

정렬 알고리즘

- 기본적인 정렬 알고리즘:
 - 버블 정렬
 - 선택 정렬
 - 삽입 정렬
- 효율적인 정렬 알고리즘:
 - 쉘 정렬
 - 힙 정렬
 - 합병 정렬
 - 퀵 정렬
- 이외에도 입력이 제한된 크기 이내에 숫자로 구성되어 있을 때에 효율적인 **기수 정렬**이 있다.

- 정렬 알고리즘
 - 내부정렬 (Internal sort)
 - 외부정렬 (External sort)
- **내부정렬**은 입력의 크기가 주기억 장치 (main memory)의 공간보다 크지 않은 경우에 수행되는 정렬이다. 예: 앞 슬라이드의 모든 정렬 알고리즘들
- 입력의 크기가 주기억 장치 공간보다 큰 경우에는, 보조 기억 장치에 있는 입력을 여러 번에 나누어 주기억 장치에 읽어 들인 후, 정렬하여 보조 기억 장치에 다시 저장하는 과정을 반복해야 한다. 이러한 정렬을 **외부정렬**이라고 한다.

- 다음의 입력에 대해 버블 정렬이 수행되는 과정을 보이시오.
 - 90 30 50 20 40 10 80 60 70
- pass 1: 30 50 20 40 10 80 60 70 90
- pass 2: 30 20 40 10 50 60 70 80 90
- pass 3: 20 30 10 40 50 60 70 80 90
- pass 4: 20 10 30 40 50 60 70 80 90
- pass 5-8: 10 20 30 40 50 60 70 80 90

- 다음의 입력에 대해 선택 정렬이 수행되는 과정을 보이시오.

– 90 30 50 20 40 10 80 60 70

- [10] 30 50 20 40 90 80 60 70
- [10 20] 50 30 40 90 80 60 70
- [10 20 30] 50 40 90 80 60 70
- [10 20 30 40] 50 90 80 60 70
- [10 20 30 40 50] 90 80 60 70
- [10 20 30 40 50 60] 80 90 70
- [10 20 30 40 50 60 70] 90 80
- [10 20 30 40 50 60 70 80] 90

셸 정렬

- 버블 정렬이나 삽입 정렬이 수행되는 과정을 살펴보면, 이웃하는 원소끼리의 자리이동으로 원소들이 정렬된다.
- 버블 정렬이 오름차순으로 정렬하는 과정을 살펴보면, 작은 (가벼운) 숫자가 배열의 앞부분으로 매우 느리게 이동하는 것을 알 수 있다.
- 예를 들어, 삽입 정렬의 경우 만일 배열의 마지막 원소가 입력에서 가장 작은 숫자라면, 그 숫자가 배열의 맨 앞으로 이동해야 하므로, 모든 다른 숫자들이 1칸씩 오른쪽으로 이동해야 한다.

- 셸 정렬 (Shell sort)은 이러한 단점을 보완하기 위해서 삽입 정렬을 이용하여 배열 뒷부분의 작은 숫자를 앞부분으로 '빠르게' 이동시키고, 동시에 앞부분의 큰 숫자는 뒷부분으로 이동시키고, 가장 마지막에는 삽입 정렬을 수행하는 알고리즘이다.

- 다음의 예제를 통해 쉘 정렬의 아이디어를 이해해보자.

30 60 90 10 40 80 40 20 10 60 50 30 40 90 80

- 먼저 간격 (gap)이 5가 되는 숫자끼리 그룹을 만든다.
- 총 15개의 숫자가 있으므로, 첫 번째 그룹은 첫 숫자인 30, 첫 숫자에서 간격이 5되는 숫자인 80, 그리고 80에서 간격이 5인 50으로 구성된다.

- 즉, 첫 번째 그룹은 [30, 80, 50]이다. 2 번째 그룹은 [60, 40, 30]이고, 나머지 그룹은 각각 [90, 20, 40], [10, 10, 90], [40, 60, 80]이다.

h=5

A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	30					80					50				
2		60					40					30			
3			90					20					40		
4				10					10					90	
5					40					60					80

- 각 그룹 별로 삽입 정렬을 수행한 결과를 1줄에 나열해보면 다음과 같다.

30 30 20 10 40 50 40 40 10 60 80 60 90 90 80


A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	30					50					80				
2		30					40					60			
3			20					40					90		
4				10					10					90	
5					40					60					80

그룹별 정렬 후

- 간격이 5인 그룹 별로 정렬한 결과를 살펴보면, 80과 90같은 큰 숫자가 뒷부분으로 이동하였고, 20과 30같은 작은 숫자가 앞부분으로 이동한 것을 관찰할 수 있다.

– 30 30 20 10 40 50 40 40 10 60 80 60 90 90 80

- 그 다음엔 간격을 5보다 작게 하여, 예를 들어, 3으로 하여, 3개의 그룹으로 나누어 각 그룹별로 삽입 정렬을 수행한다.
- 이때에는 각 그룹에 5개의 숫자가 있다. 마지막에는 반드시 간격을 1로 놓고 수행해야 한다.
- 왜냐하면 다른 그룹에 속해 서로 비교되지 않은 숫자가 있을 수 있기 때문이다.
- 즉, 모든 원소를 1개의 그룹으로 여기는 것이고, 이는 삽입 정렬 그 자체이다.

그룹1	그룹2	그룹3		그룹1	그룹2	그룹3
30	30	20		10	30	10
10	40	50		30	40	20
40	40	10		40	40	50
60	80	60		60	80	60
90	90	80		90	90	80

- 각 그룹 별로 정렬한 결과를 한 줄에 나열해보면 다음과 같다. 즉, 이것이 $h=3$ 일 때의 결과이다.

10 30 10 30 40 20 40 40 50 60 80 60 90 90 80

- 마지막으로 위의 배열에 대해 $h=1$ 일 때 알고리즘을 수행하면 아래와 같이 정렬된 결과를 얻는다.
- $h=1$ 일 때는 간격이 1 (즉, 그룹이 1개)이므로, 삽입 정렬과 동일하다.

10 10 20 30 30 40 40 40 50 60 60 80 80 90 90

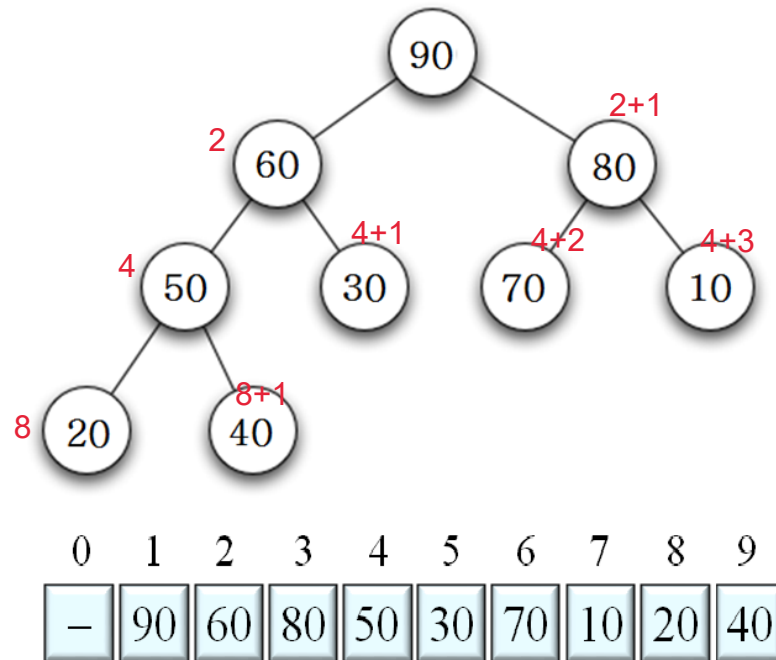
- 쉘 정렬의 수행 속도는 **간격 선정에 따라 좌우**된다.
- 지금까지 알려진 **가장 좋은 성능**을 보인 간격:
 - 1, 4, 10, 23, 57, 132, 301, 701
 - 701 이후는 아직 밝혀지지 않았다.

