9장. 정렬 알고리즘

❑ 정렬 알고리즘

- 가장 많이, 그리고 가장 잘 알려진 알고리즘
- 방법과 효율
- 빅 오 기호로 분석

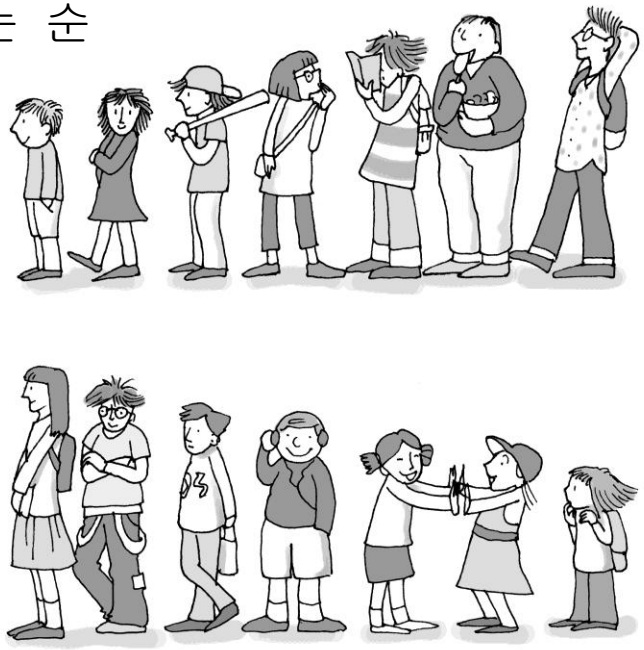
❑ 학습목표

- 정렬 알고리즘의 분류방법을 이해한다.
- 정렬 알고리즘 별로 작동원리에 대해 이해한다.
- 정렬 알고리즘 별로 효율분석 방법을 이해한다.
- 정렬 알고리즘에 따라 효율을 개선하기 위한 방법을 이해한다.

정렬의 분류

- **정렬**
  - 정렬의 대상 = 레코드
  - 정렬의 기준 = 정렬 키(Sort Key) 필드

- **오름차순, 내림차순**
  - 오름차순: 키 크기가 증가하는 순
  - 내림차순: 키 크기가 감소하는 순



정렬의 분류

❑ 내부정렬, 외부정렬

- 내부정렬
  - 정렬대상을 한꺼번에 메인 메모리에 올릴 수 있을 때
- 외부정렬
  - 정렬대상을 한꺼번에 메인 메모리로 올릴 수 없을 때
  - 메인 메모리와 보조 메모리 사이를 들락날락 하면서 정렬

정렬의 분류

- 안정 정렬(Stable Sorting)과 불안정 정렬(Unstable Sorting)
  - 1차 키: 학점
  - 2차 키: 학년
  - 1차 키로 정렬하더라도 이전의 2차 키 정렬순서가 유지됨.

성명	학년	학점	주소지
김용태	1	C	대구
정지희	1	B	서울
유일근	1	A	서울
박하영	3	B	전주
정건호	3	C	인천
김무성	3	A	수원
최석	4	A	용인

성명	학년	학점	주소지
김무성	3	A	수원
최석	4	A	용인
유일근	1	A	서울
박하영	3	B	전주
정지희	1	B	서울
김용태	1	C	대구
정건호	3	C	인천

성명	학년	학점	주소지
유일근	1	A	서울
김무성	3	A	수원
최석	4	A	용인
정지희	1	B	서울
박하영	3	B	전주
김용태	1	C	대구
정건호	3	C	인천

정렬의 분류

- 직접 정렬(Direct Sorting)과 간접 정렬(Indirect Sorting)

입력

인덱스	0		1		2	
레코드	김모	90	박모	50	최모	70

직접정렬

인덱스	0		1		2	
레코드	박모	50	최모	70	김모	90

간접정렬

	0	1	2
인덱스	0	1	2

	0	1	2
인덱스	1	2	0

선택정렬

- 가장 큰 것을 선택하여 가장 마지막 것과 스와핑

22	37	15	19	12
22	12	15	19	37
19	12	15	22	37
15	12	19	22	37
12	15	19	22	37

선택정렬의 효율

- 코드 9-1: 선택 정렬

```
void Selection(int A[ ], int N)      A는 배열이름, N은 정렬대상 레코드 수
{  for (int Last = N-1; Last >= 1; --Last) 마지막 인덱스를 왼쪽으로 이동하면서
   {  int Largest = 0;                일단 처음 것이 가장 크다고 보고
      for (int Current=1; Current<=Last; Current++) 처음부터 마지막까지
     {  if (A[Current] > A[Largest]) 현재 것이 더 크면
          Largest = Current;        현재 인덱스를 가장 큰 레코드의 인덱스로
     }
      int Temp = A[Last];           이동을 위해 마지막 레코드를 잠시 저장
     A[Last] = A[Largest];          가장 큰 레코드를 마지막으로 이동
     A[Largest] = Temp;             마지막 것을 가장 큰 레코드 위치로 이동
   }
}
```

- if 문: (N-1) + (N-2) + ... + 1 = (N-1)N/2 번 실행, 비교와 할당으로 구성
- 기타 할당문: 4(N-1)

- 효율: $2(N-1)N/2 + 4(N-1) = N^2 + 3N - 4 = O(N^2)$
- 대략적 분석: 왼쪽 위의 삼각형 면적이 비교의 횟수임. $O(N^2/2) = O(N^2)$

이중 선택정렬

- 한 번의 스캔에 최대치와 최소치를 동시에 발견하고 스와핑

  - MinMax Sorting
  - $O(N^2/2) = O(N^2)$
  - 단계의 수가 반으로 감소
  - 계수를 줄이는 노력

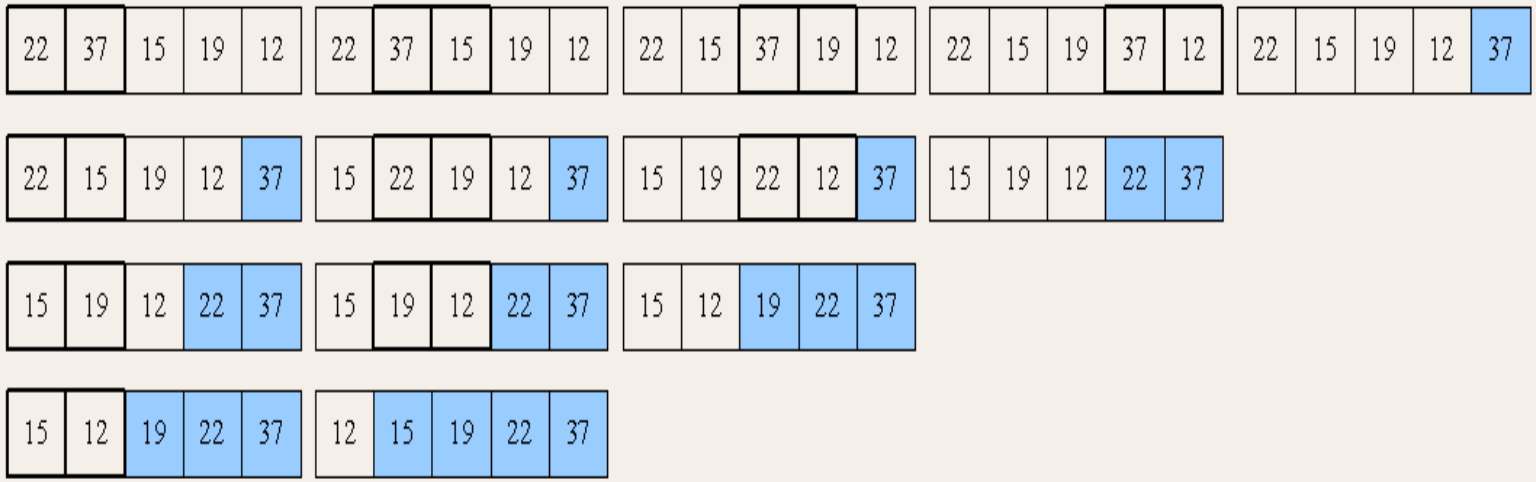
	13	40	15	12	35	14	19	17
1단계 결과	12	17	15	13	35	14	19	40
2단계 결과		13	15	17	19	14	35	
3단계 결과			14	17	15	19		
4단계 결과				15	17			

