Algoritmi trattati a lezione

Regole: Si ammette questo documento all'esame; è consentito arricchire questo documento con appunti personali. Si ammette una copia di questo documento per studente; **non** si ammette in nessun caso la condivisione dello stesso tra diversi candidati. Inoltre, non si ammettono fotocopie di questo documento che riportino appunti personali di altre persone, e non si ammette alcun altro documento al di fuori di questo. Non si farà nessuna eccezione.

```
proc InsertionSort (A)
  for (j = 2 to A.length)
    key = A[j]
    i = j - 1
    while ((i > 0) and (A[i] > key))
      A[i + 1] = A[i]
      i = i - 1
    A[i + 1] = key
```

```
proc RecursiveBinarySearch (A, low, high, k)
  if (low > high)
    then return nil
  mid = (high + low)/2
  if (A[mid] = k)
    then return mid
  if (A[mid] < k)
    then return RecursiveBinarySearch(A, mid + 1, high, k)
  if (A[mid] > k)
    then return RecursiveBinarySearch(A, low, mid - 1, k)
```

```
proc SelectionSort (A)
  for (j = 1 to A.length - 1)
    min = j
    for (i = j + 1 to A.length)
    if (A[i] < A[min])
      then min = i
    SwapValue(A, min, j)
```

```
proc Merge (A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1, ..., n1] and R[1, ..., n2] be new array
  for (i = 1 to n1) L[i] = A[p + i - 1]
  for (j = 1 to n2) R[j] = A[q + j]
  i = 1
  j = 1
  for (k = p to r)
    if (i ≤ n1)
      then
        if (j ≤ n2)
          then
            if (L[i] ≤ R[j])
              then CopyFromL(i)
              else CopyFromR(j)
          else CopyFromL(i)
      else CopyFromR(j)
```

```
proc MergeSort (A, p, r)
  if (p < r)
    then
      q = ⌊(p + r)/2⌋
      MergeSort(A, p, q)
      MergeSort(A, q + 1, r)
      Merge(A, p, q, r)
```

```
proc Partition (A, p, r)
  x = A[r]
  i = p - 1
  for (j = p to r - 1)
    if (A[j] ≤ x)
      then
        i = i + 1
        SwapValue(A, i, j)
  SwapValue(A, i + 1, r)
  return i + 1
```

```
proc QuickSort (A, p, r)
  if (p < r)
    then
      q = Partition(A, p, r)
      QuickSort(A, p, q - 1)
      QuickSort(A, q + 1, r)
```

```
proc RandomizedPartition (A, p, r)
  s = p ≤ Random() ≤ r
  SwapValue(A, s, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
    if (A[j] ≤ x)
      then
        i = i + 1
        SwapValue(A, i, j)
  SwapValue(A, i + 1, r)
  return i + 1
```

```
proc RandomizedQuickSort (A, p, r)
  if (p < r)
    then
      q = RandomizedPartition(A, p, r)
      RandQuickSort(A, p, q - 1)
      RandQuickSort(A, q + 1, r)
```

```
proc Empty (S)
  if (S.top = 0)
    then return true
  return false
```

```
proc Push (S, x)
  if (S.top = S.max)
    then return "overflow"
  S.top = S.top + 1
  S[S.top] = x
```

```
proc Pop (S)
  if (Empty(S))
    then return "underflow"
  S.top = S.top - 1
  return S[S.top + 1]
```

```
proc Enqueue (Q, x)
  if (Q.dim = Q.length)
    then return "overflow"
  Q[Q.tail] = x
  if (Q.tail = Q.length)
    then Q.tail = 1
    else Q.tail = Q.tail + 1
  Q.dim = Q.dim + 1
```

```
proc Dequeue (Q)
  if (Q.dim = 0)
    then return "underflow"
  x = Q[Q.head]
  if (Q.head = Q.length)
    then Q.head = 1
