Esercitazione su algoritmi ricorsivi

1. lunghezza di una stringa
2. contatore dei numeri pari in un array
3. ricerca sequenziale
4. uguaglianza di due stringhe
5. string matching (numero di occorrenze)
6. ordinamento per inserimento

lunghezza di una stringa

```
int Strlen_ric(char* str)
```

caso base:

stringa vuota, cioè `*str == '\0'`

soluzione `0`

caso NON base:

soluzione ricorsiva

la lunghezza di una stringa da 0 a \0 è

1+ lunghezza della sottostringa da 1 a \0

cioè `1 + Strlen_ric(str + 1)`

lunghezza di una stringa

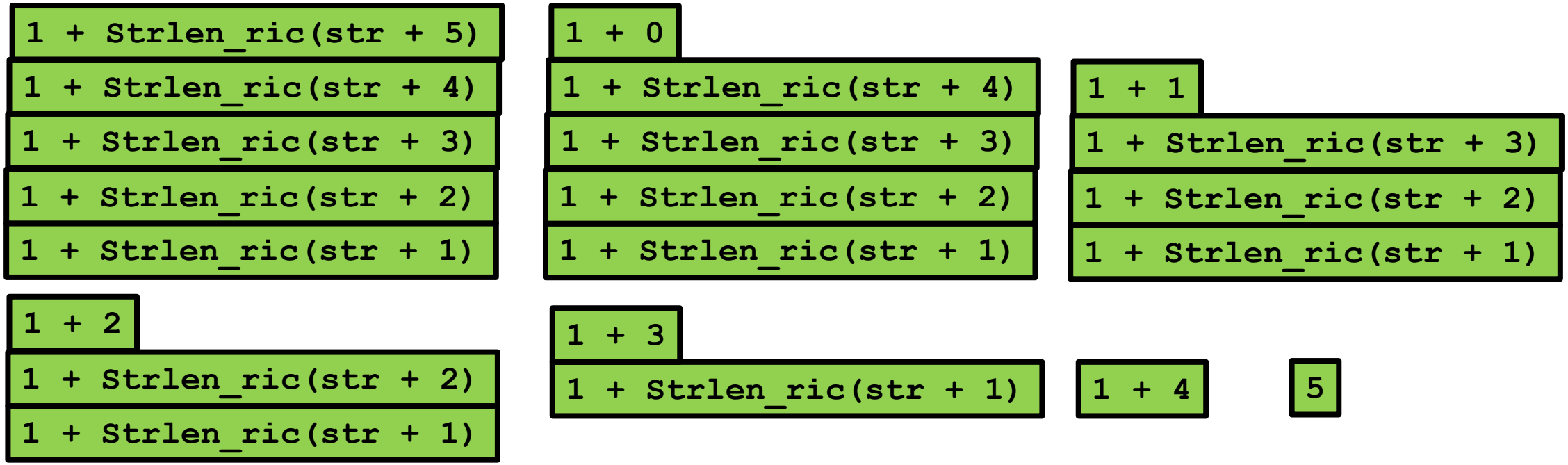
```
int Strlen_ric(char* str)
```

```
int Strlen_ric(char* str)
{
 if (*str == '\0')
        return 0;
    else
        return 1 + Strlen_ric(str + 1);
}
```

lunghezza di una stringa

```
int Strlen_ric(char* str)
{
 if (*str == '\0')
        return 0;
    else
        return 1 + Strlen_ric(str + 1);
}
```

Esercizio: costruire lo stack dei processi per la stringa in input ``pippo''



contatore dei numeri pari in un array

```
int Num_pari_ric(int a[], int n)
```

caso base:
(porzione) array **a** vuoto, cioè **n == 0**
soluzione **0**

caso NON base:
soluzione ricorsiva: il numero di componenti pari della porzione da 0 e di size **n** è
se **a[0]** è pari: 1+ numero di component pari della porzione da 1 e size **n-1**
cioè **1 + Num_pari_ric (a+1,n-1);**
altrimenti: numero di componenti pari della porzione da 1 e size **n-1**
cioè **Num_pari_ric (a+1,n-1);**

contatore dei numeri pari in un array

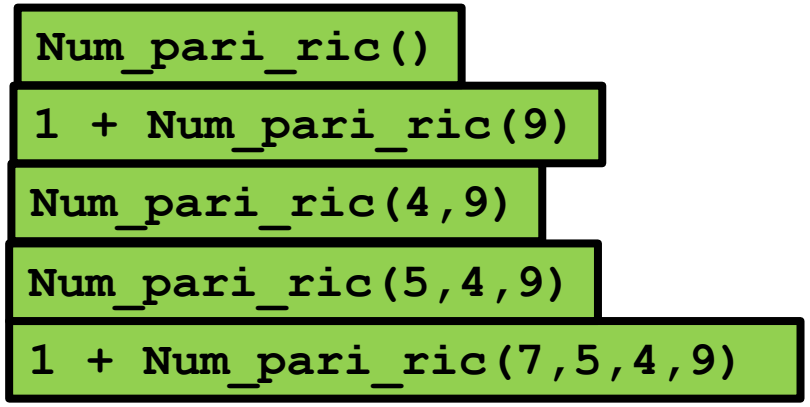
```
int Num_pari_ric(int a[], int n)
{
 if (n == 0)
        return 0;
    else
        if (e_pari(a[0]))
           return 1 + Num_pari_ric (a+1,n-1);
          else
           return Num_pari_ric (a+1,n-1);
}
```