버블정렬

- 버블정렬

버블정렬

- 가장 큰 레코드가 한 칸씩 오른쪽 끝으로 떠올라 오는 정렬
- 한 쌍씩 비교하되 이전 쌍의 둘째 레코드가 다음 쌍의 첫 레코드가 되게 중복



버블정렬

- 단계 수
  - 데이터 N개라면 버블 정렬의 단계 수는 대략 N. 어떤 단계에서 스와핑이 한번도 일어나지 않았다면 (이어지는 한 쌍끼리 모두) 정렬 완료된 것임.

코드 9-2: 버블 정렬

```
void Bubble(int A[ ], int N)  A는 배열이름, N은 정렬대상 레코드 수
{   bool Sorted = FALSE;      스와핑이 전혀 없는 단계에서 빠져나가기 위한 변수
    for (int Pass = 1; (Pass < N) && (!Sorted); ++Pass)
    {   Sorted = TRUE;        스와핑이 전혀 없다고 초기화
        for (int Current=0; Current<N-Pass; ++Current)
        {   if A[Current] > A[Current+1]   현재 것이 다음 것보다 크면
            {   int Temp = A[Current];     스왑
                A[Current] = A[Current+1];
                A[Current+1] = Temp;
                Sorted = FALSE;    스왑이 한번이라도 일어나면 다음 단계로
            ]
```

- 안쪽 루프 내부의 명령은 (N-1) + (N-2) + ... + 1 = (N-1)N/2 번 수행. 루프 내부의 비교문은 (N-1)N/2 번. 할당문 각각이 4 (N-1)N/2 번.
- 최종효율 (N-1)N/2 + 2(N-1)N = $O(N^2)$

버블정렬, 선택정렬

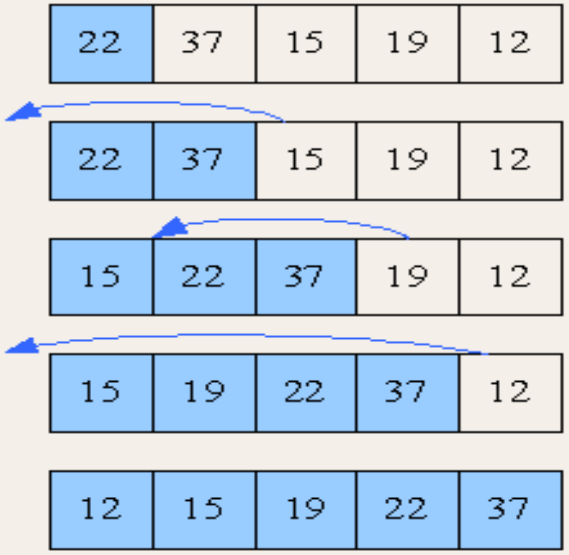
❑ 단계

- 단계별로 가장 큰 것이 가장 오른쪽으로 이동한다는 점에서 동일
- 선택정렬은 가장 큰 것과 가장 오른쪽 것이 한번에 스와핑.
- 버블 정렬은 가장 큰 것이 한 칸씩 오른 쪽으로 이동
  - 스와핑(교환, 복사)에 걸리는 시간
  - 큰 레코드에 대해서 버블 정렬은 선택 정렬보다 불리

❑ 이미 정렬된 데이터

- 버블 정렬이 최선의 효율
- 1단계에서 끝남. (스와핑이 전혀 없음). O(N)의 효율
- 선택 정렬은 여전히 $O(N^2)$의 효율.
- 가장 큰 데이터인 마지막 데이터가 자기 자신과 스왑(Self Swap)

삽입정렬



- 왼쪽 정렬된 그룹을 점차 키워 간다.
  - 1 단계:가장 왼쪽 첫 레코드 하나만 주목. 그 자체로 정렬
  - 2 단계: 다음 레코드를 왼쪽 것과 비교. 37은 22보다 크므로 그대로 둠.
  - 3 단계: 15는 37의 왼쪽으로 가야한다.
    - 풀 스왑: 삽입할 레코드가 계속적으로 왼쪽으로 옮김
    - 하프 스왑: 삽입될 레코드는 단 한번만 움직임.

삽입정렬의 효율

- 코드 9-3: 삽입 정렬

```
void Insertion(int A[ ], int N)
{ for (int Pick=1; Pick<N; ++Pick) 왼쪽부터 카드를 하나씩 집어내면서
  {  int Current = Pick;            집어낸 카드의 인덱스를 현재 인덱스로
     int Temp = A[Pick];
    for (; (Current > 0) && (A[Current-1]>Temp); --Current)
      A[Current] = A[Current-1]; 집어낸 카드 보다 크면 오른쪽으로 이동
    A[Current] = Temp;          집어낸 카드를 제 위치에 삽입 (하프 스왑)
  }
}
```

- 효율
  - 안쪽 루프 명령문은 1 + 2 + ... + (N-1) = (N-1)N/2 번 실행
  - for 문 자체에 비교 2번, 할당 1번, for 문 내부에 할당이 1번
  - 총 4(N-1)N/2 = 2(N-1)N = $O(N^2)$

- 이미 정렬된 데이터에 대해 삽입 정렬은 최선의 효율
  - 이미 정렬된 기존 사원 2000명, 신입사원 10명. 신입사원 파일을 기존사원 파일에 붙인 이후 삽입정렬. 2, 000명은 이미 정렬되어 있으므로 O(N)
  - 신입사원은 최악의 경우 배열의 맨 앞까지 가면서 비교와 스왑. 이 작업은 10명에 불과하므로 10N의 시간. 따라서 전체적인 효율은 O(N) + O(10N) = O(N)

