```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.
      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
      if A[midPoint] < x
         set lowerBound = midPoint + 1
      if A[midPoint] > x
         set upperBound = midPoint - 1
      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while

end procedure
```

Algorithm 1 Topological Sort

Input: $G = (V, E)$, a DAG

Output: L, a list of vertices in G in a total order compatible with the partial order of the DAG.

```
for v ∈ V do
    Add an attribute 'v.count' and initialize it to the indegree of v.
end for
Q ← the sent of vertices with 'count'=0.
L ← array of length |V|
i ← 1
while Q is not empty do
    v ← Q.POP
    L[i] ← v
    i++
    for u ∈ v.outgoing do
        u.count−−
        if u.count = 0 then
            Add u to Q
        end if
    end for
end while
return L
```

Binary search: O(log n)
Topo sort: O(V+E)

max flow = min cut
max flow:
set flows to 0
find path s -> t, add bottleneck capacity to flows
go until no path from s -> t
min cut:
same number but cut from top to bot, add capacities

induction:
Base case (show claim true for n = 1 or 0)
inductive assumption (assume true for n = k)
inductive step show true for n = k+1
summarize and conclude

Loop Guard: G
Post-condition: Q
Pre-condition: P
Loop invariant: L = Li

Initialization: P => L
Maintenance: Li ^ (and) G => Li+1
End: L ^ ¬G => Q

```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

$T(n) = a * T(n/b) + \Theta(n^k log^p n)$
$a \geq 1; b > 1; k \geq 0$, and p is a real number

Case-01:
If $a > b^k$, then $T(n) = \theta\left(n^{\log_b a}\right)$

Case-02: If $a = b^k$ and
- If $p < -1$, then $T(n) = \theta\left(n^{\log_b a}\right)$
- If $p = -1$, then $T(n) = \theta\left(n^{\log_b a} \cdot \log^2 n\right)$
- If $p > -1$, then $T(n) = \theta\left(n^{\log_b a} \cdot \log^{p+1} n\right)$

Case-03:
If $a < b^k$ and
- If $p < 0$, then $T(n) = O\left(n^k\right)$
- If $p \geq 0$, then $T(n) = \theta\left(n^k \log^p n\right)$

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])        ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])      ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i+1
      else if i > m
          B[k] ← A[j]; j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

HP

Answer:

Loop invariant (LV): L[1:i] is a set of vertices in the input directed acyclic graph, G, in a total order compatible with the partial order of the DAG

Avoid just listing properties. Write them out in a proper sentence.

Pre-Condition (P): $|Q| > 0; |L| = |V|; i = 1;$ All elements in L are null

Post-Condition (QP): The list L is a set of vertices in the input directed acyclic graph, G, in a total order compatible with the partial order of the DAG

Loop Guard (LG): $|Q| > 0$ (set Q is not empty)

$\neg$LG: $|Q| \leq 0$ (set Q is empty)

P

Initialization

Assume P is true. That is, assume that $i = 1$, $|L| = |V|$, $|Q| > 0$ and all elements in L are null. Since $i = 1$ and $|Q| > 0$ then we know that LV is vacuously true because L[1:i(1)] has to be sorted.

How does $|Q| > 0$ help show LV is vacuously true. Does L[1:1] need to be sorted?

LP

Maintenance

Assume we've completed i-1 iterations of the loop and we're about to enter the ith iteration. Assume LV = (LV_i) is true and G is true (thus, we have just entered the loop).

In line 8, Q gets an element popped and assigned to v

In line 9, the ith element in list L is set to v

In line 10, i is incremented by 1

In line 11, for every element u in v.outgoing we:

In line 12, we decrement u.count by 1 then we have two possible cases:

You need argue adding v to L still keeps the partial order correct.

Case 1 (u.count equal to 0): u is added to Q

Case 2 (u.count not equal to 0): pass

Why does u.count = 0 imply u is an upper bound for elements already in the list?

If u.count is equal to 0, u is the next element in a total order compatible with the partial order of the DAG because u is the outgoing vertex from the last vertice(s).

P

End When the loop invariant is true (L[1:i] is a set of vertices in the input directed acyclic graph, G, in a total order compatible with the partial order of the DAG) and the loop guard is not true (Q is empty). Then we know that that our post condition is true (the list L is a set of vertices in G is in a total order compatible with the partial order of the DAG) because i is the final element of L so L = L[1:i].

How do we know i is the last index of L? Why can't $|Q| \leq 0$ and i not be the last index.

$\boxed{\begin{array}{l} D : S \rightarrow \mathbb{N} \\ D(S) := n - i \end{array}}$, i = the iteration through the loop, n = size of input ③

```
QuickSort(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← Partition(A, p)
      QuickSort(A[1..r−1])      ⟨⟨Recurse!⟩⟩
      QuickSort(A[r+1..n])      ⟨⟨Recurse!⟩⟩
```

```
Partition(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0                ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n − 1
      if A[i] < A[n]
          ℓ ← ℓ + 1
          swap A[ℓ] ↔ A[i]
  swap A[n] ↔ A[ℓ + 1]
  return ℓ + 1
```