힙 정렬

이진트리 왼쪽 부터 있어야 한다

- 힙 (heap)은 힙 조건을 만족하는 완전 이진트리 (Complete Binary Tree)이다.
- 힙 조건이란 각 노드의 값이 자식 노드의 값보다 커야 한다는 것을 말한다.
- 노드의 값은 우선순위 (priority)라고 일컫는다.
- 힙의 루트에는 가장 높은 우선순위 (가장 큰 값)가 저장되어 있다.
- 값이 작을수록 우선순위가 높은 경우에는 가장 작은 값이 루트에 저장된다.

- 아래의 그림은 힙의 노드들이 배열에 저장된 모습을 보여주고 있다.

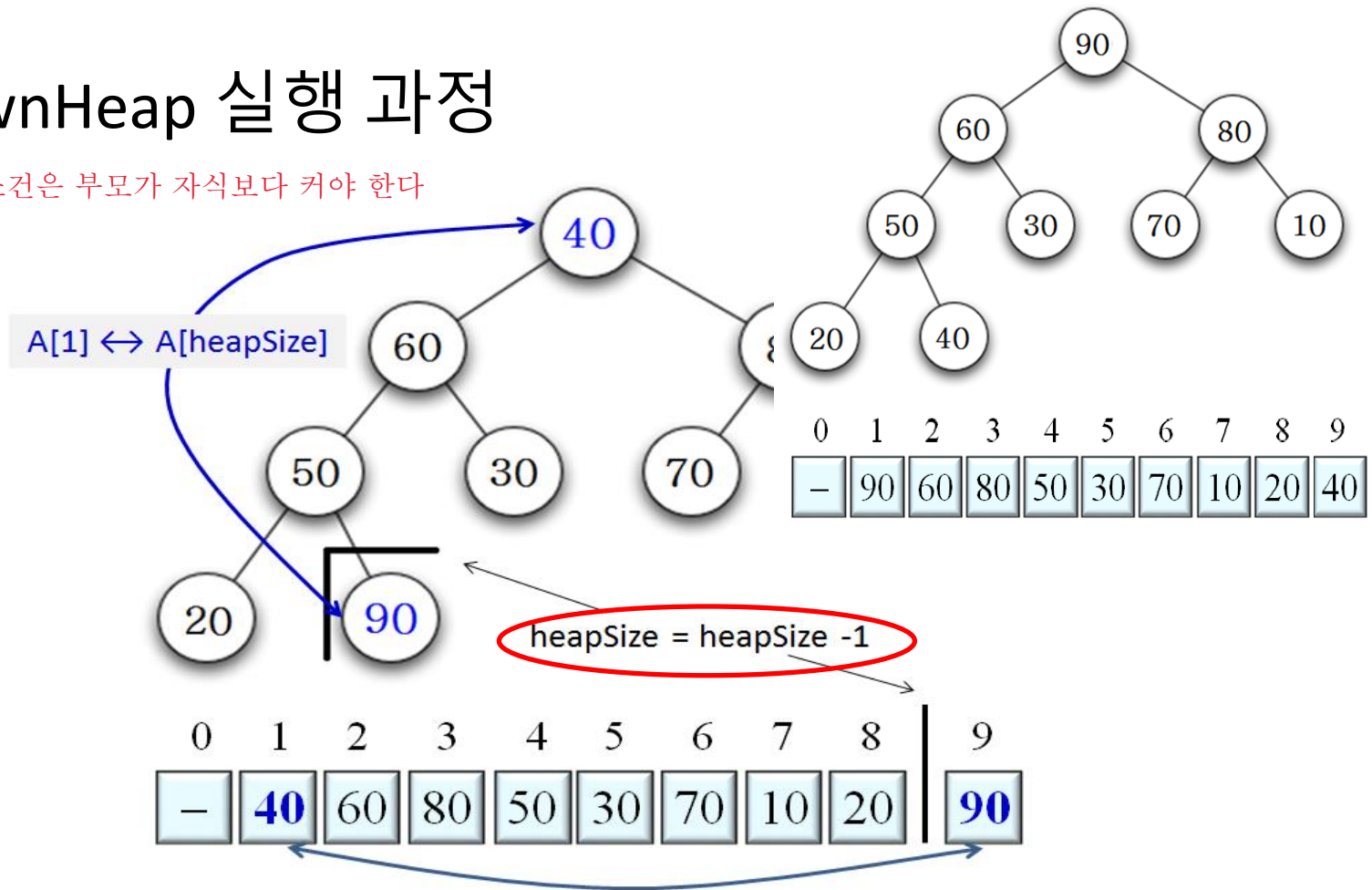


- 배열 A에 힙을 저장한다면, A[0]은 비워 두고, A[1]~A[n]까지에 힙 노드들을 트리의 층별로 좌우로 저장한다.
 - 루트의 90이 A[1]에 저장되고,
 - 그 다음 층의 60과 80이 각각 A[2]와 A[3]에 저장되며,
 - 그 다음 층의 50, 30, 70, 10이 A[4]에서 A[7]에 각각 저장되고,
 - 마지막으로 20과 40이 A[8]과 A[9]에 저장된다.

- 힙 정렬(Heap Sort)은 힙 자료 구조를 이용하는 정렬 알고리즘이다.
- 오름차순의 정렬을 위해 최대힙(maximum heap) 구성한다.
- 힙의 루트에는 가장 큰 수가 저장되므로, 루트의 숫자를 힙의 가장 마지막 노드에 있는 숫자와 바꾼다.
- 즉, 가장 큰 수를 배열의 가장 끝으로 이동시킨 것이다.
- 그리고 루트에 새로 저장된 숫자로 인해 위배된 힙 조건을 해결하여 힙 조건을 만족시키고, 힙 크기를 1개 줄인다.
- 그리고 이 과정을 반복하여 나머지 숫자를 정렬한다.
다음은 이러한 과정에 따른 힙 정렬 알고리즘이다.

