

Basandomi sugli esercizi contenuti nel PDF (che coprono settembre 2025, luglio 2025, giugno 2025, gennaio 2025, ecc.), ho elaborato uno **schema universale** per risolvere questa tipologia di problemi in F#.

Gli esercizi seguono quasi sempre lo stesso pattern:

1. **Parte (a):** Definizione ricorsiva classica (non tail-recursive).
2. **Parte (b):** Definizione tail-recursive (ricorsione di coda) usando un accumulatore.
3. **Vincoli:** Vietato usare funzioni di libreria (come List.map, List.filter), eccetto List.rev per la parte (b).

Ecco lo schema passo-passo.

1. Analisi preliminare

Prima di scrivere codice, chiediti:

- **Input:** È una lista ($x::xs$), due liste ($I1, I2$), o un numero (n)?
- **Output:** È una lista modificata, un valore singolo, o una coppia di liste?
- **Logica:** Devo trasformare (Map), filtrare (Filter), o combinare (Merge/Zip)?

2. Schema per la Parte (a): Ricorsione Classica

In questa modalità, l'operazione avviene **dopo** che la chiamata ricorsiva è tornata. La lista viene costruita "ritornando indietro".

Il Template Mentale:

```
let rec nomeFunzione parametri =
    match parametri with
    | CasoBase -> ValoreVuoto (es. [], 0, o ([] , []))
    | Testa :: Coda ->
        // 1. Calcola il risultato sul resto della lista
        let resto = nomeFunzione Coda
        // 2. Elabora la Testa corrente
        let nuovoElemento = ... logica su Testa ...
        // 3. Unisci il risultato (cons)
        nuovoElemento :: resto
```

Esempio (tratto da filterMap): Se $f x$ è Some y , aggiungo y alla lista risultato.

```
let rec filterMap f lista =
    match lista with
    | [] -> []
    | hd :: tl ->
        match f hd with
        | None -> filterMap f tl      // Scarta e continua
        | Some y -> y :: filterMap f tl // Aggiungi e continua (operazione :: dopo la chiamata)
```

3. Schema per la Parte (b): Ricorsione di Coda (Accumulatoro)

Qui l'operazione avviene **prima** della chiamata ricorsiva. Il risultato parziale viene passato in avanti tramite un parametro aggiuntivo (acc). **Attenzione:** Costruendo la lista in avanti (aggiungendo in testa), il risultato finale sarà al contrario. Devi usare List.rev alla fine.

Il Template Mentale:

```
let nomeFunzioneMain parametri =
    // Definizione della funzione interna (loop)
    let rec loop acc resto =
        match resto with
        | CasoBase -> List.rev acc // IMPORTANTE: Rovescia alla fine
        | Testa :: Coda ->
            // 1. Elabora la Testa
            let nuovoElemento = ... logica su Testa ...
            // 2. Aggiorna l'accumulatore
            let nuovoAcc = nuovoElemento :: acc
            // 3. Chiama la ricorsione passando il nuovo accumulatore
            loop nuovoAcc Coda
```

```
// Chiamata iniziale con accumulatore vuoto
loop [] parametri
```

Esempio (tratto da accFilterMap):

```
let accFilterMap f lista =
  let rec loop acc l =
    match l with
    | [] -> List.rev acc // Rovescia l'accumulatore
    | hd :: tl ->
      match f hd with
      | None -> loop acc tl           // Passa l'acc invariato
      | Some y -> loop (y :: acc) tl // Aggiungi a acc e passa avanti
  loop [] lista
```

4. Gestione dei Pattern Specifici (Cheat Sheet)

Nel PDF ci sono tre varianti principali di logica. Ecco come adattare lo schema:

A. Generazione numerica (es. gen)

Invece di match list, usi if $i < 0$.

- **Parte (a):** $f i :: gen (i-1)$
- **Parte (b):** $loop (f i :: acc) (i-1)$

B. Due Liste in Input (es. merge, vectAdd)

Il match deve guardare entrambe le liste.

- **Pattern:**

```
match l1, l2 with
| [], [] -> ...
| h1::t1, h2::t2 -> ...
| ... -> ... (gestione lunghezze diverse se serve)
```

- **Parte (b):** L'accumulatore raccoglie il risultato combinato.

C. Output Multiplo / Tuple (es. split, partition, unzip)

Qui l'accumulatore deve essere una tupla di liste.

- **Parte (a):**

```
let (l1, l2) = funzione tl // Ottieni le liste dal resto
(h :: l1, l2) // Aggiungi h alla lista giusta
```

- **Parte (b):**

```
let rec loop acc1 acc2 lista = ...
// Alla fine:
(List.rev acc1, List.rev acc2)
```

5. Checklist per l'Esame

1. **Niente List.map / List.fold:** Usa sempre match ... with.
2. **Parte (b) = List.rev:** Se usi un accumulatore per costruire una lista, al 99% devi rovesciarla nel caso base.
3. **Eccezioni:** Se le liste devono avere la stessa lunghezza (es. vectAdd), ricorda il caso $| _ [span_9](start_span), _ -> failwith "Errore"$.
4. **Tipo Option:** Se c'è Some/None (es. filterMap), usa un match interno o un if/else per estrarre il valore.

Vuoi provare ad applicare questo schema a uno degli esercizi specifici del PDF (es. swap o partition) per vedere se il risultato ti torna?