Lecture 21 : Backtracking (되추적) 알고리즘의 원리

알고리즘에서 답을 찾는 과정은 전체 solution space에서 특정 성질을 만족하는 하나의 경우(instance)를 찾는 것으로 해석될 수 있다. 예를 들어 n개 원소를 대상으로 정렬(sorting)하는 문제는 해당 원소의 모든 가능한 경우, 즉 n!(factorial)개의 가능성 중에서 하나를 선택하는 것이다. 물론 sorting을 이렇게 해결하는 경우는 없지만, 개념적으로 설명을 하자면 그렇다는 것이다. 이후에 배우겠지만 정렬의 경우 전체 원소의 수가 n개라면 최대 $n \log_2 n$번의 기본동작을 통해서 완성할 수 있으므로 전체 공간에 준비된 모든 n! 개의 경우를 모두 살펴볼 필요는 없다. 예를 들어 만일 $d_i < d_j$ $(i < j)$가 확인되면 n! 개의 결과 중에서 $d_i > d_j$인 모든 경우는 더 이상 고려될 필요가 없다. 이같이 좋은 알고리즘이란 한 번의 기본동작으로 탐색공간에서 더 이상 볼 필요가 없는 경우의 수를 최대한 많이 제거해야 한다. 즉 하나의 최적해를 위하여 빠르게 가능성이 없는 것들의 제거해나가는 것이 알고리즘의 효율을 높이는 지름길이다.

그런데 어떤 문제는 전체 search space의 모든 경우의 수를 **살펴보아야만** 답을 찾을 수 있다. 즉 최악의 경우 전체 search space를 모두 살펴보아야만 정답을 찾을 수 있는 문제가 존재하는데 이 경우라면 최악의 경우 모든 탐색공간의 조사를 피할 수 없다. 이러한 문제는 전 공간 탐색이 요구되는 문제이고, 이런 문제는 전역 탐색(exhaustive search)를 통해서만 최적의 답을 구할 수 있다. 이를 위하여 개발된 알고리즘 개발 방법론 중의 하나가 backtracking이다. 이후로 공부할 두 전역 탐색 알고리즘인 backtracking과 branch and bounding 방법론은 크게는 exhaustive searching의 한 부류이며 그것을 얼마나 적극적으로 활용하는 정도에 따라서 backtracking 알고리즘과 branch and bound 알고리즘으로 구분된다.

bactracking(BT)과 다음 장에서 배울 branch and bound(BB)은 일종의 쌍둥이 방법론이다. 두 방법을 복잡도 면에서 비교 본다면 본질적으로 다르지 않지만 이 둘은 서로 대응되는 구조를 가지고있다. backtracking은 탐색할 대상을 전 순서(total ordering) 기준으로 일렬로 정리하는 것이다. 즉 탐색 대상을 우리가 설정한 하나의 기준으로 쭉 정렬하여 (한 줄로) 이를 순차적으로 검색하는 것인데 비해서 BB는 그때 그때 상황을 봐가면서 탐색 대상을 결정하는 것이다. 예를 들어 설명해보자.

여러분이 어떤 큰 박람회, 예를 들어 전국 맛집 박람회에 갔다고 가정해보자. 전국 팔도에서 참가한 각종 맛난 음식들을 판매하는 포장마차가 큰 광장을 채우고 있으며 우리는 이들 중에서 가장 가성비가 좋은 하나의 음식을 찾으려고 한다. BT방식은 이렇게 진행된다. 각 판매가게의 이름을 사전식으로 정렬하여 그 순서대로 하나하나씩 각 가게를 탐방한다. 이 맛집 탐방을 일주일 동안 진행되므로 하루의 작업이 끝나면 숙소로 돌아가서 쉬고 그다음 날 다시 진행한다. 그런데 BT 식으로 진행하면 그날 최종적

으로 방문한 가게의 이름만 기록해두면 그 다음에 그 이후부터 진행할 수 있다. 만일 그 가게 이름이 "부산 갈매기"라고 하자. 그다음 날 다시 맛집 박람회로 갔을 때 "나주 곰탕"집을 발견했을 때 우리는 이 집을 다시 방문해야 할까? 그렇지 않다. 가게 이름을 사전식 순서로 보교해 본다면 "나주 곰탕" < "부산 갈매기"이므로 우리는 이 "나주 곰탕"은 이미 방문했다는 사실을 알게 되어 이미 방문한 이 "나주곰탕"은 skip 해야 한다. 이것이 바로 BT의 장점이다. 방문 순서가 무식하게 보이긴 하지만 방문 절차를 관리하는 입장에서 본다면 마지막 방문한 가게의 이름만 기억해두면 되기 때문에 매우 효율적으로 진행할 수 있다.

