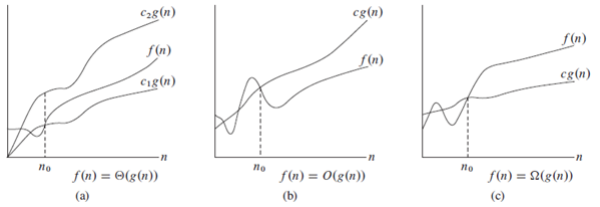
1 Basics

알고리즘은 문제 해결을 위한 일련의 구체적인 동작. 자연어, 의사 코드, 플로우 차트 등으로 표현 가능. 시간복잡도는 알고리즘의 기본적인 연산(대입, 비교, 산술 등) 횟수를 입력 크기에 대한 함수로 나타낸다. 이렇게 표현하기 위해 점근적 표기를 사용한다.

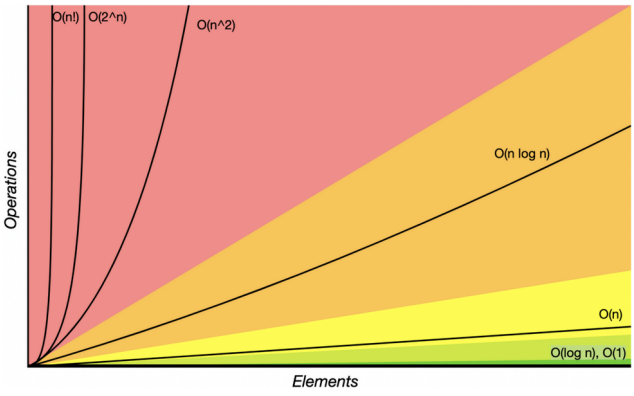


(a) Θ: n_0 보다 큰 모든 n에 대해 상한과 하한을 동시에 만족. 즉, O과 Ω가 동일한 경우. average case.

(b) O: n이 증가함에 따라 $O(g(n))$이 점근적 상한. 즉, $g(n)$은 $f(n)$의 상한. $g(n)$이 n_0보다 큰 모든 n에 대해 항상 $f(n)$보다 크다. worst case.

(c) Ω: n이 증가함에 따라 $\Omega(g(n))$이 점근적 하한. 즉, $g(n)$은 $f(n)$의 하한. $g(n)$이 n_0보다 큰 모든 n에 대해 항상 $f(n)$보다 작다. best case.

보통 시간복잡도를 O 표기로 나타내는데, 실제로는 Θ인 의미가 많음. 혹시 더 나은 방법이 있을지 모르기 때문에 Θ 표기를 지양함.



상각 분석은 점근적 분석과 달리 실제로 연산을 수행하여 총 연산 횟수의 합을 시행 횟수로 나누는 분석법.

2 Divide and Conquer

주어진 문제의 입력을 분할하여 문제를 해결하는 방식의 알고리즘. 문제의 크기가 매우 작으면 풀기 쉬워지는 경우, 또는 문제의 크기를 줄이면 trivial solution에 가까워지는 경우 사용한다.

입력 크기가 n일 때 1회 분할마다 각각의 입력 크기가 반감하기 때문에, k회 분할하는 경우 각각의 입력 크기는 $n/2^k$이며, 입력 크기가 1이 되면 더 이상 분할할 수 없으므로 $k = \log_2 n$이다.

i. 1회 분할했을 때 입력 크기: $n/2$

ii. 2회 분할했을 때 입력 크기: $n/2^2$

iii. k회 분할했을 때 입력 크기: $n/2^k$

분할로 인해 부분 문제가 매우 커지는 경우에는 부적절하다. 부분 문제의 취합 방법에 따라 분할 정복 알고리즘의 성능이 좌우된다.

2.1 Merge Sort

입력된 리스트를 두 부분으로 분할하여 각 부분 문제의 요소가 1개가 될 때까지 분할을 반복하고, 각 부분 문제를 정렬하며 병합하는 정렬 알고리즘.

```
def merge_sort(s: list[int]) -> list[int]:
  if len(s) <= 1: return s

  m = len(s) // 2
  l = merge_sort(s[:m])
  r = merge_sort(s[m:])

  li = ri = 0
  t = []
  while li < len(l) and ri < len(r):
    if l[li] < r[ri]:
      t.append(l[li])
      li += 1
    else:
      t.append(r[ri])
      ri += 1

  return t + l[li:] + r[ri:]
```

분할 시 부분 문제가 $n/2^k$개로 쪼개지므로 $O(\log n)$이고, 각 층을 병합하는 데 $O(n)$ 시간이 걸리므로, 총 시간복잡도는 $O(n \log n)$이다.

2.2 Quick Sort

머지 소트는 각 층에서 매번 새로운 리스트를 만들기 때문에 공간복잡도가 $O(n^2)$이다. 퀵 소트는 in-place

방식으로 구현할 수 있기 때문에 공간복잡도를 줄일 수 있다.

퀵 소트는 pivot을 기준으로 작은 수는 왼쪽으로, 큰 수는 오른쪽으로 분할하는 과정을 반복한다. 이를 partitioning이라고 한다.

```
def partition(a: list[int], si: int,
              ei: int, pi: int) -> int:
  pv = a[pi]
  a[si], a[pi] = a[pi], a[si]
  li, ri = si + 1, ei

  while li <= ri:
    while li <= ei and a[li] <= pv: li += 1
    while ri >= si and a[ri] > pv: ri -= 1
    if li <= ri:
      a[li], a[ri] = a[ri], a[li]
      li, ri = li + 1, ri - 1
  a[si], a[ri] = a[ri], a[si]

  return ri
```

partitioning된 각 부분 문제에 대해 퀵 정렬을 재귀적으로 실행하면 된다.

```
def quick_sort(a: list[int], si: int,
               ei: int) -> None:
  if si >= ei: return

  rpi = random.randrange(si, ei)
  pi = partition(a, si, ei, rpi)

  quick_sort(a, si, pi - 1)
  quick_sort(a, pi + 1, ei)
```

최악의 경우(항상 가장 작거나 큰 숫자가 pivot으로 선택되는 경우, 리스트가 이미 정렬되어 있는 경우) 시간복잡도는 $O(n^2)$이 된다. pivot을 랜덤하게 정했을 때 평균 시간복잡도는 $O(n \log n)$. 성능 향상을 위해 부분 문제가 작아지면 삽입 정렬을 함께 사용하기도.

2.3 Selection Problem

n개의 숫자 중 k번째로 작은 숫자를 찾는 문제. 최소 숫자를 k번 찾으면 $O(kn)$, 리스트를 정렬하고 k번째 요소를 선택하면 $O(n \log n)$ 시간이 걸린다. partition으로 이진 탐색을 하면 최악의 경우 $O(n \log n)$.

pivot을 랜덤하게 정해 한쪽으로 치우치지 않는 최선의 경우로 분할하면 부분 문제 크기가 $(3/4)^i n$배씩 줄어들기 때문에 최선의 경우 $O(\sum (3/4)^i n) = O(n)$. 최선의 경우로 분할되는 확률은 $1/2$이므로 평균 시간복잡도는 $2 \times O(n) = O(n)$.

```
def selection(a: list[int], k: int,
              si: int, ei: int) -> int:
  if si >= ei: return

  rpi = random.randrange(si, ei)
  pi = partition(a, si, ei, rpi)
  ll = (pi - 1) - si + 1

  if ll >= k:
    return selection(a, k, si, pi - 1)
  elif k == ll + 1:
    return a[pi]
  rk = k - ll - 1
  return selection(a, rk, pi + 1, ei)
```

2.4 Closest Pair Problem

2차원 평면 상에서 거리가 가장 가까운 한 쌍의 점을 찾는 문제. 모든 쌍의 조합을 계산한다면, ${}_nC_2 = \frac{n(n-1)}{2} = O(n^2)$. 대신 n개의 점을 x 좌표 기준으로 오름차순 정렬하여 반으로 분할하고, 각각의 부분 문제에서 최근점 쌍을 찾으면 더 효율적.