    else Q.head = Q.head + 1
  Q.dim = Q.dim - 1
  return x
```

```
proc Empty (S)
  if (S.head = nil )
    then return true
  return false
```

```
proc Push (S, x)
  Insert(S, x)
```

```
proc Pop (S)
  if (Empty(S))
    then return ″underflow″
  x = S.head
  Delete(S, x)
  return x.key
```

```
proc Empty (Q)
  if (Q.head = nil )
    then return true
  return false
```

```
proc Enqueue (Q, x)
  Insert(Q, x)
```

```
proc Dequeue (Q)
  if (Empty(Q))
    then return ″underflow″
  x = Q.tail
  Q.tail = x.prev
  Delete(Q, x)
  return x.key
```

```
proc Enqueue (A, i, priority)
  if (i > A.length)
    then return ″overflow″
  A[i] = priority
```

```
proc DecreaseKey (A, i, priority)
  if ((A[i] < priority) or (A[i].empty = 1))
    then return ″error″
  A[i] = priority
```

```
proc ExtractMin (A)
  MinIndex = 0
  MinPriority = ∞
  for (i = 1 to A.length)
    if ((A[i] < MinPriority) and (A[i].empty = 0))
      then
        MinPriority = A[i]
        MinIndex = i
  if (MinIndex = 0)
    then return ″underflow″
  A[MinIndex].empty = 1
  return MinIndex
```

```
proc Parent (i)
  return ⌊i/2⌋
```

```
proc Left (i)
  return 2 · i
```

```
proc Right (i)
  return 2 · i + 1
```

```
proc MinHeapify (H, i)
  l = Left(i)
  r = Right(i)
  smallest = i
  if ((l ≤ H.heapsize) and (H[l] < H[i]))
    then smallest = l
  if ((r ≤ H.heapsize) and (H[r] < H[smallest]))
    then smallest = r
  if smallest ≠ i
    then
      SwapValue(H, i, smallest)
      MinHeapify(H, smallest)
```

```
proc BuildMinHeap (H)
  H.heapsize = H.length
  for (i = ⌊H.length/2⌋ downto 1) MinHeapify(H, i)
```

```
proc HeapSort (H)
  BuildMaxHeap(H)
  for (i = H.length downto 2)
    SwapValue(H, i, 1)
    H.heapsize = H.heapsize − 1
    MaxHeapify(H, 1)
```

```
proc Enqueue (Q, priority)
  if (Q.heapsize = Q.length)
    then return ″overflow″
  Q.heapsize = Q.hepsize + 1
  Q[heapsize] = ∞
  DecreaseKey(Q, Q.heapsize, priority)
```

```
proc DecreaseKey (Q, i, priority)
  if (priority > Q[i])
    then return ″error″
  Q[i] = priority
  while ((i > 1) and (Q[Parent(i)] > Q[i]))
    SwapValue(Q, i, Parent(i))
    i = Parent(i)
```

```
proc ExtractMin (Q)
  if (Q.heapsize < 1)
    then return ″underflow″
  min = Q[1]
  Q[1] = Q[Q.heapsize]
  Q.heapsize = Q.heapsize − 1
  MinHeapify(Q, 1)
  return min
```

```
proc Enqueue (Q, i, priority)
  if (i > Q.length)
    then return ″overflow″
  Q[i] = priority
```

```
proc DecreaseKey (Q, i, priority)
  if ((Q[i] < priority) or (Q[i].empty = 1))
    then return ″error″
  Q[i] = priority
```

```
proc ExtractMin (Q)
  MinIndex = 0
  MinPriority = ∞
  for (i = 1 to Q.length)
    if ((Q[i] < MinPriority) and (Q[i].empty = 0))
      then
        MinPriority = Q[i]
        MinIndex = i
  if (MinIndex = 0)
    then return ″underflow″
  Q[MinIndex].empty = 1
  return MinIndex
```

```
proc CountingSort (A, B, k)
  let C[0, . . . , k] new array
  for (i = 0 to k) C[i] = 0
  for (j = 1 to A.length) C[A[j]] = C[A[j]] + 1
  for (i = 1 to k) C[i] = C[i] + C[i − 1]
  for (j = A.length downto 1)
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

