Riferimento Rapido per Esami

INDICE

---

1. RICONOSCIMENTO PATTERN {#riconoscimento}

Tabella Decisionale Completa

Vedi nel testo...	Pattern da usare	Complessità
"coppia (i,j)" + **ordinato**	TWO POINTERS	O(n)
"trova indice" + **ordinato** + **singolo elemento**	DIVIDE-ET-IMPERA	O(log n)
"gap" / "picco" / "centro" + ordinato	DIVIDE-ET-IMPERA	O(log n)
"albero/heap" + **campo custom** (sum, size, leaves, even)	BST ARRICCHITO	O(h)
"albero" + operazione custom **senza campo extra**	RICORSIONE STRUTTURA	O(h) o O(n)
"ricorrenza c(i,j)" / "memoizzazione" / "bottom-up"	PROGRAMMAZIONE DINAMICA	O(n²) tipico
"algoritmo greedy" / "scelta ottima" / "scheduling"	GREEDY	O(n log n) tipico
"MaxHeapify" / "BuildHeap" / "HeapSort" menzionati	FUNZIONI NOTEVOLI HEAP	O(log n) / O(n) / O(n log n)
"Insert" / "Delete" BST standard	FUNZIONI NOTEVOLI BST	O(h)

Vedi nel testo...	Pattern da usare	Complessità
"Partition" / "QuickSort"	FUNZIONI NOTEVOLI PARTITION	O(n) / O(n log n)
"anagramma" / "conta" + alfabeto piccolo	CONTATORI	O(n)
"verifica proprietà" + scansione	CICLO LINEARE	O(n)

Keyword Specifiche che Identificano

TWO POINTERS:

- "coppia di indici i, j"
- "A[i] e A[j] tali che"
- Array ordinato + relazione matematica (A[j]=2*A[i], A[i]-A[j]=k)

DIVIDE-ET-IMPERA:

- "divide et impera" esplicito
- "ricorsiva"
- Array ordinato + cerca valore/indice singolo
- Proprietà che esclude metà array

BST ARRICCHITO:

- "campo x.sum", "campo x.size", "campo x.leaves", "campo x.even"
- "arricchimento albero"
- "mantieni informazione su sottoalbero"
- **"realizza Insert(T,z)"** quando c'è campo custom

RICORSIONE ALBERI (senza campi extra):

- "dato albero/heap"
- "calcola su sottoalbero" (max, somma, conta nodi)
- "stampa nodi con proprietà"
- **NO menzion campo extra da mantenere**

PROGRAMMAZIONE DINAMICA:

- "ricorrenza c(i,j)" o "c[i]"
- "memoizzazione"
- "bottom-up"
- "massimo/minimo" + sottoproblemi sovrapposti
- Parole chiave: LCS, taglio aste, zaino, cammino, scheduling DP

GREEDY:

- "algoritmo greedy"
- "scelta ottima ad ogni passo"
- "mostrare proprietà scelta greedy"
- Problemi: scheduling, selezione attività, stop, turni

FUNZIONI NOTEVOLI:

- Nomi espliciti: "MaxHeapify", "BuildMaxHeap", "HeapSort"
- "Insert(T,z)" BST standard (senza campi custom)
- "Delete(T,z)" BST
- "Partition", "QuickSort", "MergeSort"
- "Left-Rotate", "Right-Rotate"

---

2. TWO POINTERS {#two-pointers}

Pattern: Coppia (i,j) con Relazione

Idea: Due indici scorrono array ordinato, confronto relazione, avanzo i o j.

Esempio 1: Double(A,n)

Testo: Array ordinato, trova (i,j): A[j]=2*A[i], ritorna (0,0) se non esiste.

Codice:

```
Double(A, n)
1.  i = 1
2.  j = 1
3.  while (i ≤ n) and (j ≤ n) and (2*A[i] ≠ A[j])
4.      if 2*A[i] > A[j]              // j troppo piccolo
5.          j = j + 1
6.      else                           // i troppo piccolo
7.          i = i + 1
8.  if (i ≤ n) and (j ≤ n)
9.      return (i, j)
10. else
11.     return (0, 0)
```

Invariante: Tutte le coppie (i',j') con i'<i o j'<j non soddisfano 2*A[i']=A[j'].

Esempio 2: Diff(A,n,k) - Decrescente

Testo: Array ordinato decrescente, trova (i,j): A[i]-A[j]=k.

Codice:

```
Diff(A, n, k)
1.  i = 1
2.  j = 1
3.  while (i ≤ n) and (j ≤ n) and (A[i] − A[j] ≠ k)
4.      if A[i] − A[j] < k            // j troppo avanti (A[j] troppo
grande)
5.          j = j + 1
6.      else                           // i troppo indietro
7.          i = i + 1
8.  if (i ≤ n) and (j ≤ n)
9.      return (i, j)
10. else
11.     return (0, 0)
```

Esempio 3: min(A,B) - Confronti Solo Tra Array

Testo: A e B permutazioni, trova minimo confrontando SOLO A[i] con B[j].

Codice:

```
min(A, B, n)
1.  i = 1
2.  j = 1
3.  while j ≤ n
4.      if A[i] ≤ B[j]                // B[j] non è minimo
5.          j = j + 1                 // scarto B[j]
6.      else                           // A[i] non è minimo
7.          i = i + 1                 // scarto A[i]
8.  return A[i]                        // A[i] ≤ tutti in B
```

Nota: Non serve controllare i≤n (garantito da invariante).

Template Two Pointers

```
Funzione(A, n, k)
1.  i = 1
2.  j = 1
3.  while (i ≤ n) and (j ≤ n) and (<relazione non soddisfatta>)
4.      if <confronto dice "avanzo j">
```

```
5.          j = j + 1
6.      else
7.          i = i + 1
8.  if <trovato>
9.      return (i, j)
10. else
11.     return (0, 0)
```

Riempi:

- Riga 3: Relazione da verificare (es. A[i]-A[j]≠k)
- Riga 4: Quando incrementare j? Quando incrementare i?

---

3. DIVIDE-ET-IMPERA {#divide-impera}

Pattern: Ricerca con Esclusione Metà

Idea: Divido a metà, confronto A[q], scelgo metà con soluzione, ricorro.

Esempio 1: Gap(A,p,r)

Testo: A[r]-A[p]≥r-p, trova i: A[i+1]-A[i]>1 (gap).

Codice:

```
Gap(A, p, r)
1.  if p = r - 1
2.      return p                        // base: 2 elementi, p è gap
3.  q = ⌊(p + r)/2⌋
4.  if A[q] - A[p] > q - p             // gap in [p,q]
5.      return Gap(A, p, q)
6.  else                                // gap in [q,r]
7.      return Gap(A, q, r)
```

Ricorrenza: T(n) = T(n/2) + O(1) → O(log n)

Esempio 2: Fix(A)

Testo: Array ordinato distinto, trova i: A[i]=i, o 0 se non esiste.

Codice:

```
Fix(A)
1.  return FixRec(A, 1, A.length)
```

```
FixRec(A, p, r)
1.  if p > r
2.      return 0
3.  q = ⌊(p + r)/2⌋
4.  if A[q] = q
5.      return q
6.  else if A[q] < q                    // cerco a destra
7.      return FixRec(A, q+1, r)
8.  else                                 // A[q] > q, cerco a sinistra
9.      return FixRec(A, p, q−1)
```

Perché funziona: Se A[q]>q → ∀j>q: A[j]>j (array ordinato). Analogamente per <.

Esempio 3: Over(A,p,r,x)

Testo: Array ordinato crescente, trova min i: A[i]>x, o r+1.

Codice:

```
Over(A, p, r, x)
1.  if p > r
2.      return r + 1
3.  q = ⌊(p + r)/2⌋
4.  if A[q] ≤ x                          // elemento cercato a destra
5.      return Over(A, q+1, r, x)
6.  else                                 // A[q] > x, ma potrebbe esserci più
piccolo a sx
7.      return Over(A, p, q−1, x)
```

Esempio 4: centre(A) - Array Semi-Ordinato

Testo: A semi-ordinato: ∃k: A[k+1..n]A[1..k] ordinato, A[n]<A[1]. Trova k.

Codice:

```
centre(A)
1.  return centreRec(A, 1, A.length)

centreRec(A, p, r)
1.  if r = p + 1
2.      return p
3.  q = ⌊(p + r)/2⌋
4.  if A[q] < A[p]                       // centro in [p,q]
```

```
5.      return centreRec(A, p, q)
6.  else                                // centro in [q,r]
7.      return centreRec(A, q, r)
```

Template D&C Ricerca (Una Ricorsione)

```
Func(A, p, r)
1.  if <caso base>
2.      return <soluzione diretta>
3.  q = ⌊(p + r)/2⌋
4.  if <condizione indica soluzione in [p,q]>
5.      return Func(A, p, q)
6.  else
7.      return Func(A, q+1, r)
```

Riempi:

- Riga 1: Quando non divido più? (p>r? p=r? p=r-1?)
- Riga 4: Come decido quale metà contiene soluzione?

Template D&C Calcolo (Due Ricorsioni)

```
Func(A, p, r)
1.  if p = r
2.      return A[p]
3.  q = ⌊(p + r)/2⌋
4.  sol1 = Func(A, p, q)
5.  sol2 = Func(A, q+1, r)
6.  return <combina sol1 e sol2>
```

Esempio uso: Max(A,p,r) → return max(sol1, sol2)

---

4. RICORSIONE ALBERI {#ricorsione-alberi}

Pattern: Operazioni Custom su BST/Heap

Idea: Caso base (nil/foglia), ricorsione su left/right, combina.

Esempio 1: MaxPath(T)

Testo: Albero binario, calcola massimo costo cammino radice-foglia (costo=somma chiavi).

Codice:

```
MaxPath(T)
1.  if T = nil
2.      return 0
3.  if T.left = nil and T.right = nil  // foglia
4.      return T.key
5.  maxLeft = MaxPath(T.left)
6.  maxRight = MaxPath(T.right)
7.  return T.key + max(maxLeft, maxRight)
```

Esempio 2: Insert BST con Campo x.even

Testo: BST con x.even = true se somma sottoalbero radicato in x è pari. Realizza Insert(T,z).

Codice:

```
Insert(T, z)
1.  x = T.root
2.  y = nil
3.  z.even = (z.key mod 2 = 0)         // inizializza campo z
4.  while x ≠ nil
5.      x.even = (x.even = z.even)     // aggiorna: somma cambia parità
6.      y = x
7.      if z.key < x.key
8.          x = x.left
9.      else
10.         x = x.right
11. z.p = y
12. if y = nil
13.     T.root = z
14. else if z.key < y.key
15.     y.left = z
16. else
17.     y.right = z
```

Idea chiave: Aggiorna campo custom durante la discesa (while), non dopo.

Esempio 3: IsMaxHeap(A,n) Iterativo

Testo: Verifica se A[1..n] è max-heap.

Codice:

```
IsMaxHeap(A, n)
1.  for i = 1 to ⌊n/2⌋                  // nodi interni
2.      if (2i ≤ n) and (A[i] < A[2i])
3.          return false
4.      if (2i+1 ≤ n) and (A[i] < A[2i+1])
5.          return false
6.  return true
```

Esempio 4: subseq(X,Y,m,n)

Testo: Verifica se X[1..m] è sottosequenza di Y[1..n].

Codice:

```
subseq(X, Y, m, n)
1.  if m = 0
2.      return true                     // X vuoto, sempre sottosequenza
3.  if m > n
4.      return false                    // X più lungo di Y
5.  if X[m] = Y[n]                      // match ultimo elemento
6.      return subseq(X, Y, m-1, n-1)
7.  else                                // X[m] non match, scarto Y[n]
8.      return subseq(X, Y, m, n-1)
```

Template Ricorsione Albero

```
Func(x)
1.  if x = nil
2.      return <valore base>
3.  solLeft = Func(x.left)
4.  solRight = Func(x.right)
5.  return <combina solLeft, solRight, x.key>
```

Riempi:

- Riga 2: Valore per sottoalbero vuoto (0? -∞? nil?)
- Riga 5: Come combino risultati con x.key? (somma? max? concatena?)

---

5. BST ARRICCHITO {#bst-arricchito}

Pattern: Campi Extra da Mantenere

Idea: Ogni nodo x ha campo custom (x.sum, x.size, x.leaves, x.even). Bisogna aggiornarlo durante Insert/Delete.

Esempio 1: avg(x) - Media Sottoalbero

Testo: BST con x.sum (somma chiavi sottoalbero) e x.size (numero nodi). Realizza avg(x) e Insert(T,z).

avg(x) - Triviale:

```
avg(x)
1.  if x ≠ nil
2.      return x.sum / x.size
3.  else
4.      error
```

Insert(T,z) - Aggiorna Durante Discesa:

```
Insert(T, z)
1.  y = nil
2.  x = T.root
3.  while x ≠ nil
4.      y = x
5.      x.size = x.size + 1           // aggiorna: +1 nodo
6.      x.sum = x.sum + z.key         // aggiorna: +z.key alla somma
7.      if z.key < x.key
8.          x = x.left
9.      else
10.         x = x.right
11. z.p = y
12. z.sum = z.key                      // inizializza z
13. z.size = 1
14. if y = nil
15.     T.root = z
16. else if z.key < y.key
17.     y.left = z
18. else
19.     y.right = z
```

Chiave: Aggiorna campi durante il while (righe 5-6) perché stai modificando i sottoalberi attraversati.

