 5732 5733 5734 5735 5736 5737 5738 5739 5740 5741 5742 5743
## [31] 5744 5745 5746 5747 5748 5749 5750 5751 5752 5753 5754 5755 5756
```

```
launches$launch_date[548]
```

```
## [1] "1966-05-17"
```

```
correct_dates[548]
```

```
## [1] "1966-05-17"
```

Ipotizzando di avere un vettore numerico `v` contenete alcuni valori `NA`, quale risultato ci si aspetta dall'applicazione della funzione `sum(v)`?

```
v <- c(NA, 3, 5, 5, 6)
sum(v)
```

```
## [1] NA
```

Se si applica la funzione somma ad un vettore contenente missing-value l'output generato sarà di tipo missing-value, la funzione non omette i dati mancanti di default; andrebbe specificato nell'argomento della funzione.

15.3 Percentuale di successo

I primi lanci erano caratterizzati da un'elevata probabilità di fallimento, oggi le cose fortunatamente sono cambiate. Calcoliamo il tasso di successo per i primi lanci e confrontiamolo con l'ultimo periodo disponibile del dataset.

15) Selezionare le righe da 1 a 978 (corrispondente al lancio della missione Apollo 11) mantenendo tutte le colonne.

```
subset_launches <- launches[1:978,]
```

16) Filtrare i dati mantenendo, in una nuova variabile, solamente i lanci con successo (ossia col valore della colonna category impostato a "O").

```
subset_launches_succes <- subset_launches[subset_launches$category == "O",]
```

17) Ricavare il numero di righe così ottenute e salvarlo in una variabile `n.success`.

```
n.succes <- nrow(subset_launches_succes)
```

18) Ricavare il numero di righe del dataframe originale (non filtrato) e salvarlo in una variabile `n.tot`.

```
n.tot <- nrow(subset_launches)
```

19) Calcolare la percentuale di successo come rapporto tra `n.success` e `n.tot` moltiplicato per 100.

```
per.succ <- (n.succes/n.tot) * 100
```

20) Utilizzare la funzione `round(x, digits)` per arrotondare il risultato ed eliminare i numeri decimali.

```
arr.per.succ <- round(per.succ)
```

21) Stampare a schermo il risultato con una stringa esplicativa, ad esempio:

```
print(paste("Percentuale successo primo periodo", arr.per.succ, "%"))
```

```
## [1] "Percentuale successo primo periodo 85 %"
```

Il periodo selezionato, che spazia dal lancio del primo satellite artificiale alla missione lunare, dura 12 anni. Selezioniamo ora un periodo altrettanto lungo con gli ultimi anni disponibili nel dataset, che termina nel 2018. Calcoliamo quindi il tasso di successo sul periodo 2006-2018.

22) Filtrare il dataset originale selezionando gli anni (`launch_year`) uguali o superiori a 2006.

```
launches_newera <- launches %>% filter(launch_year >= 2006)
```

Ripetere il calcolo precedente, stampare con la formattazione corretta il risultato e confrontarlo con il primo.

```
n.succes.newera <- nrow(launches_newera[launches_newera$category == "O",])
arr.per.succnewera <- round((n.succes.newera/nrow(launches_newera))*100)
print(paste("Percentuale successo ultimo periodo", arr.per.succnewera, "%"))
```

```
## [1] "Percentuale successo ultimo periodo 94 %"
```

```
print(c(arr.per.succ, arr.per.succnewera))
```

```
## [1] 85 94
```

15.4 New Space Economy

Inizialmente solo gli stati potevano farsi carico dei costi e dei rischi per i lanci spaziali, a partire dagli anni 80 però nuove aziende private si sono affacciate al neonato business dei lanci spaziali. All'europea Arianespace, la prima azienda mondiale in questo settore, si sono presto aggiunte nuove compagnie americane, russe e giapponesi. Negli ultimi anni il mercato è divenuto più competitivo grazie all'ingresso di nuove startup come SpaceX. Analizziamo questa tendenza con un grafico.

24) Ricavare dal dataset tutti i records non associati ad agenzie statali (ossia con `agency_type` diverso dal valore "state") e memorizzarli in un nuovo dataframe denominato `commercial`.

```
commercial <- launches[launches$agency_type != "state",]
```

Ipotizzando di voler ordinare un insieme di variabili categoriche {Lunedì, Martedì, Mercoledì, etc} quale ordinamento verrà applicato di default da `R`? Quale struttura dati si può utilizzare per specificare un ordine diverso?

Di default verrà utilizzato l'ordine alfabetico, per specificare un ordine differente si protebbe specificare la variabile come factor e impostare il parametro levels come si reputa più opportuno.

25) Estrarre in una nuova variabile le righe corrispondenti alla colonna `state_code`.

```
state_code <- commercial$state_code
```

26) Stampare a schermo il tipo (mode) dei dati estratti.

```
mode(state_code)
```

```
## [1] "character"
```

27) Il campo `state_code` è un esempio di variabile categorica che può assumere solo determinati valori discreti ("CYM", "J", "RU", "F", "US"). Convertire tali dati in fattori specificando manualmente l'ordine dei diversi livelli:

```
state_code <- factor(state_code, levels = c("CYM", "J", "RU", "F", "US"))
```

28) Stampare a schermo il tipo (`mode()`) dei dati convertiti ed assicurarsi che sia diverso da quello precedente la conversione.

```
mode(state_code)
```

```
## [1] "numeric"
```

29) Sostituire la colonna `state_code` del dataframe originale con la serie di dati convertita in factors.

```
commercial$state_code <- state_code
```

30) Importare, e installare se non si è già provveduto prima a farlo, le librerie `ggplot2` e `dplyr`.