```
proc RadixSort (A, d)
  for (i = 1 to d)  AnyStableSort(A) on digit i
```

```
proc ListInsert (L, x)
  x.next = L.head
  if (L.head ≠ nil )
    then  L.head.prev = x
  L.head = x
  x.prev = nil
```

```
proc ListSearch (L, k)
  x = L.head
  while (x ≠ nil ) and (x.key ≠ k) x = x.next
  return x
```

```
proc ListDelete (L, x)
  if (x.prev ≠ nil )
    then  x.prev.next = x.next
    else  L.head = x.next
  if (x.next ≠ nil )
    then  x.next.prev = x.prev
```

```
proc MakeSet (𝒮, S, x, i)
  𝒮[i].set = x
  S.head = x
  S.tail = x
```

```
proc Union (x, y)
  S_1 = x.head
  S_2 = y.head
  if (S_1 ≠ S_2)
    then
      S_1.tail.next = S_2.head
      z = S_2.head
      while (z ≠ nil )
        z.head = S_1
        z = z.next
      S_1.tail = S_2.tail
```

```
proc FindSet (x)
  return x.head.head
```

```
proc MakeSet (𝒮, S, x, i)
  𝒮[i].set = x
  S.head = x
  S.tail = x
  S.rank = 0
```

```
proc Union (x, y)
  S_1 = x.head
  S_2 = y.head
  if (S_1 ≠ S_2)
    then
      if (S_2.rank > S_1.rank)
        then
          S_temp = S_1
          S_1 = S_2
          S_2 = S_temp
      S_1.tail.next = S_2.head
      z = S_2.head
      while (z ≠ nil )
        z.head = S_1
        z = z.next
      S_1.tail = S_2.tail
      S_1.rank = S_1.rank + S_2.rank
```

```
proc MakeSet (x)
  x.p = x
  x.rank = 0
```

```
proc Union (x, y)
  x = Findset(x)
  y = Findset(y)
  if (x.rank > y.rank)
    then y.p = x
  if (x.rank ≤ y.rank)
    then
      x.p = y
      if (x.rank = y.rank)
        then y.rank = y.rank + 1
```

```
proc FindSet (x)
  if (x ≠ x.p)
    then x.p = FindSet(x.p)
  return x.p
```

```
proc TreeInOrderTreeWalk (x)
  if (x ≠ nil )
    then
      TreeInOrderTreeWalk(x.left)
      Print(x.key)
      TreeInOrderTreeWalk(x.right)
```

```
proc TreePreOrderTreeWalk (x)
  if (x ≠ nil )
    then
      Print(x.key)
      TreeInOrderTreeWalk(x.left)
      TreeInOrderTreeWalk(x.right)
```

```
proc TreePostOrderTreeWalk (x)
  if (x ≠ nil )
    then
      TreeInOrderTreeWalk(x.left)
      TreeInOrderTreeWalk(x.right)
      Print(x.key)
```

```
proc BSTTreeSearch (x, k)
  if ((x = nil ) or (x.key = k))
    then return x
  if (k ≤ x.key)
    then return BSTTreeSearch(x.left, k)
    else return BSTTreeSearch(x.right, k)
```

```
proc BSTTreeMinimum (x)
  if ((x.left = nil )
    then return x
  return BSTTreeMinimum(x.left)
```

```
proc BSTTreeSuccessor (x)
  if (x.right ≠ nil )
    then returnBSTTreeMinimum(x.right)
  y = x.p
  while ((y ≠ nil ) and (x = y.right))
    x = y
    y = y.p
  return y
```

```
proc BSTTreeInsert (T, z)
  y = nil
  x = T.root
  while (x ≠ nil )
    y = x
    if (z.key ≤ x.key)
      then x = x.left
      else x = x.right
  z.p = y
  if (y = nil )
    then T.root = z
  if ((y ≠ nil ) and (z.key ≤ y.key)
    then y.left = z
  if ((y ≠ nil ) and (z.key > y.key)
    then y.right = z
```