이때 두 부분 사이에도 최근접 쌍이 존재할 수 있으므로 왼쪽 부분의 최근접 거리와 오른쪽 부분의 최근접 거리 중 더 작은 값을 d라고 하면, 왼쪽 부분의 맨 오른쪽 점의 x 좌표에 d를 뺀 범위부터 오른쪽 부분의 맨 왼쪽 점의 x 좌표에 d를 더한 범위까지에서도 최근접 쌍을 탐색해야 한다.

2.5 Master Theorem

분할 정복 알고리즘 분석법. n개의 문제를 분할 정복으로 해결할 때, n개의 문제를 b개의 부분 문제로 분할하고 이 중 a개의 부분 문제만 연산할 때 $f(n)$ 시간이 걸린다고 하면:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Case 1

문제를 분할, 정복하는 과정의 크기가 원 문제보다 작아지는 경우.

$$\text{When} \quad f(n) = O(n^c) \quad \text{where} \quad c < c_{crit}$$
$$\text{then} \quad T(n) = \Theta(n^{c_{crit}})$$

이진 트리 전체 순회: n개의 문제를 2개($b = 2$)의 부분 문제로 나누고 이 중 2개($a = 2$)의 부분 문제애 대해 접근 연산($f(n) = O(1)$)한다.

$$\therefore T(n) = 2T(\frac{n}{2}) + O(1)$$

이때 $O(n^c)$인데 $f(n) = O(1)$이므로, c는 0. 이어서 Case 1 문제인지 확인해본다. ($c < c_{crit}$)

$$\because c_{crit} = \log_b a = \log_2 2 = 1 > c$$
$$\therefore T(n) = \Theta(n)$$

Caes 2

문제를 분할, 정복하는 과정의 크기가 원 문제와 비슷한 경우.

$$\text{When} \quad f(n) = \Theta(n^{c_{crit}} \log^k n)$$
$$\text{then} \quad T(n) = \begin{cases} \Theta(n^{c_{crit}} \log^{k+1} n) & \text{if } k > -1 \\ \Theta(n^{c_{crit}} \log\log n) & \text{if } k = -1 \\ \Theta(n^{c_{crit}}) & \text{if } k > -1 \end{cases}$$

머지 소트: n개의 문제를 2개($b = 2$)의 부분 문제로 나누고 이 중 2개($a = 2$)의 부분 문제에 대해 정렬 연산($f(n) = O(n)$)을 한다. 따라서,

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

이때 $f(n) = \Theta(n^c \log^k n)$인데 $f(n) = O(n)$이므로 $c = 1, k = 0$임을 알 수 있다. ($\because \log^0 n = 1$) 이어서 Case 2a 문제인지 확인해본다. ($c = \log_b a$)

$$\because c_{crit} = \log_b a = \log_2 2 = 1 = c$$
$$\therefore T(n) = \Theta(n \log^{0+1} n)$$
$$= \Theta(n \log n)$$

Case 3

문제가 오히려 커지는 경우.

$$\text{When} \quad f(n) = \Omega(n^c) \quad \text{where} \quad c < c_{crit}$$
$$\text{if} \quad af(\frac{n}{b}) <= kf(n) \quad \text{for} \quad k < 1$$
$$\text{then} \quad T(n) = \Theta f(n))$$

3 Greedy

최적화 문제를 해결하는 알고리즘. 항상 최적의 해를 찾는다는 의미가 아니라, 가능한 해들 중 가장 최적의 해를 찾는 것. 즉, 최적화에 '관한' 문제에 사용할 수 있다. 근시안적 선택으로 부분적인 최적해를 찾고, 이를 모아서 문제의 최적해를 찾는다. 한 번 선택한 데이터를 절대로 번복하지 않는 것이 특징.

3.1 Minimum Spanning Tree (MST)

가중치 그래프에서 사이클 없이 모든 점을 연결시킨 트리(spanning tree) 중 가중치 합이 최소가 되는 트리. spanning tree에는 항상 $n-1$개의 간선이 있다. 그보다 많으면 반드시 사이클이 생긴다.

Kruskal's Algorithm

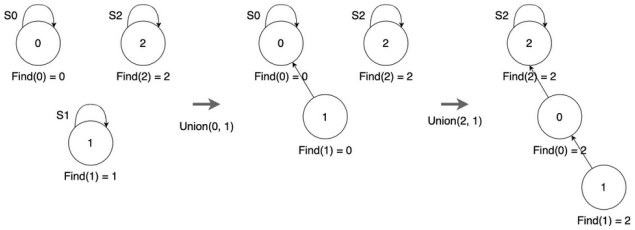
가중치 오름차순으로 n개 간선을 정렬하고, 순서대로 (greedy하게) disjoint set에 추가하여 사이클을 방지한다. 이미 set에 간선이 있는 경우 스킵한다.

아래 구현한 disjoint set에서 `make`는 $O(1)$, `find`는 상각 분석으로 $\Theta(\alpha(n))$. 이때 $\alpha(n)$은 inverse ackemann funciton으로 매우 작은 값. `union`은 `find`와 동일한 시간복잡도.

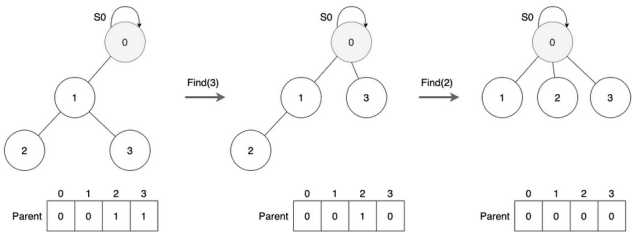
```
class DisjointSet:
  def __init__(self) -> None:
    self.p: dict[str, str] = {}
```

```
    self.l: dict[str, int] = {}

  def make(self, x: str) -> None:
    self.p[x] = x
    self.l[x] = 1

  def find(self, x: str) -> str:
    r = x
    while self.p[r] != r:
      r = self.p[r]
    while self.p[x] != r:
      p = self.p[x]
      self.p[x] = r
    return r

  def union(self, x: str,
                  y: str) -> None:
    x = self.find(x)
    y = self.find(y)
    if x == y:
      return
    if self.l[x] < self.l[y]:
      x, y = y, x
    self.p[y] = x
    self.l[x] = self.l[x] + self.l[y]
```

`union` 과정에서 최악의 경우 선형 트리가 만들어져 `find`에 $O(n)$ 시간이 소요될 수 있다.



이를 방지하기 위해 `find`할 때마다 찾고자 한 x의 상위 노드가 루트 노드가 아닌 경우 x의 상위 노드를 루트로 갱신한다. 이렇게 하면 평평한 트리가 만들어져 차후 빠르게 `find`를 수행할 수 있음.

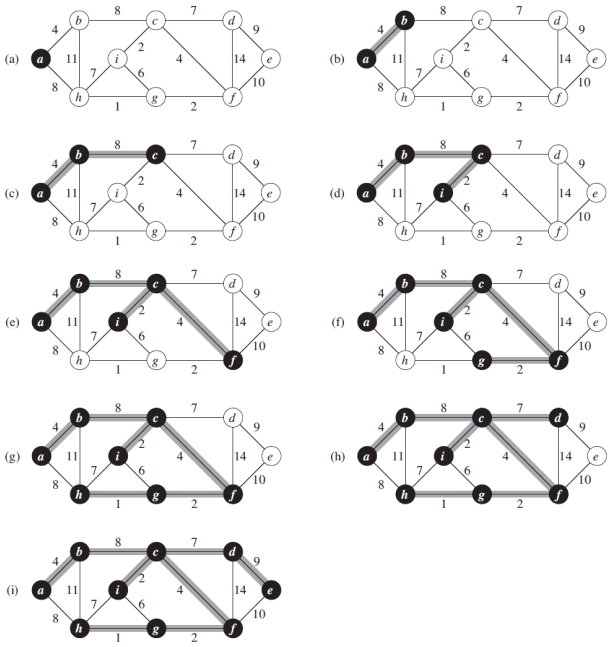


kruskal's algorithm에서는 정렬에 가장 오랜 시간이 걸리므로 시간복잡도는 $O(n \log n)$. 이때 n은 정점의 개수가 아닌 간선의 개수.

```
def kruskal_mst(
  g: tuple[list[str],
           list[tuple[str, str, int]]]
) -> list[tuple[str, str, int]]:
  r = []
  s = DisjointSet()
  e = sorted(g[1], key=lambda x: x[2])

  for v in g[0]:
    s.make(v)

  for (a, b, w) in e:
    u = s.find(a)
    v = s.find(b)
    if u != v:
      r.append((a, b, w))
      s.union(u, v)

  return r
```

Prim's Algorithm

그래프에서 임의의 정점을 선택하고 $n-1$개의 간선을 집합 T에 하나씩 추가해 트리를 구성하는 알고리즘. 이때 현재 알고 있는 가중치를 저장하는 테이블 D를 사용한다.