Esempio 2: leaves(x) - Conta Foglie

Testo: BST con x.leaves (numero foglie sottoalbero). Realizza leaves(x) e Insert(T,z).

leaves(x) - Triviale:

```
leaves(x)
1.  if x ≠ nil
2.      return x.leaves
3.  else
4.      return 0
```

Insert(T,z) - Aggiorna Foglie:

```
Insert(T, z)
1.  y = nil
2.  x = T.root
3.  while x ≠ nil
4.      y = x
5.      x.leaves = x.leaves + 1       // +1 foglia (z sarà foglia)
6.      if z.key < x.key
7.          x = x.left
8.      else
9.          x = x.right
10. z.p = y
11. z.leaves = 1                       // z è foglia
12. if y = nil
13.     T.root = z
14. else
15.     if y.left = nil and y.right = nil  // y era foglia
16.         y.leaves = 1                    // ora y ha 1 foglia (z)
17.     if z.key < y.key
18.         y.left = z
19.     else
20.         y.right = z
```

Nota: Quando inserisco z come figlio di y, se y era foglia diventa nodo interno → y.leaves cambia da 1 a 1 (mantiene solo z come foglia).

Esempio 3: range(T,k1,k2) - Stampa Chiavi in Intervallo

Testo: BST, stampa in ordine crescente tutte le chiavi k: k1 ≤ k ≤ k2.

Codice:

```
range(T, k1, k2)
1.  rangeRec(T.root, k1, k2)
```

```
rangeRec(x, k1, k2)
1.  if x ≠ nil
2.      if x.key ≥ k1                   // possibili chiavi in left
3.          rangeRec(x.left, k1, k2)
4.      if x.key ≥ k1 and x.key ≤ k2  // x è nell'intervallo
5.          print x.key
6.      if x.key ≤ k2                   // possibili chiavi in right
7.          rangeRec(x.right, k1, k2)
```

Complessità: O(h + m) dove m = numero chiavi stampate.

Esempio 4: conta(T,a,b) - Conta Nodi in Intervallo

Testo: BST, conta nodi con chiave in [a,b].

Codice:

```
conta(T, a, b)
1.  return contaRec(T.root, a, b)

contaRec(x, a, b)
1.  if x = nil
2.      return 0
3.  if b < x.key                        // solo left può contenere
4.      return contaRec(x.left, a, b)
5.  else if x.key < a                   // solo right può contenere
6.      return contaRec(x.right, a, b)
7.  else                                 // x in [a,b]
8.      return 1 + contaRec(x.left, a, b) + contaRec(x.right, a, b)
```

Template BST Arricchito Insert

Schema generale:

```
Insert(T, z)
1.  y = nil
2.  x = T.root
3.  <inizializza campo z>
4.  while x ≠ nil
5.      y = x
6.      <aggiorna campo x usando z>    // CHIAVE: aggiorna qui!
7.      if z.key < x.key: x = x.left
8.      else: x = x.right
```

```
9.  z.p = y
10. <collega z come figlio di y>
```

Riempi:

- Riga 3: Inizializza campo di z (z.sum=z.key, z.size=1, z.leaves=1, z.even=...)
- Riga 6: Come cambia `x.<campo>` quando inserisco z nel suo sottoalbero?

---

6. PROGRAMMAZIONE DINAMICA {#prog-dinamica}

Pattern: Ricorrenza con Sottoproblemi Sovrapposti

Segnali identificativi:

- "Dato ricorrenza c(i,j)"
- "Memoizzazione" o "bottom-up"
- Problemi ottimizzazione (minimo/massimo)
- Sottoproblemi si ripetono

Due approcci:

1. **Top-Down con Memoizzazione**: Ricorsione + tabella memo
2. **Bottom-Up**: Riempi tabella iterativamente

Esempio 1: LCS (Longest Common Subsequence)

Testo: Date stringhe X[1..m] e Y[1..n], trova lunghezza sottosequenza comune più lunga.

Ricorrenza:

```
c[i,j] = | 0                       se i=0 o j=0
         | c[i-1,j-1] + 1           se X[i]=Y[j]
         | max(c[i-1,j], c[i,j-1]) altrimenti
```

Bottom-Up:

```
LCS-Length(X, Y, m, n)
1.  c = new matrix[0..m, 0..n]
2.  for i = 0 to m do c[i,0] = 0
3.  for j = 0 to n do c[0,j] = 0
4.  for i = 1 to m do
5.      for j = 1 to n do
6.          if X[i] = Y[j]
7.              c[i,j] = c[i-1,j-1] + 1
8.          else if c[i-1,j] ≥ c[i,j-1]
```

```
9.              c[i,j] = c[i-1,j]
10.         else
11.             c[i,j] = c[i,j-1]
12. return c[m,n]
```

Complessità: O(m_n) tempo, O(m_n) spazio.

Esempio 2: MinPath su Scacchiera

Testo: Scacchiera n×n, vai da (1,1) a (n,n) muovendoti → o ↑. Costo u(i,j) per ↑, r(i,j) per →. Trova costo minimo.

Ricorrenza:

```
C[i,j] = | 0                                 se i=n e j=n
         | r[i,j] + C[i,j+1]                se i=n e j<n
         | u[i,j] + C[i+1,j]                se i<n e j=n
         | min(r[i,j]+C[i,j+1], u[i,j]+C[i+1,j])  altrimenti
```

Bottom-Up (riempio da (n,n) a (1,1)):

```
MinPath(u, r, n)
1.  C = new matrix[1..n, 1..n]
2.  C[n,n] = 0
3.  for j = n-1 downto 1 do
4.      C[n,j] = r[n,j] + C[n,j+1]       // ultima riga
5.  for i = n-1 downto 1 do
6.      C[i,n] = u[i,n] + C[i+1,n]       // ultima colonna
7.  for i = n-1 downto 1 do
8.      for j = n-1 downto 1 do
9.          C[i,j] = min(r[i,j] + C[i,j+1], u[i,j] + C[i+1,j])
10. return C[1,1]
```

Esempio 3: Top-Down con Memoizzazione

Testo: Stessa ricorrenza MinPath, ma approccio top-down.

Codice:

```
MinPath(u, r, n)
1.  C = new matrix[1..n, 1..n]
2.  for i = 1 to n do
3.      for j = 1 to n do
```

```
4.          C[i,j] = -1                  // non calcolato
5.  return MinPathRec(u, r, n, 1, 1, C)

MinPathRec(u, r, n, i, j, C)
1.  if C[i,j] ≠ -1                       // già calcolato
2.      return C[i,j]
3.  if i = n and j = n
4.      C[i,j] = 0
5.  else if i = n
6.      C[i,j] = r[i,j] + MinPathRec(u, r, n, i, j+1, C)
7.  else if j = n
8.      C[i,j] = u[i,j] + MinPathRec(u, r, n, i+1, j, C)
9.  else
10.     costRight = r[i,j] + MinPathRec(u, r, n, i, j+1, C)
11.     costUp = u[i,j] + MinPathRec(u, r, n, i+1, j, C)
12.     C[i,j] = min(costRight, costUp)
13. return C[i,j]
```

Ricostruzione Soluzione

Idea: Oltre a tabella C per costo ottimo, usa tabella b per decisioni.

Esempio MinPath con ricostruzione:

```
for i = n-1 downto 1 do
    for j = n-1 downto 1 do
        if r[i,j] + C[i,j+1] < u[i,j] + C[i+1,j]
            C[i,j] = r[i,j] + C[i,j+1]
            b[i,j] = "→"                  // salvo decisione
        else
            C[i,j] = u[i,j] + C[i+1,j]
            b[i,j] = "↑"
// Stampa soluzione
i = 1, j = 1
while i ≠ n or j ≠ n
    print b[i,j]
    if b[i,j] = "→": j = j + 1
    else: i = i + 1
```

Template DP Bottom-Up

Per tabelle 1D:

```
DP(input, n)
1.  C[0..n] = ...                          // inizializza casi base
2.  for i = 1 to n do
3.      C[i] = <calcola usando C[0..i-1]>
4.  return C[n]
```

Per tabelle 2D:

```
DP(input, m, n)
1.  C[0..m, 0..n] = ...                    // inizializza bordi
2.  for i = 1 to m do
3.      for j = 1 to n do
4.          C[i,j] = <calcola usando C[i-1,j], C[i,j-1], C[i-1,j-1]>
5.  return C[m,n]
```

Ordine scansione:

- **Row-major**: per righe (i=1..m, poi j=1..n dentro)
- **Column-major**: per colonne (j=1..n, poi i=1..m dentro)
- **Diagonal**: per diagonali (per LCS, taglio aste)
- **Reverse**: al contrario (da (n,n) a (1,1))

---

7. GREEDY {#greedy}

Pattern: Scelta Localmente Ottima

Segnali identificativi:

- "Algoritmo greedy"
- "Scelta ottima ad ogni passo"
- "Mostrare proprietà scelta greedy"
- Problemi scheduling, selezione attività

Due proprietà da dimostrare:

1. **Scelta Greedy**: Esiste soluzione ottima contenente la scelta greedy
2. **Sottostruttura Ottima**: Dopo scelta greedy, problema rimanente ha stessa struttura

Esempio 1: Scheduling Programmi

Testo: n programmi con lunghezza l_j. Ordine esecuzione che minimizza Σ tempi completamento.

Scelta Greedy: Esegui programma più corto prima.

Algoritmo:

```
Schedule(l, n)
1.  ordina l in modo crescente              // O(n log n)
2.  return l                                 // ordine ottimo
```

Dimostrazione Scelta Greedy:

- Sia l_1 il programma più corto
- Supponi soluzione ottima o* non inizi con l_1
- Scambia l_1 con primo programma in o* → costo diminuisce o rimane uguale
- Contraddizione: o* non era ottima, oppure esiste ottima che inizia con l_1

Esempio 2: Stop Ottimali

Testo: Viaggio con distanze d_1,...,d_n tra punti. Autonomia massima d. Minimo numero soste.

Scelta Greedy: Fai sosta appena necessario (quando non puoi proseguire).

Algoritmo:

```
stop(d, n, d_max)
1.  S[1..n-1] = 0                           // array soste
2.  dist = d[1]
3.  for i = 2 to n
4.      if dist + d[i] > d_max
5.          S[i-1] = 1                      // sosta prima di i
6.          dist = d[i]
7.      else
8.          dist = dist + d[i]
9.  return S
```

Complessità: O(n)

Dimostrazione: Ogni sosta è necessaria (non posso arrivare altrimenti). Non faccio soste inutili.

Esempio 3: Turni Postali

Testo: Richieste orari r_1,...,r_n. Turni 1 ora. Minimo numero turni per coprire tutte richieste.

Scelta Greedy: Ordina richieste, primo turno inizia alla prima richiesta, copri tutte fino a r_i + 1h.

Algoritmo:

```
time(r, n)
1.  ordina r in modo crescente
2.  k = 1
3.  t[1] = r[1]                              // primo turno
4.  i = 1
5.  while i ≤ n
6.      while i ≤ n and r[i] < t[k] + 1    // coperto da turno k
7.          i = i + 1
8.      if i ≤ n
9.          k = k + 1
10.         t[k] = r[i]                     // nuovo turno
11. return t[1..k]
```

Template Greedy

Schema generale:

```
Greedy(input)
1.  <ordina/prepara input se necessario>
2.  S = ∅                                    // soluzione
3.  while <non ho finito>
4.      x = <scelta greedy>
5.      if <x è valida>
6.          S = S ∪ {x}
7.  return S
```

Dimostrazione:

1. Enuncio scelta greedy
2. Dimostro proprietà scelta greedy (exchange argument)
3. Dimostro sottostruttura ottima
4. Concludo che greedy dà soluzione ottima

---

8. FUNZIONI NOTEVOLI {#funzioni-notevoli}

Heap (Array Representation)

Proprietà: A[i] ≥ A[2i] e A[i] ≥ A[2i+1] (max-heap)

Parent/Left/Right:

```
Parent(i) = ⌊i/2⌋
Left(i) = 2i
Right(i) = 2i + 1
```

MaxHeapify(A, i, n)

Testo: A[i] può violare proprietà heap, ma Left(i) e Right(i) sono radici di max-heap. Sistema A[i].

Codice:

```
MaxHeapify(A, i, n)
1.  l = 2i
2.  r = 2i + 1
3.  if l ≤ n and A[l] > A[i]
4.      largest = l
5.  else
6.      largest = i
7.  if r ≤ n and A[r] > A[largest]
8.      largest = r
9.  if largest ≠ i
10.     A[i] ⇔ A[largest]
11.     MaxHeapify(A, largest, n)
```

Complessità: O(log n) = O(h)

BuildMaxHeap(A, n)

Testo: Costruisci max-heap da array non ordinato.

Codice:

```
BuildMaxHeap(A, n)
1.  for i = ⌊n/2⌋ downto 1 do
2.      MaxHeapify(A, i, n)
```

Perché parto da ⌊n/2⌋: Nodi da ⌊n/2⌋+1 a n sono foglie (già heap).

Complessità: O(n) - NON O(n log n)!

HeapSort(A, n)

Codice:

```
HeapSort(A, n)
1.  BuildMaxHeap(A, n)
2.  for i = n downto 2 do
3.      A[1] ⇔ A[i]                          // max in posizione i
4.      MaxHeapify(A, 1, i-1)                // ripristina heap su A[1..i-1]
```

Invariante: A[1..i] è max-heap, A[i+1..n] ordinato.