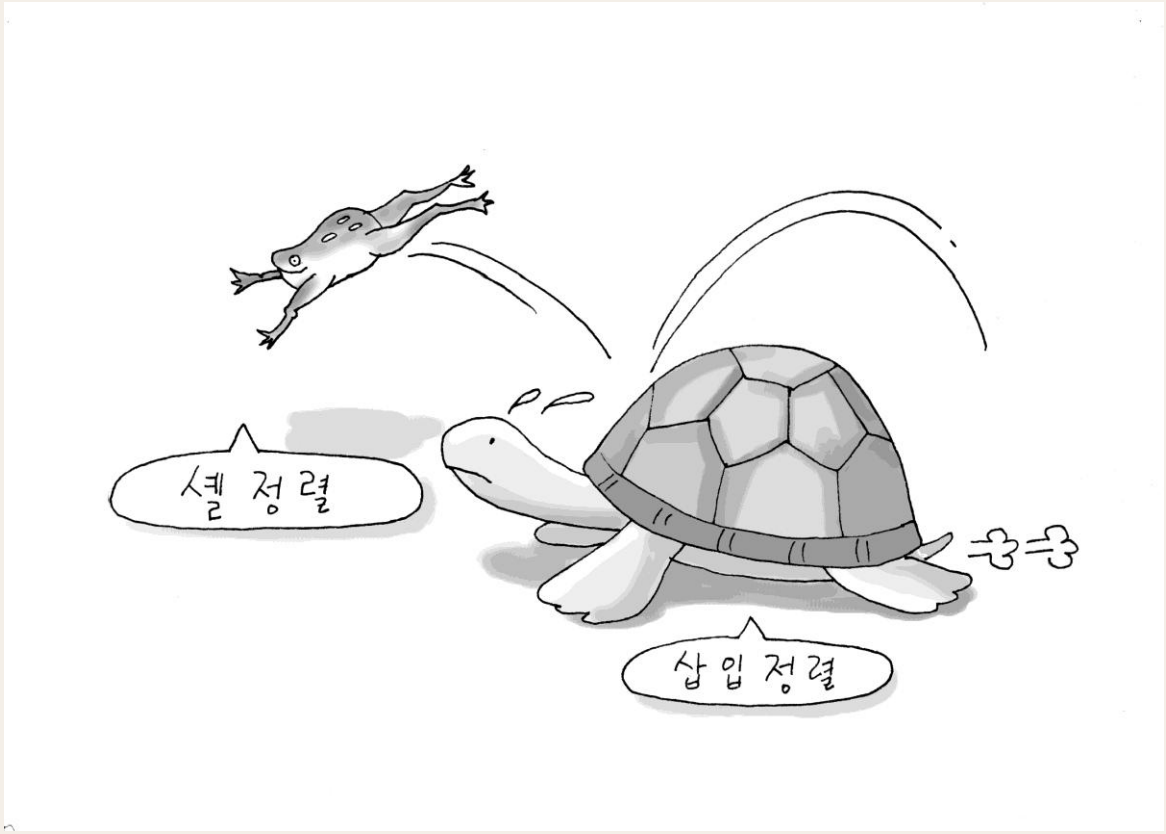
선택, 버블, 삽입

- 최악의 효율은 모두 $O(N^2)$으로서 동일
- 버블 정렬과 삽입 정렬는 안정정렬
  - 레코드들이 하나하나 순차적으로 이동(Shift)하기 때문에 원래의 순서가 유지
- 선택 정렬은 불안정 정렬
  - 스왑(Swap)에 의해 단번에 멀리 떨어진 곳으로 이동