(a) 임의의 정점 v_a을 선택하고 집합 T에 추가한다. $D = (e_b^4, e_h^8)$.

(b) $T^c \wedge min(D) = e_b$와 그에 이어진 정점 v_b가 T에 없다면 추가한다. $D = (e_b^4, e_h^8, e_c^8)$. v_b와 이어진 e_h^{11}은 D에 있는 e_h^8보다 크므로 갱신하지 않음.

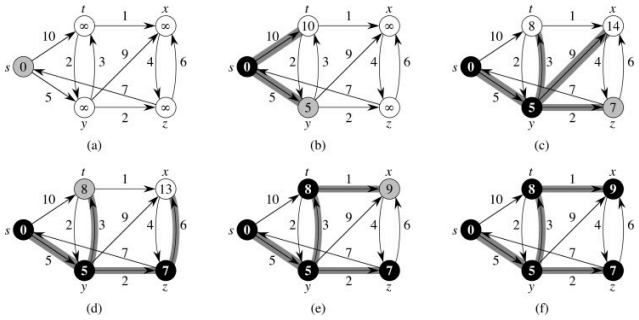
(c) $T^c \wedge min(D) = e_c$와 그에 이어진 정점 v_c가 T에 없다면 추가한다. $D = (e_b^4, e_h^8, e_c^8, e_i^2, e_f^4, e_d^7)$.

(d) 이렇게 정점과 간선을 하나씩 추가하다가 T에 추가된 간선의 개수가 $n-1$개가 되면 종료한다.

항상 T에 속하지 않은 정점을 추가하기 때문에 사이클이 만들어지지 않는다. 시간복잡도는 $O(n^2)$. 여기서 n은 정점 개수, 앞서 본 kruskal algorithm은 간선 개수이므로 일률적으로 비교가 불가하다.

3.2 Dijkstra's Algorithm

출발점에서 다른 모든 정점까지의 최단 경로를 찾는 알고리즘. 현재 정점에서 가중치가 가장 낮은 간선을 선택하고, 출발점에서 해당 간선에 이어진 정점까지의 가중치 총합을 기록해 다른 간선의 비용과 비교한다. 정점 s에서 출발해 다른 정점까지의 최단 거리를 구한다.



(a) s에서 출발. 다른 정점까지의 거리를 모르기 때문에 모두 무한대.

(b) s와 이어진 $t = 10, y = 5$ 중 더 가까운 y 선택.

(c) $t = 8$로 갱신. y와 이어진 $t = 8, x = 14, z = 7$ 중 가장 가까운 z 선택.

(d) $x = 13$으로 갱신. y와 이어진 $t = 8, z$와 이어진 정점 $x = 13$ 중 더 가까운 t 선택.

(e) $x = 9$로 갱신. t와 이어진 $x = 9, x$ 선택.

(f) s로부터 모든 정점까지의 최단 거리를 알게 됨.

시작점을 제외한 모든 정점에 대해 루프가 $(n-1)$회 반복되고, 루프 내에서 최소의 점을 찾는 데 $O(n)$, 정점에 연결된 정점의 수가 최대 $O(n-1)$개이므로 각 정점을 갱신하는 데 $O(n)$, 따라서 시간복잡도는 $(n-1) \times \{O(n) + O(n)\} = O(n^2)$.

3.3 Fractional Knapsack Problem

배낭 문제는 결국 모든 경우의 수를 살펴야 하기 때문에 trivial하지 않은 문제. 하지만 부분 배낭 문제는 그리디 알고리즘으로 해결이 가능하다.

(a) 각 물건의 단위 무게 당 가치를 계산한다: $O(n)$

(b) 가치를 기준으로 내림차순 정렬한다: $O(n \log n)$

(c) 순서대로 담는다. 공간이 부족하면 최대 용량에서 현재 용량을 뺀 만큼만 담는다: $O(n)$

3.4 Set Cover Problem

주어진 집합 U의 원소를 가장 많이 포함한 집합 S를 선택하는 과정을 반복. S를 선택하면 U에서 S에 포함되는 원소를 지운다. 이때 S를 찾으려면 남은 S_i들을 각각 U와 비교해야하므로 $O(n^2)$, 이를 n회 반복하므로 시간복잡도는 $O(n^3)$.

3.5 Job Scheduling Problem

작업 수행 시간이 중복되지 않도록 최소한의 머신에 배정하는 문제. 최적해를 찾으려면 빠른 시작 시간 작업 우선(earliest start time first) 배정해야 한다.

시작 시간이 빠른 순으로 작업 목록을 정렬하고 ($O(n \log n)$), 가장 일찍 시작하는 작업부터 배정한다. 만약 시간이 겹친다면 새로운 머신을 만들어 배정한다 ($O(m)$, m = 머신 수). 총 n번 루프를 실행하므로 시간복잡도는 $O(n \log n) + O(mn)$.

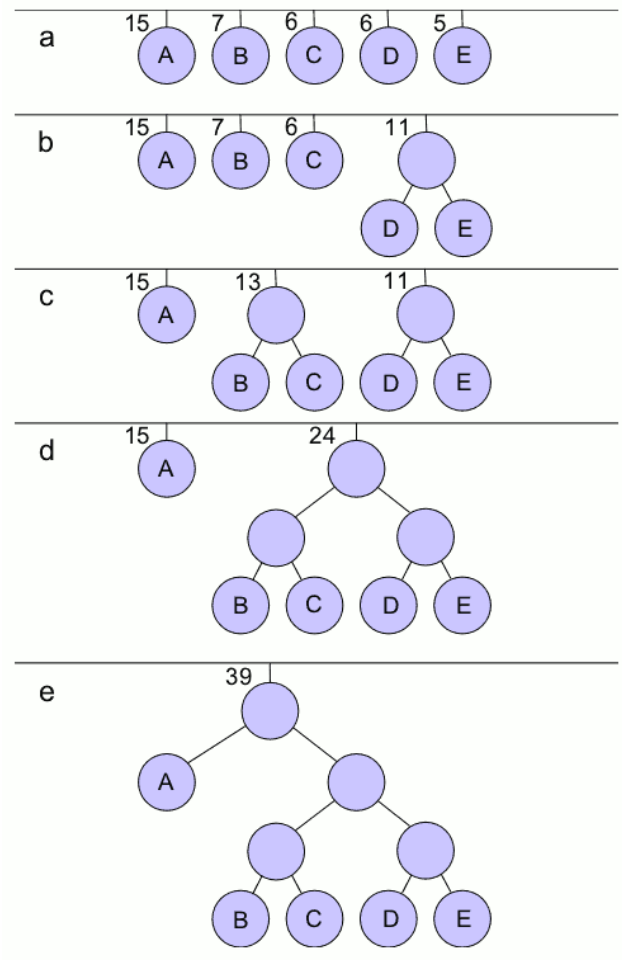
3.6 Huffman Coding

문자의 빈도수에 따라 이진 코드를 할당하고, 트리를 구성해 인코딩하는 알고리즘. 큐에서는 항상 빈도가 가장 낮은 정점 2개를 선택하며, 큐에 정점이 2개 미만이 되면 종료한다.