Complessità: O(n log n)

Insert BST Standard

Codice:

```
Insert(T, z)
1.  y = nil
2.  x = T.root
3.  while x ≠ nil
4.      y = x
5.      if z.key < x.key
6.          x = x.left
7.      else
8.          x = x.right
9.  z.p = y
10. if y = nil
11.     T.root = z
12. else if z.key < y.key
13.     y.left = z
14. else
15.     y.right = z
```

Complessità: O(h)

Delete BST Standard

Codice (con Transplant):

```
Transplant(T, u, v)                          // sostituisci sottoalbero u
con v
```

```
1.  if u.p = nil
2.      T.root = v
3.  else if u = u.p.left
4.      u.p.left = v
5.  else
6.      u.p.right = v
7.  if v ≠ nil
8.      v.p = u.p

Delete(T, z)
1.  if z.left = nil                          // caso 0 o 1 figlio sx
2.      Transplant(T, z, z.right)
3.  else if z.right = nil                    // caso 1 figlio dx
4.      Transplant(T, z, z.left)
5.  else                                      // caso 2 figli
6.      y = Minimum(z.right)                 // successor
7.      if y.p ≠ z
8.          Transplant(T, y, y.right)        // togli y dalla posizione
9.          y.right = z.right
10.         y.right.p = y
11.     Transplant(T, z, y)                  // y prende posto di z
12.     y.left = z.left
13.     y.left.p = y
```

Complessità: O(h)

Partition 2-Way (Standard QuickSort)

Codice:

```
Partition(A, p, r)
1.  x = A[p]                                  // pivot = primo elemento
2.  i = p - 1
3.  j = r + 1
4.  while true
5.      repeat j = j - 1 until A[j] ≤ x     // cerca da destra
6.      repeat i = i + 1 until A[i] ≥ x     // cerca da sinistra
7.      if i < j
8.          A[i] ⇔ A[j]
9.      else
10.         return j                          // posizione finale pivot
```

Invariante: A[p..i] ≤ x, A[j..r] ≥ x

Risultato: A[p..j] ≤ pivot, A[j+1..r] > pivot

Partition 3-Way (Per Duplicati)

Codice:

```
Partition3Way(A, p, r)
1.  x = A[p]
2.  i = p
3.  j = p
4.  k = r + 1
5.  while j < k
6.      if A[j] < x
7.          A[i] ↔ A[j]
8.          i = i + 1
9.          j = j + 1
10.     else if A[j] > x
11.         k = k - 1
12.         A[j] ↔ A[k]
13.     else                                  // A[j] = x
14.         j = j + 1
15. return i, j                               // [p..i-1] < x, [i..j-1] = x,
[j..r] > x
```

Quando usare: Array con molti duplicati (evita O(n²) su tutti uguali).

MergeSort

Codice:

```
MergeSort(A, p, r)
1.  if p < r
2.      q = ⌊(p + r)/2⌋
3.      MergeSort(A, p, q)
4.      MergeSort(A, q+1, r)
5.      Merge(A, p, q, r)

Merge(A, p, q, r)                             // unisce A[p..q] e A[q+1..r]
ordinati
1.  n1 = q - p + 1
2.  n2 = r - q
3.  L[1..n1], R[1..n2] = copia A[p..q], A[q+1..r]
4.  i = 1, j = 1
5.  for k = p to r do
6.      if i ≤ n1 and (j > n2 or L[i] ≤ R[j])
7.          A[k] = L[i]
8.          i = i + 1
9.      else
```

```
10.         A[k] = R[j]
11.         j = j + 1
```

Complessità: O(n log n)

Rotazioni Alberi

Left-Rotate(T, x):

```
Left-Rotate(T, x)
1.  y = x.right                                 // y diventa padre di x
2.  x.right = y.left                            // sottoalbero sinistro di y va
a destra di x
3.  if y.left ≠ nil
4.      y.left.p = x
5.  y.p = x.p                                   // collega y al padre di x
6.  if x.p = nil
7.      T.root = y
8.  else if x = x.p.left
9.      x.p.left = y
10. else
11.     x.p.right = y
12. y.left = x                                  // x diventa figlio sinistro di
y
13. x.p = y
```

Right-Rotate: Simmetrica (scambia left ↔ right).

---

9. CONTATORI {#contatori}

Pattern: Array di Frequenze

Idea: Array ausiliario conta occorrenze elementi.

Esempio: Anagramma(x,y)

Testo: Stringhe su {0,1}, verifica se anagrammi.

Codice:

```
Anagramma(x, y)
1.  diff[0..1] = 0
2.  for i = 1 to x.len
3.      diff[x[i]] = diff[x[i]] + 1    // conta in x
```

```
4.      diff[y[i]] = diff[y[i]] - 1    // deconta in y
5.  if (diff[0] = 0) and (x.len = y.len)
6.      return true
7.  else
8.      return false
```

Template Contatori

```
Func(A, n)
1.  count[<range>] = 0
2.  for i = 1 to n
3.      count[A[i]] = count[A[i]] + 1  // popola contatori
4.  // analizza count
5.  if <condizione su count>
6.      return true
7.  else
8.      return false
```

---

10. TEMPLATE VELOCI {#template}

Come Usare Questa Sezione

1. Vedi esercizio esame
2. Guarda Tabella Decisionale → identifichi pattern
3. Vieni qui → copi template
4. Compili placeholder → hai algoritmo

Template 1: Two Pointers

```
Func(A, n, <parametro>)
1.  i = 1
2.  j = 1
3.  while (i ≤ n) and (j ≤ n) and (<relazione NON soddisfatta>)
4.      if <confronto richiede incremento j>
5.          j = j + 1
6.      else
7.          i = i + 1
8.  if <trovato>
9.      return (i, j)
```

```
10. else
11.     return (0, 0)
```

Compila: Riga 3 (relazione), Riga 4 (quando incrementare j vs i), Riga 8 (condizione trovato)

Template 2: D&C Ricerca (Una Ricorsione)

```
Func(A, p, r)
1.  if <caso base>
2.      return <soluzione diretta>
3.  q = ⌊(p + r)/2⌋
4.  if <condizione indica soluzione in [p,q]>
5.      return Func(A, p, q)
6.  else
7.      return Func(A, q+1, r)
```

Compila: Riga 1 (caso base: p>r? p=r?), Riga 4 (come decido quale metà?)

Template 3: D&C Calcolo (Due Ricorsioni)

```
Func(A, p, r)
1.  if p = r
2.      return A[p]
3.  q = ⌊(p + r)/2⌋
4.  sol1 = Func(A, p, q)
5.  sol2 = Func(A, q+1, r)
6.  return <combina sol1 e sol2>
```

Compila: Riga 6 (come combino? max? min? somma?)

Template 4: Ricorsione Albero

```
Func(x)
1.  if x = nil
2.      return <valore base>
3.  solLeft = Func(x.left)
4.  solRight = Func(x.right)
5.  return <combina solLeft, solRight, x.key>
```

Compila: Riga 2 (base: 0? -∞? nil?), Riga 5 (combinazione)

Template 5: BST Arricchito Insert

```
Insert(T, z)
1.  y = nil
2.  x = T.root
3.  <inizializza z.campo>
4.  while x ≠ nil
5.      y = x
6.      <aggiorna x.campo usando z>         // CHIAVE!
7.      if z.key < x.key
8.          x = x.left
9.      else
10.         x = x.right
11. z.p = y
12. if y = nil: T.root = z
13. else if z.key < y.key: y.left = z
14. else: y.right = z
```

Compila: Riga 3 (campo z: z.sum=z.key? z.size=1?), Riga 6 (come aggiorna x.campo?)

Template 6: DP Bottom-Up 1D

```
DP(input, n)
1.  C[0..n] = <inizializza casi base>
2.  for i = 1 to n do
3.      C[i] = <calcola da C[0..i-1]>
4.  return C[n]
```

Compila: Riga 1 (C[0]=?), Riga 3 (ricorrenza)

Template 7: DP Bottom-Up 2D

```
DP(input, m, n)
1.  C[0..m, 0..n] = <inizializza bordi>
2.  for i = 1 to m do
3.      for j = 1 to n do
4.          C[i,j] = <calcola da C[i-1,j], C[i,j-1], C[i-1,j-1]>
5.  return C[m,n]
```

Compila: Riga 1 (bordi), Riga 4 (ricorrenza), Ordine scansione (for i poi for j? o viceversa?)

Template 8: DP Top-Down Memoizzazione

```
Init(n)
1.  C[...] = -1                                  // non calcolato
2.  return Rec(...)

Rec(...)
1.  if C[...] ≠ -1                               // già calcolato
2.      return C[...]
3.  if <caso base>
4.      C[...] = <valore base>
5.  else
6.      C[...] = <calcola ricorsivamente>
7.  return C[...]
```

Template 9: Greedy

```
Greedy(input)
1.  <ordina input se necessario>
2.  S = ∅
3.  for each elemento in input
4.      if <scelta greedy valida>
5.          S = S ∪ {elemento}
6.  return S
```

Compila: Riga 1 (ordine?), Riga 4 (scelta greedy?)

Template 10: Funzioni Notevoli - Pronti

MaxHeapify(A,i,n): Copia da sezione Funzioni Notevoli

BuildMaxHeap(A,n):

```
for i = ⌊n/2⌋ downto 1 do MaxHeapify(A, i, n)
```

HeapSort(A,n):

```
BuildMaxHeap(A,n)
for i = n downto 2 do
```

```
    A[1] ↔ A[i]
    MaxHeapify(A, 1, i-1)
```

Insert BST: Copia da sezione Funzioni Notevoli

Delete BST: Copia da sezione Funzioni Notevoli (con Transplant)

Partition 2-Way: Copia da sezione Funzioni Notevoli

Partition 3-Way: Copia da sezione Funzioni Notevoli

Checklist Veloci Pre-Consegna

Pseudocodice:

- [ ] Righe numerate
- [ ] Nome funzione corretto
- [ ] Parametri corretti

Correttezza:

- [ ] Caso base gestito
- [ ] Invariante scritto (se ciclo)
- [ ] Dimostrazione presente

Complessità:

- [ ] Tempo calcolato
- [ ] Giustificazione (iterazioni, ricorrenza)

ERRORI COMUNI DA EVITARE

❌ **Two Pointers**: Loop infinito (né i né j avanza)
✓ **Fix**: Almeno un indice avanza sempre

❌ **D&C**: Ricorsione su stesso intervallo
✓ **Fix**: Caso base esplicito PRIMA della ricorsione

❌ **Alberi**: Accesso x.left senza check nil
✓ **Fix**: `if x = nil return <base>` SEMPRE prima

❌ **DP**: Ordine scansione sbagliato (uso C[i,j] prima di calcolarlo)
✓ **Fix**: Dipendenze: C[i,j] dipende da chi? Calcola prima le dipendenze

❌ **BST Arricchito**: Aggiorno campo DOPO while
✓ **Fix**: Aggiorna DENTRO while (durante discesa)

❌ **Greedy**: Dimentico dimostrazione scelta greedy
✓ **Fix**: SEMPRE dimostra: (1) scelta greedy, (2) sottostruttura ottima

---

INVARIANTI TIPICHE (Scrivi Queste)

Two Pointers: ∀(i',j') con i'<i o j'<j: relazione non soddisfatta.

Ciclo lineare: A[1..i-1] non contiene elementi con proprietà P.

BuildMaxHeap: A[i+1..n] sono radici di max-heap.

HeapSort: A[1..i] è max-heap, A[i+1..n] ordinato crescente.

Insert BST: y è il padre del nodo da inserire.

DP: C[i,j] contiene soluzione ottima per sottoproblema (i,j).

MAPPA RAPIDA FINALE: Problema → Soluzione

Vedo...	Uso...	Sezione
"coppia (i,j)" + ordinato	Two Pointers	§2
"trova indice" + ordinato	D&C Ricerca	§3
"campo x.sum/size/leaves"	BST Arricchito	§5
"calcola max/somma albero"	Ricorsione Albero	§4
"ricorrenza c(i,j)"	DP	§6
"greedy" / "scheduling"	Greedy	§7
"MaxHeapify" / "Insert BST"	Funzioni Notevoli	§8
"anagramma"	Contatori	§9

GUIDA COMPLETA - Copre TUTTI i pattern d'esame. Consulta durante preparazione, copia template durante esame.

1.1 Anatomia di un Esercizio Tipo

Struttura standard della richiesta:

```
"Realizzare una funzione <NOME>(<PARAMETRI>) che, dato <INPUT>,
<AZIONE> e restituisce <OUTPUT>. <VINCOLI>.
Scrivere lo pseudocodice e valutare la complessità."
```

Esempio reale:

> "Realizzare una funzione Double(A,n) che, dato un array A[1,n] ordinato in senso crescente, verifica se esiste una coppia di indici i, j tali che A[j] = 2*A[i]. Restituisce la coppia se esiste e (0,0) altrimenti."

1.2 Estrai Informazioni Chiave

STEP 1: Identifica INPUT

- Tipo: array/albero/heap/valore
- Proprietà: ordinato? distinto? range specifico?
- Parametri: dimensione, indici, chiave da cercare?