	선택 정렬	버블 정렬	삽입 정렬
효율	$O(N^2)$	$O(N^2)$	$O(N^2)$
안정 정렬	No	Yes	Yes

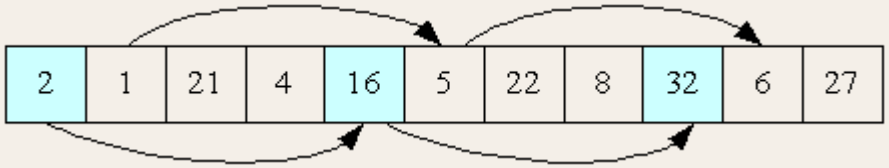
셸 정렬

❑ 셸 정렬

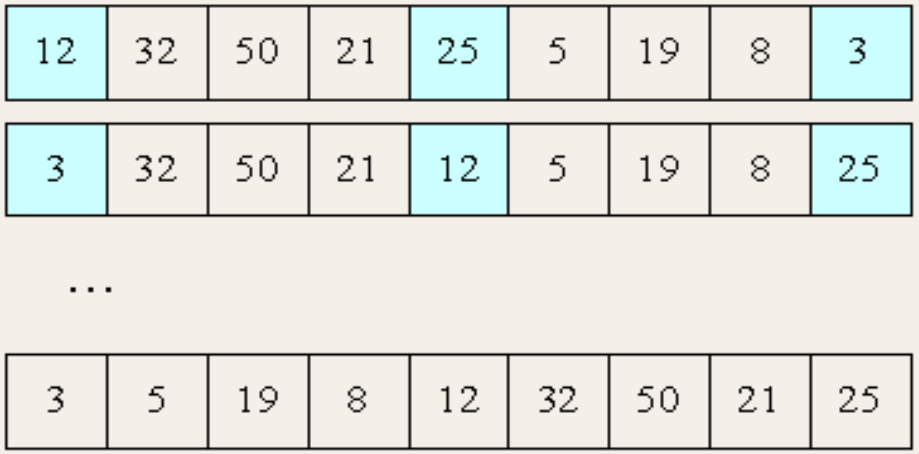


셸 정렬

- 삽입 정렬을 개선. 한 칸씩 이동하는 하는 대신 한번에 여러 칸 이동
- 4-정렬의 예

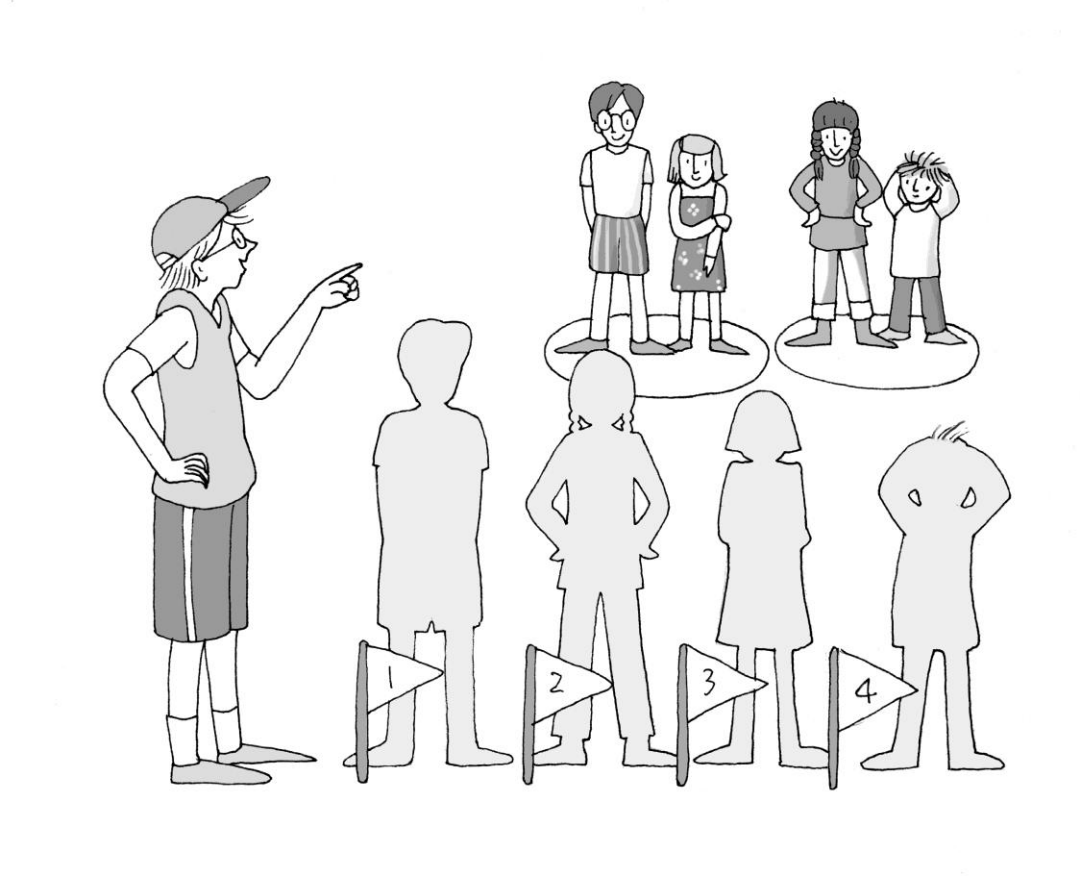


- 일련의 h-정렬
  - 최종적으로는 1-정렬. 효율은 $O(N^{3/2})$. 삽입 정렬의 $O(N^2)$보다는 빠르지만 O(NlogN)보다는 느린 효율



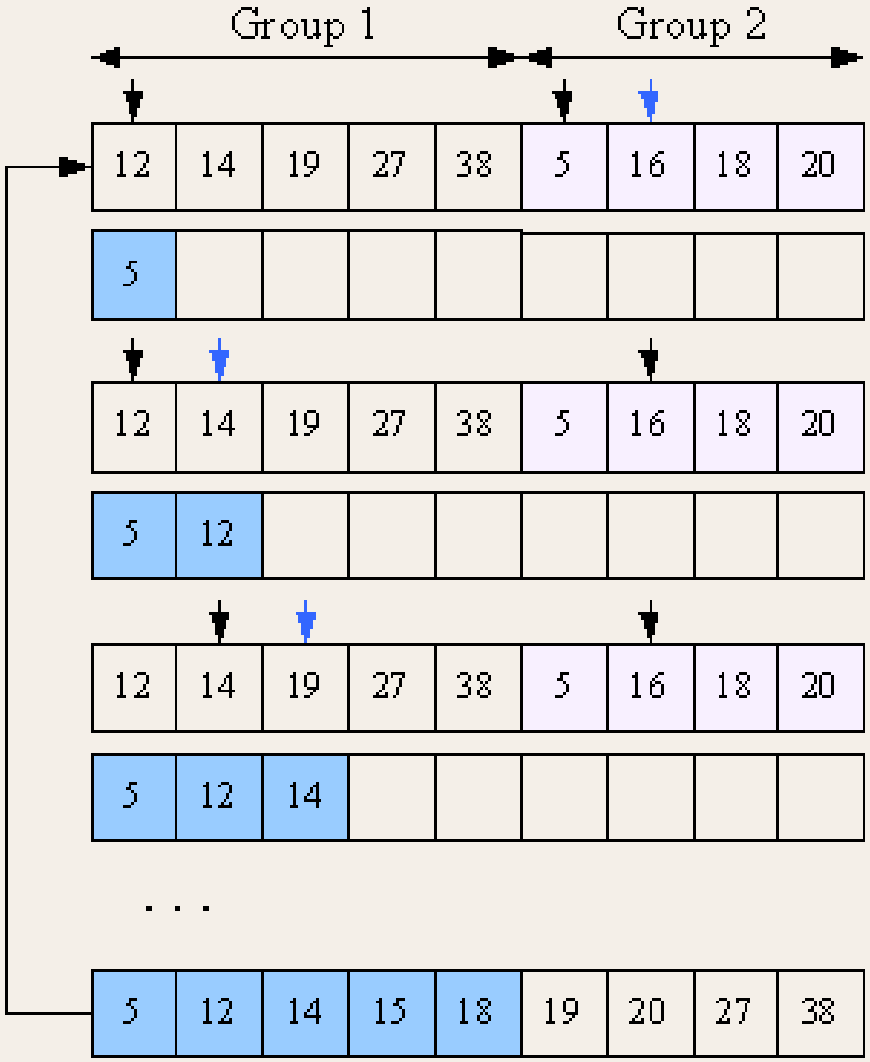
합병

- 합병



합병

- 정렬된 그룹의 합병


Group 1
Group 2
12 14 19 27 38 5 16 18 20
5
12 14 19 27 38 5 16 18 20
5 12
12 14 19 27 38 5 16 18 20
5 12 14
. . .
5 12 14 15 18 19 20 27 38

합병함수

❑ 코드 9-4: 합병 함수

```
void Merge(dataType A[ ], int F, int Mid, int L)   F, Mid, L은 그룹 분리를 위한 인덱스
{ dataType Temp[MAX];                  dataType의 배열 데이터를 가정
  int First1 = F;  int Last1 = Mid;          F부터 Mid까지가 첫 그룹
  int First2 = Mid + 1;  int Last2 = L;      (Mid+1)부터 L까지가 둘째 그룹
  int Index = First1;
  for (; (First1 <= Last1) && (First2 <= Last2); ++Index) 그룹 인덱스가 밖으로
   안 나갈 동안
  {   if (A[First1] < A[First2])       첫 그룹의 데이터가 작으면
    {   Temp[Index] = A[First1];       임시 저장 공간에 복사
        ++First1;                      포인터를 이동
     }
    else                               둘째 그룹의 데이터가 작거나 같으면
    {   Temp[Index] = A[First2];       임시 저장 공간에 복사
      ++First2;                        포인터를 이동
    }
  }
  for (; First1 <= Last1; ++First1, ++Index)    첫 그룹에 남은 데이터가 있으면
     Temp[Index] = A[First1];                  순서대로 임시 저장 공간에 복사
  for (; First2 <= Last2; ++First2, ++Index)    둘째 그룹에 남은 데이터가 있으면
     Temp[Index] = A[First2];                  순서대로 임시 저장 공간에 복사
  for (Index = F; Index <= L; ++Index)         임시 저장 공간의 데이터를
     A[Index] = Temp[Index];                   원래 배열로 복사시킴
}
```