현실에서는 의외로 많은 문제들이 exhaustive searching을 요구한다. 예를 들어 충분한 시간과 computing resource를 충분히 사용해서라도 확실한 최적해를 구해야만 한다면 exhaustive searching은 피할 수 없다. 그런데 모든 경우를 다 살펴보는 것도 실제 코딩을 통하여 해보지 않으면 현실에서는 해결할 수 없다. 이번 장에서는 다양한 경우의 exhaustive searching 문제를 backtracking 코딩을 통하여 풀어보고 그때 주의해야 할 점, 조금이라도 성능을 올릴 수 있는 tip에 대하여 공부해본다.

21.1 정답을 위하여 살펴보아야 할 전체 Search Space의 특성에 따라서 알고리즘은 다음과 같이 분류된다.

- exhaustive searching : 가장 간단한 방법, 강력하면서도, 좀 무식한 방법
  답이 있으면 언젠가는 찾는다.

- heuristic searching : 답이 될 부분을 적절히 골라서 봄. 빠르지만
  정답을 찾지 못할 수도 있다.

Q) 바둑 둘 때 기계와 사람의 차이점 (탐색공간의 크기 관점으로)
Q) 직장을 구할 때 보통 어떤 방법으로 탐색을 할까요?

되추적 알고리즘의 핵심은 "탐색공간"을 순서화하여 탐색된 것과 아닌 것을 하나의 index로 기억하는 것이다. 따라서 수행 속도는 "느리지만", 관리는 "간편"하다. 매우라고 할 정도로...
그리고 이 알고리즘의 기본 자료구조는 스택(Stack)이다.
알고리즘이 진행된 결과는 스택에 순차적으로 기록되어 있다.
backtracking의 역사책은 "Stack(스택)"이다... Stack이라고

21.2 bubble sorting은 key의 배열 가능한 모든 $n!$가지의 경우를 고려하지 않는다. 즉 exhaustive searching이 아니다.
최댓값을 찾아내는 문제를 모든 원소가 최대인지를 하나씩 검토하면
전체 비교 횟수(기본동작)가 얼마나 되겠는가를 계산해 보자.

- 엄청난 $n!$ 개의 모든 경우를 모두 고려해서 결정하지 않는다.
- 문제에 따라서는 exhaustive searching을 피할 수 없는 예도 있다.

NP-complete 문제들

21.3 Backtracking은 전형적인 exhaustive searching 방법의 한 가지이다.
- 주의사항은 모든 경우를 빠짐없이, 중복 없이 볼 수 있다는 장점이다.
- 시작하여 될 수 있는 곳까지 진행한 다음 그 단계가 안 되면
  그때 가서 backtrack을 한다. (인생에서는 있을 수 없는 일이지만.)

21.4 Backtracking의 탐색순서는 항상 in-order 방문형식의 Tree로 표현된다.
- 시작은 항상 "왼쪽"부터 (오른쪽 왼쪽은 우리가 정하기 나름)
- Depth First Searching이므로 자료구조는 반드시 Stack이 필요하다.

21.5 n-Queen Problem. 서양 장기 Chess에서 Queen은 가로, 세로 대각선으로 공격을 할 수 있다. Chess판에 n개의 Queen을 배치하는데 서로 공격하지 않도록 배치하고자 한다. 즉 어떤 2개의 Queen도 수평, 수직, 대각선에 놓이지 않아야 한다. 이렇게 N개의 Queen을 배치하는 것을 N-Queen 문제를 푼다- 라고 표현한다. 이를 참고하여 6개의 Queen이 6 by 6 chess 판 위에 모두 안정된 위치에 있도록 배치하고 그 결과를 아래 칸에 표시하시오.

Q					
		Q			

			Q		
Q					

		Q			
	Q				

			Q		
Q					
					Q

따라서 N-Queen 문제를 풀면 N개의 row와 column에 단 하나씩의 Queen이 배치되어야 하고 2N-1개의 대각성분에는 두 개 이상의 Queen이 배치되지 않아야 한다. Queen을 배치하는 방법은 여러 가지가 있겠지만 일단 우리는 row 우선순위로 제일 윗 줄부터 하나씩 배치하는 과정으로 이 문제를 풀어 보자.