STEP 2: Identifica OUTPUT

- Cosa restituisco? booleano / indice / valore / coppia?
- Caso "non trovato": cosa ritorno? (nil, 0, (0,0), false)

STEP 3: Identifica AZIONE

- Verbo chiave: "verifica", "trova", "restituisce", "ordina"
- Condizione da soddisfare: "tale che X", "se esiste Y"

STEP 4: Identifica VINCOLI

- Spazio costante? (no array ausiliari)
- Non alterare input?
- Complessità richiesta? (O(n), O(log n))

1.3 Esempio Pratico: Scompongo Double(A,n)

Categoria	Contenuto
INPUT	A[1..n] ordinato crescente, n dimensione
OUTPUT	(i,j) se A[j]=2*A[i], altrimenti (0,0)
AZIONE	Verifica esistenza coppia con relazione 2*A[i]=A[j]
VINCOLI	Spazio costante (no array extra), non alterare A
PROPRIETÀ	A ordinato → posso usare due indici che scorrono

Conclusione mentale: "Due indici i,j che scorrono A confrontando 2*A[i] con A[j]"

2. IDENTIFICARE IL PATTERN: Quale Tecnica Usare? {#identificare-pattern}

2.1 Decision Tree per Scegliere la Tecnica

```
INPUT è array ordinato?
│
├─ SÌ → Proprietà da verificare con confronti?
│  │
│  ├─ SÌ → Ricerca binaria o due indici?
│  │  ├─ Cerco singolo valore → DIVIDE-ET-IMPERA (ricerca binaria)
│  │  └─ Confronto coppie → TWO POINTERS (i, j che scorrono)
│  │
│  └─ NO → Proprietà strutturale (gap, picco, centro)?
│     └─ DIVIDE-ET-IMPERA (divido e scelgo metà con proprietà)
│
└─ NO → INPUT non ordinato
   │
   ├─ Conteggio elementi / anagrammi → ARRAY CON CONTATORI
   │
   ├─ Operazione su heap/BST → RICORSIONE SU STRUTTURA
   │
   └─ Ricerca elemento con proprietà → SCORRIMENTO LINEARE con invariante
```

2.2 Keyword che Identificano il Pattern

TWO POINTERS (2 indici i,j):

- "coppia di indici"
- "A[i] e A[j] tali che..."
- array ordinato + relazione tra elementi

DIVIDE-ET-IMPERA:

- "ricerca"
- "divide et impera" esplicito
- array ordinato + "restituisce indice"
- proprietà che permette di escludere metà

CONTATORI/TABELLE:

- "conta occorrenze"
- "anagramma"
- "elementi distinti"
- alfabeto piccolo (0,1)

RICORSIONE STRUTTURALE:

- "dato un albero/heap"
- "modifica albero aggiungendo/rimuovendo"
- "visita personalizzata"

INVARIANTE DI CICLO:

- "trova minimo confrontando solo..."
- requisiti strani (no confronti interni)
- ottimizzazione particolare

---

3. COSTRUIRE LA FUNZIONE: Passo dopo Passo {#costruire-funzione}

3.1 Schema Universale di Costruzione

FASE 1: SCHELETRO BASE

```
<NomeFunzione>(<parametri>)
1.  // Caso base / casi speciali
2.  if <condizione triviale>
3.      return <soluzione diretta>
4.
5.  // Inizializzazione variabili
6.
7.  // Corpo principale (ciclo/ricorsione)
8.
9.  // Return finale
```

FASE 2: CORPO (scegli tecnica)

- Two pointers → while con avanzamento i e/o j
- Divide-et-impera → chiamata ricorsiva su metà
- Contatori → for che popola array ausiliario
- Ricorsione struttura → if struttura.left/right, chiamate ricorsive

FASE 3: INVARIANTE/CORRETTEZZA

- Cosa è vero all'inizio di ogni iterazione/chiamata?
- Come mantengo questa verità?
- Alla fine, invariante + condizione uscita → soluzione corretta

3.2 Esempio Guidato: Double(A,n) Step-by-Step

PASSO 1: Capisco la relazione

- Voglio A[j] = 2*A[i]
- A è ordinato → se aumento i, 2*A[i] cresce
- Se aumento j, A[j] cresce
- Posso far "inseguire" 2*A[i] e A[j]

PASSO 2: Scheletro

```
Double(A, n)
1.  // Caso array vuoto
2.  if n < 2
3.      return (0, 0)
4.
5.  // Inizializzo indici
6.  i = 1
7.  j = 1
8.
9.  // Ciclo: confronto 2*A[i] con A[j]
10. while ...
11.     ...
12.
13. // Return risultato
```

PASSO 3: Logica While

- Continuo mentre non ho trovato e sono dentro l'array
- Se 2*A[i] > A[j] → j troppo piccolo, incremento j
- Se 2*A[i] < A[j] → i troppo piccolo, incremento i
- Se 2*A[i] = A[j] → trovato! esco

```
10. while (i ≤ n) and (j ≤ n) and (2*A[i] ≠ A[j])
11.     if 2*A[i] > A[j]
12.         j = j + 1
13.     else
14.         i = i + 1
```

PASSO 4: Return Finale

- Se esco con i≤n e j≤n → ho trovato
- Altrimenti → non esiste

```
15. if (i ≤ n) and (j ≤ n)
16.     return (i, j)
```

```
17. else
18.     return (0, 0)
```

SOLUZIONE COMPLETA:

```
Double(A, n)
1.  i = 1
2.  j = 1
3.  while (i ≤ n) and (j ≤ n) and (2*A[i] ≠ A[j])
4.      if 2*A[i] > A[j]
5.          j = j + 1
6.      else
7.          i = i + 1
8.  if (i ≤ n) and (j ≤ n)
9.      return (i, j)
10. else
11.     return (0, 0)
```

Invariante: ∀i' ∈ [1,i). ∀j' ∈ [1,j). 2*A[i'] ≠ A[j']
Significa: tutte le coppie (i',j') già esplorate non soddisfano la condizione.

---

4. ARRAY CON TWO POINTERS {#array-two-pointers}

4.1 Quando Usare Two Pointers

Segnali identificativi:

- Array ordinato
- "coppia di indici i,j"
- Relazione tra A[i] e A[j] (es. A[i]+A[j]=k, A[j]=2*A[i], A[i]-A[j]=k)

Schema mentale:

- Due indici partono da posizioni iniziali (spesso (1,1) o (1,n))
- Confronto A[i] con A[j] (o funzione di essi)
- Avanzo i e/o j basandomi sul confronto
- Continuo finché non trovo o finisco array

4.2 Template Two Pointers Standard

```
<Nome>Funzione(A, n, <parametro opzionale>)
1.  i = <posizione iniziale i>
2.  j = <posizione iniziale j>
3.  while (i ≤ n) and (j ≤ n) and (<condizione non trovato>)
4.      if <A[i] relazione A[j] richiede aumento j>
```

```
5.          j = j + 1
6.      else if <A[i] relazione A[j] richiede aumento i>
7.          i = i + 1
8.      // casi aggiuntivi se necessari
9.  if <trovato>
10.     return <risultato positivo>
11. else
12.     return <risultato negativo>
```

4.3 Esercizi Tipo e Soluzioni

Esempio 1: Diff(A, n, k) - Array Ordinato Decrescente

Richiesta: Dato A[1..n] ordinato decrescente, verifica se ∃ i,j: A[i] - A[j] = k. Restituisce (i,j) o (0,0).

Ragionamento:

- A decrescente → A[1] massimo, A[n] minimo
- Parto con i=1, j=1
- Se A[i]-A[j] < k → j troppo grande (A[j] troppo vicino ad A[i]), incremento j
- Se A[i]-A[j] > k → i troppo piccolo (A[i] troppo lontano da A[j]), incremento i

Soluzione:

```
Diff(A, n, k)
1.  i = 1
2.  j = 1
3.  while (i ≤ n) and (j ≤ n) and (A[i] − A[j] ≠ k)
4.      if A[i] − A[j] < k
5.          j = j + 1
6.      else
7.          i = i + 1
8.  if (i ≤ n) and (j ≤ n)
9.      return (i, j)
10. else
11.     return (0, 0)
```

Invariante: Tutte le coppie (i',j') con i'<i oppure j'<j non soddisfano A[i']-A[j']=k.

Esempio 2: min(A, B) - Confronti Solo Tra Array

Richiesta: Dati A e B permutazioni uno dell'altro, trova minimo confrontando SOLO A[i] con B[j] (mai A[i] con A[k]).

Ragionamento:

- Non posso confrontare elementi stesso array
- Se A[i] ≤ B[j] → B[j] non è minimo (esiste A[i] ≤ B[j]), avanzo j
- Se A[i] > B[j] → A[i] non è minimo (esiste B[j] < A[i]), avanzo i
- Quando j > n → A[i] è ≤ tutti gli elementi di B → A[i] è il minimo

Soluzione:

```
min(A, B, n)
1.  i = 1
2.  j = 1
3.  while j ≤ n
4.      if A[i] ≤ B[j]
5.          j = j + 1
6.      else
7.          i = i + 1
8.  return A[i]
```

Nota: Non serve controllare i≤n (garantito dall'invariante).

Esempio 3: Anagramma(x, y) - Stringhe Binarie

Richiesta: Date stringhe x,y su alfabeto {0,1}, verifica se x è anagramma di y (stessi caratteri, ordine diverso).

Ragionamento:

- Array di contatori diff[0..1]
- Incremento diff[x[i]] per ogni carattere di x
- Decremento diff[y[i]] per ogni carattere di y
- Se diff[0]=0 e diff[1]=0 (implicito) → anagrammi

Soluzione:

```
Anagramma(x, y)
1.  diff[0..1] = 0
2.  for i = 1 to x.len
3.      diff[x[i]] = diff[x[i]] + 1
4.      diff[y[i]] = diff[y[i]] - 1
5.  if (diff[0] = 0) and (x.len = y.len)
6.      return true
7.  else
8.      return false
```

4.4 Checklist Two Pointers

Prima di scrivere:

- [ ] Ho capito la relazione tra A[i] e A[j]?
- [ ] So quando incrementare i e quando j?
- [ ] Posizioni iniziali corrette (1,1 vs 1,n)?
- [ ] Condizione while corretta (quando esco)?
- [ ] Gestito caso "non trovato"?

Durante la scrittura:

- [ ] Controllo i≤n e j≤n nel while?
- [ ] Evito loop infiniti (almeno un indice avanza sempre)?
- [ ] Return finale gestisce entrambi i casi?

---

5. ARRAY CON PROPRIETÀ {#array-proprieta}

5.1 Quando Usare Questo Pattern

Segnali identificativi:

- "Trova indice i tale che `<proprietà>`"
- "Esiste un elemento con `<caratteristica>`"
- Array con struttura particolare (gap, picco, alternante, bi-ordinato, semi-ordinato)

Differenza con Two Pointers:

- Non cerco relazione tra DUE elementi
- Cerco UN elemento/indice con proprietà intrinseca

Due approcci possibili:

1. **Scorrimento lineare** con invariante (O(n))
2. **Divide-et-impera** se proprietà permette di escludere metà (O(log n))

5.2 Template Scorrimento Lineare

```
<Nome>Funzione(A, n)
1.  for i = <inizio> to <fine>
2.      if <A[i] soddisfa proprietà>
3.          return i  // o A[i], dipende dalla richiesta
4.  return <valore "non trovato">
```

Invariante tipico: A[1..i-1] non contiene elementi con la proprietà.

5.3 Template Divide-et-Impera (Esclusione Metà)

```
<Nome>Funzione(A, p, r)
1.  if <caso base>
2.      return <soluzione diretta>
3.  q = ⌊(p + r)/2⌋
4.  if <proprietà implica elemento in [p,q]>
5.      return <Nome>Funzione(A, p, q)
6.  else
7.      return <Nome>Funzione(A, q+1, r)
```

Chiave: Capire come dividere in modo da escludere metà sicuramente senza soluzione.

5.4 Esercizi Tipo e Soluzioni

Esempio 1: Gap(A, p, r) - Trova Salto Grande

Richiesta: Dato A[p..r] con A[r]-A[p] ≥ r-p, trova i: A[i+1]-A[i] > 1 (gap).

Ragionamento D&C:

- Divido a metà in q
- Se A[q]-A[p] > q-p → c'è gap in [p,q] (più crescita che posizioni)
- Altrimenti → gap in [q,r]
- Caso base: 2 elementi → gap è sempre in p

Soluzione:

```
Gap(A, p, r)
1.  if p = r - 1
2.      return p
3.  q = ⌊(p + r)/2⌋
4.  if A[q] - A[p] > q - p
5.      return Gap(A, p, q)
6.  else
7.      return Gap(A, q, r)
```

Complessità: T(n) = T(n/2) + O(1) → O(log n)

Esempio 2: Fix(A) - Trova i Tale Che A[i]=i

Richiesta: Dato A[1..n] ordinato distinto, trova i: A[i]=i, o 0 se non esiste.

Osservazione chiave:

- Se A[i] > i → ∀j>i: A[j]>j (array cresce + vincolo ordine)
- Se A[i] < i → ∀j<i: A[j]<j
- Quindi posso fare ricerca binaria!

