다음과 같은 상황에서 A이전 항목들은 모두 정렬되었고, A 위치에 최솟값 M을 넣는다고 가정하자.

3장. 해답

<3.1절>

3.1

A L G O R I T H M
A G L O R I T H M
A G H O R I T L M
A G H I R O T L M
A G H I L O T R M
A G H I L M T R O
A G H I L M O R T
A G H I L M O R T
A G H I L M O R T

3.2

7 4 9 6 3 8 7 5
3 4 9 6 7 8 7 5
3 4 9 6 7 8 7 5
3 4 5 6 7 8 7 9
3 4 5 6 7 8 7 9
3 4 5 6 7 8 7 9
3 4 5 6 7 7 8 9
3 4 5 6 7 7 8 9
3 4 5 6 7 7 8 9

3.3

2 2 1
1 2 2
1 2 2
1 2 2

3.4*

- - - A ? ? ? B ? ? C ? ? M ? ? ?

이때, A와 M을 바로 바꾸면 안정성을 만족하지 않는 경우가 발생할 수 있다. 안정성을 만족하기 위해서는 먼저 A와 동일한 항목 (예를 들어, B, C)들을 모두 찾고, 이들을 순서대로 다음 자리로 옮긴다. 예를 들어, 위의 경우에는 C는 M위치, B는 C위치, A는 B 위치로 먼저 옮긴 후 M을 A위치로 옮기면 된다.

- - - M ? ? ? A ? ? B ? ? C ? ? ?

3.5

```
bChanged = True
while bChanged :
    bChanged = False
    for i in range(1, 2*n ) :
        if A[i-1] == 1 and A[i] == 0 :
            A[i-1], A[i] = 0, 1
            bChanged = True
```

<3.2절>

3.6

이진 탐색을 사용할 수 있다. 4장 이진탐색 참조.

3.7*

```
def sentinel_search(A, key):
    n = len(A)
    A.append(key)
    i = 0
    while A[i] != key :
        i += 1
    if i == n : return -1
    return I
```

<3.3절>

3.8

(1) 96 * 5

(2) 96 * 1

(3) 96 * 2

3.9

예를 들어, 다음과 같은 텍스트와 패턴은 최악의 입력이다.

텍스트: TTTTT...T?

패턴: TT...TY

매 위치에서 패턴의 길이(m) 만큼 비교해야 하고, 텍스트의 맨 마지막 위치까지 처리해야 하기 때문이다. 전체 비교 횟수는 m * (n-m+1)번 이다.

3.10

한꺼번에 여러 칸을 건너뛰고 검사할 수 있다(6장의 호스풀 알고리즘 참조)

3.11*

(1) $O(n^2)$

```
def count_substr(str, A, B):
    count = 0
    n = len(str)
    for i in range (n-1) :
        if str[i] == A :
            for j in range (i+1, n) :
                if str[j] == B :
                    count += 1
    return count
```

(2) $O(n)$

step1: 모든 A의 위치와 B의 위치를 순서대로 찾음 --> 각각 리스트에 저장

step2: 각각의 A위치 이후에 있는 B의 개수를 셈

두 단계 모두 $O(n)$ 시간이 걸림 --> $O(n)$

<3.4절>

3.12*

(1) $dist(i,j) = |x_i - x_j|$

(2) 정렬한 후 인접 항목의 거리를 계산함. 시간복잡도가 $O(nlog_2 n)$로 줄어듦

3.13

유클리드 거리, Hamming distance, Hausdorff distance 등

3.14 $O(kn^2)$

<3.5절>

3.15*

```
def is_safe(g, v, pos, path):
    if g[ path[pos-1] ][v] == 0: return False
    for vertex in path:
        if vertex == v: return False
    return True

def hamiltonian_recur(g, path, pos):
    n = len(g)
    if pos == n:
        if g[path[pos-1]][path[0]] == 1: return True
        else: return False

    for v in range(1, n):
        if is_safe(g, v, pos, path) == True:
            path[pos] = v
            if hamiltonian_recur(g, path, pos+1) == True:
                return True
            path[pos] = -1
    return False

def hamiltonian_cycle(g):
    n = len(g)
    path = [-1] * (n+1)
    path[0] = path[n] = 0     # 0번부터 출발하자.

    if hamiltonian_recur(g, path, 1) == False:
        print ("해밀토니언 사이클 없음")
        return False
    else :
        print ("해밀토니안 사이클: ", path)
    return True

g1 = [  [0, 1, 0, 1, 0],
```

```
        [1, 0, 1, 1, 1],
        [0, 1, 0, 0, 1],
        [1, 1, 0, 0, 1],
        [0, 1, 1, 1, 0], ]
hamiltonian_cycle(g1)
```

3.16*

```
def hamiltonian_recur(g, path, pos):
    n = len(g)
    if pos == n:
        if g[path[pos-1]][path[0]] == 1:  # 마지막 정점-첫 정점
            cycles.append(list(path))
            return True
        else: return False

    for v in range(1, n):
        if is_safe(g, v, pos, path) == True:
            path[pos] = v
            hamiltonian_recur(g, path, pos+1)
            path[pos] = -1

cycles = []
def hamiltonian_cycle_all(g):
    global cycles
    n = len(g)
    path = [-1] * (n+1)
    path[0] = path[n] = 0     # 0번부터 출발하자.

    cycles=[]
    hamiltonian_recur(g, path, 1)
    print ("해밀토니언 사이클 개수: ", len(cycles))
    for i in range(len(cycles)) :
        print ("₩t ", cycles[i])
```

3.17* 문제 16에서 구한 모든 사이클들 중에 경로의 길이가 가장 작인 사이클을 출력하면 됨.