(a) 각 문자의 빈도수로 우선순위 큐를 만든다.

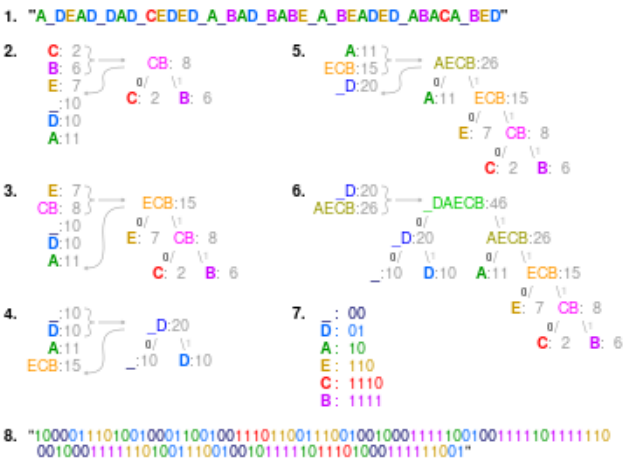
(b) 빈도수가 가장 작은 D(6), E(5) 정점을 pop해 새 정점 DE(11)의 하위 정점으로 만든다.

(c) 빈도수가 가장 작은 B(7), C(6) 정점을 pop해 새 정점 BC(13)의 하위 정점으로 만든다.



(d) 빈도수가 가장 작은 BC(13), DE(11) 정점을 pop 해 새 정점 BCDE(24)의 하위 정점으로 만든다.

(e) 빈도수가 가장 작은 BCDE(24), A(15) 정점을 pop 해 새 정점 BCDEA(39)의 하위 정점으로 만든다.



여기서 왼쪽 간선에는 0, 오른쪽 간선에는 1을 부여하면 인코딩이 된다. 시간복잡도는 $O(n \log n)$.

4 Dynamic Programming

최적화 문제를 해결하는 알고리즘. 문제를 쪼개서 해결한다는 점에서 분할 정복과 똑같지만, 다른 문제의 부분 문제를 사용한다는 점에서 다름. 가령 피보나치 수열을 분할 정복으로 구현하면 같은 문제를 중복해서 풀기 때문에 시간복잡도가 $O(2^n)$.

```
def f(n: int) -> int
  if n <= 1: return n
  return f(n - 1) + f(n - 2)
```

동적 계획법으로 구현하면 효율적이다. 메모이제이션을 이용해 계산한 해를 테이블에 저장해두면 $O(n)$.

```
m = [0, 1] + [-1 for _ in range(n - 1)]
def f(n: int) -> int:
  if n <= 1:
    return n
  if m[n] == -1:
    m[n] = f(n - 1) + f(n - 2)
  return m[n]
```

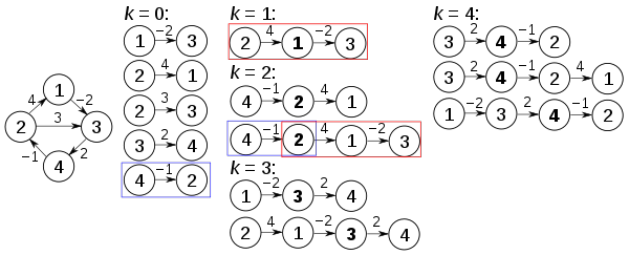
동적 계획법은 최적성 원칙 또는 최적 부분 구조라는 특성을 가지고 있음. 부분 문제들의 최적해를 이용해 원 문제에 대한 최적해를 구할 수 있다는 것. 즉, 문제의 최적해 속에 부분 문제의 최적해가 포함되어 있고, 부분 문제의 해 속에 그 보다 작은 부분 문제의 해가 포함되어 있다.

4.1 Floyd-Warshall Algorithm

모든 정점 쌍의 모든 최단 경로를 찾는 알고리즘. 다익스트라 알고리즘($O(n^2)$)을 n회 수행한 것과 같은 $O(n^3)$ 시간이 걸린다.

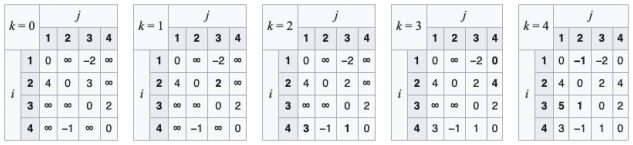
경로 표기 $D^k_{i,j}$는 정점 $1, 2, \cdots, k$의 경유를 고려한 정점 i부터 j까지의 모든 경로 중 가장 짧은 경로의 거리. (k를 경유할 수도, 경유하지 않을 수도 있다.) 따라서 $D^0_{i,j}$는 두 정점을 잇는 간선의 가중치를 의미한다.

- $k = 0$: 어떤 정점도 경유하지 않았을 때 도달할 수 있는 정점 목록.
- $k = 1$: 정점 1을 경유하는 최단 경로는 $D^1_{2,3}$ 하나. $D^0_{2,3}$보다 짧으므로 거리를 업데이트한다.



- $k = 2$: 정점 2를 경유하는 최단 경로의 목록. 이때 $D^2_{4,3}$는 $D^0_{4,2}$와 $D^1_{2,3}$의 합이다. 즉, $D^k_{i,j} = min(D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j})$. 다른 부분 문제의 해를 사용하는 동적 계획법의 예.
- $k = 3$: 정점 3을 경유하는 최단 경로의 목록.
- $k = 4$: 정점 4를 경유하는 최단 경로의 목록.

마지막 단계에서 모든 정점 간의 최단 거리를 알게 된다. 표로 표현하면 아래와 같다. 업데이트된 거리는 볼드체로 표기.



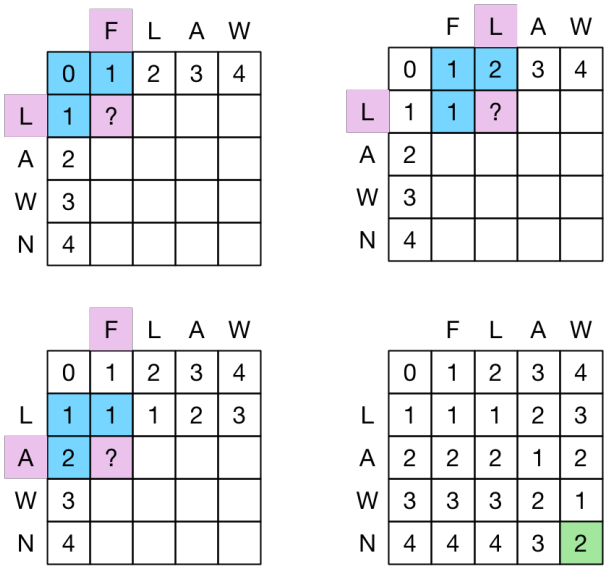
4.2 Matrix Chain Multiplication

연속적인 행렬 곱을 효율적으로 계산하기 위한 연산 순서를 찾는 알고리즘. 행렬의 결합법칙을 이용.

```
def matrix_chain_mult(p: list[int],
                      n: int) -> int:
  m = [[0 for x in range(n)]
       for x in range(n)]

  for i in range(1, n):
    m[i][i] = 0

  for L in range(2, n):
    for i in range(1, n-L + 1):
      j = i + L-1
      m[i][j] = sys.maxsize
      for k in range(i, j):
        q = m[i][k] + m[k + 1][j]
            + p[i-1]*p[k]*p[j]
        if q < m[i][j]:
          m[i][j] = q

  return m[1][n-1]
```

4.3 Minimum Edit Distance

두 문자열의 유사도를 판단하는 알고리즘. 어떤 문자열을 최소한의 횟수로 `insert`, `replace`, `delete`하여 다른 문자열로 변형. 모든 부분 문제에 대해 최소 동작 테이블을 만든다. 이때 →: `insert`, ↘: `replace`, ↓: `delete`.