합병정렬 메인함수

❑ 코드 9-5: 합병 정렬의 메인함수

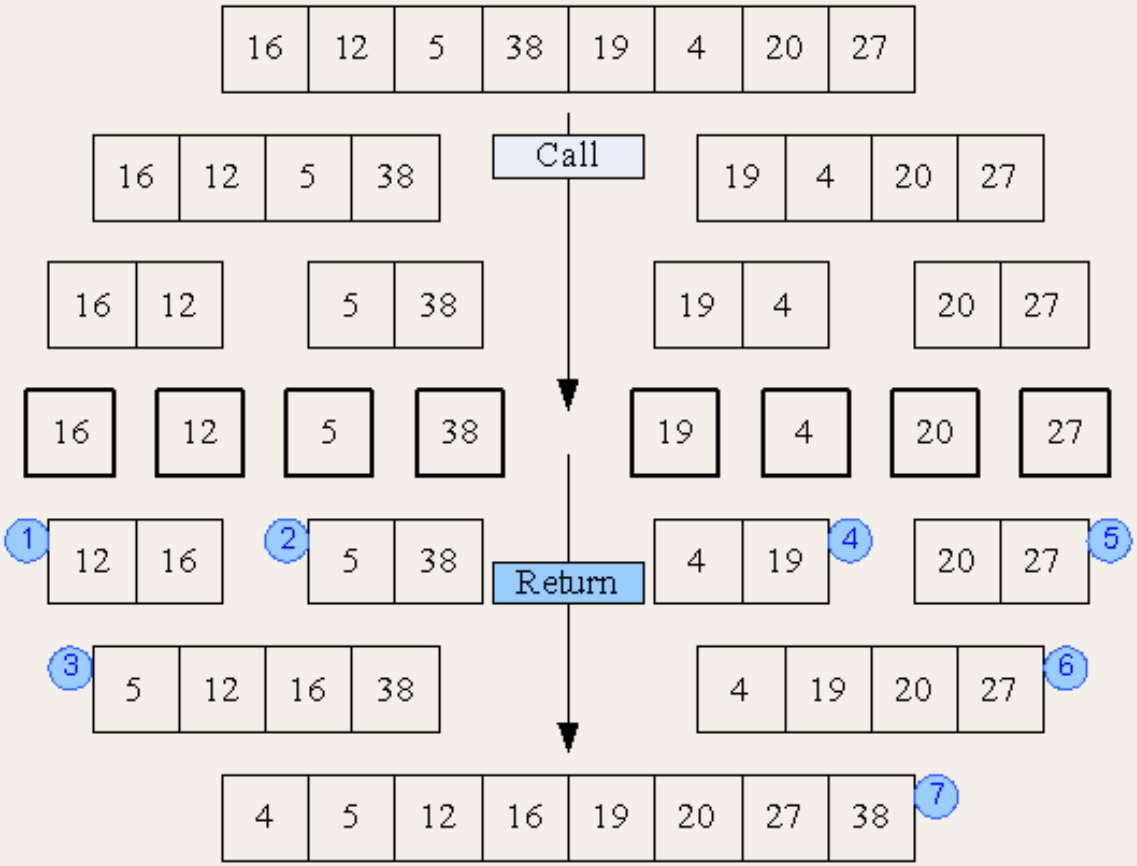
```
void MergeSort(int A[ ], int First, int Last)
{ if (First < Last)
 {   int Middle = (First + Last) / 2;
    MergeSort(A, First, Middle);
    MergeSort(A, Middle+1, Last);
    Merge(A, First, Middle, Last);
   }
 }
```

❑ 합병정렬

- 반 잘라서 왼쪽 재귀호출, 오른쪽 재귀호출. 결과를 합병
- 베이스 케이스로부터 빠져 나와 호출함수로 되돌아 오는 과정에서 Merge(A, First, Middle, Last) 즉, 합병에 의해 정렬

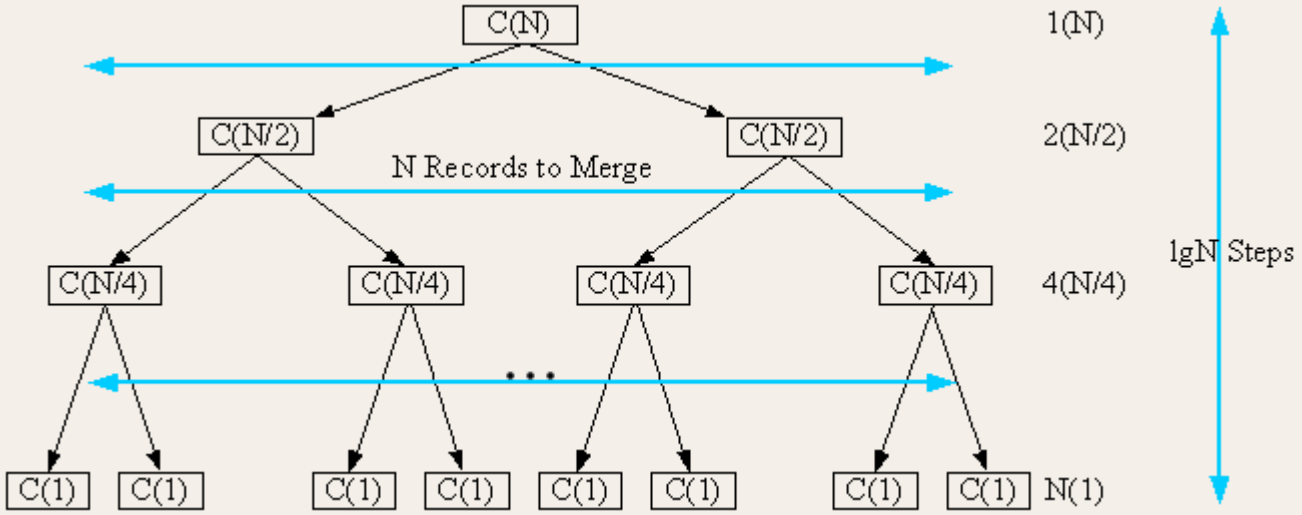
합병정렬 들어가고 나오기

- 합병정렬 들어가고 나오기



합병정렬의 효율

- **효율**
  - O(NlgN)
  - 호출의 단계 수가 lgN
  - 각 단계별로 합병에 O(N)



삽입정렬과 합병정렬

❑ 효율

- $O(N^2)$ 대 $O(NlgN)$
- 좋은 알고리즘은 슈퍼 컴퓨터보다 낫다.

- 삽입정렬

	$N=10^3$	$N=10^6$	$N=10^9$
PC	순간적	2.8 시간	317년
수퍼 컴	순간적	1초	1.7 주

- 합병정렬

	$N=10^3$	$N=10^6$	$N=10^9$
PC	순간적	1초	18분
수퍼 컴	순간적	순간적	순간적

쾌속

- 쾌속

쾌속정렬

❑ 합병정렬과 쾌속정렬

- 재귀호출의 순서에 유의
- 실제 정렬작업이 어디서 일어나는지 유의

MergeSort	QuickSort
{ MergeSort (Left); MergeSort (Right); Merge; }	{ Partition; QuickSort(Left); QuickSort(Right); }