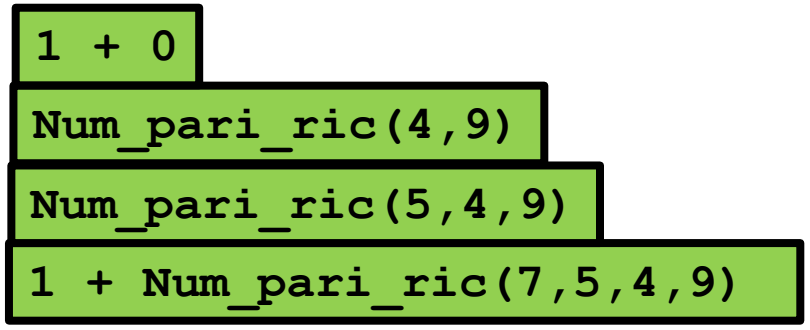
```
logical e_pari(int x) {
      return !(x%2);
}
```

contatore dei numeri pari in un array

```
int Num_pari_ric(int a[], int n){
 if (n == 0)
        return 0;
    else
        if (e_pari(a[0]))
           return 1 + Num_pari_ric (a+1,n-1);
          else
           return Num_pari_ric (a+1,n-1);
}
```

Esercizio: costruire lo stack dei processi per l'array in input `{12,7,5,4,9}`





ricerca sequenziale

```
int appartiene_ric(int A[],int n,int chiave)
```

caso base:
(porzione) array `A` vuoto, cioè `n == 0`
soluzione `0`
se `A[n-1]` è uguale alla `chiave`:
soluzione `1`

caso NON base: (`A[n-1]` è diversa dalla `chiave`)
soluzione ricorsiva:
appartenenza della `chiave` alla porzione di inizio `0` e size `n-1`
cioè `appartiene_ric(A,n-1,chiave);`

ricerca sequenziale

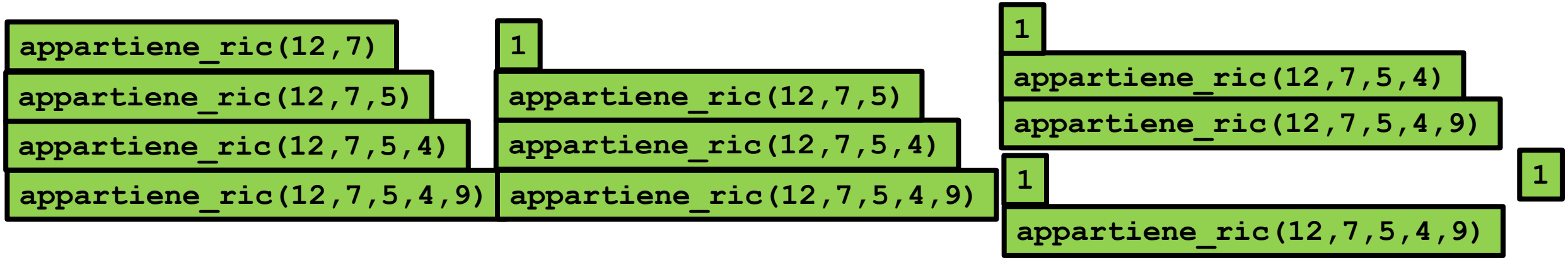
```
int appartiene_ric(int A[],int n,int chiave)
```

```
int appartiene_ric(int A[],int n,int chiave){
    if (n == 0)
        return 0;
    else
    {
        if(A[n-1]== chiave)
            return 1;
          else
            return appartiene_ric(A,n-1,chiave);
     }
}
```

ricerca sequenziale

```
int appartiene_ric(int A[],int n,int chiave){
    if (n == 0)
        return 0;
    else
    {
        if(A[n-1]== chiave)
            return 1;
         else
            return appartiene_ric(A,n-1,chiave);
     }
}
```

Esercizio: costruire lo stack dei processi per `A={12,7,5,4,9}, chiave=7`



uguaglianza di due stringhe

```
int uguaglianza_str_ric(char * str1, char * str2)
```

caso base:

stringhe vuote, cioè `*str1 == '\0' && *str2 == '\0'`

soluzione `1`

una stringa vuota e l'altra no

soluzione `0`

il primo carattere delle due stringhe è diverso

soluzione `0`

caso NON base: (il primo carattere delle due stringhe è uguale)

soluzione ricorsiva

determina l'uguaglianza della sottostringa di `str1` da 1 a \0 e della sottostringa di `str2` da 1 a \0