```
proc BSTTreeDelete (T, z)
  if (z.left = nil )
    then BSTTransplant(T, z, z.right)
  if ((z.left ≠ nil ) and (z.right = nil ))
    then Transplant(T, z, z.left)
  if ((z.left ≠ nil ) and (z.right ≠ nil ))
    then
      y = BSTTreeMinimum(z.right)
      if (y.p ≠ z)
        then
          BSTTransplant(T, y, y.right)
          y.right = z.right
          y.right.p = y
      BSTTransplant(T, z, y)
      y.left = z.left
      y.left.p = y
```

```
proc BSTTreeTransplant (T, u, v)
  if (u.p = nil )
    then T.root = v
  if ((u.p ≠ nil ) and (u = u.p.left))
    then u.p.left = v
  if ((u.p ≠ nil ) and (u = u.p.right))
    then u.p.right = v
  if (v ≠ nil )
    then v.p = u.p
```

```
proc BSTTreeLeftRotate (T, x)
  y = x.right
  x.right = y.left
  if (y.left ≠ T.nil )
    then y.left.p = x
  y.p = x.p
  if (x.p = T.nil )
    then T.root = y
  if ((x.p ≠ T.nil ) and (x = x.p.left))
    then x.p.left = y
  if ((x.p ≠ T.nil ) and (x = x.p.right))
    then x.p.right = y
  y.left = x
  x.p = y
```

```
proc RBTreeInsert (T, z)
  y = T.nil
  x = T.root
  while (x ≠ T.nil )
    y = x
    if (z.key < x.key)
      then x = x.left
      else x = x.right
  z.p = y
  if (y = T.nil )
    then T.root = z
  if ((y ≠ T.nil ) and (z.key < y.key)
    then y.left = z
  if ((y ≠ T.nil ) and (z.key ≥ y.key)
    then y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED
  RBTreeInsertFixup(T, z)
```

```
proc RBTreeInsertFixup (T, z)
  while (z.p.color = RED)
    if (z.p = z.p.p.left)
      then RBTreeInsertFixUpLeft(T, z)
      else RBTreeInsertFixUpRight(T, z)
  T.root.color = BLACK
```

```
proc RBTreeInsertFixupLeft (T, z)
  y = z.p.p.right
  if (y.color = RED)
    then
    z.p.color = BLACK
    y.color = BLACK
    z.p.p.color = RED
    z = z.p.p
    else
    if (z = z.p.right)
      then
      z = z.p
      LeftRotate(T, z)
    z.p.color = BLACK
    z.p.p.color = RED
    TreeRightRotate(T, z.p.p)
```

```
proc RBTreeInsertFixupRight (T, z)
  y = z.p.p.left
  if (y.color = RED)
    then
    z.p.color = BLACK
    y.color = BLACK
    z.p.p.color = RED
    z = z.p.p
    else
    if (z = z.p.left)
      then
      z = z.p
      TreeRightRotate(T, z)
    z.p.color = BLACK
    z.p.p.color = RED
    TreeLeftRotate(T, z.p.p)
```

```
proc BTreeSearch (x, k)
  i = 1
  while ((i ≤ x.n) and (k > x.key_i)) i = i + 1
  if ((i ≤ x.n) and (k = x.key_i))
    then return (x, i)
  if (x.leaf = true )
    then return nil
  DiskRead(x.c_i)
  return BTreeSearch(x.c_i, k)
```

```
proc BTreeCreate (T)
  x = Allocate()
  x.leaf = true
  x.n = 0
  DiskWrite(x)
  T.root = x
```