퀘속정렬

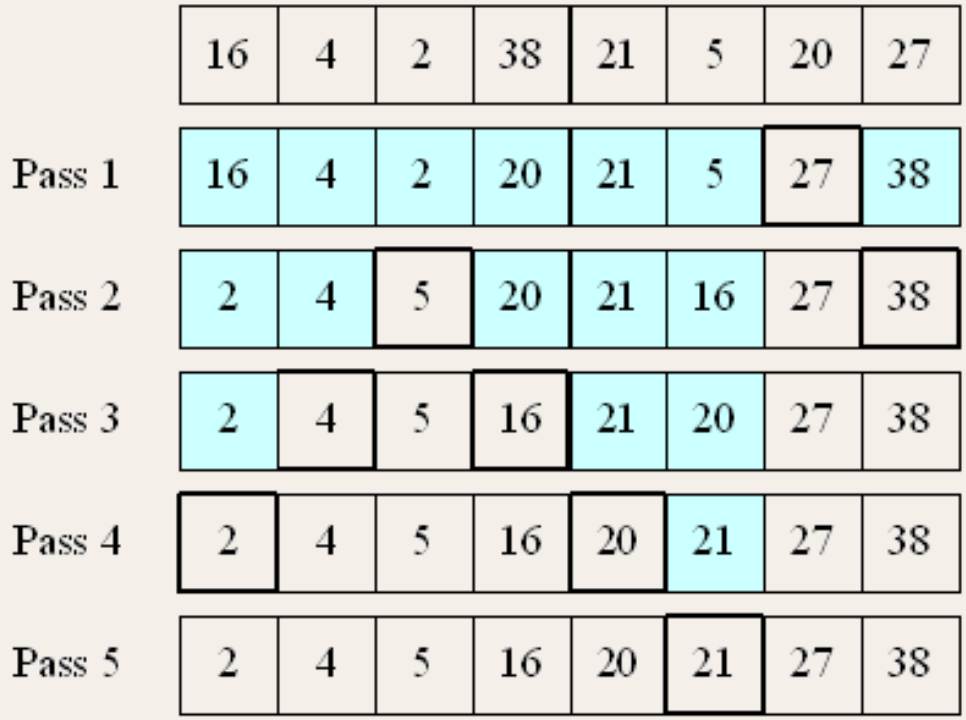
- 코드 9-6: 파티션 함수

```
int partition(int A[ ], int first, int last)
{  int low, high, pivotindex, p;
   p = A[last];                      마지막 요소를 피벗으로
   low = first;                      업 포인터를 처음으로
   high = last-1;                    다운 포인터를 마지막 직전 요소로
   while (low < high)                크로스 오버가 없을 때까지
   {  while (p > A[low]) low++;      피벗보다 크거나 같은 것 찾기
      while (p < A[high]) high--;    피벗보다 작거나 같은 것 찾기
      if (low < high)                크로스 오버가 아니면
         Swap(A, low, high);         작은 것과 큰 것을 스왑
   }
   Swap(A[low], A[last]);            업 포인터 레코드와 피벗 레코드를 스왑
   return (low);                     피벗 인덱스를 리턴
}
```

쾌속정렬 메인함수

❑ 코드 9-7: 쾌속 정렬의 메인함수

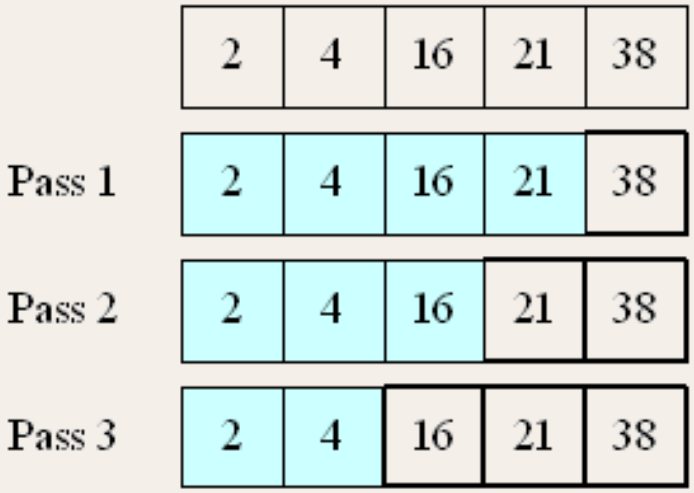
```
void QuickSort(int A[ ], int First, int Last)
{ if (First < Last)
{   int PivotIndex = Partition(A, First, Last);
   QuickSort(A, First, PivotIndex-1);
   QuickSort(A, PivotIndex+1, Last);
 }
}
```



쾌속정렬

- **효율**
  - 단계의 수에 따라 결정
  - 단계의 수는 파티션 결과 좌우 정확히 양분되면 lgN
  - 균형(Balancing)이 좋을 때 효율은 O(NlgNO

- **정렬된 데이터에 최악의 효율**
  - 파티션 결과 하나씩만 나가 떨어진
  - 단계의 수는 거의 N



...

쾌속정렬의 균형

❑ 파티션 방법

- 더 나은 균형을 위하여 같은 키에서도 스와핑

❑ 피벗의 선택 (“세개 중 메디안” 파티션)

- 처음 세 개, 마지막 세 개, 처음, 마지막, 중간 것
- 랜덤 함수에 의한 추출
- 어느 경우든 선택을 위한 시간이 소요됨.

❑ 시스템 정렬

- 샘플정렬
  - 램덤하게 여러 샘플 추출, 정렬, 그 중 중간값을 피벗으로
- 벤틀리 매클로이 방식
  - 한번에 3개의 샘플을 사용하여 메디안을 구함
  - 3번 반복하여 3개의 메디안을 구한 뒤, 그 중의 메디안 값을 피벗으로

❑ 파티션의 최종단계

- 예를 들어 데이터 10개 이하
- 직접 삽입정렬에 의해 정렬
- 재귀호출에 따른 활성화 레코드 생성,복원에 따르는 시간 배제

합병정렬, 쾌속정렬

❑ 효율비교

- 합병정렬은 매 단계마다 완벽한 균형
- 합병정렬은 최악의 경우에도 O(NlgN)을 보장
- 쾌속정렬은 최악의 경우 $O(N^2)$
- 쾌속정렬은 임시저장공간이 불필요한 제자리 연산(In-Place Computation)
- 합병정렬은 임시저장공간에 옮겨 가고 옮겨오는 시간이 필요
- 평균적으로는 쾌속정렬이 빠름

정렬방식 비교

❑ 합병정렬

	$N=10^3$	$N=10^6$	$N=10^9$
PC	순간적	1초	18분
수퍼 컴	순간적	순간적	순간적

❑ 쾌속정렬

	$N=10^3$	$N=10^6$	$N=10^9$
PC	순간적	0.3초	6분
수퍼 컴	순간적	순간적	순간적

❑ 삽입, 쾌속, 합병

	삽입 정렬	쾌속 정렬	합병 정렬
최악의 효율	N^2	N^2	NlgN
최선의 효율	N	NlgN	NlgN
평균적 효율	N^2	NlgN	NlgN
이미 정렬된 데이터	N	N^2	NlgN
반대로 정렬된 데이터	N^2	N^2	NlgN
공간	N	N	2N
안정정렬	Yes	No	YES

외부정렬

- 기본적으로 합병정렬
  - 입력 파일: D A T A S T R U C T U R E A N D A L G O R I T H M
  - 메인 메모리 용량이 세개 단위로 읽어 들여 정렬 가능하다고 가정

- 1단계:
  - 파일 1: ADT * RTU * GOR *
  - 파일 2: AST * AEN * HIT *
  - 파일 3: CRU * ADL * M *