```
library(dplyr)
library(ggplot2)
```

31) La libreria `dplyr` offre diverse funzioni per manipolare i dati, compresa la funzione `group_by` che utilizzeremo per raggruppare i lanci per anno e contare, con la funzione `n()`, il numero di lanci in ogni gruppo. Riportare nello script la seguente riga di codice:

```
grouped <- commercial %>% group_by(launch_year, state_code) %>%
  summarise(n = n(), .groups = "drop")
```

32) Eseguire l'istruzione ed esplorare, tramite l'interfaccia di RStudio o funzione `head()`, il dataframe ritornato.

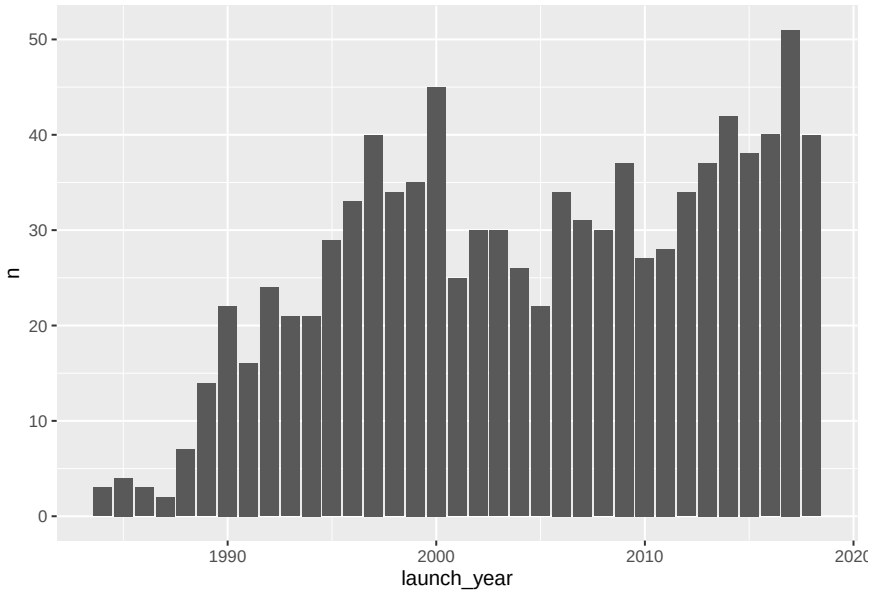
```
head(grouped)
```

```
## # A tibble: 6 x 3
##   launch_year state_code     n
##         <int> <fct>      <int>
## 1        1984 F              3
## 2        1985 F              4
## 3        1986 F              3
## 4        1987 F              2
## 5        1988 F              7
## 6        1989 F              7
```

33) Visualizzare il numero di lanci per anno con un grafico `ggplot` specificando il corretto mapping tra colonne del dataframe e assi (`launch_year` sull asse x e `n` sull'asse y) ed utilizzando il layer `geom_bar(position="stack", stat="identity")` per ottenere un grafico a barre.

```
plot1 <- ggplot(grouped, mapping = aes(x = launch_year, y = n)) +
  geom_bar(position = "stack", stat = "identity")
plot1
```

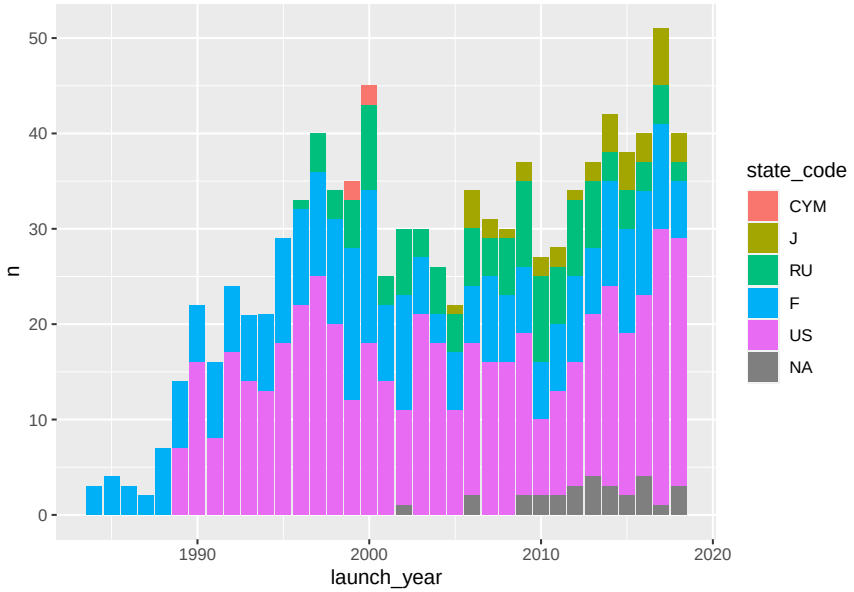
```
## Warning: Removed 1 rows containing missing values (`position_stack()`).
```



34) Utilizzare colori diversi per evidenziare i diversi paesi coinvolti aggiungendo il parametro `fill=state_code` alla funzione `aes()` Lo schema del codice diventa quindi:

```
plot1 <- ggplot(grouped, mapping = aes(x = launch_year, y = n, fill = state_code)) +
  geom_bar(position = "stack", stat = "identity")
plot1
```

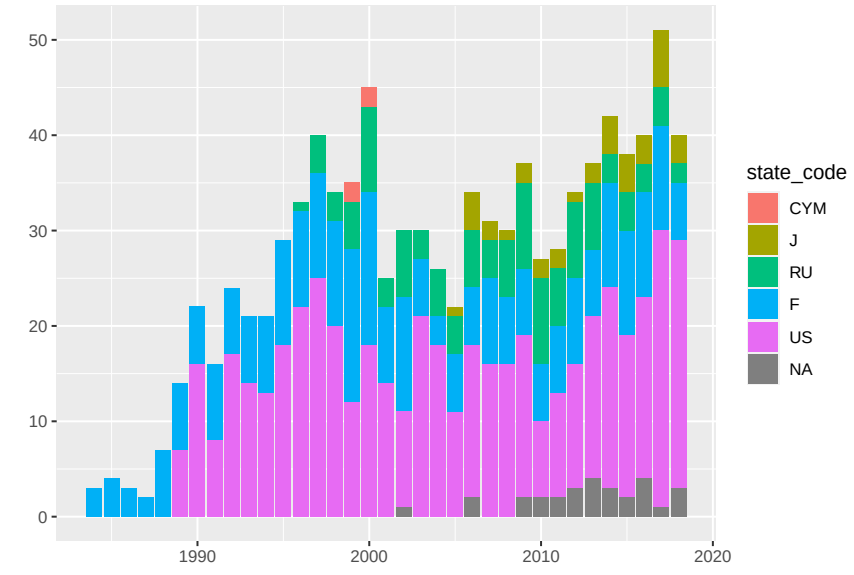
```
## Warning: Removed 1 rows containing missing values (`position_stack()`).
```



Eliminare titoli degli assi impostando di conseguenza una stringa vuota per i layers `ylab` e `xlab`.

```
plot1 <- ggplot(grouped, mapping = aes(x = launch_year, y = n, fill = state_code)) +
  geom_bar(position = "stack", stat = "identity") +
  labs(x = "", y = "")
plot1
```

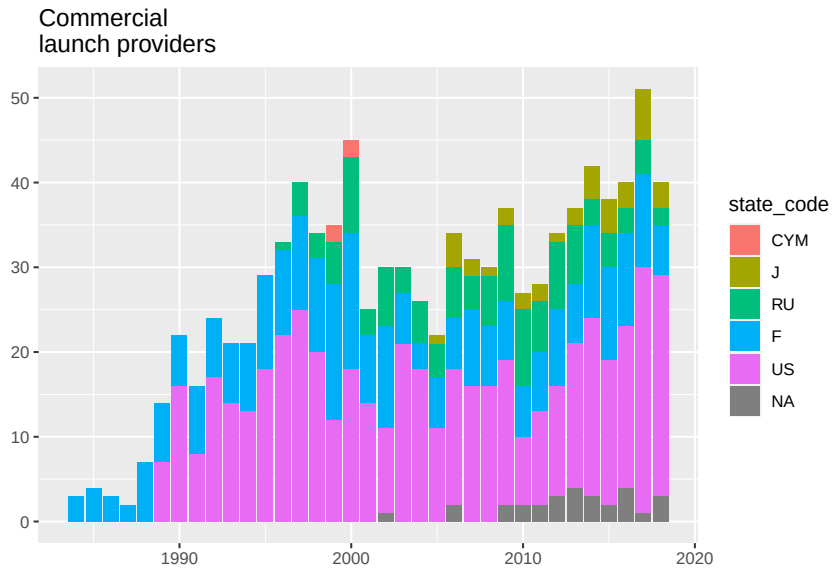
```
## Warning: Removed 1 rows containing missing values (‘position_stack()‘).
```



Aggiungere il titolo “Commercial launch providers” con il layer `ggtitle()` (opzionale) andare a capo dopo “Commercial”.

```
plot1 <- ggplot(grouped, mapping = aes(x = launch_year, y = n, fill = state_code)) +
  geom_bar(position = "stack", stat = "identity") +
  labs(x = "", y = "") +
  ggtitle("Commercial \nlaunch providers")
plot1
```

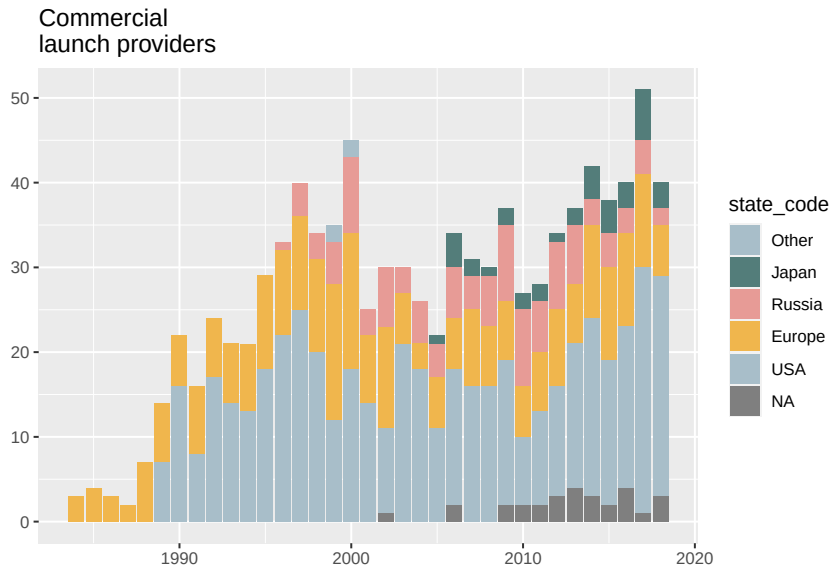
```
## Warning: Removed 1 rows containing missing values (‘position_stack()‘).
```



Aggiungere al plot il seguente layer per specificare nuovi colori e nuove etichette per gli stati:

```
plot1 <- ggplot(grouped, mapping = aes(x = launch_year, y = n, fill = state_code)) +
  geom_bar(position = "stack", stat = "identity") +
  labs(x = "", y = "") +
  ggtitle("Commercial \nlaunch providers") +
  scale_fill_manual(values=c("#a8bec9","#537d7a","#e79b96","#f0b64d","#a8bec9"),
                    labels=c('Other', 'Japan', 'Russia', "Europe", "USA"))
plot1
```

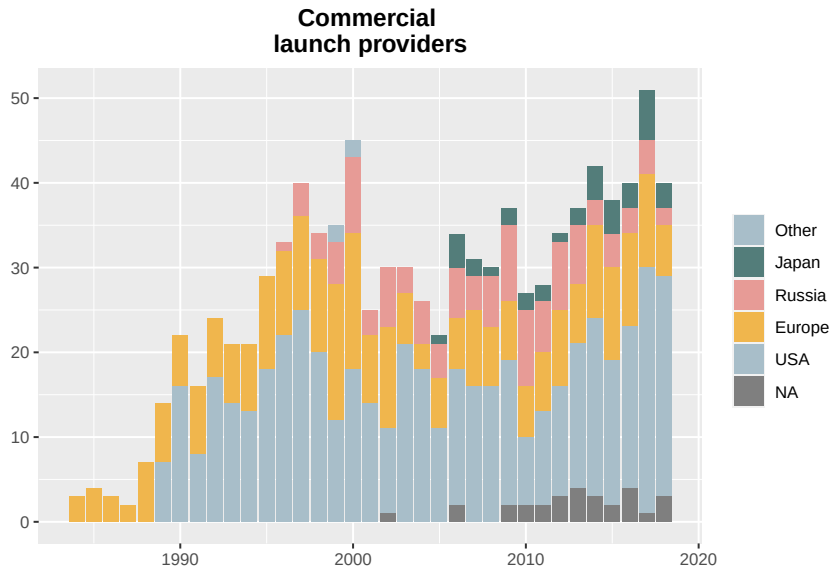
```
## Warning: Removed 1 rows containing missing values (‘position_stack()‘).
```



38) Aggiungere al plot il seguente layer per cambiare l'aspetto grafico (legenda senza titolo, titolo del grafico centrato ed in grassetto):

```
plot1 <- ggplot(grouped, mapping = aes(x = launch_year, y = n, fill = state_code)) +
  geom_bar(position = "stack", stat = "identity") +
  labs(x = "", y = "") +
  ggtitle("Commercial \nlaunch providers") +
  scale_fill_manual(values=c("#a8bec9","#537d7a","#e79b96","#f0b64d","#a8bec9"),
                    labels=c('Other', 'Japan', 'Russia', "Europe", "USA")) +
  theme(legend.title = element_blank(),
        plot.title = element_text(hjust = 0.5, face = "bold"))
plot1
```

```
## Warning: Removed 1 rows containing missing values (‘position_stack()‘).
```



16 Esercitazione 2 - Grain deal

16.1 Importare i dati

2) Creare un nuovo script e importare il dataset.

```
fao <- read.csv("fao.csv", header = TRUE)
```

3) Creare un vettore contenente le seguenti stringhe: “Element”, “Item”, “Value”.

```
string <- c("Element", "Item", "Value")
```

4) Creare un vettore contenente i seguenti numeri:

```
v <- c(2006, 3, 2010, 5, 2015, 5, 2020)
```

Come posso verificare il tipo dei dati?

Posso verificare il tipo di dati attraverso la funzione `typeof()` Cosa mi aspetto di ottenere dai due vettori?

Il primo sarà di tipo `"character"` e il secondo `"double"`.

```
typeof(string)
```

```
## [1] "character"
```

```
typeof(numeric())
```

```
## [1] "double"
```

Quali sono le differenze tra liste e vettori?

Le liste possono contenere elementi di natura diversa, ad esempio nuemerici, caratteri, booleani, ecc. Invece, per quanto riguarda i vettori possono essere contenuti soltanto elementi dello stesso tipo.

In linea di principio, potrei convertire un vettore in una lista? Si

Posso anche convertire una lista in un vettore?

No, neanche se la lista contiene elementi dello stessa tipologia.

5) Creare una nuova variabile `fao.world` filtrando il dataset originale, mantenendo solamente le righe con `Area` uguale a “World” e le colonne `Element`, `Item` e `Value`.

```
fao.world <- fao[fao$Area == "World", c("Element", "Item", "Value")]
```

6) Filtrare ulteriormente il dataset mantenendo solamente le righe con `Element` diverso da “Export Quantity” (e mantenendo tutte le colonne)

```
library(dplyr)
fao.world <- fao.world %>% filter(Element != "Export Quantity")
```

7) Ripetere l'operazione sulle le righe con `Element` diverso da "Import Quantity"

```
fao.world <- fao.world %>% filter(Element != "Import Quantity")
```

La maggior parte della produzione di cereali al mondo non è destinata all'uomo, bensì è utilizzata per l'alimentazione degli animali, come biofuel o per altri usi (ad esempio olio per sapone). Analizziamo, a partire dagli ultimi dati FAO disponibili, l'utilizzo e la produzione dei principali cereali: mais, grano e riso.

8) Filtrare, creando una nuova variabile di nome `food`, le righe con `Element` posto uguale a `Food` (mantenendo tutte le colonne).

```
food <- fao.world %>% filter(Element == "Food")
```

9) Estrarre la colonna `Value` dal dataframe filtrato.

