Binary search

- Problema:
  - input: un array *ordinato* A, ed un elemento x da cercare
  - output: se $x \in A$, un indice i tale che $A[i] = x$, altrimenti *NIL*
- parametri:
  - A e x come sopra
  - p e q sono i limiti del sottoarray $A[p..q]$ di A in cui cercare x

```
BIN-SEARCH(A, x, p, q)
if q < p
  return NIL
i := ⌊(q + p)/2⌋
if A[i] = x
  return i
elseif A[i] > x
  return BIN-SEARCH(A, x, p, i-1)
else return BIN-SEARCH(A, x, i+1, q)
```

- Ricorrenza per `BIN-SEARCH`:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n < 1 \\ T(n/2) + \Theta(1) & \text{altrimenti} \end{cases}$$

Binary search (iterativo)

```
BIN-SEARCH_IT(A, x)
p := 1
q := A.length
while p ≤ q
  i := ⌊(q + p)/2⌋
  if A[i] = x
    return i
  elseif A[i] > x
    q := i-1
  else p := i+1
return NIL
```

Ricerca lineare

- Problema:
  - input: un arrany *A non ordinato* e un elemento *x* da cercare
  - output: se $x \in A$, un indice *i* tale che $A[i] = x$, altrimenti *NIL*

```
LIN-SEARCH(A, x)
for i := 1 to A.length
  if A[i] = x
    return i
  else i := i+1
return NIL
```

Algoritmo `EXACT-SUM`

- Problema:
  - *input*: un array A di valori (interi) e un valore x
  - *output*: *true* se ci sono 2 indici, i e j, in A, tali che $i \neq j$ e $A[i] + A[j] = x$, altrimenti *false*

```
EXACT-SUM(A, x)
  MERGE-SORT(A, 1, A.length)
  i := 1
  while i ≤ A.length and A[i] ≤ x
    i := i+1
  i := i-1
  if i < 1
    return false
  j := 1
  while j < i
    if A[i] + A[j] = x
      return true
    elseif A[i] + A[j] < x
      j := j+1
    else i := i-1
  return false
```

Algoritmo N_INV

- Problema:
  - *input*: un sottoarray di elementi $A[p..r]$
  - *output*: il numero di inversioni in A[p..r] (laddove per *inversione* si intende una coppia i, j di indici tali che $i < j$ e $A[i] > A[j]$)

```
N_INV(A, p, r)
  if r ≤ p
    return 0
  q := ⌊(p + r)/2⌋
  n_inv := N_INV(A,p,q)+N_INV(A,q+1,r)
  n_inv := n_inv + MERGE-INV(A,p,q,r)
  return n_inv
```

Algoritmo MERGE-INV

- Problema:
  - *input*: 2 array *ordinati* $A[p..q]$ e $A[q+1..r]$
  - *output*: l'array ordinato $A[p..r]$ ottenuto fondendo i 2 array di input, e il numero di inversioni tra gli elementi degli array

```
MERGE-INV (A, p, q, r)
n1 := q – p + 1
n2 := r – q
crea array L[1..n1+1] e R[1..n2+1]
for i := 1 to n1
  L[i] := A[p + i - 1]
for j := 1 to n2
  R[i] := A[q + j]
L[n1 + 1] := ∞
R[n2 + 1] := ∞
i := 1
j := 1
n_inv := 0
for k := p to r
  if L[i] ≤ R[j]
    A[k] := L[i]
    i := i + 1
  else A[k] := R[j]
    j := j + 1
    n_inv := n_inv + n1 – i + 1
return n_inv
```

Algoritmo `FIND-MAXIMUM-SUBARRAY`

- Problema:
  - *input*: un sottoarray di interi *A*[*low*..*high*]
  - *output*: il sottoarray la cui somma degli elementi è massima

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high = low
  return (low, high, A[low])
else mid := ⌊(low + high)/2⌋
  (left_low, left_high, left_sum) :=
      FIND-MAXIMUM-SUBARRAY(A,low,mid)
  (right_low, right_high, right_sum) :=
      FIND-MAXIMUM-SUBARRAY(A,mid+1,high)
  (cross_low, cross_high, cross-sum) :=
      FIND-MAX-CROSSING-SUBARRAY(A,low,mid,high)
  if left_sum ≥ right_sum and left_sum ≥ cross_sum
    return (left_low, left_high, left_sum)
  elseif right_sum ≥ cross_sum
    return (right_low, right_high, right_sum)
  else return (cross_low, cross_high, cross_sum)
```

Algoritmo `FIND-MAX-CROSSING-SUBARRAY`

- Problema:
  - *input*: due sottoarray contigui di interi $A[low..mid]$ e $A[mid+1..high]$
  - *output*: il sottoarray a cavallo dei 2 in input la cui somma degli elementi è massima

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    left_sum := -∞
    sum := 0
    for i := mid downto low
      sum := sum + A[i]
      if sum > left_sum
        left_sum := sum
        max_left := i
    right_sum := -∞
    sum := 0
    for j := mid+1 to high
      sum := sum + A[j]
      if sum > right_sum
        right_sum := sum
        max_right := j
    return (max_left, max_right, left_sum + right_sum)
```