- 2단계: (3개씩 합병)
  - 파일 4: AACDRSTTU *
  - 파일 5: AADELNRTU *
  - 파일 6: GHIMORT *

- 3단계: (3개씩 합병)
  - 파일 1: AAAACDDEGHILMNORRSTTTTUU *

외부정렬

- 1단계:
  - 파일 1: ADT * RTU * GOR *
  - 파일 2: AST * AEN * HIT *
  - 파일 3: CRU * ADL * M *
- 2단계: (2개씩 합병)
  - 파일 4: AADSTT * AADELN * M
  - 파일 5: CRRTUU * GHIORT *
- 3단계: (2개씩 합병)
  - 파일 1: AACDRRSTTTUU * M
  - 파일 2: AADEGHILNORT *
- 4단계:(2개씩 합병)
  - 파일 3: AAAACDDEGHILNORRSTTTTUU *
  - 파일 4: M
- 5단계(2개씩 합병)
  - 파일 1: AAAACDDEGHILMNORRSTTTTUU *

외부정렬

합병

- 모든 블록이 합병에 참가하기만 하면 합병의 순서는 관계없음

효율

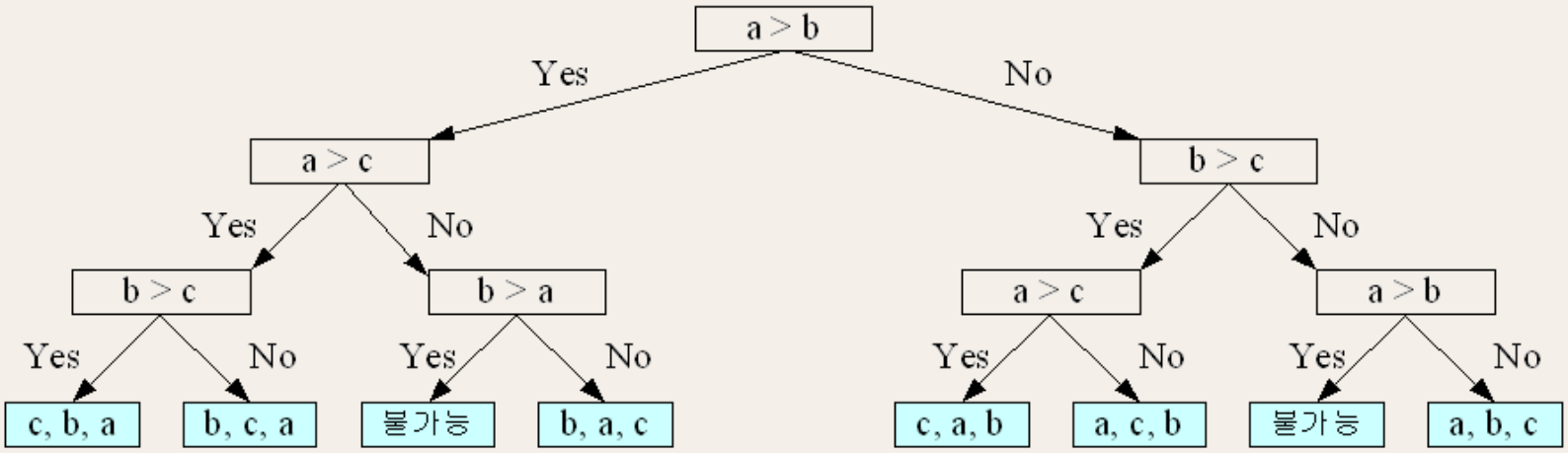
- CPU 연산시간 << 입출력 시간
- 외부정렬의 효율은 입출력시간에 좌우됨
- 몇 단계에 걸쳐 파일 출력이 일어나는가가 관건
- 1 단계 실행결과 정렬된 블록의 개수는 N/M 개

p개 단위의 합병

- 합병 한번에 블록의 개수는 1/p 씩 줄어듬
- 따라서 정렬의 효율은 대략 $\log_p(N/M)$

최선의 정렬효율

- 버블정렬의 결정트리
  - 3개의 키이므로 크기 순서의 조합은 3! = 6
  - 불필요한 비교를 감안하면 일반적으로 리프 노드의 수는 N! 개 이상



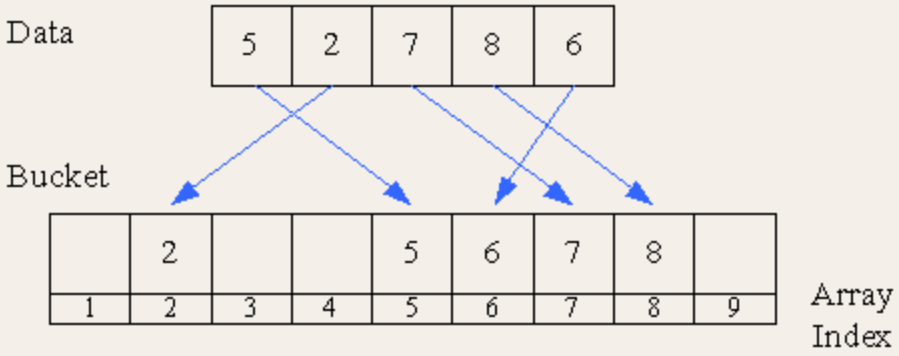
- 트리의 높이는 최소한 $\log_2(N!)$ 보다 크고 최소 비교의 회수는 $\log_2(N!)$.
- 스털링(Stirling)의 공식: $\log_2(N!) \geq \log_2(N/e)N = N\log_2 N - N\log 2e$.
- 따라서 정렬에서 가능한 최소 비교회수는 $N\log_2 N$

버켓정렬

- 버켓정렬

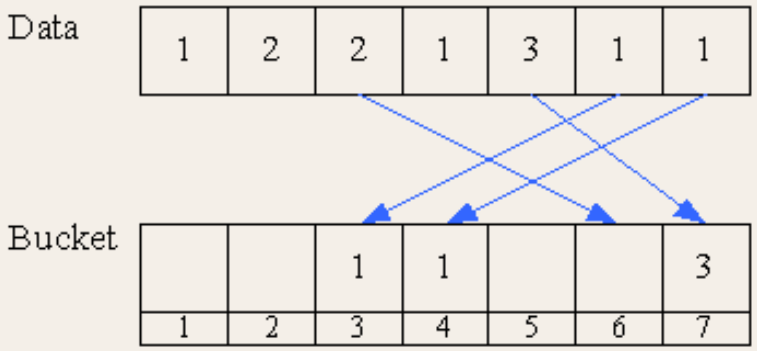
버켓정렬

- **방법**
  - 키 자체를 배열 인덱스로 하여 별도 배열로 옮김
- **효율**
  - O(N). 키 비교에 의한 정렬이 아니므로 O(NlgN)을 능가할 수 있음
- **단점**
  - 버켓 배열의 인덱스보다 큰 키가 들어올 수 없음
  - 버켓의 크기를 최대 인덱스 크기로 설정. 메모리 공간낭비 가능성
  - 이를 개선한 것이 셈 정렬



