ซูโดโค้ดของอัลกอริทึมต่าง ๆ (Pseudocodes)

Decrease-and-Conquer

Algorithm 1 Insertion Sort($A[0 \dots n-1]$)

Require: $A[0 \dots n-1]$, an arbitrary array
Ensure: $A[0 \dots n-1]$, sorted in **non-decreasing** order

```
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 And A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    end while
    A[j + 1] ← v
end for
```

Algorithm 2 JohnsonTrotter(n)

Require: n, a positive integer
Ensure: A list of all permutations of $\{1, 2, \dots, n\}$

Initialize the first permutation with $\overleftarrow{1}, \overleftarrow{2}, \dots, \overleftarrow{n}$
while the last permutation has a mobile element **do** ▷ *An element is called mobile if its arrow points to a small adjacent number*
    Find its largest mobile element k
    Swap k with the adjacent element k's arrow points to
    Reverse the direction of all the elements that are larger than k
    Add the new permutation to the list
end while

Algorithm 3 LexicographicPermute(n)

Require: n, a positive integer
Ensure: A list of all permutations of $\{1, 2, \dots, n\}$ in lexicographic order

Initialize the first permutation with $12 \dots n$
while the last permutation has two consecutive elements in increasing order **do**
    Let i be its largest index such that $a_i < a_{i+1}$ ▷ $a_{i+1} > a_{i+2} > \cdots > a_n$
    Find the largest index j such that $a_i < a_j$ ▷ $j \geq i+1$ *since* $a_i < a_{i+1}$
    Swap a_i with a_j ▷ $a_{i+1}a_{i+2} \dots a_n$ *will remain in decreasing order*
    Reverse the order of the elements from a_{i+1} to a_n inclusive
    Add the new permutation to the list
end while

Algorithm 4 Binary Search($A[0 \dots n-1], K$)

Require: $A[0 \dots n-1]$ sorted in ascending order and a search key K
Ensure: An index of the array's element that is equal to K or -1 if there is no such element

$l \leftarrow 0$
$r \leftarrow n-1$
while $l \leq r$ **do**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K = A[m]$ **then**
    Return m
  else if $K < A[m]$ **then**
    $r \leftarrow m-1$
  else
    $l \leftarrow m+1$
  end if
end while
Return -1

Algorithm 5 LomutoPartition($A[l \dots r]$)

Require: A subarray $A[l \dots r]$ of array $A[0 \dots n-1]$, defined by its left and right indices l and r ($l \leq r$)
Ensure: Partition of $A[l \dots r]$ and the new position of the pivot

$p \leftarrow A[l]$
$s \leftarrow l$
for i from $l+1$ to r **do**
  if $A[i] < p$ **then**
    $s \leftarrow s+1$
    Swap($A[s]$, $A[i]$)
  end if
end for
Swap($A[l]$,$A[s]$)
Return s

Algorithm 6 Quickselect($A[l \dots r], k$)

Require: A subarray $A[l \dots r]$ of array $A[0 \dots n-1]$ or orderable elements and integer k ($1 \leq k \leq r-l+1$)
Ensure: The value of the kth smallest element in $A[l \dots r]$

$s \leftarrow LomutoPartition(A[l \dots r])$ ▷ *Or another partition algorithm*
if $s = l+k-1$ **then**
  Return $A[s]$
else if $s > l+k-1$ **then**
  $Quickselect(A[l \dots s-1], k)$
else
  $Quickselect(A[s+1 \dots r], l+k-1-s)$
end if

Divide-and-Conquer

Algorithm 7 Mergesort($A[0 \dots n-1]$)

Require: An array $A[0 \dots n-1]$ of orderable elements
Ensure: Array $A[0 \dots n-1]$ sorted in nondecreasing order

```
if n > 1 then
    Copy A[0 ... ⌊n/2⌋ − 1] to B[0 ... ⌊n/2⌋ − 1]
    Copy A[⌊n/2⌋ ... n − 1] to C[0 ... ⌈n/2⌉ − 1]
    Mergesort(B[0 ... ⌊n/2⌋ − 1])
    Mergesort(C[0 ... ⌈n/2⌉ − 1])
    Merge(B, C, A)
end if
```