예를 들어 LAWN를 FLAW로 변형하는 경우 최종적으로 2번의 변경이 필요하다.

(a) L → F: $\min((1+1), (0+1), (1+1)) = 1$.

(b) L → FL: $\min((1+1), (1+0), (2+1)) = 1$.

(c) LA → F: $\min((2+1), (1+1), (1+1)) = 2$.

(d) LAWN → FLAW: $\min((3+1), (2+1), (1+1)) = 2$.

```
def minimum_edit_distance(x: str,
                          y: str) -> int:
  (m, n) = (len(x), len(y))

  t = [[0 for _ in range(n + 1)]
       for _ in range(m + 1)]

  for i in range(m + 1): t[i][0] = i
  for j in range(n + 1): t[0][j] = j

  c = 0
  for i in range(1, m + 1):
    for j in range(1, n + 1):
      if x[i - 1] == y[j - 1]:
        c = 0
      else:
        c = 1
```

```
      t[i][j] = min(t[i][j - 1] + 1,
                    t[i - 1][j - 1] + c,
                    t[i - 1][j] + 1)

  return t[m][n]
```

시간복잡도는 테이블을 만드는 시간: $O(mn)$.

4.4 1-0 Knapsack Problem

n개의 물건과 각 물건의 무게 w_i와 가치 v_i가 주어지고, 배낭의 용량이 c일 때, 배낭에 담을 수 있는 물건의 최대 가치를 찾는 문제. 각 물건이 배낭에 담기지 않는 경우는 0, 담긴 경우는 1로 취급.

```
def knapsack(s: list[tuple[int, int]],
             c: int) -> int:
  sl = len(s)
  t = [[0 for _ in range(c)]
       for _ in range(sl - 1)]

  for i in range(1, sl):
    for w in range(1, c):
      wi = s[0][i]
      vi = s[1][i]
      if (wi > w):
        t[i, w] = t[i - 1, w]
      else:
        t[i, w] = max(t[i - 1, w],
                      t[i - 1, w - wi] + si)
  return t[n, c]
```

배낭 문제는 NP-Complete 문제. P 문제는 다항 시간 내에 해결할 수 있는 문제, NP 문제는 다항 시간 내에 해결할 수 없는 문제. NP-Hard 문제는 NP에 속하는 어떤 문제를 다항 시간 내에 다른 문제로 변경할 수 있는 문제. NP-Complete 문제는 NP이면서 NP-Hard인 문제.

4.5 Coin Change Problem

동전 거스름돈 문제를 그리디하게 해결하려면 적절한 동전 단위가 전제되어야 한다. 다양한 단위의 거스름돈을 다루기 위해 0부터 n원까지의 테이블을 만든다. 동전의 단위가 $d_{16} = 16$, $d_{10} = 10$, $d_5 = 5$, $d_1 = 1$ 원일 때 20원을 거슬러줘야 한다면.

(a) 0부터 20까지의 배열 C를 $C[0] = 0$, 그 외는 무한대로 초기화.

(b) $C[i_1]$: $d_1 \leq i_1$, $C[i_1 - d_1] + 1 = 1$.

(c) $C[i_2]$: $d_1 \leq i_2$, $C[i_2 - d_1] + 1 = 2$.

(d) $C[i_5]$: $d_5 \leq i_5$, $C[i_5 - d_5] + 1 = 1$. $d_1 \geq i_5$ 조건도 참이긴 하지만, $C[i_5 - d_1] + 1 < C[i_5] = C[4] + 1 < C[i_5] = 4 < 1$ 조건을 만족하지 않음.

(e) $C[i_{20}]$: $d_{16} \leq i_{20}$, $C[i_{20} - d_{16}] + 1 = 5$. 그런데 $C[i_{20} - d_{10}] + 1 < C[i_{20}] = C[10] + 1 < 5 = 2 < 5$ 조건을 만족하므로 2로 갱신.

5 Sorting

5.1 Bubble Sort

입력 배열을 순회하며 두 요소를 선택한 뒤, 정렬되어 있다면 그대로 두고 아니라면 swap하며 정렬.

```
def bubble_sort(A: list[int]):
  for i in range(len(A) - 1, 0, -1):
    for j in range(i):
      if A[j] > A[j + 1]:
        A[j], A[j + 1] = A[j + 1], A[j]
```

요소의 입력 순서대로 뒤쪽에 있는 요소와 앞쪽에 있는 요소를 비교해 swap하므로 stable하다. 단, L4에서 크거나 같음(>=) 비교를 하면 값이 같은 요소도 swap하게 되므로 unstable해진다.

5.2 Selection Sort

입력 배열을 순회하며 커서 뒤의 최소값과 커서의 값(정렬되지 않은 맨 앞쪽 값)을 swap하며 정렬.

```
def selection_sort(A: list[int]):
  for i in range(len(A) - 1):
    min_idx = i
      for j in range(i + 1, len(A)):
        if A[j] < A[min_idx]:
          min_idx = j
      A[i], A[min_idx] = A[min_idx], A[i]
```

커서 값과 최소값을 swap하는 순서가 기존 순서와 상관없이 일으나므로 stable하지 않다. 단, 값을 swap하지 않고 insert하도록 구현하면 stable해진다.

```
def stable_selection_sort(A: list[int]):
  n = len(A)
  for i in range(n-1):
    min_idx = i
    for j in range(i+1, n):
      if A[min_idx] > A[j]:
        min_idx = j
    key = A[min_idx]
    while min_idx > i:
      A[min_idx] = A[min_idx-1]
      min_idx -= 1
    A[i] = key
```

5.3 Insertion Sort

입력이 이미 정렬되어 있으면 모든 원소가 제자리에 있게 되므로 최선의 경우 $O(n)$. 따라서 원소 개수가 적거나 거의 정렬되어 있을 때 좋은 성능을 보인다.

```
def insertion_sort(A: list[int]):
  for i in range(1, len(A)):
    k = A[i]
    j = i - 1
    while j >= 0 and k < A[j]:
      A[j + 1] = A[j]
      j -= 1
    A[j + 1] = k
```

입력 요소의 순서를 유지한 상태에서 자신보다 큰 요소들을 한 자리씩 뒤로 밀고 그 앞에 insert하는 방식이므로 stable하다. 단, 4번째 줄에서 크거나 같음(>=) 비교를 하면 값이 같은 요소도 뒤로 밀게 되므로 unstable해진다.

5.4 Shell Sort

삽입 정렬을 응용한 알고리즘. 대략 정렬을 해놓고 삽입 정렬을 한다.

1. 우선 gap에 따라 숫자를 여러 그룹으로 나누고, 각 그룹 내에서 삽입 정렬을 한다.
2. 이어서 gap을 줄이며 같은 방식으로 정렬을 수행한다.
3. 마지막에는 반드시 gap을 1로 놓고 정렬한다. 다른 그룹에 속해 서로 비교되지 않은 숫자가 있을 수 있기 때문. 이는 삽입 정렬 그 자체다.

```
def shell_sort(A: list[int]):
  h = len(A) // 2
  while h > 0:
    for i in range(h, len(A)):
      k = A[i]
      j = i
      while j >= h and k < A[j - h]:
        A[j] = A[j - h]
        j = j - h
      A[j] = k
    h = h // 2
```

시간복잡도는 $O(n^2)$. gap에 따라 성능이 좌우된다. 가장 좋은 성능의 gap으로 알려진 히바드 간격 $2^k - 1$

은 $O(n^{1.5})$. 많은 실험으로 통해 $O(n^{1.25})$로 알려짐. 입력 크기가 매우 크지 않은 경우 좋은 성능을 보인다.

5.5 Heap Sort

힙을 이용하는 정렬. 최대 힙은 항상 상위 노드의 값이 하위 노드의 값보다 큰 완전 이진트리. 과정을 $(n-1)$번 반복하므로 총 시간 복잡도는 $O(n) + (n-1) * O(\log n) = O(n \log n)$.