1부터 I번째 row까지 Queen이 배치되어 있다. 이제 $i+1$번째 row에 Queen을 배치해보자. 고려되는 순서는 가장 왼쪽 cell부터이다. 만일 왼쪽 k번째 cell에 Queen을 배치할 수 없다면 $k+1$번째 Queen을 배치하여 문제가 없는지 조사한다. 이렇게 진행하여 가장 오른쪽 cell까지 진행해도 답이 없다면 이 $i+1$번째 줄에 넣는 것은 포기하고 바로 그 전 단계로 back하여 그 i번째 row에 놓인 Queen의 위치를 한 칸 오른쪽으로 움직여본다.

21.6 수도쿠(Sudoku) 퀴즈 풀기 : 9 X 9 칸은 9개의 sub 3 X 3 서브 그리드(sub grid)로 구성되어 있다. 각 sub grid에 1부터 9까지의 숫자를 채워 넣고 각 행과 열은 모두 서로 다른 숫자가 되도록 해야 한다.

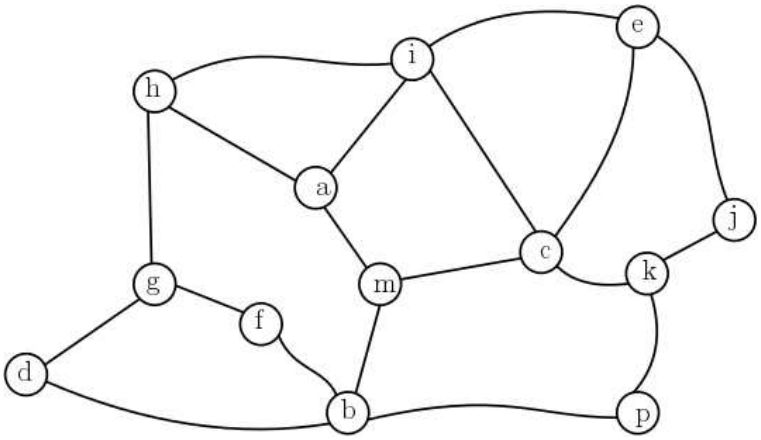
		6		5	4	9		
1				6			4	2
7				8	9			
	7				5		8	1
	5		3	4		6		
4		2						
	3	4				1		
9			8				5	
			4			3		7

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

이 문제는 BT 방식으로 해결해보시오.

21.7 Combinatorial Sum : array[ ] , K
array 에서 선택된 원소의 합이 K가 되는 multi-set. (반복도 허용된다.)
Input : arr[ ] = [2, 4, 6, 8], K = 8
Output : [2, 2, 2, 2], [2, 2, 4], [2, 6], [4, 4], [8]

21.8 Vertex Graph Coloring by Backtracking, 아주 어려운 NP-complete 문제
- 주어진 그래프가 m-coloring인지를 밝힌다. $k=1,2, \cdots n$
어떤 회사에서 신입사원 연수를 간다. 그래프에서 edge는 친한 관계이다. 친한 사람끼리는 같은 방에 넣지 않는다. 몇 개의 방이 필요한가 ?

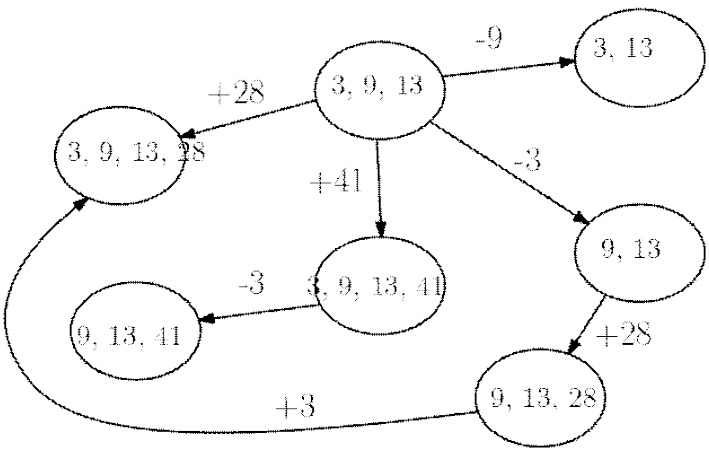


21.9 Sum of Subset Problem: S에서 몇 개의 원소를 고르되 그 골라낸 원소의 총합이 정확하게 C가 되도록 해야 한다. Instance는 집합 S와 C, 2개로 구성된다.

예 - **S={ 3, 7, 9, 13, 24, 28, 41 }, C = 51**

이 문제를 상태 공간 (State-Space Graph) 방법으로 접근해 보자.

$$S = [3, 7, 9, 13, 24, 28, 41], C = 51$$



상태 공간은 해답이 될 수 있는 각 상태를 그래프 노드로 표현하고 한 상태에서 다른 상태로의 변환(transition)이 가능하면, 그 둘 상태를 연결하는 edge를 추가한다. 변화하는 방법은 어떤 부분 집합에서 하나의 원소를 제거하거나 그 안에 없는 하나의 원소를 추가하는 방법으로 구성된다. 위 그림은 상태공간 그래프의 일부만을 표현한 것이다. 실제로는 더 많은 edge가 요구된다.