```
proc BTreeSplitChild (x, i)
  z = Allocate()
  y = x.c_i
  z.leaf = y.leaf
  for j = 1 to t − 1 z.key_j = y.key_{j+t}
  if (y.leaf = false )
    then for j = 1 to t z.c_j = y.c_{j+t}
  y.n = t − 1
  for j = x.n + 1 downto i + 1 x.c_{j+1} = x.c_j
  x.c_{i+1} = z
  for j = x.n downto i x.key_{j+1} = x.key_j
  x.key_i = y.key_t
  x.n = x.n + 1
  DiskWrite(y)
  DiskWrite(z)
  DiskWrite(x)
```

```
proc BTreeInsert (T, k)
  r = T.root
  if (r.n = 2 · t − 1)
    then
    s = Allocate()
    T.root = s
    s.leaf = false
    s.n = 0
    s.c_1 = r
    BTreeSplitChild(s, 1)
    BTreeInsertNonFull(s, k)
    else BTreeInsertNonFull(r, k)
```

```
proc BTreeInsertNonFull (x, k)
  i = x.n
  if (x.leaf = true )
    then
    while ((i ≥ 1) and (k < x.key_i))
      x.key_{i+1} = x.key_i
      i = i − 1
    x.key_{i+1} = k
    x.n = x.n + 1
    DiskWrite(x)
    else
    while ((i ≥ 1) and (k < x.key_i)) i = i − 1
    i = i + 1
    DiskRead(x.c_i)
    if (x.c_i.n = 2 · t − 1)
      then
      BTreeSplitChild(x, i)
      if (k > x.key_i)
        then i = i + 1
    BTreeInsertNonFull(x.c_i, k)
```

```
proc HashInsert (T, k)
  let x be a new node with key k
  i = h(k)
  ListInsert(T[i], x)
```

```
proc HashSearch (k)
  i = h(k)
  return ListSearch(T[i], k)
```

```
proc HashDelete (k)
  i = h(k)
  x = ListSearch(T[i], k)
  ListDelete(T[i], x)
```

```
proc OaHashInsert (T, k)
  i = 0
  repeat
    j = h(k, i)
    if (T[j] = nil )
      then
      T[j] = k
      return j
      else i = i + 1
  until (i = m)
  return "overflow"
```

```
proc OaHashSearch (T, k)
  i = 0
  repeat
    j = h(k, i)
    if (T[j] = k)
      then return j
    i = i + 1
  until ((T[j] = nil ) or (i = m))
  return nil
```

```
proc HashComputeModulo (w, B, m)
  let d = |w|
  z_0 = 0
  for (i = 1 to d) z_{i+1} = ((z_i · B) + a_i) mod m
  return z_d + 1
```

```
proc BreadthFirstSearch (G, s)
  for (u ∈ G.V \ {s})
    u.color = WHITE
    u.d = ∞
    u.π = nil
  s.color = GREY
  s.d = 0
  s.π = nil
  Q = ∅
  Enqueue(Q, s)
  while (Q ≠ ∅)
    u = Dequeue(Q)
    for (v ∈ G.Adj[u])
      if (v.color = WHITE)
        then
          v.color = GRAY
          v.d = u.d + 1
          v.π = u
          Enqueue(Q, v)
    u.color = BLACK
```

```
proc DepthFirstSearch (G)
  for (u ∈ G.V)
    u.color = WHITE
    u.π = nil
  time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisit(G, u)
```

```
proc DepthVisit (G, u)
  time = time + 1
  u.d = time
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then
        v.π = u
        DepthVisit(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time
```

```
proc CycleDet (G)
  cycle = False
  for (u ∈ G.V) u.color = WHITE
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisitCycle(G, u)
  return cycle
```

```
proc DepthVisitCycle (G, u)
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then DepthVisitCycle(G, v)
    if (v.color = GREY)
      then cycle = true
  u.color = BLACK
```

```
proc TopologicalSort (G)
  for (u ∈ G.V) u.color = WHITE
  L = ∅
  time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisitTS(G, u)
  return L
```

```
proc DepthVisitTS (G, u)
  time = time + 1
  u.d = time
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then DepthVisitTS(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time
  ListInsert(L, u)
```