Algorithm 8 Merge($B[0 \dots p-1], C[0 \dots q-1], A[0 \dots p+q-1]$)

Require: Arrays $B[0 \dots p-1]$ and $C[0 \dots q-1]$ both sorted
Ensure: Sorted array $A[0 \dots p+q-1]$ of the elements of B and C

```
i ← 0; j ← 0; k ← 0
while i < p And j < q do
    if B[i] ≤ C[j] then
        A[k] ← B[i]; i ← i + 1
    else
        A[k] ← C[j]; j ← j + 1
    end if
    k ← k + 1
end while
if i = p then
    Copy C[j ... q − 1] to A[k ... p + q − 1]
else
    Copy B[i ... p − 1] to A[k ... p + q − 1]
end if
```

Algorithm 9 Quicksort($A[l \dots r]$)

Require: Subarray of array $A[0 \dots n-1]$, defined by its left and right indices l and r
Ensure: Subarray $A[l \dots r]$ sorted in nondecreasing order

```
if l < r then
    s ← Partition(A[l ... r])          ▷ s is a split position
    Quicksort(A[l ... s − 1])
    Quicksort(A[s + 1 ... r])
end if
```

Algorithm 10 HoarePartitioning($A[l \dots r]$)

Require: Subarray $A[0 \dots n-1]$, defined by its left and right indices l and r ($l < r$)
Ensure: Partition of $A[l \dots r]$, with the split position returned as this function's value

$p \leftarrow A[l]$
$i \leftarrow l$
$j \leftarrow r + 1$
repeat
  Repeat $i \leftarrow i + 1$ **Until** $A[i] \geq p$
  Repeat $j \leftarrow j - 1$ **Until** $A[j] \leq p$
  Swap($A[i]$, $A[j]$)
until $i \geq j$
Swap($A[i]$,$A[j]$) ▷ *Undo last swap when* $i \geq j$
Swap($A[l]$,$A[j]$)
Return j

Algorithm 11 EfficientClosestPair(P, Q)

Require: An array P of $n \geq 2$ points in the Cartesian plane sorted in nondecreasing order of their x coordinates and an array Q of the same points sorted in nondecreasing order of the y coordinates
Ensure: Euclidean distance between the closest pair of points

if $n \leq 3$ **then**
  Return the minimal distance found the Brute-Force algorithm
else
  Copy the first $\lceil n/2 \rceil$ points of P to array P_l
  Copy the same $\lceil n/2 \rceil$ points of Q to array Q_l
  Copy the remaining $\lfloor n/2 \rfloor$ points of P to array P_r
  Copy the same $\lfloor n/2 \rfloor$ points of Q to array Q_r
  $d_l \leftarrow EfficientClosestPair(P_l, Q_l)$
  $d_r \leftarrow EfficientClosestPair(P_r, Q_r)$
  $d \leftarrow Min(d_l, d_r)$
  $m \leftarrow P[\lceil n/2 \rceil - 1].x$
  Copy all the points of Q for which $|x - m| < d$ into array $S[0 \dots num - 1]$
  $dminsq \leftarrow d^2$
  for $i \leftarrow 0$ to $num - 2$ **do**
    $k \leftarrow i + 1$
    while $l \leq num - 1$ and $(S[k].y - S[i].y)^2 < dminsq$ **do**
      $dminsq \leftarrow Min((S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2, dminsq)$
      $k \leftarrow k + 1$
    end while
  end for
end if
Return $sqrt(dminsq)$

Transform-and-Conquer

Algorithm 12 PresortElementUniqueness($A[0 \ldots n-1]$)

Require: An array $A[0 \ldots n-1]$ of orderable elements
Ensure: "True" if A has no equal elements, "False" otherwise

```
Sort the array A
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] then
        Return "False"
    end if
end for
Return "True"
```