Ragionamento:

- Confronto A[q] con q
- Se A[q] = q → trovato
- Se A[q] < q → cerco a destra (a sinistra sicuro A[j]<j)
- Se A[q] > q → cerco a sinistra (a destra sicuro A[j]>j)

Soluzione:

```
Fix(A)
1.  return FixRec(A, 1, A.length)

FixRec(A, p, r)
1.  if p > r
2.      return 0
3.  q = ⌊(p + r)/2⌋
4.  if A[q] = q
5.      return q
6.  else if A[q] < q
7.      return FixRec(A, q+1, r)
8.  else
9.      return FixRec(A, p, q−1)
```

Esempio 3: centre(A) - Centro Array Semi-Ordinato

Richiesta: A[1..n] semi-ordinato: ∃k: A[k+1..n]A[1..k] ordinato e A[n]<A[1]. Trova k.

Ragionamento:

- Divido a metà in q
- Confronto A[q] con A[p]
- Se A[q] < A[p] → centro è in [p,q] (parte sinistra contiene il "salto")
- Se A[q] > A[p] → centro è in [q,r] (parte destra contiene il "salto")
- Caso base: 2 elementi → centro è p

Soluzione:

```
centre(A)
1.  return centreRec(A, 1, A.length)

centreRec(A, p, r)
1.  if r = p + 1
2.      return p
3.  q = ⌊(p + r)/2⌋
4.  if A[q] < A[p]
5.      return centreRec(A, p, q)
6.  else
7.      return centreRec(A, q, r)
```

5.5 Checklist Proprietà Array

Per Scorrimento Lineare:

- [ ] Invariante chiaro (cosa è vero su A[1..i-1])?
- [ ] Condizione if corretta per riconoscere proprietà?
- [ ] Return nel posto giusto (dentro/fuori ciclo)?

Per Divide-et-Impera:

- [ ] Caso base identificato (quando non divido più)?
- [ ] Criterio divisione corretto (quale metà contiene soluzione)?
- [ ] Non genero ricorsione infinita (dimensione diminuisce)?
- [ ] Dimostrazione correttezza per induzione?

---

6. DIVIDE-ET-IMPERA SU ARRAY {#divide-impera-array}

6.1 Quando Usare D&C su Array

Segnali identificativi:

- Testo dice "divide et impera" oppure "ricorsiva"
- Array ordinato + ricerca valore/indice
- Problema si può ridurre a sottoproblema dimensione metà
- Complessità richiesta O(log n)

6.2 Schema Mentale D&C

TRE DOMANDE FONDAMENTALI:

1. **CASO BASE**: Quando il problema è "abbastanza piccolo" da risolvere direttamente?
   - Array 1 elemento? 2 elementi? Vuoto?
2. **DIVIDE**: Come divido il problema?
   - Quasi sempre: q = ⌊(p+r)/2⌋ (divido a metà)
3. **IMPERA**: Su quale metà ricorro?
   - Entrambe le metà (come MergeSort) → O(n log n)
   - Solo una metà (come ricerca binaria) → O(log n)
   - Come decido quale metà? Confronto A[q] con qualcosa

6.3 Template D&C Ricerca (Una Ricorsione)

```
<Nome>Funzione(A, p, r, <parametri>)
1.  // Caso base
2.  if <problema piccolo>
3.      return <soluzione diretta>
4.
5.  // Divide
6.  q = ⌊(p + r)/2⌋
7.
8.  // Confronto per decidere dove ricorrere
9.  if <condizione indica soluzione in [p,q]>
10.     return <Nome>Funzione(A, p, q, <parametri>)
11. else
12.     return <Nome>Funzione(A, q+1, r, <parametri>)
```

6.4 Template D&C Calcolo (Due Ricorsioni)

```
<Nome>Funzione(A, p, r)
1.  // Caso base
2.  if p = r
3.      return A[p]
4.
5.  // Divide
6.  q = ⌊(p + r)/2⌋
7.
8.  // Impera (risolvi entrambe le metà)
9.  sol1 = <Nome>Funzione(A, p, q)
10. sol2 = <Nome>Funzione(A, q+1, r)
11.
12. // Combina
13. return <combina sol1 e sol2>
```

6.5 Esercizi Tipo e Soluzioni

Esempio 1: Over(A, p, r, x) - Primo Elemento > x

Richiesta: Array ordinato crescente, trova indice minimo i: A[i]>x, o r+1 se non esiste.

Ragionamento:

- Ricerca binaria modificata
- Confronto A[q] con x
- Se A[q] ≤ x → elemento cercato è in [q+1,r] (a destra)
- Se A[q] > x → elemento potrebbe essere q oppure a sinistra

Soluzione:

```
Over(A, p, r, x)
1.  if p > r
2.      return r + 1
3.  q = ⌊(p + r)/2⌋
4.  if A[q] ≤ x
5.      return Over(A, q+1, r, x)
6.  else
7.      return Over(A, p, q−1, x)
```

Nota sottile: Perché ricorro su q-1 e non su q quando A[q]>x?

- Voglio trovare il MINIMO i con A[i]>x
- Se A[q]>x, potrebbe esserci elemento più piccolo a sinistra
- Quando ritorno dalla ricorsione su [p,q-1], se non ho trovato niente, q è la risposta

---

Esempio 2: Max(A, p, r) - Massimo con D&C

Richiesta: Trova massimo in A[p..r] con divide et impera.

Ragionamento:

- Divido a metà
- Trovo max sinistra e max destra ricorsivamente
- Ritorno il maggiore dei due

Soluzione:

```
Max(A, p, r)
1.  if p = r
2.      return A[p]
3.  q = ⌊(p + r)/2⌋
4.  m1 = Max(A, p, q)
5.  m2 = Max(A, q+1, r)
6.  if m1 > m2
```

```
7.      return m1
8.  else
9.      return m2
```

Complessità: T(n) = 2T(n/2) + O(1) → O(n)

Esempio 3: subseq(X, Y, m, n) - X Sottosequenza di Y

Richiesta: Verifica se X[1..m] è sottosequenza di Y[1..n] (X appare in Y nello stesso ordine, non necessariamente contiguo).

Ragionamento:

- Caso base: m=0 → X vuoto, sempre sottosequenza
- Caso base: m>n → X più lungo di Y, impossibile
- Confronto X[m] con Y[n] (ultimi elementi)
- Se uguali → controllo X[1..m-1] in Y[1..n-1]
- Se diversi → X[1..m] potrebbe essere in Y[1..n-1]

Soluzione:

```
subseq(X, Y, m, n)
1.  if m = 0
2.      return true
3.  if m > n
4.      return false
5.  if X[m] = Y[n]
6.      return subseq(X, Y, m-1, n-1)
7.  else
8.      return subseq(X, Y, m, n-1)
```

Complessità: O(n) - al massimo scorro Y una volta.

6.6 Checklist D&C

Prima di scrivere:

- [ ] Caso base: quando smetto di dividere?
- [ ] Divide: come scelgo q? (quasi sempre ⌊(p+r)/2⌋)
- [ ] Impera: ricorro su una o entrambe le metà?
- [ ] Combina (se 2 ricorsioni): come unisco soluzioni?

Correttezza:

- [ ] Dimostrazione per induzione su n (dimensione sottoproblema)
- [ ] Caso base: n piccolo (0,1,2) risolto correttamente?
- [ ] Passo induttivo: assumo correttezza su sottoproblemi più piccoli, dimostro correttezza su problema dimensione n

Complessità:

- [ ] Scrivo ricorrenza: T(n) = ...
- [ ] Risolvo ricorrenza (contando chiamate + lavoro per chiamata)

---

7. RICORSIONE SU STRUTTURE {#ricorsione-strutture}

7.1 Quando Usare Ricorsione su Alberi/Heap

Segnali identificativi:

- "Dato un albero..." oppure "Dato un heap..."
- Operazione custom su BST/Heap (non standard Insert/Delete)
- "Calcola proprietà di sottoalbero"
- "Modifica nodo e mantieni proprietà struttura"

7.2 Schema Mentale Ricorsione Alberi

STRUTTURA NODO:

```
Nodo x:
  x.key      // chiave
  x.left     // sottoalbero sinistro
  x.right    // sottoalbero destro
  x.p        // padre (opzionale)
  x.<campo>  // campo custom (es. x.even, x.sum)
```

PATTERN RICORSIVO CLASSICO:

```
Funzione(x)
1.  if x = nil
2.      return <valore base>
3.  <calcola su x.left ricorsivamente>
4.  <calcola su x.right ricorsivamente>
5.  <combina risultati + informazione x.key>
6.  return <risultato>
```

7.3 Template Visita Custom

```
VisitaCustom(x)
1.  if x ≠ nil
2.      <azione pre>                        // PreOrder
3.      VisitaCustom(x.left)
4.      <azione in>                         // InOrder
5.      VisitaCustom(x.right)
6.      <azione post>                       // PostOrder
```

Scelta ordine:

- **PreOrder** (pre): Elaboro prima il nodo, poi i figli
- **InOrder** (in): BST → produce sequenza ordinata
- **PostOrder** (post): Elaboro prima i figli, poi combino nel nodo

7.4 Template Proprietà Custom BST

Problema tipo: BST con campo aggiuntivo `x.<campo>` da mantenere.

Esempio: BST con x.even = true se somma sottoalbero radicato in x è pari.

Soluzione Insert modificato:

```
Insert(T, z)
1.  // Standard: cerca posizione
2.  x = T.root
3.  y = nil
4.  z.even = (z.key mod 2 = 0)           // inizializza campo z
5.  while x ≠ nil
6.      x.even = (x.even = z.even)       // aggiorna campo lungo il cammino
7.      y = x
8.      if z.key < x.key
9.          x = x.left
10.     else
11.         x = x.right
12. // Collega z
13. z.p = y
14. if y = nil
15.     T.root = z
16. else if z.key < y.key
17.     y.left = z
18. else
19.     y.right = z
```

Idea chiave: Aggiorna campo custom durante la discesa (while) perché stai modificando il sottoalbero.

7.5 Template Calcolo Ricorsivo su Albero

Esempio: MaxPath(T) - Costo massimo cammino radice-foglia (costo = somma chiavi).

Ragionamento:

- Caso base: nodo nil → costo 0
- Caso foglia: T.key (solo nodo corrente)
- Caso interno: T.key + max(MaxPath(T.left), MaxPath(T.right))

Soluzione:

```
MaxPath(T)
1.  if T = nil
2.      return 0
3.  if T.left = nil and T.right = nil
4.      return T.key                        // foglia
5.  maxLeft = MaxPath(T.left)
6.  maxRight = MaxPath(T.right)
7.  return T.key + max(maxLeft, maxRight)
```

7.6 Template Verifica Proprietà Albero

Esempio: IsMaxHeap(A, i, n) - Verifica se A[i..n] è max-heap.

Ragionamento iterativo:

- Scorro i nodi interni (i = 1..⌊n/2⌋)
- Per ciascuno, verifico A[i] ≥ A[2i] e A[i] ≥ A[2i+1]

Soluzione iterativa:

```
IsMaxHeap(A, n)
1.  for i = 1 to ⌊n/2⌋
2.      if (2i ≤ n) and (A[i] < A[2i])
3.          return false
4.      if (2i+1 ≤ n) and (A[i] < A[2i+1])
5.          return false
6.  return true
```

Soluzione ricorsiva:

```
IsMaxHeap(A, i, n)
1.  l = 2i
2.  r = 2i + 1
3.  if l > n
4.      return true                         // foglia, ok
5.  leftOk = (A[i] ≥ A[l]) and IsMaxHeap(A, l, n)
6.  if r > n
```

```
7.      rightOk = true                  // no figlio destro
8.  else
9.      rightOk = (A[i] ≥ A[r]) and IsMaxHeap(A, r, n)
10. return leftOk and rightOk
```

7.7 Checklist Ricorsione Strutture

Prima di scrivere:

- [ ] Caso base: struttura vuota (nil)? Foglia?
- [ ] Ricorro su left e right? Oppure solo uno?
- [ ] Combino risultati ricorsivi? Come?
- [ ] Campo custom da mantenere? Dove lo aggiorno?

Durante la scrittura:

- [ ] Controllo x ≠ nil prima di accedere x.left/right?
- [ ] Gestisco caso un solo figlio (left o right nil)?
- [ ] Se modifico struttura, mantengo proprietà (BST, heap)?

Correttezza:

- [ ] Dimostrazione per induzione su altezza/dimensione sottoalbero
- [ ] Invariante di ciclo (se uso while invece di ricorsione)

---

8. TEMPLATE VUOTI DA COMPLETARE {#template-vuoti}

8.1 Two Pointers su Array Ordinato

Problema: Trova coppia (i,j) con relazione R(A[i], A[j]) = vero.

```
<Nome>(A, n)
1.  i = ___                             // posizione iniziale i
2.  j = ___                             // posizione iniziale j
3.  while (i ≤ n) and (j ≤ n) and (_____________) // condizione "non
ancora trovato"
4.      if _____________                // caso 1: incremento j
5.          j = j + 1
6.      else if _____________           // caso 2: incremento i
7.          i = i + 1
8.      // eventuali altri casi
9.  if _____________                    // condizione "trovato"
10.     return (i, j)                   // o altro risultato positivo
```

```
11. else
12.     return (0, 0)                    // o altro risultato negativo
```

Compilare:

- Riga 1-2: Scegli posizioni iniziali (1,1) oppure (1,n)
- Riga 3: Condizione: continua finché non ho trovato
- Riga 4,6: Quando incrementare i vs j? Guarda relazione
- Riga 9: Come so di aver trovato?