cioè `uguaglianza_str_ric(str1+ 1, str2+ 1);`

uguaglianza di due stringhe

```
int uguaglianza_str_ric(char * str1, char * str2)
```

```
int uguaglianza_str_ric(char * str1, char * str2) {
if ((*str1 == '\0' && *str2 != '\0')|| (*str1 != '\0'
    && *str2 == '\0'))
        return 0;
if (*str1 == '\0' && *str2 == '\0')
   return 1;
  else
    if (*str1 == *str2)
        return uguaglianza_str_ric(str1+ 1, str2+ 1);
      else
        return 0;
}
```

uguaglianza di due stringhe

```
int uguaglianza_str_ric(char * str1, char * str2)
```

```
int uguaglianza_str_ric(char * str1, char * str2) {
if ((*str1 == '\0' && *str2 != '\0')|| (*str1 != '\0'
    && *str2 == '\0'))
        return 0;
if (*str1 == '\0' && *str2 == '\0')
        return 1;
  else
    if (*str1 == *str2)
        return uguaglianza_str_ric(str1+ 1, str2+ 1);
      else
        return 0;
}
```

Esercizio: costruire lo stack dei processi per le seguenti stringhe in input:
``proffvw'', ``proftvw''

Esercizio: costruire lo stack dei processi per le seguenti stringhe in input:
``proffvw'', ``proffvw''

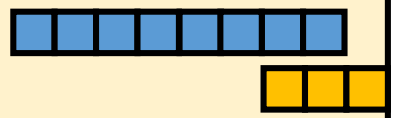
string matching

```
int string_matching_ric(char *testo, char *chiave)
```

caso base:

dopo l'ultimo allineamento a destra `testo - chiave`, cioè la `chiave` non è più allineata con una sottostriga di `testo` della stessa lunghezza

cioè `*(testo+strlen(chiave)-1) == '\0'`



soluzione `0`

caso NON base:

soluzione ricorsiva

determina l'uguaglianza della sottostringa di `testo` da 0 a lunghezza di `chiave` con la `chiave`, usando `strncmp`

se sono uguali: la soluzione è 1 + il numero di occorrenze di `chiave` nelle successive sottostringhe di `testo`

cioè `1 + string_matching_ric(testo+1,chiave);`

altrimenti: la soluzione è il numero di occorrenze di `chiave` nelle successive sottostringhe di `testo`

cioè `string_matching_ric(testo+1,chiave);`

string matching

```
int string_matching_ric(char *testo, char *chiave)
```

```
int string_matching_ric(char *testo, char *chiave) {
 int ris ;
 if (*(testo+strlen(chiave)-1) == '\0')
     return 0;
   else {
     ris = strncmp(testo,chiave,strlen(chiave));
     if (ris == 0)
        return 1 + string_matching_ric(testo+1,chiave);
      else
        return string_matching_ric(testo+1,chiave);
   }
}
```

string matching

```
int string_matching_ric(char *testo, char *chiave) {
 int ris ;
 if (*(testo+strlen(chiave)-1) == '\0')
     return 0;
   else {
     ris = strncmp(testo,chiave,strlen(chiave));
     if (ris == 0)
        return 1 + string_matching_ric(testo+1,chiave);
      else
        return string_matching_ric(testo+1,chiave);
   }
}
```

Esercizio: costruire lo stack dei processi per le seguenti stringhe in input:

``proffvwffvfvwffvkprffv'', ``ffv''

ordinamento per inserimento

```
void ord_ins_ric(int A[], int n)
```

caso base:
(porzione) array **A** di size 1, cioè `n == 1`
soluzione `nessuna azione`

caso NON base: (porzione di **A** di size > 1)
soluzione ricorsiva:
ordina la porziona di **A** di inizio 0 e size **n-1**
cioè `ord_ins_ric(A, n - 1)`

elimina solo il `for` esterno;
dopo che lo stack è stato riempito, si riattiva il processo che agisce sulla porzione di inizio 0 e size 2 che esegue il solito ciclo `while` "a sinistra" per inserire `el_da_ins`
e poi si riattiva il processo che agisce sulla porzione di inizio 0 e size 3 che esegue il solito ciclo `while` "a sinistra" per inserire `el_da_ins`
e così via

ordinamento per inserimento

```
void ord_ins_ric(int A[], int n)
```

```
void ord_ins_ric(int A[], int n) {
 int el_da_ins, j;
 if (n <= 1)
     return;
 ord_ins_ric(A, n - 1);
 el_da_ins = A[n - 1];
 j = n - 2;
 while (j >= 0 && el_da_ins < A[j])
    { A[j + 1] = A[j];
      j--; }
 A[j + 1] = el_da_ins;
}
```

ordinamento per inserimento

```
void ord_ins_ric(int A[], int n) {
 int el_da_ins, j;
 if (n <= 1)
    return;
 ord_ins_ric(A, n - 1);
 el_da_ins = A[n - 1];
 j = n - 2;
 while (j >= 0 && el_da_ins < A[j])
   { A[j + 1] = A[j];
     j--; }
 A[j + 1] = el_da_ins;
}
```

Esercizio: costruire lo stack dei processi per il seguente array in input:
{11,7,9,12,3,10}