```
food$Value
```

```
## [1] 504621 616308 145551
```

Quale struttura dati ci si aspetta di ottenere dalla colonna estratta? (vettore, lista...) Quale tipo di dati? (stringhe, numeri ...)

Ci si aspetta di ottenere una struttura di tipo vettore contenete dati di tipo numerico (`integer`).

10) Sommare i valori così ottenuti e stampare il risultato a schermo aggiungendo una descrizione testuale del tipo:

```
print(paste("Human food", sum(food$Value), "x1000 tonnes"))
```

```
## [1] "Human food 1266480 x1000 tonnes"
```

La funzione `which.max(v)` ritorna la posizione (indice) del massimo valore contenuto nel vettore `v` in input; corrisponde, in matematica, alla funzione `argmax`.

Creare un vettore con i seguenti numeri: 21 47 52 70 92 33 67 e applicare ad esso la funzione `which.max()`.

a) Quale risultati ci si aspetta?

```
v <- c(21, 47, 52, 70, 92, 33, 67)
ind <- which.max(v)
ind
```

```
## [1] 5
```

Viene riportato la posizione dell'elemento massimo contenuto nel vettore: nel nostro caso il quinto.

b) Utilizzare l'indice ottenuto con la funzione `which.max()` per indicizzare il vettore e ricavare così l'elemento corrispondente

```
v[ind]
```

```
## [1] 92
```

Dato un vettore numerico `v`, le seguenti istruzioni sono equivalenti?

```
max(v) == v[which.max(v)]
```

```
## [1] TRUE
```

Ipotizzando di avere un vettore `v` di 4 elementi e un vettore `t` di 6 elementi, quale risultato conterrà la variabile `l` dopo la seguente istruzione?

```
v <- c(1:4)
t <- c(1:6)
l <- c(length(v), length(t))
l
```

```
## [1] 4 6
```

11) Applicare la funzione `which.max()` alla colonna `Value` del dataframe `food` per ottenere l'indice del cibo più consumato.

```
indx <- which.max(food$Value)
```

12) Ricavare, data la posizione (riga) del cibo più consumato, il nome (`Item`) del prodotto corrispondente, ossia: indicizzare il dataframe `food` selezionando la riga indicata dalla funzione `which.max()` e la colonna `Item`.

```
max_food <- food$Item[indx]
```

13) Stampare il risultato a schermo con una descrizione testuale del tipo:

```
print(paste("Most consumed human food:", max_food))
```

```
## [1] "Most consumed human food: Rice"
```

14) Ripetere l'analisi effettuata sulla categoria "Food" anche con gli elementi di tipo "Feed".

```
feed <- fao.world %>% filter(Element == "Feed")
indx <- which.max(feed$Value)
max_feed <- feed$Item[indx]
print(paste("Most consumed human feed:", max_feed))
```

```
## [1] "Most consumed human feed: Maize"
```

15) Utilizzare gli operatori di congiunzione (and) e disuguaglianza (diverso) per individuare gli elementi (colonna `Element` del dataframe) diversi da "Food" e diversi da "Feed", salvare il risultato in un vettore di valori booleani (maschera).

```
maschera <- fao.world$Element != "Food" & fao.world$Element != "Feed"
```

16) Utilizzare la maschera così ottenuta per filtrare le righe del dataframe, mantenendo tutte le colonne.

```
masch.df <- fao.world[maschera,]
```

17) Stampare, analogamente a quanto fatto prima, la somma sui valori della colonna Value del dataframe filtrato, ottenendo un risultato del tipo:

```
print(paste("Other uses", sum(masch.df$Value), "x1000 tonnes"))
```

```
## [1] "Other uses 436932 x1000 tonnes"
```

18) Quale tipo di dati ci si aspetta di ottenere per le variabili a, b e c?

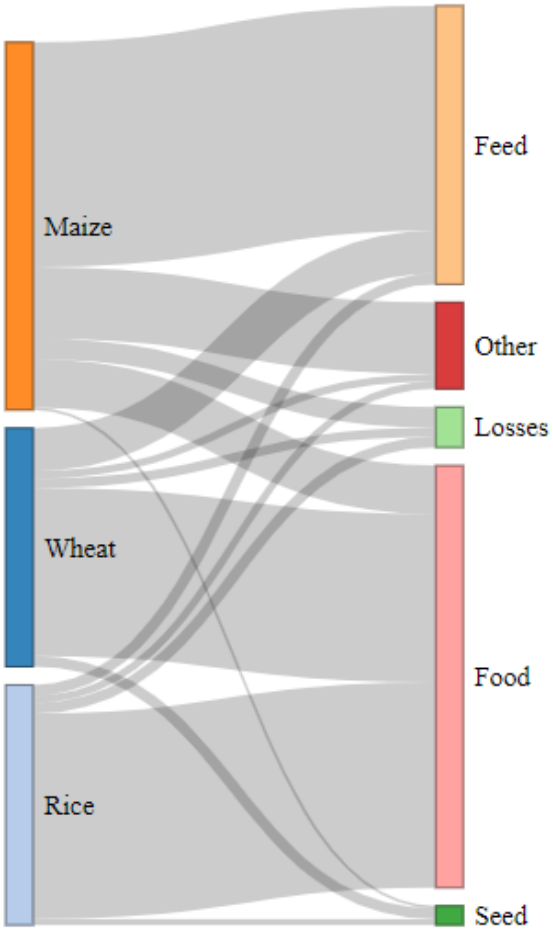
```
a = 5L
b = 5
c = "5"
```

Si otterrano in ordine: integer, double, character.

19) Installare (se non già presenti) ed importare le librerie `networkD3`, `ggplot2` e `dplyr`.

20) Incollare ed eseguire il seguente blocco di codice per visualizzare graficamente, tramite la libreria `networkD3`, l'utilizzo dei diversi alimenti presi in considerazione.