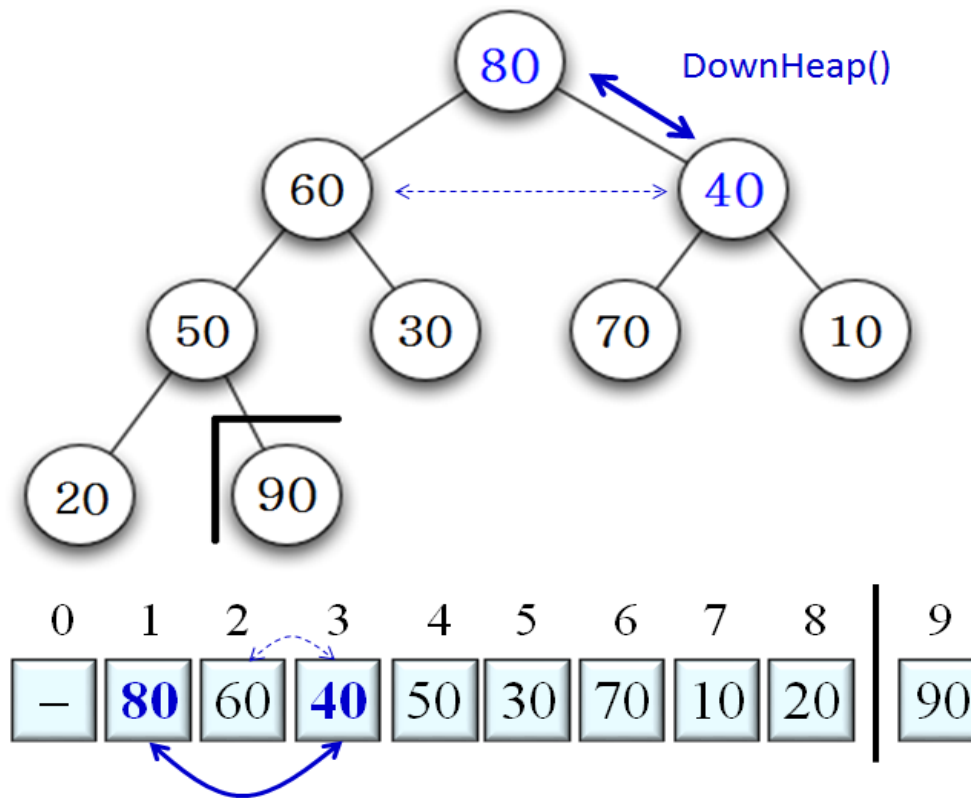
• DownHeap 실행 과정

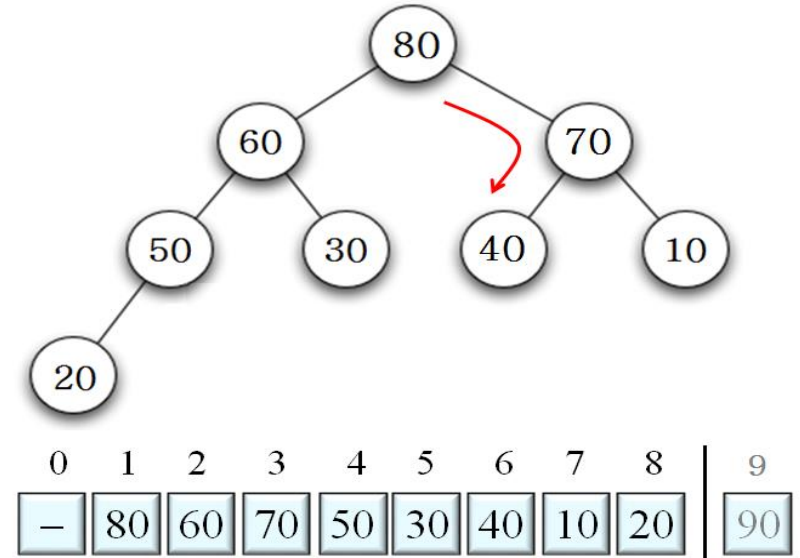
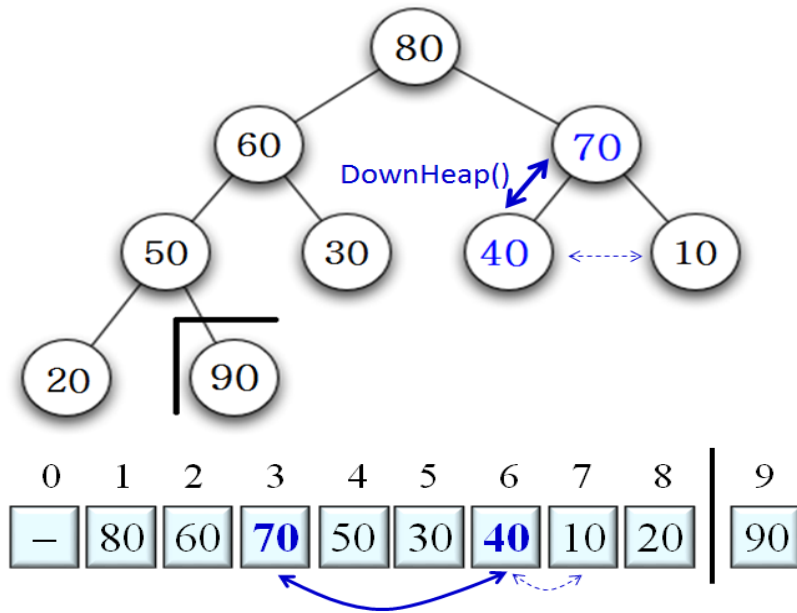
힙 조건은 부모가 자식보다 커야 한다



- Line 4에서 힙의 마지막 노드 40과 루트 90을 바꾸고, 힙의 노드 수(heapsize)가 1개 줄어든 것을 보이고 있다.

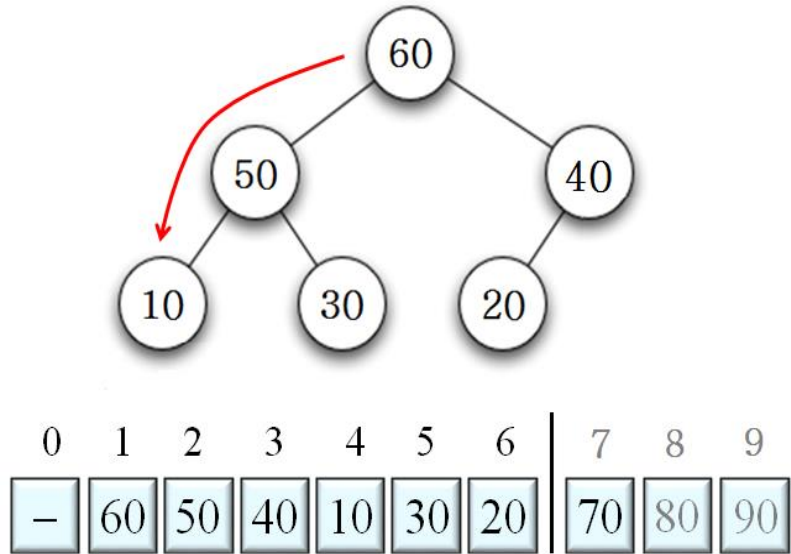
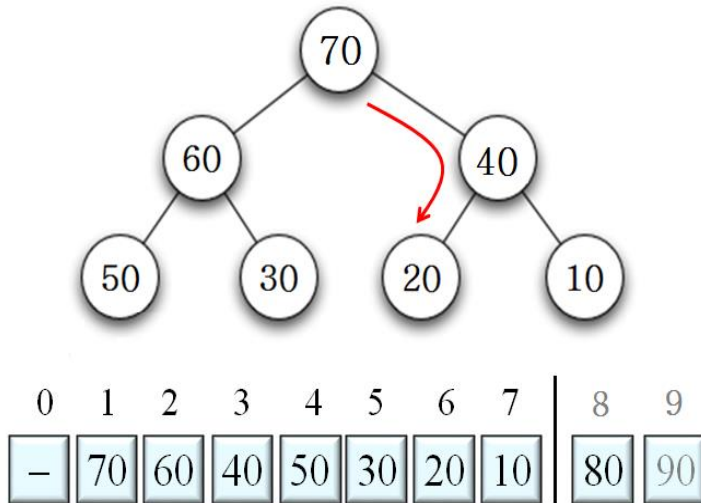
- 다음 그림은 새로이 루트에 저장된 40이 루트의 자식 노드들 (60과 80)보다 작아서 힙 조건이 위배되므로 자식 노드들 중에서 큰 자식 노드 80과 루트 40이 교환된 것을 보여준다.



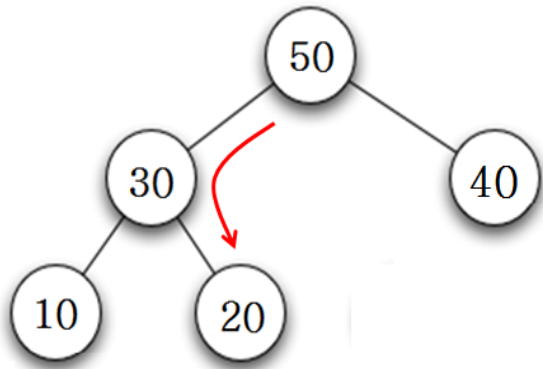


- 40은 다시 자식 노드들 (70과 10) 중에서 큰 자식 70과 비교하여, 힙 조건이 위배되므로 70과 40을 서로 바꾼다.
- 그 다음엔 더 이상 자식 노드가 없으므로 힙 조건이 만족되므로 `DownHeap`을 종료한다.