- Complessità $T(n) = \Theta(n)$
- Ricorrenza per `FIND-MAXIMUM-SUBARRAY`:

$$T(n)=\begin{cases}\Theta(1) & \text{se } n<2\\ 2T(n/2)+\Theta(n) & \text{altrimenti}\end{cases}$$

- Quindi $T_{\texttt{FIND-MAXIMUM-SUBARRAY}}=\Theta(n\log(n))$

Algoritmo FIND-MAX-SUBARRAY-LIN

```
FIND-MAX-SUBARRAY-LIN(A)
s := e := 1
tot := A[1]
s_crs := 1
tot_crs := A[1]
for i := 2 to A.length
  if tot_crs+A[i] > tot
    s := s_crs
    e := i
    tot := tot_crs+A[i]
  if tot_crs ≤ 0
    s_crs := i
    tot_crs := A[i]
  else tot_crs := tot_crs + A[i]
return (s,e,tot)
```

- Complessità $T(n) = \Theta(n)$

Elemento di mezzo di una lista

- Problema:
  - *input*: la testa (*h*) e la coda (*t*) di una lista doppiamente concatenata
  - *output*: l'elemento di mezzo della lista

```
LIST-MIDDLE (h, t)
if h = t or h.next = t
  return h
return LIST-MIDDLE (h.next, t.prev)
```

- Versione alternativa:
  - *input*: la testa (*h*) di una lista doppiamente concatenata e il numero *n* dei suoi elementi
  - *output*: l'elemento di mezzo della lista

```
LIST-MIDDLE2 (h, n)
curr := h
for i := 1 to ⌈n / 2⌉ -1
  curr := curr.next
return curr
```

- Entrambi gli algoritmi sono $\Theta(n)$

Ricerca binaria in una lista

```
BIN-LIST-SEARCH (L, k)
if L.head = NIL
  return NIL
h := L.head
t := L.head
while t.next ≠ NIL
  t := t.next
return BIN-SUBLIST-SEARCH(h, t, k)
```

- dove l'algoritmo `BIN-SUBLIST-SEARCH` prende come input la testa e la coda di una lista (e la chiave da cercare) ed è il seguente:

```
BIN-SUBLIST-SEARCH(h, t, k)
m := LIST-MIDDLE(h, t)
if m = NIL or m.key = k
  return m
if k < m.key
  return BIN-SUBLIST-SEARCH(h,m.prev,k)
else return BIN-SUBLIST-SEARCH(m.next,t,k)
```

- $T_{\text{BIN-LIST-SEARCH}} = O(n)$

INSERTION-SORT in una lista doppiamente concatenata

```
LIST-INSERTION-SORT(L)
1 if L.head ≠ NIL
2   curr := L.head.next
3   while curr ≠ NIL
4     i := curr
5     while i.prev ≠ NIL and
            i.prev.key > i.key
6       swap i.key ↔ i.prev.key
7       i := i.prev
8       curr := curr.next
```

- E' di fatto lo stesso algoritmo di INSERTION-SORT, quindi $T_{\text{LIST-INSERTION-SORT}}$ è $O(n^2)$

- Se la lista fosse singolarmente concatenata, il ciclo **while** delle linee 5-7 dovrebbe essere modificato, in quanto no potremmo muoverci “all'indietro” lungo la lista, quindi per trovare il “posto giusto” per l'elemento corrente dovremmo ogni volta partire da *L.head*; comunque, la complessità del ciclo sarebbe ancora $O(n)$, e la complessità globale sarebbe ancora $O(n^2)$

MERGE-SORT in una lista doppiamente concatenata

```
LIST-MERGE-SORT(L)
1 if L.head ≠ NIL
2   h := L.head
3   t := L.head
4   while t.next ≠ NIL
5     t := t.next
6   SUBLIST-MERGE-SORT(h, t)
```

- dove SUBLIST-MERGE-SORT è il seguente

```
SUBLIST-MERGE-SORT(h, t)
1  if h ≠ t
2    m := LIST-MIDDLE(h, t)
3    SUBLIST-MERGE-SORT(h, m)
4    SUBLIST-MERGE-SORT(m.next, t)
5    SUBLIST-MERGE(h, m, t)
```

- SUBLIST-MERGE può essere fatto in modo da avere complessità temporale $\Theta(n)$ (esercizio per casa), quindi di fatto l'unica differenza tra MERGE-SORT e SUBLIST-MERGE-SORT è che il tempo $D(n)$ che serve per dividere il problema in problemi più piccoli (cioè la linea 2) è ora $\Theta(n)$ invece di $\Theta(1)$, ma il termine $C(n) + D(n)$ nella ricorrenza è ancora $\Theta(n)$ ($C(n)$, il tempo per ricombinare le soluzioni dei sottoproblemi è ed era $\Theta(n)$), quindi la complessità globale rimane la stessa