8.2 Divide-et-Impera Ricerca (Una Ricorsione)

Problema: Trova elemento con proprietà P in array ordinato.

```
<Nome>(A, p, r)
1.  if _____________                  // caso base
2.      return _____________          // soluzione diretta
3.  q = ⌊(p + r)/2⌋
4.  if _____________                  // condizione: elemento in [p,q]
5.      return <Nome>(A, p, q)
6.  else
7.      return <Nome>(A, q+1, r)
```

Compilare:

- Riga 1: Quando non divido più? (p=r? p>r? p=r-1?)
- Riga 2: Cosa ritorno nel caso base?
- Riga 4: Come decido se cercare a sinistra o destra?

8.3 Ricorsione su Albero Binario

Problema: Calcola proprietà ricorsiva sull'albero.

```
<Nome>(x)
1.  if x = nil
2.      return _____________          // valore base (0? nil? -∞?)
3.  if _____________                  // caso foglia (opzionale)
4.      return _____________
5.  solLeft = <Nome>(x.left)
6.  solRight = <Nome>(x.right)
7.  return _____________              // combina con x.key
```

Compilare:

- Riga 2: Cosa ritorno per sottoalbero vuoto?
- Riga 3: Serve caso speciale foglia?
- Riga 7: Come combino solLeft, solRight, x.key?

8.4 Scorrimento Lineare con Invariante

Problema: Trova elemento con proprietà P (non ordinato).

```
<Nome>(A, n)
1.  for i = 1 to n
2.      if _________________                // A[i] soddisfa P?
3.          return _________________        // ritorno i o A[i]
4.  return _________________                // valore "non trovato"
```

Invariante: A[1..i-1] non contiene elementi con proprietà P.

---

9. ERRORI TIPICI DA EVITARE {#errori-tipici}

9.1 Errori su Array

❌ Indici fuori range

```
// SBAGLIATO
for i = 1 to n+1    // accesso A[n+1]!
    if A[i] > A[i+1]
```

✓ Corretto

```
for i = 1 to n-1    // ultima iterazione: A[n-1] vs A[n]
    if A[i] > A[i+1]
```

❌ Condizione while non termina

```
// SBAGLIATO: né i né j avanzano in caso speciale
while i ≤ n and j ≤ n
    if A[i] = A[j]
        // non avanzo nessuno!
```

✓ Corretto

```
while i ≤ n and j ≤ n and A[i] ≠ A[j]
    if ...
        i = i + 1
    else
        j = j + 1
// esco quando A[i] = A[j]
```

9.2 Errori su Divide-et-Impera

❌ Ricorsione infinita (dimensione non diminuisce)

```
// SBAGLIATO
if p < r
    q = (p + r)/2
    return Func(A, p, q+1)    // dimensione non diminuisce mai se p=r-1
```

✓ Corretto

```
if p ≥ r
    return ...    // caso base esplicito
q = ⌊(p + r)/2⌋
if ...
    return Func(A, p, q)      // dimensione diminuisce
else
    return Func(A, q+1, r)
```

❌ Caso base mancante

```
// SBAGLIATO: cosa ritorno quando p > r?
Func(A, p, r)
    q = (p + r)/2
    if ...
        return Func(A, p, q)
    else
        return Func(A, q+1, r)
```

✓ Corretto

```
Func(A, p, r)
    if p > r                    // SEMPRE caso base prima!
        return ...
    q = (p + r)/2
    ...
```

9.3 Errori su Alberi

❌ Accesso a campo senza controllo nil

```
// SBAGLIATO
Func(x)
    return Func(x.left) + Func(x.right)    // se x=nil → errore
```

✓ Corretto

```
Func(x)
    if x = nil
        return 0
    return Func(x.left) + Func(x.right)
```

❌ Dimenticare caso un solo figlio

```
// SBAGLIATO: cosa faccio se x.right = nil?
maxChild = max(Func(x.left), Func(x.right))
```

✓ Corretto

```
if x.left = nil and x.right = nil
    return x.key    // foglia
if x.left = nil
    return Func(x.right)
if x.right = nil
    return Func(x.left)
// caso normale: entrambi i figli
return max(Func(x.left), Func(x.right))
```

9.4 Errori di Invariante

❌ **Invariante non preservato**

```
// Invariante: "A[1..i] ordinato"
for i = 2 to n
    // inserisco A[i] in A[1..i-1] ordinato
    // ma non mantengo A[1..i] ordinato!
    if A[i] < A[i-1]
        A[i] ⇔ A[i-1]    // scambio solo con precedente, non basta
```

✓ **Corretto**

```
for i = 2 to n
    key = A[i]
    j = i - 1
    while j > 0 and A[j] > key    // scorro indietro finché necessario
        A[j+1] = A[j]
        j = j - 1
    A[j+1] = key
```

---

10. CHECKLIST FINALE PRIMA DI CONSEGNARE

10.1 Pseudocodice

- [ ] Righe numerate (1, 2, 3, ...)?
- [ ] Nome funzione corretto (quello richiesto)?
- [ ] Parametri corretti (nome e ordine)?
- [ ] Variabili inizializzate prima dell'uso?
- [ ] Indentazione chiara (if dentro for, ecc.)?

10.2 Correttezza

- [ ] Caso base/speciale gestito (array vuoto, nil, n<2)?
- [ ] Invariante scritto esplicitamente?
- [ ] Dimostrazione correttezza (induzione o invariante)?
- [ ] Gestiti tutti i casi (if completo con else)?
- [ ] Return in tutti i percorsi?

10.3 Complessità

- [ ] Complessità tempo calcolata?
- [ ] Complessità spazio calcolata (se richiesta)?
- [ ] Giustificazione (conteggio iterazioni, ricorrenza)?
- [ ] Forma finale semplificata (O(n), O(log n), O(n²))?

10.4 Stile

- [ ] Commenti solo dove necessario (no ovvietà)?
- [ ] Nomi variabili standard (i,j,p,r,q per indici)?
- [ ] No codice ridondante?
- [ ] Leggibile a prima vista?

APPENDICE: Mappa Rapida Problema → Soluzione

Se il problema dice...	**Usa tecnica...**	**Template...**
"coppia (i,j) tale che" + ordinato	Two Pointers	i=1, j=1, while con confronto
"trova indice i tale che" + ordinato	D&C Ricerca	ricorsione su metà con if confronto
"calcola max/min/somma" + albero	Ricorsione Struttura	if nil return base, ricorri left/right
"verifica proprietà" + scorrimento	Ciclo con Invariante	for i, if proprietà return
"conta/anagramma" + alfabeto piccolo	Array Contatori	diff[0..k], incrementi/decrementi
"divide et impera" esplicito	D&C Standard	caso base + $q=\lfloor(p+r)/2\rfloor$ + ricorsioni
"ordinato crescente/decrescente"	Sfrutta Ordine	Two pointers o D&C ricerca
"mantieni campo custom" + BST	Modifica Insert/Delete	aggiorna campo durante discesa

---

Fine della Guida - Ricapitolando:

1. **Leggi bene** la richiesta: input, output, vincoli
2. **Identifica pattern**: two pointers? D&C? ricorsione?
3. **Parti da template** appropriato e adattalo
4. **Scrivi invariante** e dimostrazione correttezza
5. **Calcola complessità** e verifica sia richiesta
6. **Controlla checklist** prima di consegnare

Ricorda: Gli esami premiano soluzioni **semplici e corrette**, non soluzioni "ingegnose" ma sbagliate. Segui i template del corso, non inventare approcci complicati.

1.1 Definizioni Base

Max-Heap: Ogni nodo ha chiave ≥ dei suoi figli
Min-Heap: Ogni nodo ha chiave ≤ dei suoi figli

Rappresentazione su array A[1..n]:

- Radice: A[1]
- Parent(i) = ⌊i/2⌋
- Left(i) = 2i
- Right(i) = 2i + 1
- Nodi foglia: A[⌊n/2⌋ + 1 .. n]

1.2 MaxHeapify(A, i) - Sistemare un Nodo Singolo

Quando usare: Nodo in posizione `i` viola la proprietà di heap, ma i suoi sottoalberi sono già max-heap.

Template Pseudocodice:

```
MaxHeapify(A, i)
1.  l = 2i                          // figlio sinistro
2.  r = 2i + 1                      // figlio destro
3.  max = i                         // assumi che il massimo sia i
4.  if l ≤ A.heapsize and A[l] > A[max]
5.      max = l                     // aggiorna se sinistro è maggiore
6.  if r ≤ A.heapsize and A[r] > A[max]
7.      max = r                     // aggiorna se destro è maggiore
8.  if max ≠ i                      // se i non è il massimo
9.      A[i] ↔ A[max]               // scambia i con il figlio maggiore
10.     MaxHeapify(A, max)          // ricorsione sul figlio
```

Complessità: O(log n) = O(h) con h altezza sottoalbero

Invariante (per correttezza ricorsiva):

- I sottoalberi sinistro e destro di i sono max-heap
- Dopo MaxHeapify(A, i), il sottoalbero radicato in i è max-heap

---

1.3 BuildMaxHeap(A) - Costruire Heap da Array

Quando usare: Trasformare un array non ordinato in un max-heap.

Template Pseudocodice:

```
BuildMaxHeap(A)
1.  A.heapsize = A.length
2.  for i = ⌊A.length/2⌋ downto 1      // parti dall'ultimo nodo non-foglia
3.      MaxHeapify(A, i)
```

Complessità: O(n) (analisi ammortizzata, non O(n log n))

Invariante di ciclo:

- **Inizio**: i = ⌊n/2⌋, tutti i nodi > i sono radici di max-heap (sono foglie)
- **Mantenimento**: Dopo MaxHeapify(A, i), il nodo i è radice di max-heap; left(i) e right(i) erano già heap
- **Conclusione**: i = 0, tutti i nodi > 0 sono max-heap → A[1] è radice del max-heap completo

---

1.4 HeapSort - Ordinamento in Loco

Template Pseudocodice:

```
HeapSort(A)
1.  BuildMaxHeap(A)                    // costruisci heap: O(n)
2.  for i = A.length downto 2
3.      A[1] ↔ A[i]                    // porta massimo in coda
4.      A.heapsize = A.heapsize - 1    // riduci heap
5.      MaxHeapify(A, 1)               // ripristina heap sulla radice
```

Complessità: O(n log n)

Invariante di ciclo:

- A[1..i] è max-heap
- A[i+1..n] è ordinato
- A[1..i] ≤ A[i+1..n]

1.5 Code con Priorità - Operazioni Dinamiche

Extract-Max (Rimuovi e Restituisci Massimo)

```
Extract-Max(A)
1.  if A.heapsize < 1
2.      error "heap underflow"
3.  max = A[1]                         // salva massimo
```

```
4.  A[1] = A[A.heapsize]              // sposta ultimo in radice
5.  A.heapsize = A.heapsize - 1
6.  MaxHeapify(A, 1)                  // ripristina heap
7.  return max
```

Complessità: O(log n)

Insert (Inserimento con MaxHeapifyUp)

Idea: Inserisci in coda, poi fai "salire" l'elemento confrontandolo col padre.

```
MaxHeapifyUp(A, i)
1.  while i > 1 and A[Parent(i)] < A[i]
2.      A[i] ↔ A[Parent(i)]           // scambia con padre
3.      i = Parent(i)                 // sali
```

```
Insert(A, key)
1.  A.heapsize = A.heapsize + 1
2.  A[A.heapsize] = -∞                // valore temporaneo basso
3.  IncreaseKey(A, A.heapsize, key)   // aumenta chiave a "key"
```

```
IncreaseKey(A, i, key)
1.  if key < A[i]
2.      error "nuova chiave minore della corrente"
3.  A[i] = key
4.  MaxHeapifyUp(A, i)                // fai salire il nodo
```

Complessità: O(log n)

Invariante MaxHeapifyUp:

- Ad ogni iterazione, se A[i] > A[Parent(i)], scambia e ripeti
- Termina quando A[i] ≤ A[Parent(i)] o i = 1

1.6 Checklist Heap

Prima di consegnare, verifica:

- [ ] Hai usato Parent(i) = ⌊i/2⌋, Left(i) = 2i, Right(i) = 2i+1?
- [ ] Hai controllato i ≤ A.heapsize prima di accedere ad A[i]?
- [ ] MaxHeapify parte dal nodo con violazione, non dalla radice

- [ ] BuildMaxHeap parte da ⌊n/2⌋ (ultimo nodo interno), non da n
- [ ] Invariante di ciclo: inizializzazione, mantenimento, conclusione
- [ ] Complessità calcolata correttamente (O(h) per MaxHeapify, O(n) per Build)

---

2. ALBERI BINARI DI RICERCA (BST) {#bst}

2.1 Proprietà BST

Invariante fondamentale:

- Per ogni nodo x: tutte le chiavi nel sottoalbero sinistro ≤ x.key
- Per ogni nodo x: tutte le chiavi nel sottoalbero destro ≥ x.key

Struttura nodo:

```
Nodo x:
  x.key     // chiave
  x.left    // puntatore figlio sinistro
  x.right   // puntatore figlio destro
  x.p       // puntatore padre (opzionale)
```

2.2 Visite - Template Ricorsivo

InOrder (Simmetrica) - Produce Sequenza Ordinata

```
InOrder(x)
1.  if x ≠ nil
2.      InOrder(x.left)              // visita sottoalbero sinistro
3.      print x.key                  // visita nodo corrente
4.      InOrder(x.right)             // visita sottoalbero destro
```

Proprietà: InOrder su BST produce chiavi in ordine crescente.
Complessità: Θ(n) per visitare tutti i nodi.