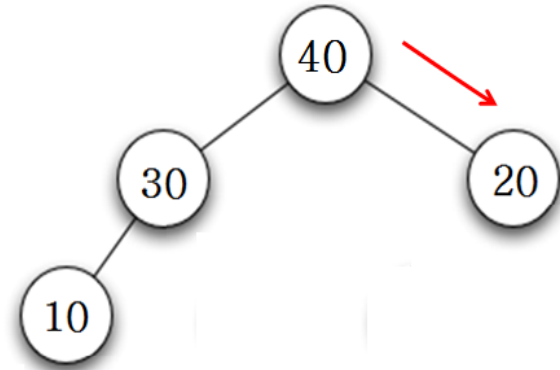
- DownHeap 예제 이은 HeapSort 수행



- 80이 20과 교환된 후 DownHeap을 수행한 결과 (왼쪽)
- 70이 10과 교환된 후 DownHeap을 수행한 결과 (오른쪽)

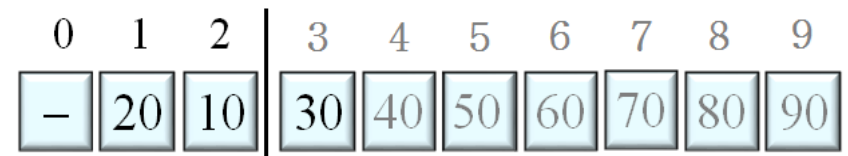
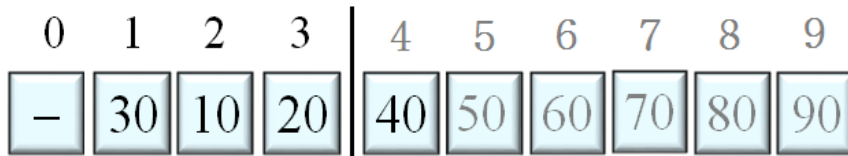
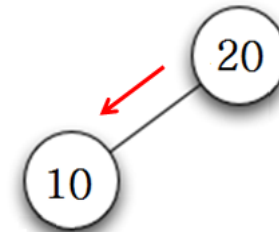
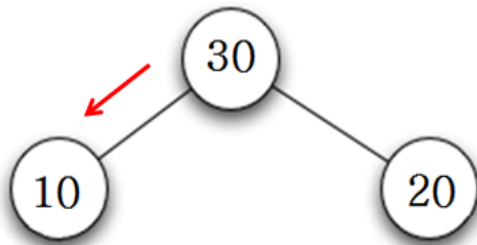


0	1	2	3	4	5	6	7	8	9
-	50	30	40	10	20	60	70	80	90



0	1	2	3	4	5	6	7	8	9
-	40	30	20	10	50	60	70	80	90

- 60이 20과 교환된 후 DownHeap을 수행한 결과 (왼쪽)
- 50이 20과 교환된 후 DownHeap을 수행한 결과 (오른쪽)



- 40이 10과 교환된 후 DownHeap을 수행한 결과 (왼쪽)
- 30이 10과 교환된 후 DownHeap을 수행한 결과 (오른쪽)

10

0	1	2	3	4	5	6	7	8	9
-	10	20	30	40	50	60	70	80	90

- 마지막에 힙의 크기가 10이 되면 힙 정렬을 마친다.

기수 정렬LSD(Least significant Digit)

- 기수 정렬 (Radix Sort)이란 비교정렬이 아니고, 숫자를 부분적으로 비교하는 정렬 방법이다.
- 기 (radix)는 특정 진수를 나타내는 숫자들이다.
 - 예를 들어, 10진수의 기는 0, 1, 2, ..., 9이고, 2진수의 기는 0, 1이다.
- 기수 정렬은 제한적인 범위 내에 있는 숫자에 대해서 각 자릿수 별로 정렬하는 알고리즘이다.
- 기수 정렬의 가장 큰 장점은 어느 비교정렬 알고리즘보다 빠르다.

입력	1의 자리	10의 자리	100의 자리
----	-------	--------	---------

089	07 0	9 1 0	0 35
-----	-------------	--------------	-------------

070	91 0	1 3 1	0 70
-----	-------------	--------------	-------------

035	⇒ 13 1	⇒ 0 3 5	⇒ 0 89
-----	---------------	----------------	---------------

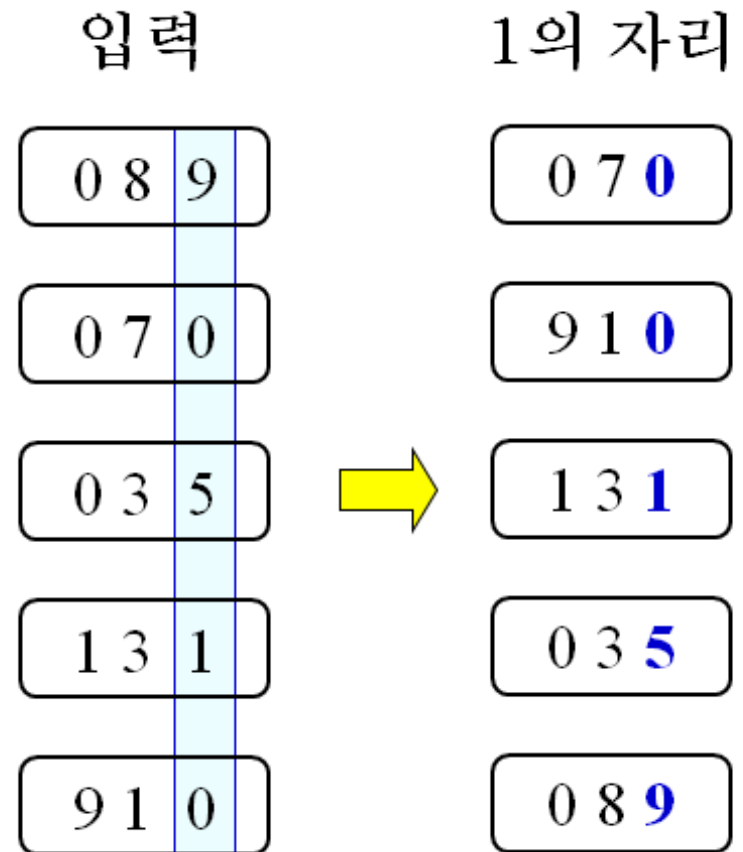
131	03 5	07 0	1 31
-----	-------------	-------------	-------------