Esercizio R.12

```
SEARCH(T, k)
SEARCH-NODE(T.root, k)

SEARCH-NODE(x, k)
if x = NIL or k = x.key
  return x
if k < x.key
   return SEARCH-NODE(x.fc, k)
else return SEARCH-NODE(x.rs, k)

INSERT(T, n)
 y := NIL
 x := T.root
 while x ≠ NIL
   y := x
   if n.key < x.key
     x := x.fc
   else x := x.rs
 n.p := y
 if y = NIL
   T.root := n     // L'albero era vuoto
elsif n.key < y.key
   y.fc := n
elsif y.p = NIL  //creo una nuova radice
   n.fc := y
   T.root := n
   y.p := n
else
   y.rs := n
   n.ls := y
   n.p := y.p
```

Esercizio R.17

```
Set_Depth_LR(N)
if N = NIL
  return -1
N.depth_L := Set_Depth_LR(N.left)+1
N.depth_R := Set_Depth_LR(N.right)+1
return max(N.depth_L, N.depth_R)

Set_Depth_PMAX(N, d)
if N ≠ NIL
  N.depth_P := d
  N.depth_MAX := max(N.depth_L, N.depth_R,
                     N.depth_P)
  Set_Depth_PMAX(N.left, max(d,N.depth_R)+1)
  Set_Depth_PMAX(N.right, max(d,N.depth_L)+1)

Search_Min_Depth(N)
if N = NIL
  return NIL
l := Search_Min_Depth(N.left)
r := Search_Min_Depth(N.right)
min := N
if (l ≠ NIL and l.depth_MAX < min.depth_MAX)
  min := l
if (r ≠ NIL and r.depth_MAX < min.depth_MAX)
  min := r
return min
```

Esercizio NR.13

```
TreeSearchUpper(T,v)
 if T.root = NIL
   return NIL
 TreeFindNextGE(T.root,v,NIL)

TreeFindNextGE(N, k, NGE)
if N = NIL
  return NGE
if N.key = k
  return N
elsif N.key > k
  return TreeFindNextGE(N.left,k,N)
else return TreeFindNextGE(N.right,k,NGE)
```

- Complessità O(h), che nel caso di albero bilanciato è O(log n)

Esercizio NR.9

```
TreeInterval(T,L,U)
N := TREE-SEARCH(T.root,L)
while N ≠ NIL and N.key ≤ U
  print N.key
  N := TreeIntervalWalkUpTo(N.right, U, N)
  N := TREE-SUCCESSOR(N)
```

```
TreeIntervalWalkUpTo(N, U, last)
if N = NIL
  return last
last := TreeIntervalWalkUpTo(N.left, U, last)
if N.key ≤ U
  print N.key
  return TreeIntervalWalkUpTo(N.right, U, N)
else return last
```

- La complessità di `TreeIntervalWalkUpTo` è O(n'), con *n'* numero dei nodi del sottoalbero di radice *N*. Diverse invocazioni di `TreeIntervalWalkUpTo` interessano sottoalberi diversi, ed al massimo il numero di nodi vistato dalle loro invocazioni è $l = U - L + 1$
- A ogni invocazione di `TREE-SUCCESSOR` si risale nell'albero di almeno un livello (perché, se non siamo ancora arrivati ad *U*, l'ultimo nodo visitato da `TreeIntervalWalkUpTo` è il massimo del sottoalbero di radice *N*, il cui successore è un antenato di *N*
- In totale la complessità è $O((\log n)^2 + l)$
- In realtà si può anche fare in tempo $O((\log n) + l)$

OS-RANK

```
OS-RANK(T, x)
r := x.left.size + 1
y := x
while y ≠ T.root
  if y = y.p.right
    r := r + y.p.left.size + 1
  y := y.p
return r
```

Esercizio R.7

```
CamminoNonProibito(G, s, t, P)
for each vertex u ∈ G.V – {s}
  u.color := WHITE
  u.dist := ∞
s.color := GREY
s.dist := 0
for i := 1 to P.length
  P[i].color := RED
ENQUEUE(Q, s)
while Q ≠ ∅ do
  u := DEQUEUE(Q)
  for each v ∈ u.Adj
    if v = t
      return u.dist + 1
    else if v.color = WHITE
      v.color := GRAY
      v.dist := u.dist +1
      ENQUEUE(Q, v)
  u.color := BLACK
return ∞
```

Esercizio R.13

```
Buoni-cattivi(G)
for each vertex u ∈ G.V
  u.color := UNDEF
for each vertex u ∈ G.V
    if u.color = UNDEF
    ris := visita_e_partiziona(u, B)
    if ris = false
      return false
return true
```

```
visita_e_partiziona(u, c)
u.color := c
for each v ∈ u.Adj
  if v.color = U
    if c = B
      ris := visita_e_partiziona(v, C)
    else ris := visita_e_partiziona(v, B)
    if ris = false
      return ris
  else if v.color = c
    return false
return true
```