Q) 만일 N개의 원소로 구성된 문제의 상태는 몇 개가 가능한가 ?

S) 모든 가능한 부분 집합의 개수이므로 그 답은 $2^{|N|}$이다.

21.10 0/1 방식으로 배낭 채우기 문제 (0/1 방식은 해당되는 원소를 완전히 넣거나(1), 또는 넣지 않는 경우(0)만 가능한 상황이다.

- 단순한 기계적인 Backtracking에서 약간의 기술을 더 해보자.
- 핵심은 **답의 "가능성"이 없을 경우에는 더 이상 탐색을 하지 않는다.**
  Solution space Pruning(잘라내기)

일단 물품을 용량의 크기로 정렬한다.
T_i = { 3, 5, 11, 15, 30, 36, 42, 70 }

a) T_i $(i \geq k)$ 물품을 넣을 경우 배낭이 넘치는 경우
b) 그 이하의 용량을 잘라서 넣는 것을 허용할 경우에,
   보장되는 이득이 현재 확보된 이득보다 작을 경우
   (Partial Knapsack으로 bound 값을 설정할 수 있다.)

21.11 지능적 Backtracking의 원리 = Branch and Bound 방식과 동일

if 현재 확보된 이득 > X 방향으로 갈 때 얻을 이득의 최대치
:
then : 우리는 이 X 방향으로 진행하는 일을 일단 포기한다.

Q) 현재 연봉이 3500이다. 이직할 업체가 다음과 같다. 여러분은 어떤 순서로 고려해 볼 것인지 자신의 기준을 말하시오.

업체	A	B	C	D	E	F
최소연봉	3000	4000	2800	1700	1000	3800
최대연봉	5000	4300	3100	6500	8000	5000

21.12 전형적인 문제는 0/1 Knapsack이다. 이 문제를 Branch and Bound를 이용하면 무식한 backtracking보다는 조금 더 나은 시간에 풀 수 있다. 문제는 어떻게 Bound 할 것인가이다. 즉 non-promising node(가망이 없는 노드)를 어떻게 이른 시간에 제거할 것인가이다. 즉 현재까지 확보한 이득(배낭에 들어있는 이득)과 배낭의 남은 공간에 채워 넣을 수 있는 이득의 예상치를 예상하는 것이다. 이것은 넣을 물건을 잘라서 Partial Knapsack으로 처리했을 때 확보되는 이득의 한계치를 사용하면 된다. 왜냐하면 아래와 같기 때문이다.

Partial Knapsack의 성능 > 0/1 Knapsack의 성능

21.13 모든 가능한 Permutation 출력하기. 먼저 떠오르는 "무식한" 방법은 여러 개의 for loop을 사용하는 것이다. 아래는 3개의 문자로 구성되는 모든 permutation을 출력하는 코드이다.[1)]

```
N=5
for i in range(N+1):
    for j in range(i, N+1):
        for k in range(j, N+1):
            print(i, j, k )
```

물론 python의 itertools package를 활용하면 아래와 같이 가능하다.

```
import itertools
L=[1,2,3,4,5]
print("\n itertools.permutations(L,3)")
for x in itertools.permutations( L, 3):
    print(x)
```

만일 5자리의 숫자로 만들어지는 모든 가능한 permutation을 출력한다면 ?
>> 이러한 작업에 필수적인 자료구조는 Stack이다. Stack Stack !!!

{1, 2, 3, 4, 5}로 구성되는 5!개의 모든 조합을 stack으로 구성해보자.

5	**4**	**5**	**3**	**4**	**3**	**5**					**1**
4	**5**	**3**	**5**	**3**	**4**	**4**					**2**
3	**3**	**4**	**4**	**5**	**5**	**2**	**.**	**.**	**.**		**3**
2	**2**	**2**	**2**	**2**	**2**	**3**					**4**
1	**1**	**1**	**1**	**1**	**1**	**1**					**5**
t=1	t=2	t=3	t=4	t=5	t=6		.	.	.		t=5!

21.14 6 개의 모든 permutation을 stack으로 생성하려고 한다. 다음 stack의 구조를 보고 그 다음 순서로 스택에 채워질 숫자를 모두 채워 넣으시오.