910	08 9	0 8 9	9 10
-----	-------------	--------------	-------------

여기에 텍스트 입력

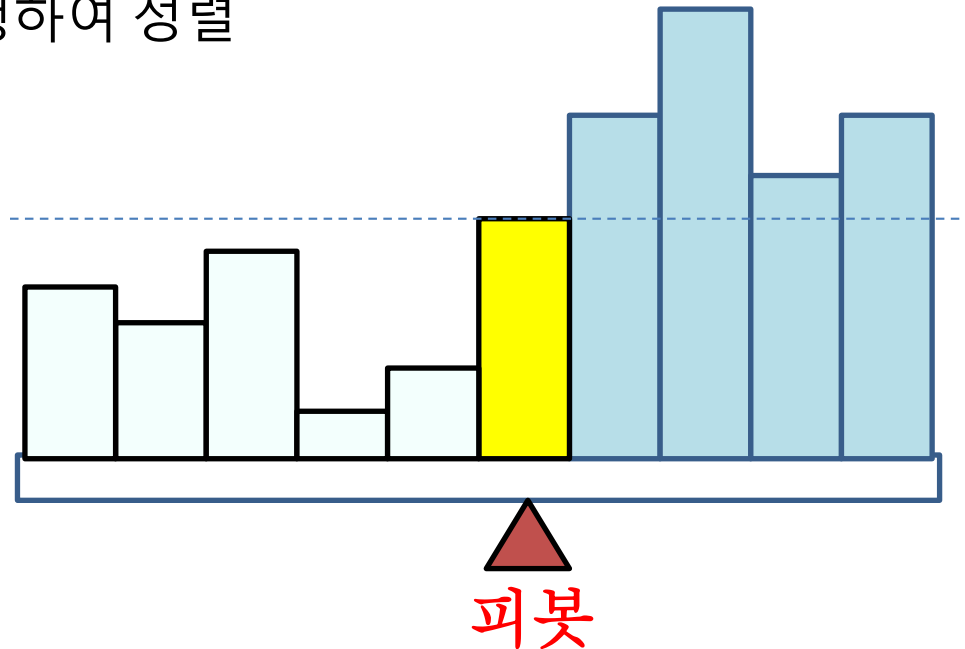
- 앞의 예제와 같이 5개의 3자리 십진수가 입력으로 주어지면, 가장 먼저 각 숫자의 1의 자리만 비교하여 작은 수부터 큰 수를 정렬한다.
- 그 다음엔 10의 자리만을 각각 비교하여 정렬한다. 이때 반드시 지켜야 할 순서가 있다.
- 예제에서 10의 자리가 3인 131과 035가 있는데, 10의 자리에 대해 정렬될 때 131이 반드시 035 위에 위치하여야 한다.
- 10의 자리가 같은데 왜 035가 131 위에 위치하면 안 되는 것일까?
 - 그 답은 1의 자리에 대해 정렬해 놓은 것이 아무 소용이 없게 되기 때문이다.

- 예제에서 $i=1$ 일 때, 입력의 각 숫자의 1의 자리 수만을 보면, 9, 0, 5, 1, 0이므로, 1의 자리가 '0'인 숫자가 2개, '1'인 숫자가 1개, '5'인 숫자가 1개, '9'인 숫자가 1개이다.
- 따라서 1의 자리가 '0'인 숫자 070과 910, '1'인 숫자 131, '5'인 숫자 035, 마지막으로 '9'인 숫자 089 순으로 입력의 숫자들이 정렬된다.

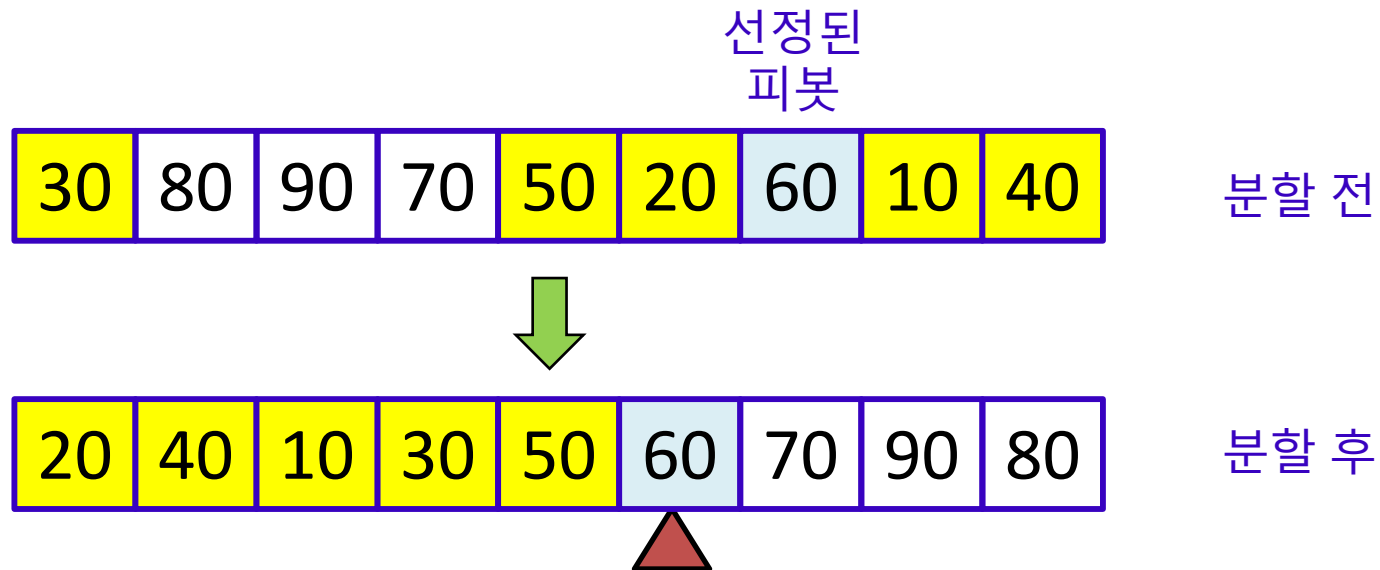


퀵 정렬

- 퀵 정렬 알고리즘은 문제를 2개의 부분문제로 분할하는데, 각 부분문제의 크기가 일정하지 않은 형태의 분할 정복 알고리즘
- 퀵 정렬은 **피벗 (pivot)**이라 일컫는 배열의 원소(숫자)를 기준으로 피벗보다 작은 숫자들은 왼편으로, 피벗보다 큰 숫자들은 오른편에 위치하도록 분할하고, 피벗을 그 사이에 놓는다.
- 퀵 정렬은 분할된 부분문제들에 대하여서도 위와 동일한 과정을 재귀적으로 수행하여 정렬



- 피벗은 분할된 왼편이나 오른편 부분에 포함되지 않는다. 피벗이 60이라면, 60은 [20 40 10 30 50]과 [70 90 80] 사이에 위치한다.



퀵 정렬 알고리즘

QuickSort(A, left, right)

입력: 배열 $A[\text{left}] \sim A[\text{right}]$

출력: 정렬된 배열 $A[\text{left}] \sim A[\text{right}]$

1. if (left < right) {
2. 피벗을 $A[\text{left}] \sim A[\text{right}]$ 중에서 선택하고, 피벗을 $A[\text{left}]$ 와 자리를 바꾼 후, 피벗과 배열의 각 원소를 비교하여 피벗보다 작은 숫자들은 $A[\text{left}] \sim A[p-1]$ 로 옮기고, 피벗보다 큰 숫자들은 $A[p+1] \sim A[\text{right}]$ 로 옮기며, 피벗은 $A[p]$ 에 놓는다.
3. **QuickSort**(A, left, p-1) // 피벗보다 작은 그룹
4. **QuickSort**(A, p+1, right) // 피벗보다 큰 그룹
- }

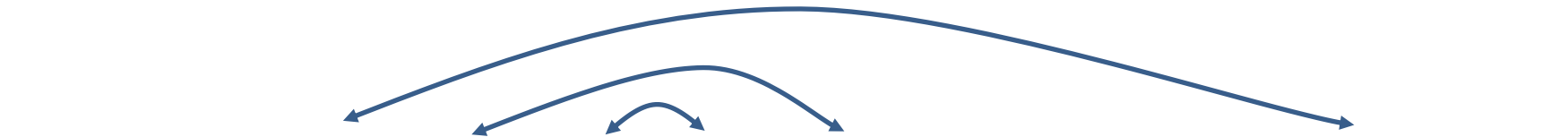
QuickSort(A,0,11) 호출

0	1	2	3	4	5	6	7	8	9	10	11
6	3	11	9	12	2	8	15	18	10	7	14


피벗 $A[6]=8$ 이라면, line 2에서 아래와 같이 차례로 원소들의 자리를 바꾼다. 먼저 피벗을 가장 왼쪽으로 이동시킨다.