```
library(networkD3)
nodes <- data.frame(name=c(fao.world$Item, fao.world$Element) %>% unique())
fao.world$IDsource <- match(fao.world$Item, nodes$name)-1
fao.world$IDtarget <- match(fao.world$Element, nodes$name)-1
p <- sankeyNetwork(Links = fao.world, Nodes = nodes, Source = "IDsource", Target = "IDtarget", Value =
p
```



16.2 Grano ucraino

Russia e Ucraina combinate forniscono il 28% del grano commercializzato a livello mondiale, contribuendo a procurare il 12% delle calorie scambiate sui mercati e rifornendo, in larga parte, i paesi più poveri del mondo. Per queste ragioni, la guerra in corso ha serie ripercussioni sulla fame nel mondo. Analizziamo i danni collaterali sul mercato del grano individuando i paesi più esposti e calcolando la loro dipendenza dalle importazioni di grano provenienti dall'Ucraina. 21) Creare un nuovo script `R` ed importare le librerie `ggplot2` e `dplyr`.

Ipotizzando di avere un dataframe, oppure una matrice, `D` e conoscendo il significato delle istruzioni `nrow()` e `ncol()`, cosa ci si aspetta di ottenere dall'istruzione seguente?

```
D[nrow(D, ncol(D))]
```

Si ottiene il valore dell'ultima variabile dell'ultima osservazione.

22) Caricare il file "wheat_trade.csv" disponibile su moodle.

```
wheat_trade <- read.csv("wheat_trade.csv", header = TRUE, sep = "\t")
```

Cosa rappresenta invece il carattere speciale `\n` in `R`? Permette di mandare a capo un testo.

23) Filtrare, in una nuova variabile denominata "dependencies", il dataframe, mantenendo solo le importazioni dall'ucraina (righe con `Source` posto a "Ukraine")

```
dependencies <- wheat_trade %>% filter(Source == "Ukraine")
```

24) Filtrare una funzione di nome `compute.perc` che, date in ingresso due quantità `a` e `b`, calcola la percentuale come $\frac{a}{b}\%$ Il risultato dev'essere arrotondato a 2 cifre decimali.

```
compute.perc <- function(a, b){
  perc <-round( a/b * 100, 2)
  return(perc)
}
```

25) Provare la funzione su alcune coppie di numeri, ad esempio 3 - 5 e 2 - 3

```
compute.perc(3,5)
```

```
## [1] 60
```

```
compute.perc(2, 3)
```

```
## [1] 66.67
```

26) Riportare ed eseguire la seguente linea di codice per applicare la funzione `compute.perc` ad ogni riga del dataframe, aggiungendo il risultato in una nuova colonna `Percentage`.

```
dependencies <- dependencies %>% rowwise() %>%
  mutate(Percentage = compute.perc(Trade.Value, Tot.imports))
```

27) Esplorare il risultato con `head()` oppure con l'interfaccia di RStudio.

```
head(dependencies)
```

```
## # A tibble: 6 x 5
## # Rowwise:
##   Country  Source  Trade.Value Tot.imports Percentage
##   <chr>    <chr>         <dbl>       <dbl>      <dbl>
## 1 Benin    Ukraine        1110     4735382       0.02
## 2 Djibouti Ukraine     4071684    13407421      30.4
## 3 Algeria  Ukraine     1358099  1640493157       0.08
## 4 Egypt    Ukraine  1220179917  5195225572      23.5
## 5 Ethiopia Ukraine    39630626   320062300      12.4
## 6 Kenya    Ukraine    16793370   400218250       4.2
```

28) Riportare la seguente linea di codice per riordinare la tabella in base al valore contenuto nella colonna `Percentage`.

```
dependencies <- arrange(dependencies, desc(Percentage))
#con desc riordina dal più grande al più piccolo
```

Selezionare, in una nuova variabile `selected`, le prime 40 righe del dataframe riordinato.

```
selected <- dependencies[1:40,]
```

Dato un vettore `v`, quale delle seguenti istruzioni è equivalente a

```
v[1:3]
```

- v $c(1, 2, 3)$

  CORRETTO

- v $1, 2, 3$

  SBAGLIATO

- v $seq(1, 3, 2)$

  SBAGLIATO

- v $c(TRUE, TRUE, TRUE, TRUE)$

  SBAGLIATO

Estrarre, in una variabile `order`, la colonna `Country` dal dataframe `selected`.

```
order <- selected$Country
```

La funzione `rev(v)` permette di invertire l'ordine degli elementi di un vettore.

```
v <- c(1, 2, 5, 6, 8, 2)
rev(v)
```

```
## [1] 2 8 6 5 2 1
```

31) Sovrascrivere il valore del vettore `order` con il risultato della funzione `rev()` ad esso applicato (cioè invertire l'ordine degli elementi di `order`)