1) 물론 python에서 import itertools을 사용하면 대부분의 조합론 계산이 가능하다.

3					
5					
1					
4					
2					
6					

1					
5					
3					
4					
6					
2					

1					
5					
3					
4					
6					
2					

1					
2					
3					
6					
5					
4					

Lecture 22 : 되추적(Backtracking) 알고리즘의 실전 응용

22.1 Branch and Bound 방법과 Backtracking 의 비교 분석

- Backtrack : 기계적
- Branch and Bound : 지능적
- 속도 ? 머리를 쓰는 것이 항상 좋은 것은 아니다.
  "머리를 쓰는 비용까지를 생각해야 한다."

예) 회식비 총액이 116,500원인 경우 4명이 더치페이를 하는 방법

- Branch and Bound를 쓸 만큼 문제가 복잡하고 이득이 있어야 한다.
- Binary Search가 Linear Search보다 항상 더 빠르지 않다.

각 알고리즘에 따른 자료구조 유형 (아주 중요함)

적용할 알고리즘	주된 자료구조
Divide and Conquer	Stack
Dynamic Programming	k-dimensional Table (array)
Backtracking	Stack
Branch and Bound	Queue and Priority Queue

22.2 지능적이라고 생각되는 방법이 "항상" 더 나은 결과를 보장하지는 않는다.

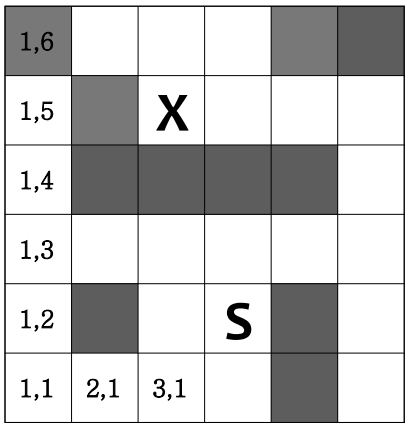
- Branch and Bound 역시 exhaustive searching의 일종이다.
- 각 방법을 비교해 봅시다.

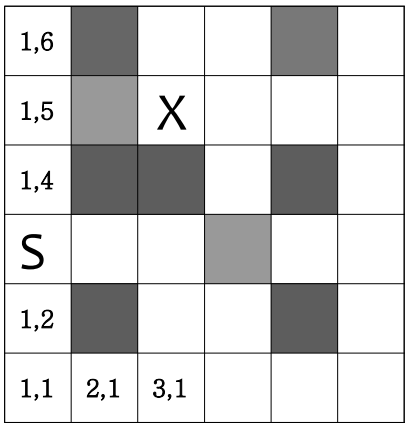
비교항목	Heuristics	Backtracking	Branch and Bound	Optimal algorithm
답이 있으면 반드시 찾는가?				
코딩 작업은 간단한가 ?				
수행속도가 빠른가?				
최악의 경우에는 ?				

22.3 Backtracking에서 조금 더 “머리”를 쓰면 Branch and Bound 방법이 된다.

- 먼저 가능한 발전 방향이 있는 경우에 Branch를 한다.
- 현재 Branch 된 “살아있는” 노드 중에 가장 “똘똘한 놈”을 지원한다.
- “자손”들 중에서 가장 똘똘한 놈을 찾아서 키우는 작업을 계속한다.

예) <맨해튼>에서 길 찾아가기 (S에서 T까지 가는 길을 “빨리” 찾으시오.