0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14

- 그 다음엔 피벗보다 큰 수와 피벗보다 작은 수를 다음과 같이 각각 교환




0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14




0	1	2	3	4	5	6	7	8	9	10	11
2	3	7	6	8	12	9	15	18	10	11	14

0	1	2	3
2	3	7	6


0	1	2	3
6	3	7	2



0	1	2	3
2	3	6	7



0	1	0	1	0	1
2	3	3	2	2	3



시간복잡도

- 퀵 정렬의 성능은 피벗 선택이 좌우한다. 피벗으로 가장 작은 숫자 또는 가장 큰 숫자가 선택되면, 한 부분으로 치우치는 분할을 야기한다.

피벗

1	17	42	9	18	23	31	11	26
---	----	----	---	----	----	----	----	----



1	9	42	17	18	23	31	11	26
---	---	----	----	----	----	----	----	----



...

1	9	11	17	18	23	26	31	42
---	---	----	----	----	----	----	----	----



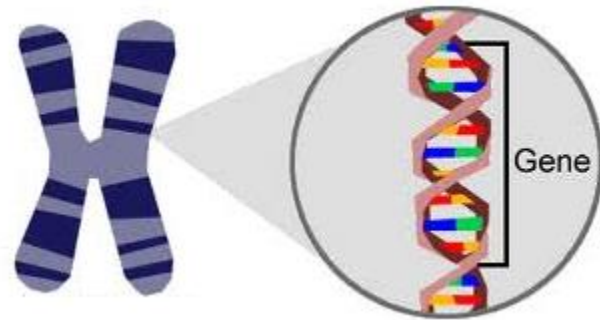
피벗 선정 방법

- 랜덤하게 선정하는 방법
- 숫자의 중앙값으로 선정하는 방법: 가장 왼쪽 숫자, 중간 숫자, 가장 오른쪽 숫자 중에서 중앙값으로 피벗을 정한다. 아래의 예제를 보면, 31, 1, 26 중에서 중앙값인 26을 피벗으로 사용한다.

31	17	42	9	1	23	18	11	26
----	----	----	---	---	----	----	----	----

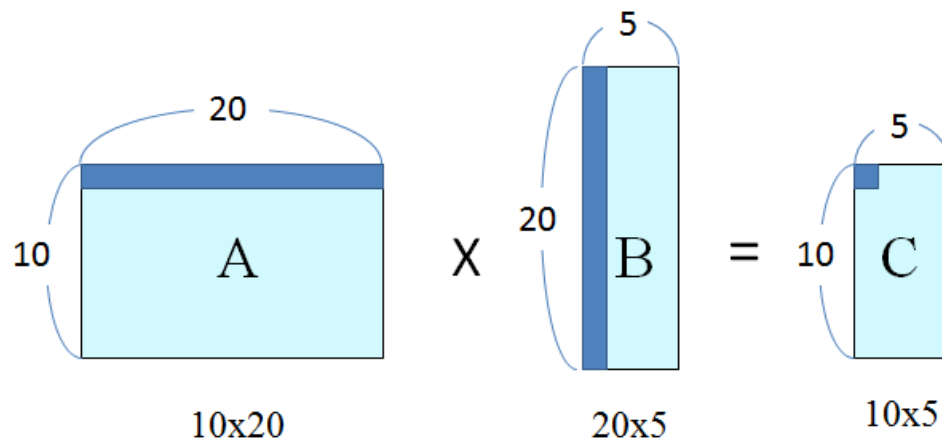
응 용

- 켄 정렬은 커다란 크기의 입력에 대해서 가장 좋은 성능을 보이는 정렬 알고리즘이다.
- 켄 정렬은 실질적으로 어느 정렬 알고리즘보다 좋은 성능을 보인다.
- 생물 정보 공학(Bioinformatics)에서 특정 유전자를 효율적으로 찾는데 접미 배열(suffix array)과 함께 켄 정렬이 활용된다.

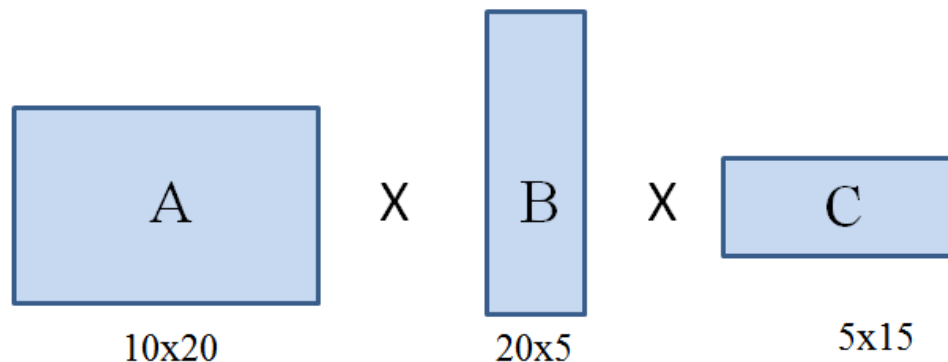


연속 행렬 곱셈

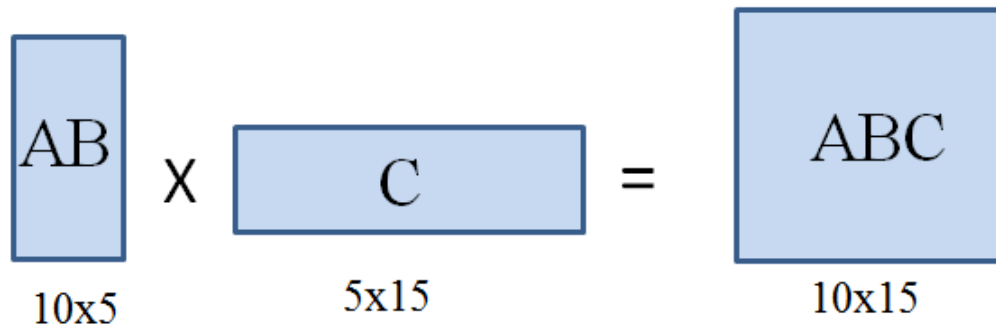
- 연속 행렬 곱셈 (Chained Matrix Multiplications) 문제는 연속된 행렬들의 곱셈에 필요한 원소간의 최소 곱셈 횟수를 찾는 문제이다.
- 10×20 행렬 A와 20×5 행렬 B를 곱하는데 원소간의 곱셈 횟수는 $10 \times 20 \times 5 = 1,000$ 이다. 그리고 두 행렬을 곱한 결과 행렬 C는 10×5 이다.



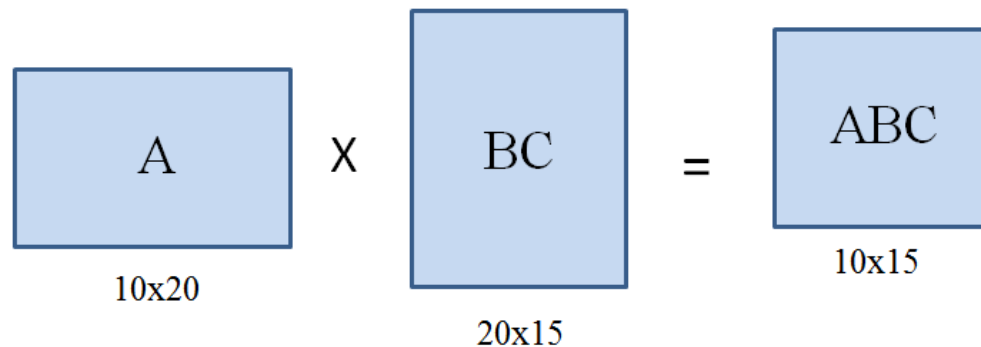
- 행렬 C의 1개의 원소를 위해서 행렬 A의 1개의 행에 있는 20개 원소와 행렬 B의 1개의 열에 있는 20개의 원소를 각각 곱한 값을 더해야 하므로 20회의 곱셈이 필요하다.
- 3개의 행렬을 곱해야 하는 경우
- 연속된 행렬의 곱셈에는 **결합 법칙이 허용**된다.
- 즉, $A \times B \times C = (A \times B) \times C = A \times (B \times C)$ 이다.
- 다음과 같이 행렬 A가 10×20 , 행렬 B가 20×5 , 행렬 C가 5×15 라고 하자.