Algorithm 13 PresortMode($A[0 \ldots n-1]$)

Require: An array $A[0 \ldots n-1]$ of orderable elements
Ensure: The array's mode

```
Sort the array A
i ← 0                                  ▷ Current run begins at position i
modefrequency ← 0                      ▷ Highest frequency seen so far
while i ≤ n − 1 do
    runlength ← 1
    runvalue ← A[i]
    while i + runlength ≤ n − 1 And A[i + runlength] = runvalue do
        runlength ← runlength + 1
    end while
    if runlength > modefrequency then
        modefrequency ← runlength
        modevalue ← runvalue
    end if
    i ← i + runlength
end while
Return modevalue
```

Algorithm 14 HeapBottomUp($H[1 \dots n]$)

Require: An array $H[1 \dots n]$ of orderable elements
Ensure: A heap $H[1 \dots n]$

```
for i ← ⌊n/2⌋ Downto 1 do
    k ← i
    v ← H[k]
    heap ← "False"
    while Not heap And 2 × k ≤ n do
        j ← 2 × k
        if j < n then                          ▷ There are two children
            if H[j] < H[j + 1] then
                j ← j + 1
            end if
        end if
        if v ≥ H[j] then
            heap ← "True"
        else
            H[k] ← H[j]
            k ← j
        end if
    end while
    H[k] ← v
end for
```

Space-Time Tradeoffs

Algorithm 15 ComparisonCountingSort($A[0 \dots n-1]$)

Require: An array $A[0 \dots n-1]$ of orderable elements
Ensure: Array $S[0 \dots n-1]$ of A's elements sorted in nondecreasing order

```
for i ← 0 to n − 1 do
    Count[i] ← 0
end for
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j] then
            Count[j] ← Count[j] + 1
        else
            Count[i] ← Count[i] + 1
        end if
    end for
end for
for i ← 0 to n − 1 do
    S[Count[i]] ← A[i]
end for
Return S
```

Algorithm 16 DistributionCountingSort($A[0 \ldots n-1], l, u$)

Require: An array $A[0 \ldots n-1]$ of integers between l and u ($l \leq u$)
Ensure: Array $S[0 \ldots n-1]$ of A's elements sorted in nondecreasing order

```
for j ← 0 to u − 1 do
    D[j] ← 0                                  ▷ Initialize frequencies
end for
for i ← 0 to n − 1 do
    D[A[i] − l] ← D[A[i] − l] + 1             ▷ Compute frequencies
end for
for j ← 0 to u − 1 do
    D[j] ← D[j − 1] + D[j]                    ▷ Reuse for distribution
end for
for i ← n − 1 Downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
end for
Return S
```

Algorithm 17 ShiftTable($P[0 \ldots m-1]$)

Require: Pattern $P[0 \ldots m-1]$ and an alphabet of possible characters
Ensure: $Table[0 \ldots size-1]$ indexed by the alphabet's characters and filled with shift.

```
for i ← 0 to size − 1 do
    Table[i] ← m
end for
for j ← 0 to m − 2 do
    Table[P[j]] ← m − 1 − j
end for
Return Table
```

Algorithm 18 HorspoolMatching($P[0 \ldots m-1], T[0 \ldots n-1]$)

Require: Pattern $P[0 \ldots m-1]$ and text $T[0 \ldots n-1]$
Ensure: The index of the left end of the first matching substring or -1 if there are no matches

```
ShiftTable(P[0 ... m − 1])                    ▷ Generate Table of shifts
i ← m − 1                                     ▷ Position of pattern's right end
while i ≤ n − 1 do
    k ← 0                                     ▷ Number of matched characters
    while k ≤ m − 1 And P[m − 1 − k] = T[i − k] do
        k ← k + 1
    end while
    if k = m then
        Return i − m + 1
    else
        i ← i + Table[T[i]]
    end if
end while
Return −1
```