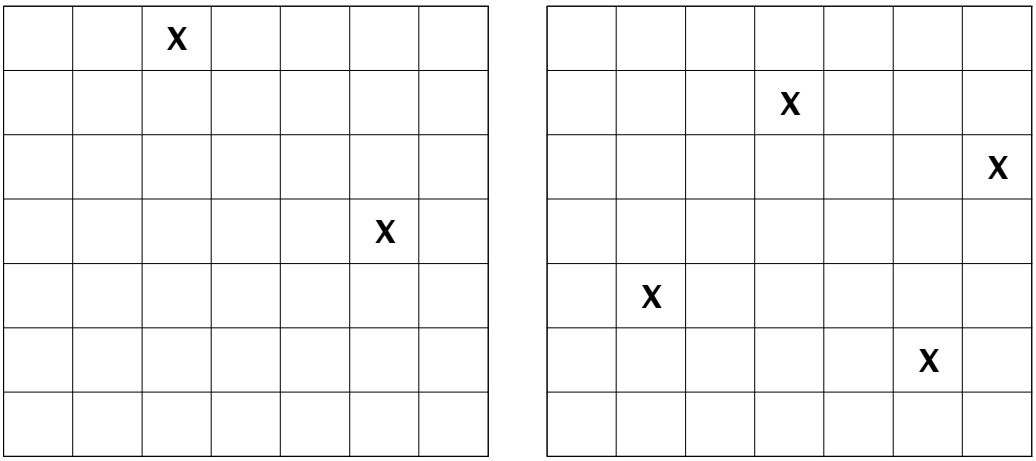




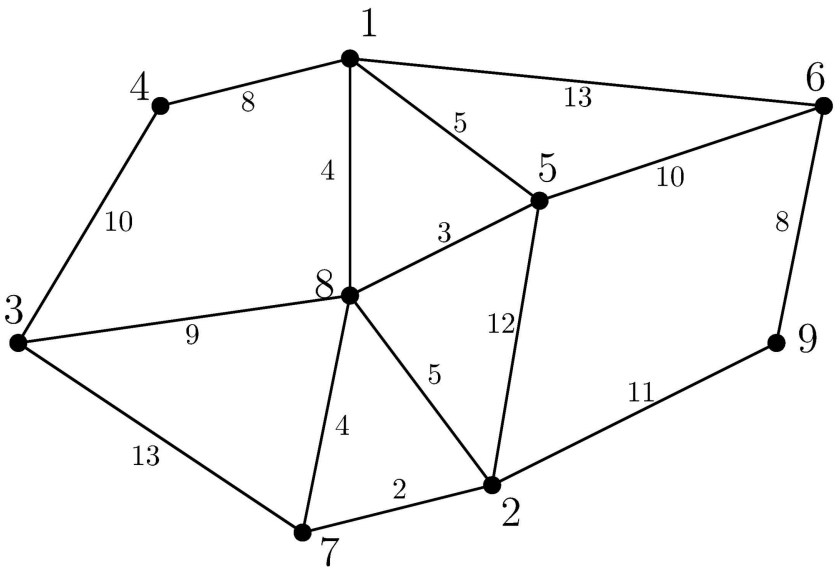
22.4 다이어트 식사를 위한 재료 장보기: 음식 재료를 선택하여 (단백질, 지방, 탄수화물, 비타민)=(100, 100, 100, 10)이상이 되도록 하는 방법. 단 필요한 영양을 확보할 수 있는 재료의 비용이 최소가 되어야 한다. 일단 Greedy 방식으로 먼저 풀어보자. 기준은 가격이 싼 재료부터 하나씩 추가하여 조건을 만족할 때까지 진행한다.

재료	단백질	지방	탄수화물	비타민	가격/W
1	30	55	10	8	100
2	60	10	10	2	70
3	10	80	50	0	50
4	40	30	30	8	60
5	60	10	70	2	120
6	20	70	50	4	40

22.5 아까는 무식하게 Backtracking으로 푼 n-Queen Problem의 좀 멋있게 풀어 봅시다. i번째 줄에서 다음 i+1 번째 줄로 내려갈 때, 정해진 순서(예를 들면 왼쪽부터 오른쪽)대로 하지 않고 가장 “가능성”이 있는 길부터 먼저 탐색해보자. 다음은 n-Queen Problem인데 각 검은색 cell은 장애물을 나타낸다. 이것은 Queen의 공격을 막아준다. 이 문제를 Branch and Bound로 풀어 보자.



22.6 다음 지점을 모두 방문하여 물건을 배달하고 돌아오는 최소경로 구하기
(단 출발은 8번 vertex에서 시작한다.)



22.7 Branch and Bound는 인공지능에서 말하는 A* 알고리즘의 원형(Prototype)이 된다. 즉 max{ (현재까지 확보한 이득) + (예상되는 이득) }이 되는 노드부터 더 깊게 검사를 해본다. 비유하자면 여러분은 졸업 후 가질 수 있는 직업 {프로그램 디자이너, 가수, 격투기 선수, 변호사, 국회의원, 일용노동자.. } 중에서 직업을 탐색해나가는 과정과 유사하다.

이 방법은 성능은 현재 promising 한 노드 중에서 가장 "좋은" 노드를 빠르게 선택하고 가망이 없는 노드 (non-promising)를 빨리 제거하는 것이다. 이를 위한 자료구조가 support 해야 할 동작은 다음이다.

- `get_max_priority( Q )`
- `insert-queue( N )`
- `delete-queue( X )`

22.8. Backtracking 순서는 하나의 Stack에 기록 가능하다. (코딩의 단순함.)
n=6인 리스트 [1, 2, 3, 4, 5, 6]의 6! 개의 permutation을 구하시오.
Q) Stack을 사용해서 모든 permutation을 생성할 수 있다. 아래 표에서 다음 단계의 Stack의 내용을 채우시오.