- 먼저 $A \times B$ 를 계산한 후에 그 결과 행렬과 행렬 C 를 곱하기 위한 원소간의 곱셈 횟수를 세어 보면, $A \times B$ 를 계산하는데 $10 \times 20 \times 5 = 1,000$ 번의 곱셈이 필요하고, 그 결과 행렬의 크기가 10×5 이므로, 이에 행렬 C 를 곱하는데 $10 \times 5 \times 15 = 750$ 번의 곱셈이 필요하다.
- 총 $1,000 + 750 = 1,750$ 회의 원소의 곱셈이 필요하다.



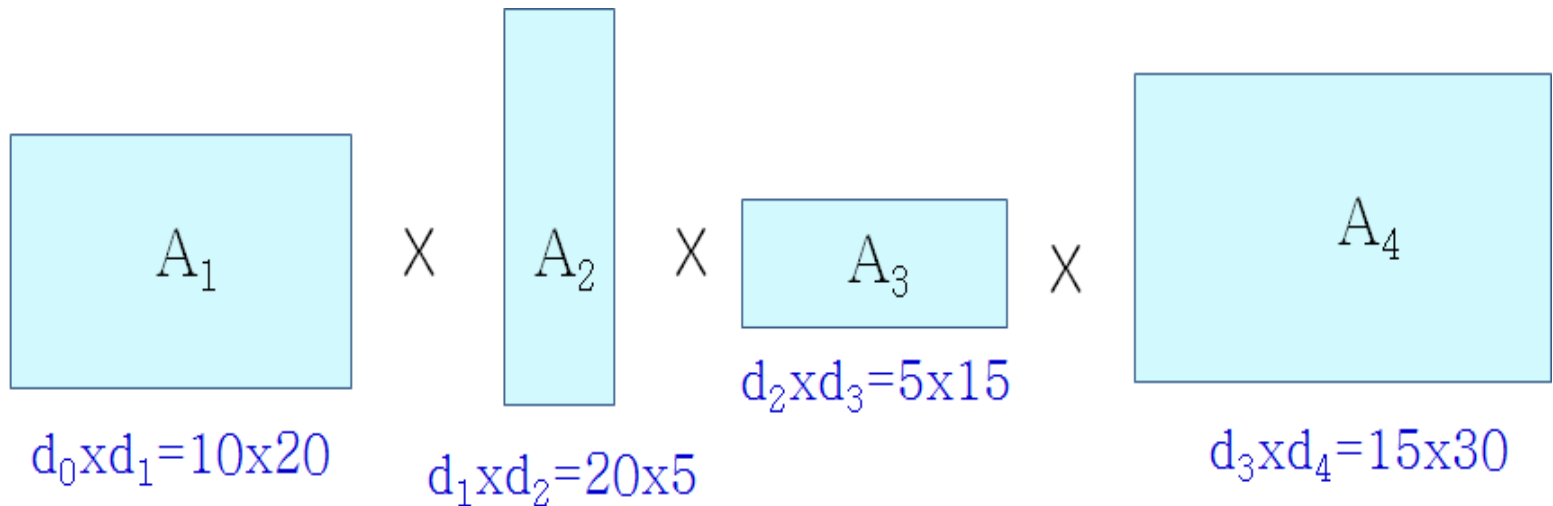
- 이번엔 BxC를 먼저 계산한 후에 행렬 A를 그 결과 행렬과 곱하면, BxC를 계산하는데 $20 \times 5 \times 15 = 1,500$ 번의 곱셈이 필요하고, 그 결과 20×15 행렬이 만들어지므로, 이를 행렬 A와 곱하는데 $10 \times 20 \times 15 = 3,000$ 번의 곱셈이 필요하다.
- 따라서 총 $1,500 + 3,000 = 4,500$ 회의 곱셈이 필요하다.



- 동일한 결과를 얻음에도 불구하고 원소간의 곱셈 횟수가 $4,500 - 1,700 = 2,800$ 이나 차이 난다.
- 따라서 연속된 행렬을 곱하는데 필요한 원소간의 곱셈 횟수를 최소화시키기 위해서는 적절한 행렬의 곱셈 순서를 찾아야 한다.
- 주어진 행렬의 순서를 지켜서 이웃하는 행렬끼리 반드시 곱해야 하기 때문
- 예를 들어, $A \times B \times C \times D \times E$ 를 계산하려는데, B를 건너뛰어서 $A \times C$ 를 수행한다든지, B와 C를 건너뛰어서 $A \times D$ 를 먼저 수행할 수 없다.
- 따라서 다음과 같은 부분문제들이 만들어진다.

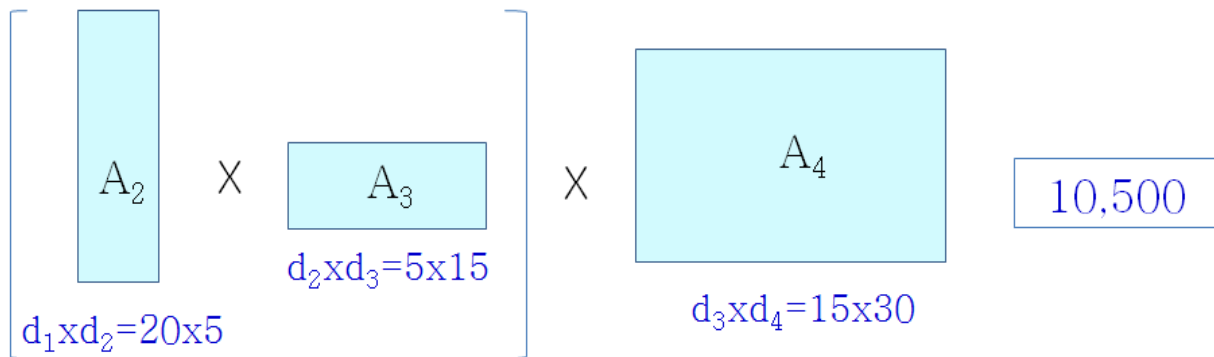
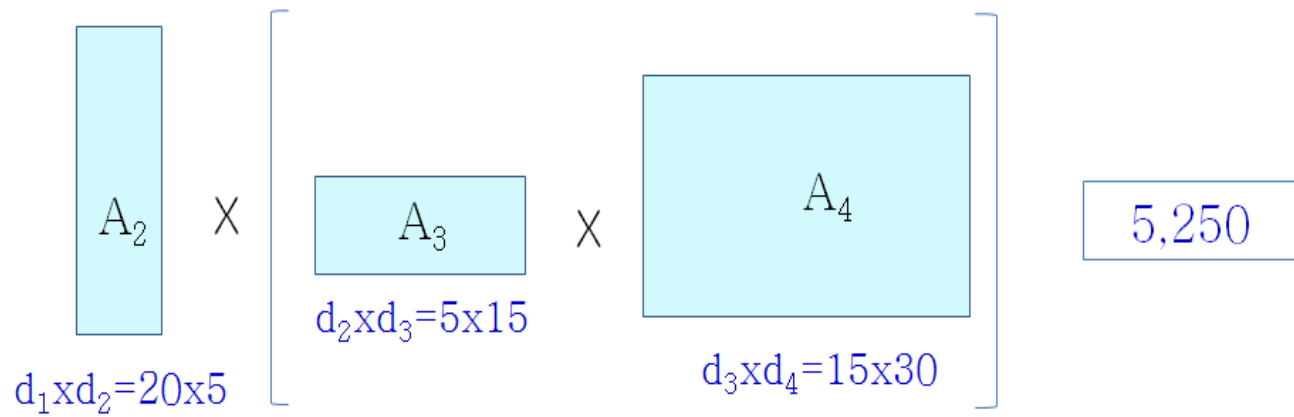
MatrixChain 알고리즘의 수행 과정