```
proc StronglyConnectedComponents (G)
  for (u ∈ G.V)
    u.color = WHITE
    u.π = nil
  time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisit(G, u)
  for (u ∈ G.V)
    u.color = WHITE
    u.π = nil
  time = 0
  L = ∅
  for (u ∈ G^T.V in rev. finish time order)
    if (u.color = WHITE)
      then DepthVisit(G^T, u)
    ListInsert(L, u)
  return L
```

```
proc MST-Prim (G, w, r)
  for (v ∈ G.V)
  do
    v.key = ∞
    v.π = nil
  r.key = 0
  Q = G.V
  while (Q ≠ ∅)
  do
    u = ExtractMin(Q)
    for (v ∈ G.Adj[u])
    do
      if ((v ∈ Q) and (W(u, v) < v.key))
      do
        v.π = u
        v.key = W(u, v)
```

```
proc MST-Kruskal (G, w)
  T = ∅
  for (v ∈ G.V)
    do MakeSet(v)
  SortNoDecreasing(G.E)
  for ((u, v) ∈ G.E − in order)
  do
    if (FindSet(u) ≠ FindSet(v))
      then
        T = T ∪ {(u, v)}
        Union(u, v)
  return A
```

```
proc DAGShortestPath (G, s)
  TopologicalSort(G)
  InizializeSingleSource(G, s)
  for (u ∈ G.V − in order)
    for (v ∈ G.Adj[u]) Relax(u, v, W)
```

proc ***InizializeSingleSource*** (G, s)
- **for** $(v \in G.V)$
  - $v.d = \infty$
  - $v.\pi =$ **nil**
- $s.d = 0$

proc ***Relax*** (u, v, W)
- **if** $(v.d > u.d + W(u, v))$
  - **then**
  - $v.d = u.d + W(u, v)$
  - $v.\pi = u$

proc ***Bellman-Ford*** (G, s)
- $InizializeSingleSource(G, s)$
- **for** $i = 1$ **to** $|G.V| - 1$
  - **for** $((u, v) \in G.E)$ $Relax(u, v, W)$
- **for** $((u, v) \in G.E)$
  - **if** $(v.d > u.d + W(u, v))$
  - **then return false**
- **return true**

proc ***Dijkstra*** (G, s)
- $InizializeSingleSource(G, s)$
- $S = \emptyset$
- $Q = G.V$
- **while** $(Q \neq \emptyset)$
  - $u = ExtractMin(Q)$
  - $S = S \cup \{u\}$
  - **for** $(v \in G.Adj[u])$
    - **if** $(v \in Q)$
    - **then** $Relax(u, v, W)$

proc ***ExtendShortestPaths*** (L, W)
- $n = L.rows$
- **let** L' **be a new matrix**
- **for** $i = 1$ **to** n
  - **for** $j = 1$ **to** n
    - $L'_{ij} = \infty$
    - **for** $k = 1$ **to** n
      - $L'_{ij} = min\{L'_{ij}, L_{ik} + W_{kj}\}$
- **return** $i + 1$

proc ***SlowAllPairsMatrix*** (W)
- $n = L.rows$
- $L^1 = W$
- **for** $m = 2$ **to** $n - 1$
  - $L^m = ExtendShortestPaths(L^{m-1}, W)$
- **return** L^{n-1}

proc ***FastAllPairsMatrix*** (W)
- $n = L.rows$
- $L^1 = W$
- $m = 1$
- **while** $(m < n - 1)$
  - **do**
  - $L^{2 \cdot m} = ExtendShortestPaths(L^m, L^m)$
  - $m = 2 \cdot m$
- **return** L^m

proc ***Floyd-Warshall*** (W)
- $n = W.rows$
- $D^0 = W$
- **for** $k = 1$ **to** n
  - **let** D^k **be a new matrix**
  - **for** $i = 1$ **to** n
    - **for** $j = 1$ **to** n
      - $D^k_{ij} = min\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\}$