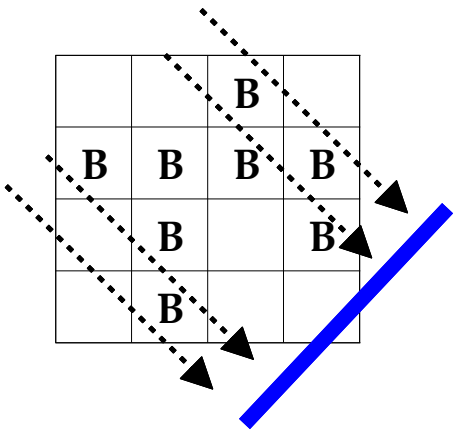
1. 시작은 [1,2,3, n]이며
2. 끝은 [n, n-1, ...3, 2, 1]
3. max last permutation를 찾는다.
   last permutation은 top에서
   $a_{top} < a_{top-1} < a_{top-2} \cdots < a_{top-i} > a_{top-(i+1)}$ 인 i를 찾는다.
   따라서 top부터 i까지의 변화는 모두 끝이 났으므로, 이들 중에서
   가장 작은 것과 $a_{top-(i+1)}$과 교체한 뒤에

4. 나머지 원소를 정렬해서 다시 stack에 넣는다.

6	5	6			5	I	2	I	6			
5	6	4			I	5	I	2	4			
4	4	5			2	2	5	5	2			
3	3	3			6	6	6	6	I			
2	2	2			4	4	4	4	5			
I	I	I			3	3	3	3	3	?		

22.9 되추적을 이용한 단층사진 재구성하기[2)]

[문제] $N \times N$ 그리드에 하나의 물체가 있다. 이 물체에 상하좌우 45도씩 4번의 엑스선을 주사하여 그 단면을 필름에 담는다. 우리는 4장의 필름에 담긴 정보를 이용해서 이 물체의 원래 모습을 재구성하고자 한다. 물체는 방사선을 통과하는 부분과 잘 통과하지 못하는 부분(예를 들면 뼈)이 있다. 아래 예를 보자. 먼저 위에서 밑으로 크기 10인 방사선을 주사하면 B 하나에 대하여 그 강도가 1씩 감소되어 바닥에 감지되는 그 값은 [9,7,8,8]이 된다. 즉 원래 주사한 광선의 초기값을 빼면 불투과(opaque) 성분 셀 개수가 나온다. 우리는 이 개수가 바로 필름에 기록된다고 생각하자. 아래는 ↓, →, ↘, ↙ 방향으로 주사했을 때 얻는 4개의 vector 값은 각각 `[1,3,2,2]`, `[1,4,2,1]`, `[0,2,2,1,2,1,0]`, `[0,1,2,2,2,1,0]`이 된다.



위 그림에서 화살표는 광선 방향이며 파란색은 필름을 나타낸다. 여러분은 4개 감광판 결과를 보고 원래의 내부 모양을 정확하게 재구성해야 한다. 이것은 backtracking으로 풀면 좋다. 단 물체 B는 반드시 상하좌우 4방향으로 연결된 하나의 연결 덩어리이라는 가정을 활용하여 많은 상태를 잘라낼 수 있기 때문에 속도를 올릴 수 있다.

[입출력] 입력화일 ct.inp 의 첫 줄에는 차원 N ($3 \le N \le 8$)이 주어진다. 그 다음에는 ↓,

2) Computerized Tomography: 컴퓨터 단층 촬영방법: 필요한 단면을 횡단하는 방사선의 흡수에 관한 정보 또는 방사능 분포에 관한 정보를 기억, 축적하여 컴퓨터에 의해 재편성하여 단면상을 얻는 방법.