1. 리스트를 최대 힙으로 만든다. 이때 순서는 BFS. heapify로 $O(n)$ 시간이 든다.
2. 루트와 마지막 노드를 교환한다. 이제 마지막 노드는 정렬된 상태이므로, 힙 트리에서 마지막 노드를 제거한다.
3. 깨진 힙을 다운 힙으로 재구조화한다. 다운 힙 시에는 대상 노드와 그 하위 두 노드 중 더 큰 것과 교환한다. 이때 heapify에 드는 시간은 트리의 높이인 $O(\log n)$.
4. 2, 3을 반복한다.

힙을 배열에 저장할 때는 `A[0]`를 비워두고, `A[1]`부터 `A[n]`까지 힙 노드들을 트리의 층별로 저장한다. 가령 `A[1]`에 루트 노드를 저장했다면, `A[2]`에는 왼쪽 하위 노드를, `A[3]`에는 오른쪽 하위 노드를 저장한다. 완전 이진 트리이기 때문에 아래의 공식이 유효하다.

- `A[i]`의 상위 노드: `A[i / 2]`
- `A[i]`의 왼쪽 하위 노드: `A[2 * i]`
- `A[i]`의 오른쪽 하위 노드: `A[2 * i + 1]`

최대 힙으로 구현된 힙 정렬은 루트 노드와 마지막 노드를 교환하는 과정에서 앞에 있는 중복 요소가 그 뒤에 배치된다. 따라서 unstable하다.

5.6 Radix Sort

지금까지 다룬 정렬 알고리즘은은 숫자와 숫자를 비교를 하는 비교 정렬(comparison sort)이었음. 반면 기수 정렬은 숫자를 한 자리씩 부분적으로 비교함.

기수 정렬은 비교 정렬과 달리 숫자를 부분적으로 비교한다. 1의 자리부터 k번째 자리로 진행하는 LSD 기수 정렬을 살펴보자. k자리 부터 1의 자리로 진행하는 방식은 MSD라고 한다.

1. 우선 각 숫자의 1의 자리를 비교해 정렬한다.
2. 다음엔 10의 자리를 비교해 정렬한다. 중복되는 숫자가 있는 경우 전 단계의 순서를 그대로 따라야 한다. 이것을 안정성이라고 한다.
3. 마찬가지로 100의 자리, 1000의 자리를 비교해 안정 정렬한다.

기수 정렬의 시간복잡도는 $O(k(n+r))$. 이때 자리수 k나 상수 r이 입력 크기 n보다 작으면 $O(n)$이다. 이는 각 자리에 들어갈 숫자가 전제되기 때문에 가능한 것. 숫자를 그대로 인덱스로 쓸 수 있다. 문자열이라도 어떤 문자만 사용할지 전제하면 가능하다. 이렇게 한정된 조건에서만 사용할 수 있기 때문에 정렬 문제의 하한이 $O(n)$이라고 할 수는 없음.

비교 정렬의 하한

비교 정렬을 이용하는 경우 정렬 문제의 하한을 알아보자. 문제의 하한이란 그 문제를 풀기 위한 어떠한 알고리즘이라도 해를 구하려면 적어도 하한의 시간복잡도만큼 시간이 걸린다는 뜻. 3개의 숫자를 정렬할 때 결정 트리는 리프 노드의 수가 3!개인 이진트리. 여기에는 정렬에 불필요한 내부 노드가 없다. 따라서 어느 경우라도 서로 다른 3개의 숫자를 정렬하려면 적어도 3번의 비교가 필요하다.

n개의 서로 다른 숫자를 정렬하는 결정 트리의 높이를 계산하려면 이진트리의 속성을 이용한다: k개의 리프 노드가 있는 이진 트리의 높이는 $\log k$보다 크다. 따라서 $n!$개의 리프를 가진 결정 트리의 높이는 $\log(n!)$보다 크다. 이때 $\log(n!) = O(n \log n)$이므로, 비교 정렬의 하한은 $O(n \log n)$.

안정성

입력에 중복된 숫자가 있을 때 정렬 후에도 중복된 숫자의 순서가 입력 순서와 동일한 것을 말한다. 가령 $3a, 2b, 4c, 3d, 1e$가 입력됐을 때 안정한 정렬(stable sort)의 결과는 $1e, 2b, 3a, 3d, 4c$가 되어야 한다. 기수 정렬이 이를 지키지 않으면 전 단계의 정렬이 무효해진다.

안정 정렬은 원래 원소들 간의 상대적인 순서를 유지하면서 정렬해야 할 때, 정렬 기준이 여러 개일 때 필요하다. 버블 정렬, 삽입 정렬, 병합 정렬은 안정적이다. 선택 정렬, 쉘 정렬, 퀵 정렬, 힙 정렬은 불안정하다. 병합 정렬의 경우 분할된 부분 리스트가 기존 순서를 유지하고 있고, 이를 병합할 때도 순서를 유지한다. 반면 퀵 정렬의 경우 pivot을 기준으로 원소를 왼쪽, 오른쪽 부분으로 재배치하고 정렬을 하기 때문에 기존 순서를 유지할 수 없다.

5.7 External Sort

내부 정렬(Internal sort): 입력의 크기가 메인 메모리의 공간보다 크지 않은 경우에 수행되는 정렬. 지금까지 다룬 정렬 알고리즘은 모두 내부 정렬.

외부정렬(External sort): 입력의 크기가 메인 메모리보다 큰 경우. 보조 기억 장치에 있는 입력을 여러 번에 나누어 메인 메모리에 읽어 들인 후 정렬하여 보조 기억 장치에 다시 저장하는 과정을 반복한다.

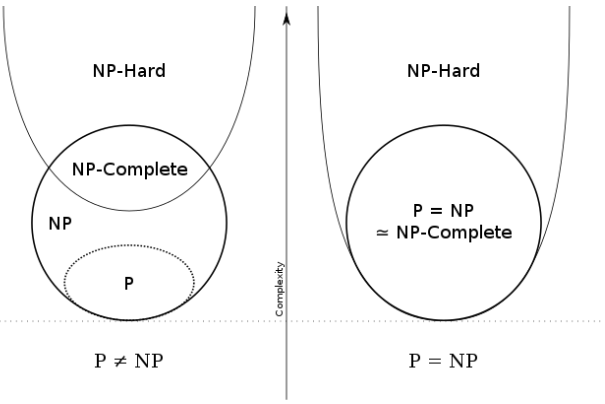
1. HDD A에서 순차적으로 데이터를 읽어 메인 메모리에 로드한다.
2. 메인 메모리는 데이터가 로드될 때마다 내부 정렬을 하고 HDD B에 저장한다.
3. HDD B에 저장된 데이터 블록 2개를 선택하고 각 블록의 데이터를 부분적으로 메인 메모리에 로드해 병합한다.
4. 메인 메모리는 두 블록을 번갈아가며 순차적으로 HDD A에 저장한다.

외부정렬은 전체 데이터를 몇 번 처리하는지에 따라 시간복잡도를 측정. 전체 데이터를 읽고 쓰는 것을 패스라고 한다. 입력 크기가 N이고 메인 메모리 크기가 M이라면 시간복잡도는 $O(\log(N/M))$.

6 NP-Complete

문제는 크게 두 개 집합으로 분류할 수 있다.

- 다항식 시간복잡도 알고리즘으로 해결되는 문제 집합.