PreOrder (Anticipata)

```
PreOrder(x)
1.  if x ≠ nil
2.      print x.key
3.      PreOrder(x.left)
4.      PreOrder(x.right)
```

PostOrder (Posticipata)

```
PostOrder(x)
1.  if x ≠ nil
2.      PostOrder(x.left)
3.      PostOrder(x.right)
4.      print x.key
```

2.3 Search (Ricerca)

Template Ricorsivo:

```
Search(x, k)
1.  if x = nil or x.key = k
2.      return x
3.  if k < x.key
4.      return Search(x.left, k)
5.  else
6.      return Search(x.right, k)
```

Template Iterativo (preferibile per efficienza spaziale):

```
Search-Iterative(x, k)
1.  while x ≠ nil and x.key ≠ k
2.      if k < x.key
3.          x = x.left
4.      else
5.          x = x.right
6.  return x
```

Complessità: O(h) con h altezza albero (O(log n) se bilanciato, O(n) se degenere)

2.4 Minimum e Maximum

```
Minimum(x)                          Maximum(x)
1.  while x.left ≠ nil              1.  while x.right ≠ nil
2.      x = x.left                  2.      x = x.right
3.  return x                        3.  return x
```

Complessità: O(h)

2.5 Successor (Successore)

Definizione: Nodo con chiave minima maggiore di x.key (= successivo in InOrder).

Casi:

1. Se x ha sottoalbero destro → Successor(x) = Minimum(x.right)
2. Altrimenti → Successor(x) = primo antenato di cui x è nel sottoalbero sinistro

Template:

```
Successor(x)
1.  if x.right ≠ nil                    // caso 1: ha figlio destro
2.      return Minimum(x.right)
3.  y = x.p                             // caso 2: risali
4.  while y ≠ nil and x = y.right
5.      x = y
6.      y = y.p
7.  return y
```

Complessità: O(h)

2.6 Insert (Inserimento)

Idea: Scendi cercando la posizione (come Search), poi inserisci come foglia.

Template Standard:

```
Insert(T, z)                            // T è l'albero, z è il nuovo nodo
1.  x = T.root                          // inizia dalla radice
2.  y = nil                             // y sarà il padre di z
3.  while x ≠ nil                       // cerca posizione
4.      y = x                           // salva padre candidato
5.      if z.key < x.key
6.          x = x.left
7.      else
8.          x = x.right
9.  z.p = y                             // collega z al padre y
10. if y = nil                          // albero vuoto
11.     T.root = z
12. else if z.key < y.key
13.     y.left = z
14. else
15.     y.right = z
```

Complessità: O(h)

Invariante while (righe 3-8):

- y è sempre il padre del nodo x corrente
- Alla fine del ciclo, y è il padre dove inserire z

2.7 Delete (Cancellazione) - Casi

Casi di cancellazione:

1. **z ha 0 figli** (foglia): rimuovi direttamente
2. **z ha 1 figlio**: sostituisci z con l'unico figlio
3. **z ha 2 figli**:
   - Trova y = Successor(z) (che avrà al più 1 figlio destro)
   - Sostituisci z con y
   - Rimuovi y dalla sua posizione originale

Funzione Ausiliaria Transplant (sostituisce u con v):

```
Transplant(T, u, v)
1.  if u.p = nil                        // u è radice
2.      T.root = v
3.  else if u = u.p.left               // u è figlio sinistro
4.      u.p.left = v
5.  else                                // u è figlio destro
6.      u.p.right = v
7.  if v ≠ nil
8.      v.p = u.p
```

Template Delete Completo:

```
Delete(T, z)
1.  if z.left = nil                     // caso 1: 0 figli o solo destro
2.      Transplant(T, z, z.right)
3.  else if z.right = nil               // caso 2: solo figlio sinistro
4.      Transplant(T, z, z.left)
5.  else                                // caso 3: 2 figli
6.      y = Minimum(z.right)            // trova successore
7.      if y.p ≠ z                      // successore non è figlio diretto
8.          Transplant(T, y, y.right)  // rimuovi y, sostituisci con suo
destro
9.          y.right = z.right
10.         y.right.p = y
11.     Transplant(T, z, y)             // sostituisci z con y
```

```
12.     y.left = z.left
13.     y.left.p = y
```

Complessità: O(h)

2.8 Costruire BST Bilanciato da Array Ordinato

Quando usare: Dato A[1..n] ordinato, costruire BST di altezza minima.

Template Divide-et-Impera:

```
BST(A)
1.  return BST-Rec(A, 1, A.length)

BST-Rec(A, p, r)
1.  if p > r
2.      return nil                    // caso base: intervallo vuoto
3.  m = ⌊(p + r)/2⌋                  // mediano
4.  x = MkNode(A[m])                  // crea nodo con chiave A[m]
5.  x.left = BST-Rec(A, p, m-1)       // ricorsione su metà sinistra
6.  x.right = BST-Rec(A, m+1, r)      // ricorsione su metà destra
7.  return x
```

Complessità: T(n) = 2T(n/2) + Θ(1) → T(n) = Θ(n) (Master Theorem caso 1)

2.9 Checklist BST

- [ ] Proprietà BST rispettata in tutte le operazioni?
- [ ] Insert: scendi fino a trovare nil, poi inserisci come foglia
- [ ] Delete: gestiti correttamente i 3 casi (0, 1, 2 figli)?
- [ ] Visite ricorsive: ordine corretto (pre/in/post)?
- [ ] Complessità O(h): esplicitato h = altezza albero?
- [ ] Dimostrazione correttezza per induzione su altezza/dimensione sottoalbero

---

3. ARRAY: Invarianti di Ciclo e Manipolazione {#array}

3.1 Principi Fondamentali

Invariante di ciclo: proprietà che rimane vera ad ogni iterazione.

Schema di dimostrazione:

1. **Inizializzazione**: invariante vera prima del primo ciclo
2. **Mantenimento**: se invariante vera prima iterazione i, è vera dopo iterazione i
3. **Conclusione**: quando ciclo termina, invariante + condizione uscita → correttezza

---

3.2 Insertion Sort - Esempio Canonico

Idea: Mantieni A[1..j-1] ordinato, inserisci A[j] nella posizione corretta.

Template:

```
InsertionSort(A)
1.  for j = 2 to A.length
2.      key = A[j]                      // elemento da inserire
3.      i = j - 1                       // posizione precedente
4.      while i > 0 and A[i] > key
5.          A[i+1] = A[i]               // sposta a destra
6.          i = i - 1
7.      A[i+1] = key                    // inserisci key
```

Invariante (riga 1, inizio iterazione j):

- A[1..j-1] contiene gli elementi originali di A[1..j-1] ma in ordine crescente

Dimostrazione:

- **Inizio**: j = 2 → A[1..1] ordinato (singolo elemento)
- **Mantenimento**: A[1..j-1] ordinato, inserisci A[j] → A[1..j] ordinato
- **Conclusione**: j = n+1 → A[1..n] ordinato

Complessità:

- Caso migliore (array già ordinato): $\Theta(n)$
- Caso peggiore (array ordinato al contrario): $\Theta(n^2)$

3.3 Problemi Comuni su Array

3.3.1 Ricerca Elemento

Ricerca lineare:

```
LinearSearch(A, x)
1.  for i = 1 to A.length
2.      if A[i] = x
```

```
3.          return i
4.  return nil
```

Complessità: O(n)

Ricerca binaria (se array ordinato):

```
BinarySearch(A, x, p, r)
1.  if p > r
2.      return nil                      // intervallo vuoto
3.  q = ⌊(p + r)/2⌋
4.  if A[q] = x
5.      return q
6.  else if x < A[q]
7.      return BinarySearch(A, x, p, q-1)
8.  else
9.      return BinarySearch(A, x, q+1, r)
```

Complessità: O(log n)

3.3.2 Manipolazione Indici - Pattern Ricorrenti

Pattern "Two Pointers":

- Indice sinistro `i`, indice destro `j`
- Scorrono verso centro o in direzioni opposte
- Esempio: Partition di QuickSort

Pattern "Sliding Window":

- Indici `p` e `r` delimitano finestra
- Finestra scorre/espande su array
- Esempio: somma sottointervalli

Pattern "In-place Rearrangement":

- Riorganizza array senza spazio extra
- Usa scambi A[i] ↔ A[j]
- Esempio: separare pari/dispari

3.3.3 Trovare Elemento con Proprietà

Template generico:

```
FindProperty(A, p, r)
1.  for i = p to r
2.      if <condizione su A[i]>
3.          return i                    // o A[i], dipende dal problema
4.  return <valore di errore>
```

Invariante tipico:

- A[p..i-1] non contiene elementi con la proprietà cercata
- Se esiste elemento con proprietà in A[p..r], sarà trovato

3.4 Checklist Array

- [ ] Invariante di ciclo esplicitato (inizializzazione, mantenimento, conclusione)?
- [ ] Gestiti correttamente i casi limite (array vuoto, singolo elemento)?
- [ ] Indici sempre validi (1 ≤ i ≤ n, attenzione a i-1 e i+1)?
- [ ] Se ordini/modifichi array, è richiesto ordine stabile?
- [ ] Complessità: contato numero iterazioni e costo operazioni interne?

---

4. DIVIDE-ET-IMPERA: Schema e Varianti di Partizione {#divide-et-impera}

4.1 Schema Generale

Paradigma:

1. **Divide**: Dividi problema in sottoproblemi più piccoli
2. **Impera**: Risolvi ricorsivamente i sottoproblemi (caso base: problema piccolo, soluzione diretta)
3. **Combina**: Unisci soluzioni sottoproblemi per ottenere soluzione problema originale

Template Pseudocodice:

```
DivideEtImpera(A, p, r)
1.  if <caso base>                      // problema piccolo
2.      return <soluzione diretta>
3.  <dividi problema in sottoproblemi>
4.  sol1 = DivideEtImpera(sottoprob1)
5.  sol2 = DivideEtImpera(sottoprob2)
6.  ...
7.  return <combina sol1, sol2, ...>
```

4.2 MergeSort - Esempio Canonico

Idea: Dividi array a metà, ordina ricorsivamente, fondi (merge) risultati.

Template:

```
MergeSort(A, p, r)
1.  if p < r                          // almeno 2 elementi
2.      q = ⌊(p + r)/2⌋               // divide
3.      MergeSort(A, p, q)            // ordina metà sinistra
4.      MergeSort(A, q+1, r)          // ordina metà destra
5.      Merge(A, p, q, r)             // fondi
```

Merge (fusione): Unisce A[p..q] e A[q+1..r] (già ordinati) in A[p..r] ordinato.

```
Merge(A, p, q, r)
1.  n1 = q − p + 1
2.  n2 = r − q
3.  crea array L[1..n1+1] e R[1..n2+1]
4.  for i = 1 to n1
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n2
7.      R[j] = A[q + j]
8.  L[n1+1] = ∞                       // sentinelle
9.  R[n2+1] = ∞
10. i = 1
11. j = 1
12. for k = p to r
13.     if L[i] ≤ R[j]
14.         A[k] = L[i]
15.         i = i + 1
16.     else
17.         A[k] = R[j]
18.         j = j + 1
```

Complessità:

- Ricorrenza: $T(n) = 2T(n/2) + \Theta(n)$
- Master Theorem caso 2: $T(n) = \Theta(n \log n)$

Correttezza: Induzione su dimensione array.

- Caso base (n=1): array già ordinato
- Passo induttivo: se MergeSort ordina array di dimensione < n, allora ordina array di dimensione n (divide in 2 parti < n, ordina ricorsivamente, Merge preserva ordine)

4.3 QuickSort - Partizione 2-Way (Standard)

Idea: Scegli pivot, partiziona in ≤pivot e >pivot, ordina ricorsivamente.

Template:

```
QuickSort(A, p, r)
1.  if p < r
2.      q = Partition(A, p, r)          // partiziona
3.      QuickSort(A, p, q-1)            // ordina parte sinistra
4.      QuickSort(A, q+1, r)            // ordina parte destra
```

Partition 2-Way (pivot = A[p], primo elemento):

```
Partition(A, p, r)
1.  x = A[p]                            // pivot
2.  i = p - 1                           // indice parte ≤ pivot
3.  j = r + 1                           // indice parte > pivot
4.  while true
5.      repeat
6.          j = j - 1
7.      until A[j] ≤ x                  // trova elemento ≤ pivot da destra
8.      repeat
9.          i = i + 1
10.     until A[i] ≥ x                  // trova elemento ≥ pivot da sinistra
11.     if i < j
12.         A[i] ⇔ A[j]                 // scambia
13.     else
14.         return j                    // j è la posizione finale del pivot
```

Invariante Partition (durante while):

- A[p..i] contiene elementi ≤ x
- A[j..r] contiene elementi ≥ x
- Elementi in A[i+1..j-1] non ancora classificati

Complessità QuickSort:

- Caso peggiore: $T(n) = T(n-1) + \Theta(n) \rightarrow T(n) = \Theta(n^2)$ (partizioni sbilanciate)
- Caso medio/migliore: $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \log n)$

Ottimizzazione: Randomized pivot (scegli casualmente pivot in [p,r])

```
RandomizedPartition(A, p, r)
1.  i = Random(p, r)                    // indice casuale
2.  A[p] ⇔ A[i]                         // porta pivot random in prima
```

```
posizione
3.  return Partition(A, p, r)          // usa Partition standard
```

4.4 QuickSort - Partizione 3-Way (Tripartition)

Quando usare: Array con molti elementi duplicati.