→, ↘, ↙ 방향의 필름 결과가 4개의 줄에 순서대로 들어온다. 여러분은 이것을 바탕으로 투과지역(-. minus)과 불투과 셀(B)인 $N \times N$ 행렬로 재구성해야 한다. 각 기호 사이에 하나의 공백이 있다.

ct.inp	ct.out
4 1 3 2 2 // ↓ 1 4 2 1 // → 0 2 2 1 2 1 0 // ↘ 0 1 2 2 2 1 0 // ↙	- - B B B B B B - B - B - B - -

이 문제를 풀기 위한 기본 함수는 한 줄에 k개의 'B'가 있을 때 이들을 순서화하는 것이다. $k=3$ 이고 $N=6$ 일 때 살펴보자. 이것은 6자리 Binary number 중에서 '1'이 3개인 모든 숫자를 크기순으로 정렬하는 것과 동일하다.

index	1	2	3	4	5	6	
1	B	B	B				
2	B	B		B			
3	B	B			B		
4	B	B				B	reset
5	B		B	B			
6	B		B		B		
7	B		B			B	reset
8	B			B	B		
9	B			B		B	
10	B				B	B	
11		B	B	B			reset
12		B	B		B		
13		B	B			B	
14		B		B	B		
15		B		B		B	
16		B			B	B	
17			B	B	B		
18			B	B		B	
19			B		B	B	
20				B	B	B	
21							

$k=4$ 이고 $N=7$ 일 때 같은 과정을 반복해 보자.

index	1	2	3	4	5	6	7	
1	B	B	B	B				
2	B	B	B		B			
3	B	B	B			B		
4	B	B	B				B	reset
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								

22.10 되추적(backtracking) 알고리즘의 성능 개선하기

간결한 것은 그 자체로 미덕이다. (괴테, 독일을 철학자)

되추적 방법을 사용했을 때의 worst case는 모든 탐색공간을 살펴보는 것이지만 평균적으로는 그보다 작은 공간을 살펴보게 된다. 실제 worst case에 해당되는 경우는 흔하지 않다는 말이다.

문제는 탐색공간을 얼마나 줄이는가- 이다. 단 탐색공간을 줄이기 위해서는 이 줄이는 방법을 계산해야 하는데 이 역시 비용이 든다. 예를 들어 시간을 아껴 쓰기 위해서는 심사숙고하여 시간을 줄일 수 있도록 생활을 정리하고 계획을 세워야 한다.

예를 들어 Planner를 잘 활용하고 일기를 쓰면 시간을 더 활용할 수 있지만, 이 부가적인 작업(extra work)을 하는 시간까지도 전체 성능 평가에 고려해야 한다. 2시간 고민해서 1시간을 절약할 수 있다면 이것은 도리어 손해가 된다. 이것이 backtracking과 branch and bound의 경계를 나누는 중요한 관점이다. 즉 단순한 backtracking과 복잡한 A.I. 기법 중에서 어떤 것이 더 우수한지는 문제와 상황에 따라서 달라진다. 때로는 전체 space를 모두 고려하는 단순 searching 알고리즘의 성능이 더 빠를 수 있다.

Backtracking 이해를 위한 실습지 (Working sheet)

(1) 5-Queen 문제 풀어보기 (진행과정은 모두 stack에 표시되어야 함)

2,1																
1,1																
1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***

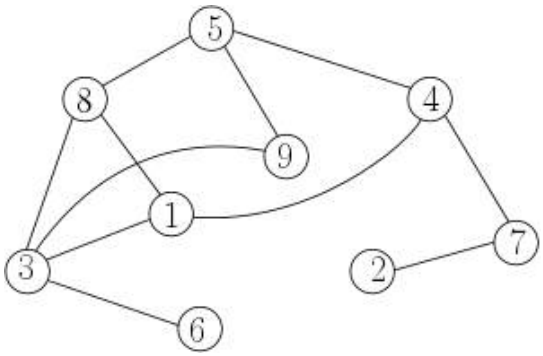
(2) Sum of Subset S={ *3, 6, 15, 23, 35* }, K=50

1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***

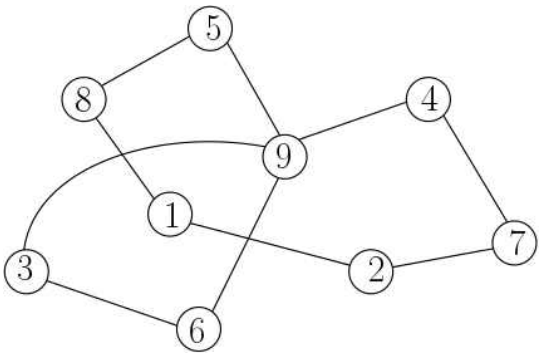
(3) 방정식 $23x+12y+9z+4t=100$ 을 만족시키는 정수해 (x,y,z,t) 구하기

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*

(4) 그래프 색칠하기 (주어진 그래프를 k개의 색으로 칠하기)



$G_1(V,E)$



$G_2(V,E)$

Find a 3-Coloring / 4-Coloring for graph above.

Vertex	1	2	3	4	5	6	7	8
Col.								

stack													
9													
8													
7													
6													
5													
4													
3													
2													
1													
단계	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***