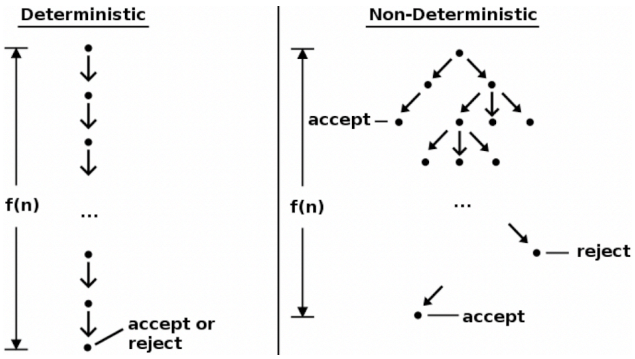


  - **P**: 다항식 시간 내에 결정론적으로 해를 찾을 수 있는 문제의 집합. 즉, 시간복잡도가 $O(n^k)$에 포함되는 경우.
  - **NP**: 다항식 시간 내에 비결정론적으로 해결되는 문제의 집합. 즉, NP 문제는 비결정 튜링 머신으로 다항식 시간에 해결 가능하다. 문제의 조건을 만족하는 해가 존재하는지 판단하는 결정 문제를 다항식 시간에 확인할 수 있는 경우에도 NP 문제.
- 다항식보다 큰 시간복잡도 알고리즘으로 해결되는 문제 집합.
  - **NP-Complete**: NP 문제 중 지수 시간의 시간복잡도를 가진 알고리즘으로 해결되는 문제 집합.
  - **NP-Hard**: 어떤 NP 문제보다도 해결하기 어려운 문제.

P 문제 집합이 NP 문제 집합에 속하는 이유: P 문제를 해결하는데 다항식 시간이 걸리므로, 이를 NP 알고리즘이 문제의 해를 다항식 시간에 확인하는 것과 대응시킬 수 있기 때문.

6.1 Turing Machine

튜링 머신은 입력에 대해 출력을 내는 가상의 컴퓨터. 비결정 튜링 머신은 특정 상태에서 움직일 수 있는 다음 상태의 개수가 하나로 정해져 있지 않은 튜링 머신을 말한다. 일정 시간 안에 해를 찾을 수도, 못 찾을 수도 있다. NP 문제가 비결정 튜링 머신으로 해결된다는 건 비결정 튜링 머신이 아주 운이 좋은 경우 다항식

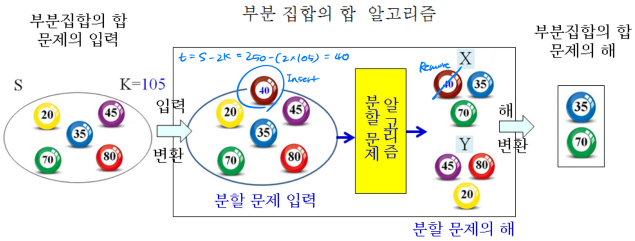


시간에 해결할 수 있다는 것. 유한상태기계는 입력에 따라 상태가 변화하는 튜링 머신.

6.2 Reduction

NP 완전 문제 A를 해결하기 위해 문제 B를 해결하는 알고리즘을 이용하는 것. 이렇게 하면 밖에서는 A 문제를 푼 것으로 보인다.

가령 문제 A는 원소의 합이 K가 되게 만드는 부분집합을 찾는 'subset sum' 문제. $O(2^n)$ 시간이 걸리는 NP 완전 문제임. 문제 B는 두 부분 집합 X, Y의 합을 같게 만드는 'partition' 문제.



변환의 시간복잡도는 입력 변환 시간, 문제 B 알고리즘의 수행 시간, 출력 변환 시간의 총합.

입력과 출력의 변환은 다항식 시간에 가능하다. 이때 문제 A는 문제 B로 reducible하다. 양쪽으로 reducible하다면 transitive한 관계. 추이 관계로 NP-C 문제들이 서로 얽혀 있어서 어느 하나의 문제에 대해 다항식 시간에 해를 구할 수 있다면 다른 모든 NP-C 문제도 다항식 시간에 해를 구할 수 있다.

7 Approximation

NP-C 문제는 실세계에서 광범위하게 활용되지만, 다항식 시간에 해결할 방법이 발견되지 않음. 근사 알고리즘은 최적해 찾는 걸 포기하고 다항식 시간에 모든 입력에 대한 해를 찾음. 단, worst case에서 근사해가 최적해에 얼마나 가까운지 나타내는 근사 비율을 함께 제시해야 함.

7.1 Traveling Salesman Problem

임의의 도시에서 출발해 다른 모든 도시를 한 번씩만 방문하고 출발지로 돌아오는 경로의 길이를 최소화하는 문제.

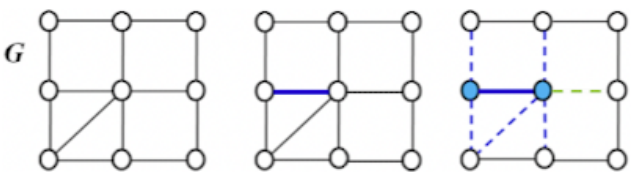
1. MST를 만든다.
2. 한 도시에서 출발해 모든 도시를 방문하고 돌아오는 순서를 구한다.
3. 앞서 구한 순서를 따라 방문하되, 중복 방문하는 도시를 순서에서 제거한다. (단, 마지막 도시는 출발 도시와 같으므로 제거하지 않는다.)

이때 근사 비율은 MST 선분의 가중치 합(M)을 간접적인 최적해로 사용해 구한다. 실제 최적해 값도 M보다는 같거나 클 것이기 때문이다. 앞의 알고리즘에 따르면 근사 비율은 $2M/M = 2$. 즉, 근사해의 값이 최적해의 2배를 넘지 않는다.

7.2 Vertex Cover

그래프에서 각 선분의 양 끝점 중 적어도 하나의 끝점을 포함하는 점들의 집합 중 최소 크기의 집합을 찾는 문제. CCTV를 설치한다고 하면 모든 간선을 감시할 수 있는 위치를 찾는 것.

따라서 사실 정점이 아니라 간선을 선택하는 것. 간선 하나를 선택하면 정점 두 개를 선택하는 셈. 임의의 선분을 선택하면 주변의 커버된 선분은 이후 선택하지 않는다. 이렇게 선택된 선분 집합을 극대 매칭이라고 함.



극대 매칭을 찾기 위해 하나의 간선을 선택할 때 $O(n)$, 그래프의 간선 수가 m이라면 각 선분에 대해 $O(n)$ 시간. 따라서 $O(mn)$.

근사 비율은 극대 매칭을 간접적인 최적해로 사용해 구한다. 어떠한 정점 커버라도 극대 매칭에 있는 선분을 커버해야 하기 때문. 근사해의 값은 극대 매칭 선분 수의 2배. 따라서 근사 비율은 극대 매칭의 각 선분의 양 끝점의 수를 극대 매칭의 간선 수로 나눈 2.

7.3 Bin Packing

n개의 물건과 용량이 C인 통이 있을 때, 모든 물건을 가장 적은 수의 통에 채우는 문제. 물건을 통에 넣으려면 통에 그 물건이 들어갈 여유가 있어야 한다. 그리디하게 근사하는 몇가지 방법이 있음.

- 최초 적합: 첫 번째 통부터 차례로 살펴보며, 가장 먼저 여유가 있는 통을 선택.
- 다음 적합: 직전에 물건을 넣은 통에 여유가 있으면 선택.
- 최선 적합: 기존 통 중에서 물건이 들어가면 남는 부분이 가장 작은 통을 선택.
- 최악 적합: 기존 통 중에서 물건이 들어가면 남는 부분이 가장 큰 통을 선택.

다음 적합은 직전에 사용한 통만 살펴보면 되므로 $O(n)$. 이때 근사 비율은 최적해의 2배를 넘지 않는 2.

나머지 방법은 물건을 넣을 때마다 모든 통을 살펴봐야 하므로 $O(n^2)$. 2개 이상의 통이 1/2 이하로 차 있는 경우는 없다. 따라서 2.

7.4 Job Scheduling

작업의 시작 시각이 아니라 수행 시간을 고려. 그리디하게 근사할 수 있음. 가장 빠르게 현재 작업을 수행할 수 있는 기계에 배정한다. n개의 작업에 대해 m개 기계의 종료 시간을 확인해야 하므로 $O(nm)$. 근사 비율은 2.

7.5 Clustering

K-Means와 동일한 문제. n개의 점에 대해 k개의 센터까지의 거리를 각각 계산해야 하므로 $O(k^2n)$. 근사 비율은 2.