```
order <- rev(order)
```

Dato un vettore `v`, le seguenti istruzioni sono equivalenti?

- sort(v, decreasing = FALSE)
- rev(sort(v, decreasing = TRUE))

Si, la prima funzione riordina dal valore più piccolo al valore più grande invece attraverso il secondo comando prima viene ordinato dal più grande al più piccolo e successivamente invertito l'ordine.

Cosa dire invece delle seguenti istruzioni?

- rev(sort(v, decreasing=TRUE))

1

- min(v)

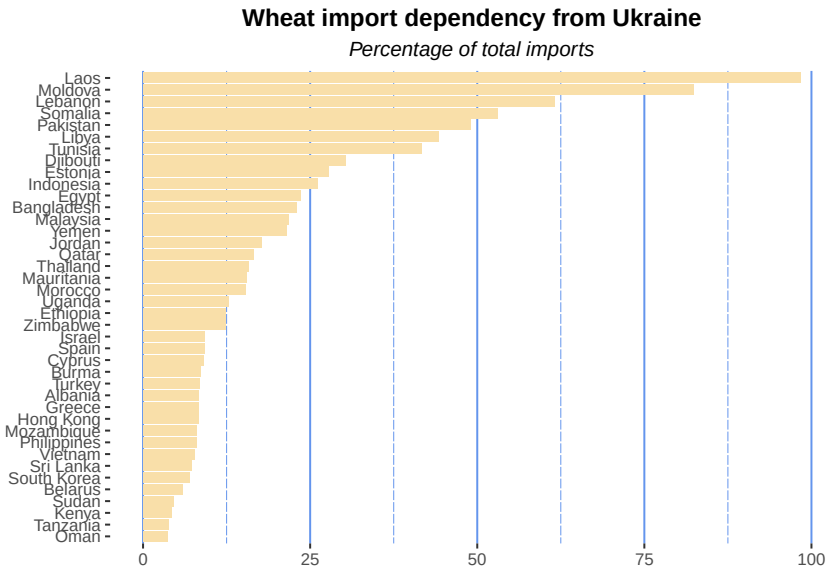
Anche in questo caso il risultato è lo stesso.

32) Sostituire la colonna `Country` del dataframe `selected` con i valori (della stessa colonna) convertiti in factors utilizzando come livelli il vettore `order` precedentemente creato; lo schema da seguire è il seguente:

```
selected$Country <- factor(selected$Country, levels = order)
```

33) Creare e visualizzare il seguente plot, impostando correttamente il mapping della funzione `aes()`, mappando la colonna `Country` sull'asse x e `Percentage` sull'asse y.

```
p = ggplot(selected, aes(x = Country, y = Percentage)) +
  geom_bar(stat="identity", fill="#f9dfa9") +
  coord_flip() +
  xlab('') +
  ylab('') +
  ggtitle("Wheat import dependency from Ukraine",
          subtitle="Percentage of total imports") +
  theme(legend.position = 'none',
        plot.title = element_text(hjust=0.5, face="bold"),
        plot.subtitle = element_text(hjust=0.5, face="italic"),
        panel.background = element_blank(),
        panel.grid.major.x = element_line(color='cornflowerblue'),
        panel.grid.minor.x = element_line(color='cornflowerblue'))
p
```



34) Lo stesso risultato, ma esteso a tutti i paesi, può essere rappresentano su una mappa con il seguente codice, avendo cura di importare le librerie `maps` e `mapproj`.

```
library(maps)
```

```
## Warning: il pacchetto 'maps' è stato creato con R versione 4.1.3
```

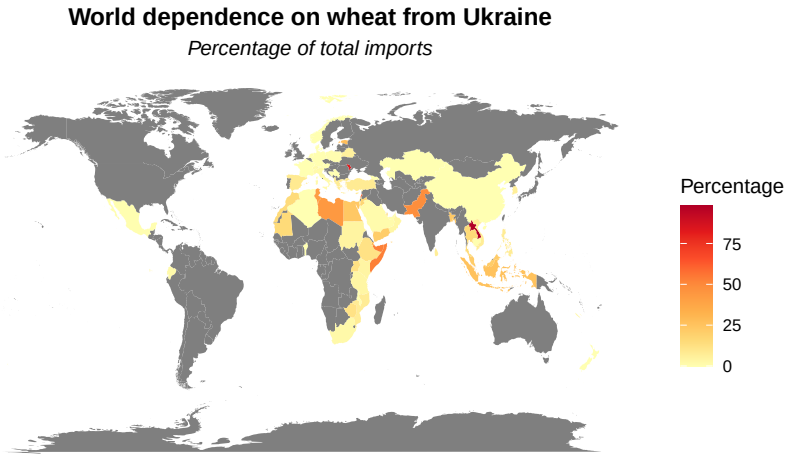
```
##
## Caricamento pacchetto: 'maps'
```

```
## Il seguente oggetto è mascherato da 'package:purrr':
##
##     map
```

```
library(mapproj)
```

```
## Warning: il pacchetto 'mapproj' è stato creato con R versione 4.1.3
```

```
world <- map_data("world")
mymap <- right_join(dependencies, world, by=c("Country" = "region"))
p = ggplot(mymap, aes(long, lat, group=group, fill=Percentage)) +
  geom_polygon() + coord_fixed(1.3) + # aspect ratio
  scale_fill_distiller(palette ="YlOrRd", direction=1) +
  ggtitle("World dependence on wheat from Ukraine",
          subtitle="Percentage of total imports") +
  xlab('') +
  ylab('') +
  theme(plot.title = element_text(hjust=0.5, face="bold"),
        plot.subtitle = element_text(hjust=0.5, face="italic"),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        panel.background = element_blank(),
        axis.text.x=element_blank(),
        axis.ticks.x=element_blank(),
        axis.text.y=element_blank(),
        axis.ticks.y=element_blank())

p
```



17 Simulazione d'esame

17.1 Domanda 1: funzione personalizzata e/o algoritmo

Scrivi uno script `R` che definisca una funzione personalizzata chiamata `conta_quanti_sup` che prenda in input un vettore e un numero `k`, e restituisca il conteggio di quante volte i numeri nel vettore sono superiori a `k`.

```
conta_quanti_sup <- function(v, k){
  if(!is.vector(v) | !is.numeric(k)){
    stop("Gli input devono essere, in ordine, un vettore e un numero")
  }
  counter <- 0
  for(elem in v){
    if(elem > k){
      counter <- counter + 1
    }
  }
  return(counter)
}

v <- sample(1:100, 13, replace = TRUE)
k <- sample(1:100, 1)
conta_quanti_sup(v, k)
```

```
## [1] 11
```

17.2 Domanda 2: analisi di dataframe

Utilizzando il dataset `iris` predefinito in `R`, crea un dataframe chiamato `dati_iris` e completa le seguenti operazioni:

a) Filtra le righe in cui la variabile "Species" è uguale a "setosa".