3.18* 0-1 배낭 채우기 문제. 9.5절 참조

3.19* 일 배정 문제. 9.6절 참조

3.20* 일 배정 문제에 대한 헝가리안 알고리즘
https://www.geeksforgeeks.org/hungarian-algorithm-assignment-problem-set-1-introduction/

3.21 모든 숫자에 대한 순열 생성하고, 정렬의 조건을 만족하는 것을 찾음. $O(n!)$

3.22* 모든 부분집합을 구하고, 그 부분집합의 숫자의 합이 전체 원소의 합의 절반인지를 검사함. --> 복잡도 $O(2^n)$

<3.6절>

3.23*

(1)

```
adjMat = [   [ 0, 1, 0, 1, 1, 0, 0 ],
             [ 1, 0, 1, 0, 1, 1, 0 ],
             [ 0, 1, 0, 0, 0, 0, 1 ],
             [ 1, 0, 0, 0, 1, 0, 0 ],
             [ 1, 1, 0, 1, 0, 0, 0 ],
             [ 0, 1, 0, 0, 0, 0, 1 ],
             [ 0, 0, 1, 0, 0, 1, 0 ] ]
```

(2)

```
graph={   'A'  : { 'B' ,  'D' ,  'E' },
          'B'  : { 'A' ,  'C' ,  'E' ,  'F' },
          'C'  : { 'B' ,  'G' },
          'D'  : { 'A' ,  'E' },
          'E'  : { 'A' ,  'B' ,  'D' },
          'F'  : { 'B' ,  'G' },
          'G'  : { 'C' ,  'F' }
      }
```

(3) A – B – C – G – F – E – B

(4) G

(5) A – B – D – E – C – F – G

3.24

(1)

```
def dfs_recur(adj, vtx, visited, id) :
    print(vtx[id], end=' ')
    visited[id] = True
    for v in range(len(vtx)) :
        if visited[v]==False and adj[id][v] != 0 :
            dfs_recur(adj, vtx, visited, v)
    return

def dfs(adj, vtx, s):
    n = len(vtx)
    visited = [False]*n
    dfs_recur(adj, vtx, visited, s)
```

(2)

```
from queue import Queue
def bfs(adj, vtx, s):
    n = len(vtx)
    visited = [False]*n
    Q = Queue()
    Q.put(s)
    visited[s] = True
    while not Q.empty() :
        s = Q.get()
        print(vtx[s], end=' ')
        for v in range(len(vtx)) :
            if visited[v]==False and adj[s][v] != 0 :
                Q.put(v)
                visited[v] = True
    return
```

3.25 인접 리스트 방식이 더 효율적이다.

3.26*

```
def st_dfs_recur(adj, vtx, visited, id) :
    visited[id] = True
    for v in range(len(vtx)) :
        if visited[v]==False and adj[id][v] != 0 :
            print("(%s,%s)"%(vtx[id], vtx[v]), end=" ") # (v,u)간선 추가
            st_dfs_recur(adj, vtx, visited, v)
```

```
    return

def st_dfs(adj, vtx, s):
    n = len(vtx)
    visited = [False]*n
    st_dfs_recur(adj, vtx, visited, s)
    return
```

3.27*

(1)

```
def color_graph_DFS(g, color, pos, c):
    if color[pos] != -1 and color[pos] != c:
        return False

    color[pos] = c
    ans = True
    for i in range(len(g)):
        if g[pos][i] == 1:      # (pos,i) 간선이 있는 경우
            if color[i] == -1:  # i가 칠해지지 않았으면
                if not color_graph_DFS(g, color, i, 1-c):
                    return False
            elif color[i] != 1-c:
                return False

    return True

def is_bipartite_DFS(g):
    color = [-1] * len(g)
    return color_graph_DFS(g, color, 0, 1)
```

(2)

```
import queue
def is_bipartite_BFS(g):
    n = len(g)
    colorArr = [-1] * n
    colorArr[0] = 1     # 시작 정점 --> 0
    Q = queue.Queue()
    Q.put(0)
```

```
    while not Q.empty():
        u = Q.get()
        for v in range(n):
            if g[u][v] == 1 :
                if colorArr[v] == -1:
                    colorArr[v] = 1 - colorArr[u]   # 다른 색 할당
                    Q.put(v)
                elif colorArr[v] == colorArr[u]:    # 이분 그래프 아님.
                    return False
    return True

g1 = [[0, 1, 0, 1],
      [1, 0, 1, 0],
      [0, 1, 0, 1],
      [1, 0, 1, 0] ]

print("BFS 방식 탐색 --> ", "Yes" if is_bipartite_BFS(g1) else "No")
print("DFS 방식 탐색 --> ", "Yes" if is_bipartite_DFS(g1) else "No")
```