8 Solution Finding

NP-C 문제는 n이 너무 크지 않다면, 느리더라도 모든 경우를 다 해보고 해결할 수 있다. 미로 탐색의 경우 오른손을 짚고 가는 DFS(Depth First Search)로 해결 가능. 하지만 최단 경로가 아니다.

8.1 Backtracking

해를 찾는 도중에 막히면(도달한 리프 노드가 해가 아니라면) 되돌아가서 다시 해를 찾는 기법. 특정 조건을 최소화/최대화하는 조합을 찾는 최적화 문제 또는 결정 문제를 해결할 수 있다.

트리를 만들어서 모든 리프 노드에서의 값을 확인해본다. 단, 현재 알고 있는 최적의 값(최단 경로)보다 값이 더 커지는 시점에 그 하위 트리는 탐색하지 않는 pruning을 할 수 있다.

백트래킹의 시간 복잡도는 상태 공간 트리의 노드 수에 비례. 최악의 경우 모든 노드를 탐색하는 완결 탐색(exhaustive search)과 같아지므로 지수 시간이 걸린다. 평균적으로는 pruning을 하기 때문에 훨씬 효율적임.

8.2 Branch-and-Bound

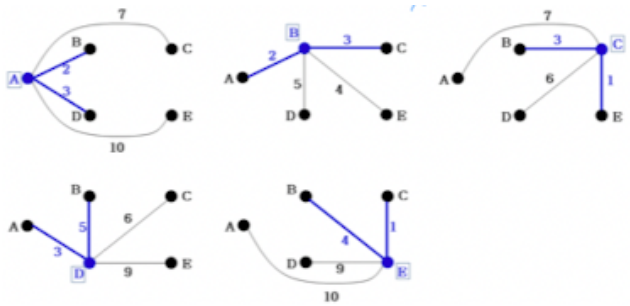
트리의 각 노드(state)에 한정값(bound)을 부여하고, 노드의 한정값을 활용해 pruning함으로써 백트래킹보다 빠르게 해를 찾는다. 백트래킹과 다른 점은 어떤 노드의 가능성이 높을 지 예측하고, 가능성이 높은 노드부터 탐색한다는 것.한정값을 잘못 정하면 단순히 루트 노드와 가까운 노드부터 탐색하게 되므로 BFS와 같아진다. 미래를 고려해 정해야 하는데, 대충 정하면 더 나은 경로가 있음에도 선택하지 않는 문제가 생길 수도. Branch-and-Bound는 깊이 우선이 아니라 최선 우선이기 때문에 'best first search'.

공짜 점심은 없다. 일반적으로 백트래킹보다 효율적이지만, 메모리 사용량이 커질수도. 백트래킹은 스택을 쓰기 때문에 트리의 깊이만큼만 메모리를 사용함. 따라서 공간복잡도가 $O(n)$. 하지만 best first search는 pruning이 잘 안 된 경우 큐 크기가 엄청 커질 수 있

음. 참고로 BFS는 exhaustive함. BFS는 다음 레벨의 모든 노드를 큐에 담아야 하기 때문에 공간복잡도가 $O(n!)$.

TSP

임의의 위치 x에서의 한정값은 x까지의 길이에 x를 떠나 남은 점을 한번씩 방문하고 시작점으로 돌아오는 경로의 예측 길이.



1. 초기 한정값: 각 노드에서의 가장 작은 간선 두 개의 합을 모두 더하고 2로 나눈다.

A, B , [A, D], [A, C], [A, E] 중 하나를 선택.

A, B 를 선택: A를 거쳐온 B위치에서 한정값을 계산.

A* Search

branch-and-bound의 개선판. $f(n) = g(n) + h(n)$으로 노드의 가치를 표현할 수 있다.

1. f: 시작점에서 현재 노드를 지나 도착점까지의 예상 거리.
2. g: 시작점에서 현재 노드까지의 실제 거리.
3. h: 현재 노드에서 도착점까지의 예상 거리.

이때 h를 휴리스틱 함수라고 부르며, 길 찾기의 경우 보통 유클리드 거리나 맨해튼 거리를 사용. h에 따라 최적해를 만들지 못하기도 한다. h는 실제 도착점까지의 거리보다는 짧거나 같아야 루트와 가까운 노드가 선택될 수 있다. 이러한 함수를 admissible heuristic function이라고 부름. 최초로 도달한 리프가 최적해임을 보장할 수 있음.

그렇다고 h를 0으로 하면 무조건 가까운 노드가 선택되므로 BFS와 같아짐. BFS는exhaustive함. 이것도 exhaustive해서 공간복잡도가 트레이드오프이긴 한데, pruning이 잘 돼서 빠르기 때문에 사용.

Genetic Algorithm

여러 해를 임의로 생성해 초기 세대로 놓고, 루프를 통해 현재 세대의 해로부터 다음 세대의 해를 생성한다. 루프가 끝났을 때 마지막 세대에서 가장 우수한 해를 반환한다.

1. 선택 연산: 여러 세대의 후보해 중에서 우수한 후보해를 선택할 때는 적합도의 확률에 따라 랜덤하게 선택한다. 이 방법을 roulette wheel이라고 한다. 적합도가 높은 후보해가 미래에도 좋을 것이라는 보장이 없기 때문에 랜덤 선택하는 것.
2. 교차 연산: 선택 연산을 통해 얻은 우수한 후보해보다 더 우수한 후보해를 생성하는 것. 교차 연산을 수행할 후보해의 수는 0.2에서 1.0 범위의 교차율(crossover rate)를 이용해 조절한다.
3. 돌연변이 연산: 작은 확률(mutation rate)로 후보해의 일부분을 변형. mutation rate는 보통 (1/population size) ~ (1/length) 범위. population size는 한 세대의 후보해 수, length는 후보해를 이진 표현했을 때 비트 수.

다음 세대가 현재 세대보다 우수하지 않다면 종료한다. 최적해를 보장하지 않지만, 기존의 알고리즘으로 해결하기 어려운 경우 최적해에 가까운 해를 찾을 때 적절하다.

Simulated Annealing

이와 같은 기법을 iterative method라고 함. 솔루션을 만들어 가는게 아니라 찾아가는 것에 가까움. 어떤 시작점에서 솔루션을 조금씩 발전시켜 나간다. 따라서 시작점에 여러 후보해를 만들어 둬야 한다. 이때 후보해를 임의로 만드는데, 그래서 randomized algorithm이라고도 함.

```
set random solution s
set initial T # temperature (e.g. T = 1000)
repeat:
  for i = 1 to k: # k is number of loop at T
    selete a solution s'
      in neighbors of s randomley
    d = s' - s
    if (d < 0): s = s' # if neighbor
                       #   s' is superior to s
```

```
    else:
      q = random number in (0, 1)
      if (q < p): s = s' # p is exploration
                         #   probability
  T = A * T # multiply constant A less than 1
            #   by T to calculate a new T
until
return s
```

9 Realworld Problems

9.1 설탕 배달

```
def g(a: int, b: int) -> int:
  if a == -1:
    return b
  elif b == -1:
    return a
  else:
    return min(a, b)

def f(n: int) -> int:
  d = [0] + [-1 for _ in range(n)]
  for i in range(1, len(d)):
    a, b = -1, -1
    if i >= 5 and d[i - 5] != -1:
      a = d[i - 5] + 1
    if i >= 3 and d[i - 3] != -1:
      b = d[i - 3] + 1
    d[i] = g(a, b)
  return d[n]
```

10 Appendix

- $\log_b a = c, \quad b^c = a$
- $\log_b 1 = 0$
- $\log_b b = 1$
- $\log^0 a = a$
- $\log^1 a = \log a$
- $O(n^c \log^k n) = O(1), \quad c = 0, \quad k = 0.$
- $O(n^c \log^k n) = O(n), \quad c = 1, \quad k = 0.$
- $O(n^c \log^k n) = O(n \log n), \quad c = 1, \quad k = 1.$