Idea: Partiziona in <pivot, =pivot, >pivot.

Template 3-Way Partition:

```
Partition3Way(A, p, r)
1.  x = A[p]                           // pivot
2.  i = p                              // fine zona < pivot
3.  j = p                              // inizio zona non analizzata
4.  k = r                              // inizio zona > pivot
5.  while j ≤ k
6.      if A[j] < x
7.          A[i] ↔ A[j]
8.          i = i + 1
9.          j = j + 1
10.     else if A[j] > x
11.         A[j] ↔ A[k]
12.         k = k − 1
13.     else                           // A[j] = x
14.         j = j + 1
15. return (i, j)                      // ritorna intervallo zona =pivot
```

Risultato: A[p..i-1] < x, A[i..j-1] = x, A[j..r] > x

QuickSort con 3-Way:

```
QuickSort3Way(A, p, r)
1.  if p < r
2.      (i, j) = Partition3Way(A, p, r)
3.      QuickSort3Way(A, p, i−1)       // ordina < pivot
4.      QuickSort3Way(A, j, r)         // ordina > pivot (= pivot già al
posto)
```

Vantaggio: Evita caso pessimo O(n²) su array con tutti elementi uguali.

4.5 Select (Selezione k-esimo elemento) - QuickSelect

Problema: Dato A[1..n] e k, trova elemento che sarebbe in posizione k se A fosse ordinato.

Idea: Usa Partition, poi ricorri solo sulla parte che contiene k.

Template:

```
QuickSelect(A, p, r, k)
1.  if p = r                          // caso base: 1 elemento
2.      return A[p]
3.  q = Partition(A, p, r)            // partiziona
4.  if k = q                          // pivot è k-esimo
5.      return A[q]
6.  else if k < q
7.      return QuickSelect(A, p, q-1, k)  // k è a sinistra
8.  else
9.      return QuickSelect(A, q+1, r, k)  // k è a destra
```

Complessità:

- Caso peggiore: $T(n) = T(n-1) + \Theta(n) \rightarrow T(n) = \Theta(n^2)$
- Caso medio: $T(n) = T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n)$ (Master Theorem caso 3)

Nota: Esiste Select deterministico in O(n) worst-case (mediana delle mediane), ma non richiesto nel corso.

4.6 Casi Tipici Divide-et-Impera

4.6.1 Ricerca in Array Ordinato

Esempio: Trova elemento > x in array ordinato.

```
FindGreater(A, p, r, x)
1.  if p > r
2.      return r + 1                  // nessun elemento > x
3.  q = ⌊(p + r)/2⌋
4.  if A[q] ≤ x
5.      return FindGreater(A, q+1, r, x)  // cerca a destra
6.  else
7.      return FindGreater(A, p, q-1, x)  // cerca a sinistra
```

Ricorrenza: $T(n) = T(n/2) + O(1) \rightarrow T(n) = O(\log n)$

4.6.2 Trovare Elemento con Proprietà

Esempio: Gap in array (A[i+1] - A[i] > 1).

Precondizione: A[r] - A[p] ≥ r - p (garantisce esistenza gap)

```
FindGap(A, p, r)
1.  if p = r - 1
2.      return p                        // caso base: 2 elementi, p è gap
3.  q = ⌊(p + r)/2⌋
4.  if A[q] - A[p] > q - p              // gap a sinistra
5.      return FindGap(A, p, q)
6.  else                                // gap a destra
7.      return FindGap(A, q, r)
```

Ricorrenza: T(n) = T(n/2) + O(1) → T(n) = O(log n)

4.6.3 Massimo/Minimo

```
Max(A, p, r)
1.  if p = r
2.      return A[p]                     // caso base
3.  q = ⌊(p + r)/2⌋
4.  m1 = Max(A, p, q)                   // max sinistra
5.  m2 = Max(A, q+1, r)                 // max destra
6.  return max(m1, m2)                  // combina
```

Ricorrenza: T(n) = 2T(n/2) + O(1) → T(n) = O(n) (Master Theorem caso 1)

Nota: Soluzione non ottimale (lineare iterativo è più semplice), ma dimostra schema D&C.

4.7 Master Theorem - Analisi Complessità

Forma ricorrenza: T(n) = aT(n/b) + f(n)

Dove:

- a = numero sottoproblemi
- n/b = dimensione ciascun sottoproblema
- f(n) = costo divide + combina

Casi:

1. **Caso 1**: f(n) = O(n^(log*b(a) - ε)) per ε > 0*
   → T(n) = Θ(n^log_b(a))
   _Dominano le foglie della ricorsione

2. **Caso 2**: f(n) = Θ(n^log*b(a)*)
   → *T(n) = Θ(n^log_b(a) · log n)*
   _Bilancio tra foglie e livelli interni
3. **Caso 3**: f(n) = Ω(n^(log*b(a) + ε)) per ε > 0, e af(n/b) ≤ kf(n) per k < 1*
   → *T(n) = Θ(f(n))*
   _Domina il costo divide/combina

Esempi Applicazione:

Ricorrenza	a	b	f(n)	log_b(a)	Caso	Soluzione
T(n) = 2T(n/2) + O(n)	2	2	n	1	2	Θ(n log n)
T(n) = 2T(n/2) + O(1)	2	2	1	1	1	Θ(n)
T(n) = T(n/2) + O(1)	1	2	1	0	2	Θ(log n)
T(n) = T(n/2) + O(n)	1	2	n	0	3	Θ(n)
T(n) = 4T(n/2) + O(n²)	4	2	n²	2	2	Θ(n² log n)

4.8 Checklist Divide-et-Impera

- [ ] Caso base esplicito e corretto (problema piccolo → soluzione diretta)?
- [ ] Divisione problema: sottoproblemi indipendenti?
- [ ] Combinazione soluzioni: preserva correttezza?
- [ ] Ricorrenza esplicitata: T(n) = ... ?
- [ ] Complessità calcolata con Master Theorem (se applicabile)?
- [ ] Correttezza dimostrata per induzione su dimensione input?
- [ ] Gestiti correttamente indici array (p, q, r)?

---

5. STRATEGIE GENERALI DI RISOLUZIONE

5.1 Flusso di Lavoro Standard

Prima di scrivere pseudocodice:

1. **Comprendi il problema**:
   - Input: cosa ricevi? Formato, vincoli, precondizioni?
   - Output: cosa devi restituire? Formato, postcondizioni?
   - Proprietà: cosa deve essere vero alla fine?

2. **Identifica paradigma**:
   - Operazioni su heap/BST → usa template strutture dati
   - Ordinamento/ricerca → array con invarianti
   - Divisione problema → divide-et-impera
   - Ottimizzazione su sequenza → (greedy/DP, non trattati qui)
3. **Progetta invariante/schema ricorsivo**:
   - Cicli → invariante (cosa è vero ad ogni iterazione?)
   - Ricorsione → caso base + passo induttivo
4. **Scrivi pseudocodice**:
   - Inizia da template appropriato
   - Adatta alle specifiche del problema
   - Numera le righe (1, 2, 3, ...)
5. **Dimostra correttezza**:
   - Invarianti: inizializzazione, mantenimento, conclusione
   - Ricorsione: induzione su dimensione/altezza
6. **Calcola complessità**:
   - Conta operazioni elementari
   - Cicli: sommatorie
   - Ricorsione: ricorrenza + Master Theorem

5.2 Errori Comuni da Evitare

Heap:

- ❌ Dimenticare di controllare i ≤ A.heapsize prima di accedere A[i]
- ❌ BuildMaxHeap parte da n invece di ⌊n/2⌋
- ❌ MaxHeapify chiamato su nodo che non ha sottoalberi heap

BST:

- ❌ Dimenticare caso albero vuoto (T.root = nil)
- ❌ Insert non collega x.p (puntatore padre)
- ❌ Delete non gestisce tutti e 3 i casi

Array:

- ❌ Indici fuori range (i = 0 o i = n+1 senza controllo)
- ❌ Invariante non preservato dopo modifica array
- ❌ Dimenticare caso array vuoto o singolo elemento

Divide-et-Impera:

- ❌ Caso base mancante o errato
- ❌ Ricorsione su problema di dimensione uguale (loop infinito)
- ❌ Combinazione soluzioni non corretta
- ❌ Indici p, q, r confusi (p ≤ q < r?)

5.3 Schema di Correttezza per Induzione

Per ricorsione (usato in BST, D&C):

Teorema: `<funzione>(input di dimensione n)` produce output corretto.

Dimostrazione per induzione su n:

- **Caso base** (n = n_0, solitamente 0 o 1):
  - Mostra che per input più piccolo, output è corretto direttamente
- **Passo induttivo** (n > n_0):
  - **Ipotesi induttiva**: assume funzione corretta per input dimensione < n
  - **Tesi**: dimostra funzione corretta per input dimensione n
  - Usa ipotesi induttiva sulle chiamate ricorsive + logica locale

Esempio BST Insert:

- Caso base: albero vuoto → inserisci come radice (corretto)
- Passo induttivo: albero non vuoto di altezza h
  - Ipotesi: Insert corretto su alberi altezza < h
  - Inserisco z: scendo ricorsivamente (altezza diminuisce), poi collego z come foglia
  - Per ipotesi induttiva, sottoinserimento corretto → inserimento corretto

---

Per cicli (usato in array, heap):

Invariante: <proprietà P> è vera all'inizio di ogni iterazione.

Dimostrazione:

1. **Inizializzazione**: P vera prima primo ciclo (i = i_0)
2. **Mantenimento**: se P vera prima iterazione i, è vera prima iterazione i+1
3. **Conclusione**: quando ciclo termina, P + condizione uscita → proprietà desiderata

Esempio InsertionSort:

- Invariante: A[1..j-1] ordinato
- Inizio: j = 2 → A[1..1] ordinato (1 elemento)

- Mantenimento: se A[1..j-1] ordinato, dopo inserimento A[j] → A[1..j] ordinato
- Conclusione: j = n+1 → A[1..n] ordinato

5.4 Prompt per Autovalutazione

Prima di consegnare, chiediti:

Correttezza:

- [ ] Ho dimostrato correttezza (induzione/invariante)?
- [ ] Ho gestito tutti i casi limite (vuoto, singolo elemento, ...)?
- [ ] Precondizioni rispettate? Postcondizioni garantite?

Complessità:

- [ ] Ho calcolato complessità tempo? È richiesta?
- [ ] Ho usato Master Theorem (se D&C)? Caso corretto?
- [ ] Complessità spazio (se richiesta)? Array ausiliari?

Stile accademico:

- [ ] Pseudocodice numerato (1, 2, 3, ...)?
- [ ] Nomi variabili chiari (p, r per indici; x, y per nodi; ...)?
- [ ] Commenti essenziali (no ovvietà, sì chiarimenti)?
- [ ] Utilizzo template standard del corso (non "overengineering")?

---

6. RIEPILOGO COMPLESSITÀ STANDARD

Operazione	Struttura	Complessità
MaxHeapify	Heap	O(log n)
BuildMaxHeap	Heap	O(n)
HeapSort	Heap	O(n log n)
Extract-Max	Heap	O(log n)
Insert (Heap)	Heap	O(log n)
Search	BST	O(h)
Insert	BST	O(h)
Delete	BST	O(h)
InOrder	BST	Θ(n)

Operazione	Struttura	Complessità
InsertionSort	Array	$O(n^2)$
MergeSort	Array	$\Theta(n \log n)$
QuickSort (medio)	Array	$\Theta(n \log n)$
QuickSort (pessimo)	Array	$\Theta(n^2)$
QuickSelect (medio)	Array	$\Theta(n)$
BinarySearch	Array ord	$O(\log n)$

Note:

- h = altezza albero (O(log n) se bilanciato, O(n) se degenere)
- Θ = tight bound (caso migliore = caso peggiore)
- O = upper bound (caso peggiore)

Fine della Guida

Appendice: Template Vuoti da Completare

Heap - Operazione Custom

```
<NomeOperazione>(A, <parametri>)
1.  // Inizializzazione variabili
2.
3.  // Corpo algoritmo
4.
5.  // Return/modifica
```

Correttezza: <invariante/induzione>

Complessità: O(...)

BST - Operazione Custom

```
<NomeOperazione>(T, <parametri>)
1.  // Caso base
2.  if <condizione vuoto>
3.      return <valore base>
4.  // Caso ricorsivo
```

```
5.  <logica>
6.  return <risultato>
```

Correttezza: Induzione su altezza/dimensione sottoalbero

Complessità: O(h) o Θ(n)

Array - Ciclo con Invariante

```
<NomeFunzione>(A, <parametri>)
1.  // Inizializzazione
2.
3.  for i = <start> to <end>
4.      // Corpo ciclo
5.      <operazioni>
6.
7.  return <risultato>
```

Invariante: All'inizio iterazione i, <proprietà P>

Complessità: O(n) o O(n^2)

Divide-et-Impera

```
<NomeFunzione>(A, p, r)
1.  if <caso base>
2.      return <soluzione diretta>
3.  q = <divisione>
4.  sol1 = <NomeFunzione>(A, p, q)
5.  sol2 = <NomeFunzione>(A, q+1, r)
6.  return <combina sol1, sol2>
```

Ricorrenza: T(n) = aT(n/b) + f(n)

Complessità: Θ(...) (Master Theorem caso ...)