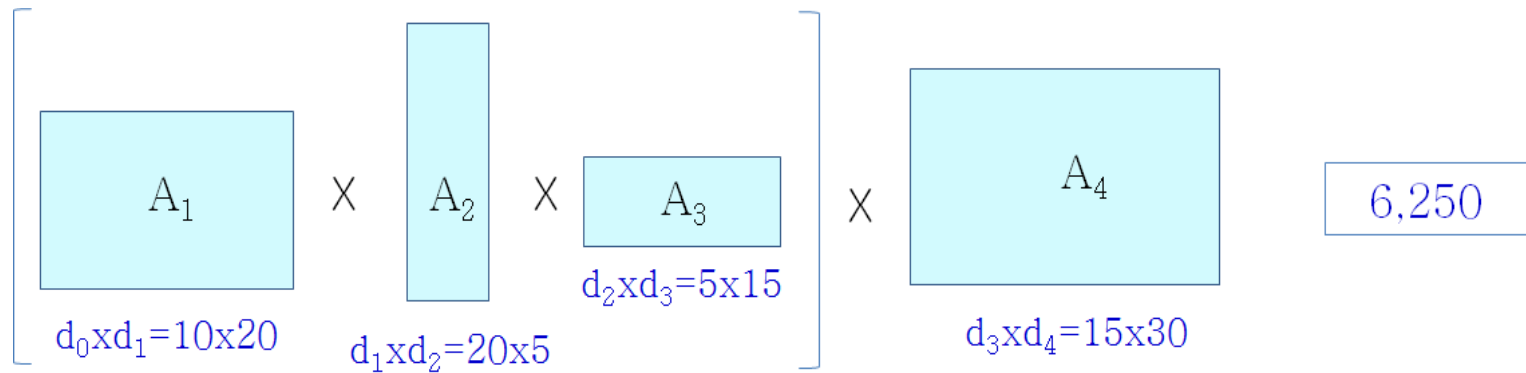
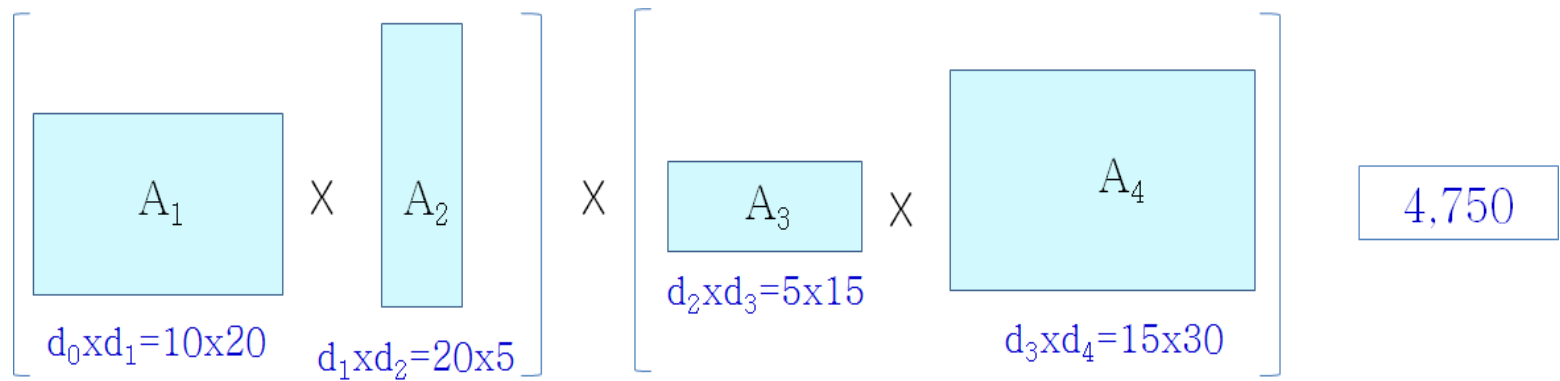
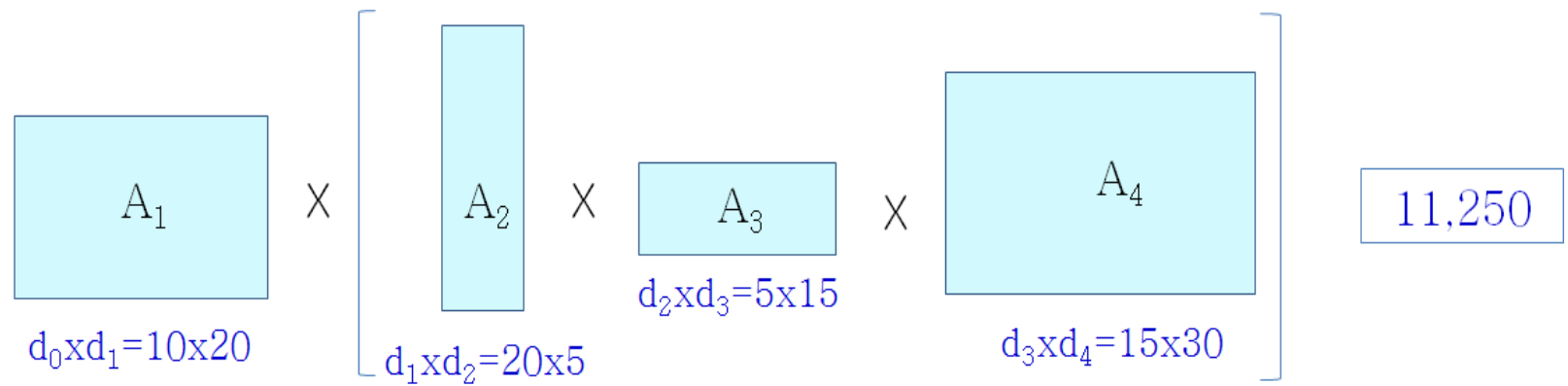
- A_1 이 10x20, A_2 가 20x5, A_3 이 5x15, A_4 가 15x30이다.



$$\begin{array}{c}
 \boxed{A_1} \\
 d_0 \times d_1 = 10 \times 20
 \end{array}
 \times
 \left[\begin{array}{c}
 \boxed{A_2} \\
 d_1 \times d_2 = 20 \times 5
 \end{array}
 \times
 \begin{array}{c}
 \boxed{A_3} \\
 d_2 \times d_3 = 5 \times 15
 \end{array}
 \right]
 \boxed{4,500}$$

$$\left[\begin{array}{c}
 \boxed{A_1} \\
 d_0 \times d_1 = 10 \times 20
 \end{array}
 \times
 \begin{array}{c}
 \boxed{A_2} \\
 d_1 \times d_2 = 20 \times 5
 \end{array}
 \right]
 \times
 \begin{array}{c}
 \boxed{A_3} \\
 d_2 \times d_3 = 5 \times 15
 \end{array}
 \boxed{1,750}$$





- 따라서 최종해는 4,750번이다. 먼저 $A_1 \times A_2$ 를 계산하고, 그 다음엔 $A_3 \times A_4$ 를 계산하여, 각각의 결과를 곱하는 것이 가장 효율적이다. 다음은 알고리즘이 수행된 후의 배열 c이다.

C	1	2	3	4
1	0	1,000	1,750	4,750
2		0	1,500	5,250
3			0	2,250
4				0