```
dati_iris <- iris
dati_iris_filtered <- dati_iris[dati_iris$Species == "setosa", ]
table(dati_iris_filtered$Species)
```

```
##
##     setosa versicolor  virginica
##         50          0          0
```

b) Calcola la media della variabile "Sepal.Length" per ciascuna specie.

```
media <- setNames(rep(0, 3), c("setosa", "versicolor", "virginica"))
for(species in unique(dati_iris$Species)){
  data <- dati_iris[dati_iris$Species == species, ]
  media[species] <- mean(data$Sepal.Length)
}
media
```

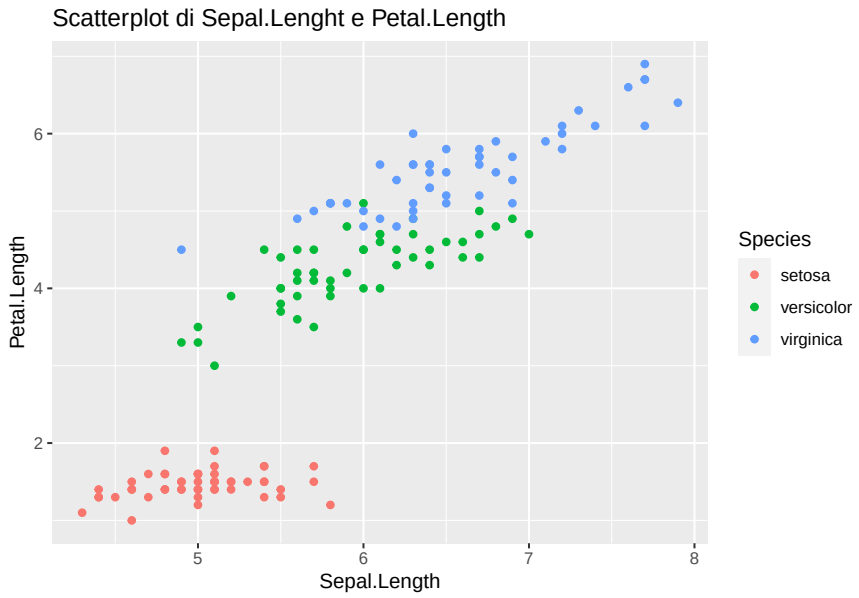
```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

```
#Alternativa con dplyr
library(dplyr)
dati_iris %>%
  group_by(Species) %>%
  summarise(media_Sepal.Length = mean(Sepal.Length))
```

```
## # A tibble: 3 x 2
##   Species    media_Sepal.Length
##   <fct>                   <dbl>
## 1 setosa                   5.01
## 2 versicolor               5.94
## 3 virginica                6.59
```

c) Visualizza un grafico a dispersione (scatterplot) con "Sepal.Length" sull'asse x e "Petal.Length" sull'asse y per tutte le specie.

```
library(ggplot2)
ggplot(dati_iris, aes(Sepal.Length, Petal.Length, col = Species)) +
  geom_point() +
  ggtitle("Scatterplot di Sepal.Lenght e Petal.Length")
```



17.3 Domanda 3: analisi di dataframe

Supponi di avere un dataset chiamato `dati_clienti` contenente le seguenti variabili:

– `ID_Cliente`: Identificativo univoco del cliente – `Sesso`: Genere del cliente (Maschio/Femmina) – `Età`: Età del cliente – `Spesa`: Importo speso dal cliente – `Soddisfazione`: Livello di soddisfazione del cliente (da 1 a 5)

```
ID_Cliente <- sample(1:100, 50, replace = FALSE)
Sesso <- sample(c("M", "F"), 50, replace = TRUE)
Età <- sample(20:80, 50, replace = TRUE)
Spesa <- runif(50, 1000, 2000)
Soddisfazione <- sample(c(1:5), 50, replace = TRUE)
dati_clienti <- data.frame(
  ID_Cliente = ID_Cliente,
  Sesso = Sesso,
  Età = Età,
  Spesa = Spesa,
  Soddisfazione = Soddisfazione
)
head(dati_clienti)
```

```
##   ID_Cliente Sesso Età    Spesa Soddisfazione
## 1         89     M  27 1395.893             1
## 2         32     F  63 1064.928             1
## 3         25     F  51 1225.886             5
## 4         87     M  77 1054.629             5
## 5         35     M  55 1670.282             2
## 6         40     F  64 1297.742             2
```

Utilizzando il pacchetto `dplyr` in `R`, completa le seguenti operazioni:

a) Filtra i dati per includere solo i clienti di genere femminile.

```
table(dati_clienti$Sesso)
```

```
##
##  F  M
## 28 22
```

```
dati_clienti_f <- dati_clienti %>%
  filter(Sesso == "F")
nrow(dati_clienti_f)
```

```
## [1] 28
```

b) Calcola la media della spesa dei clienti femminili.

```
dati_clienti_f %>%
  summarise(media_spesa = mean(Spesa))
```

```
##   media_spesa
## 1    1484.208
```

c) Raggruppa i dati per età e calcola la media della soddisfazione per ogni fascia d'età.

```
dati_clienti %>%
  group_by(Età) %>%
  summarise(media_soddisfazione = mean(Soddisfazione))
```

```
## # A tibble: 29 x 2
##      Età media_soddisfazione
##    <int>               <dbl>
##  1    26                2.33
##  2    27                1
##  3    28                2
##  4    29                3.33
##  5    30                2
##  6    32                5
##  7    33                1
##  8    35                5
##  9    37                1
## 10    42                4
## # i 19 more rows
```