셈 정렬

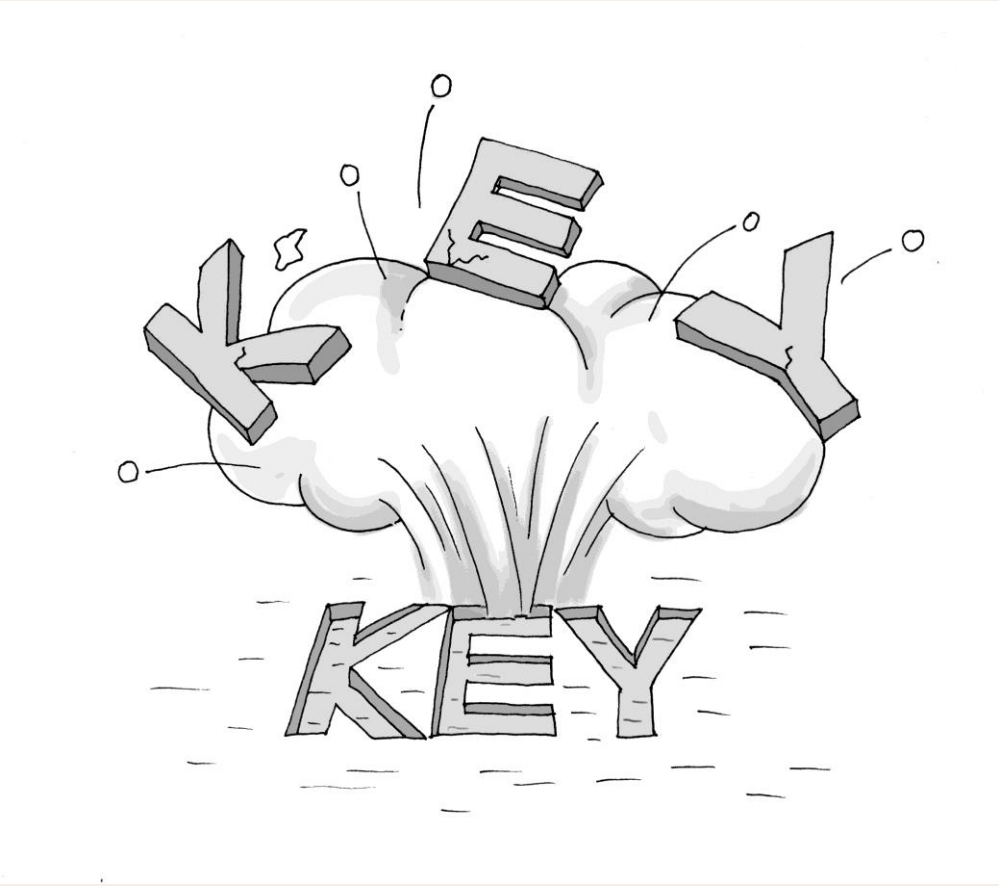
- 분포세기(Distribution Counting) 또는 셈 정렬(Count Sorting)
  - 중복 키 허용
  - 키 빈도를 Count[ ] 배열에 넣음. Count[1]= 4, Count[2] = 2, Count[3] = 1
  - 누계를 구하면 Count[1] = 4, Count[2] = 4 + 2 = 6, Count[3] = 6 + 1 = 7
  - 오른쪽에서 왼쪽으로 스캔 해 가면서 버켓에 삽입
  - 삽입 위치는 현재의 카운트 값. 삽입 직후에는 카운트 값을 하나씩 줄임



- 효율
  - O(N)로서 안정정렬

기수정렬

- 기수 = 키의 구성요소. 분해된 문자열, 분해된 문자.



기수정렬

❑ 기수정렬

- Radix = 뿌리
- 문자열 키의 뿌리는 문자. 문자의 뿌리는 비트
- 문자열 단위로 비교하는 대신 문자열을 잘라서 문자 단위로 비교

❑ 기수

- 8비트 ASCII 코드라면 256가지 레이딕스
- 16비트 유니코드(Unicode)라면 65,536 가지의 레이딕스
- 숫자를 문자열로 간주하면 숫자 키에 대해서도 기수 정렬을 가할 수 있음.

LSD 기수정렬

- LSD
  - 문자열 오른쪽(Least Significant Digit)에서 왼쪽으로
  - 제대로 동작하는 이유는 안정성 때문. 왼쪽 키이가 같다면 오른쪽 키이는 이전에 정렬된 순서를 유지함
  - 문자 단위의 정렬은 셈 정렬을 사용
  - 문자열 길이 W일 때, 효율은 O(WN).

0	a	d	d
1	c	a	b
2	f	e	e
3	b	a	d
4	d	a	d
5	b	e	e
6	b	e	d
7	a	c	e
			⇧

0	c	a	b
1	a	d	d
2	b	a	d
3	d	a	d
4	b	e	d
5	f	e	e
6	b	e	e
7	a	c	e
		⇧	

0	c	a	b
1	b	a	d
2	d	a	d
3	a	c	e
4	a	d	d
5	b	e	d
6	f	e	e
7	b	e	e
	⇧		

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	d
7	f	e	e

[표 9-20] B

LSD 기수정렬

단점

- 최종 단계의 정렬이 이루어지기 전까지는 조금이라도 정렬된 흔적이 없음
- 가변 길이 키에 적용하기 어려움.
- cold 와 old 라는 키에 이 방식을 가하면 뒤에서부터 올라오므로 마지막 첫 자리에서 cold의 c와 비교되어야 하는 것이 old에는 없으므로 아무것도 없음을 표시하는 널 문자(Null Character)와 비교.
- ASCII 코드 표에 의하면 c는 십진수 99, 널 문자는 십진수 0이므로 old, cold 순의 오름차순이 된다. 이는 잘못된 정렬
- 마지막 부근의 자릿수 정렬에 많은 시간을 허비. 만약 키가 Kim Pak Cho Rho 등이라면 사실은 첫 자리만 보고도 정렬이 가능한 것이다.

패딩

- cold, old, at를 비교하자면 cold, old0, at00 등으로 마지막에 0을 보충
- 0은 아스키 값 십진수 47로서 문자보다는 그 값이 작음
- 숫자를 문자열로 취급할 경우에는 거꾸로 처음에 0을 넣어서 비교
- 1611, 315, 17을 비교하자면 1611, 0315, 0017로 비교
- 패딩은 번거로운 작업으로서 키의 최대 길이를 찾아내는 시간, 패딩 하는 시간을 요한다.

MSD 기수정렬

MSD

- 왼쪽 (MSD: Most Significant Digit) 에서 오른쪽으로 진행
- 셈 정렬(Distribution Counting)을 사용. O(WN) = O(N)
- 이전의 모든 문자가 일치하는 것들만 다음 문자를 비교
- 앞부분이 유일(Uniqueness) 해지면 더 이상 비교 대상에서 제외

0	a	d	d
1	c	a	b
2	f	e	e
3	b	a	d
4	d	a	d
5	b	e	e
6	b	e	d
7	a	c	e
	⇧		

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	d
7	f	e	e
		⇧	

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	d
7	f	e	e
			⇧

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	d
7	f	e	e

기수교환 정렬

❑ 기수교환 정렬

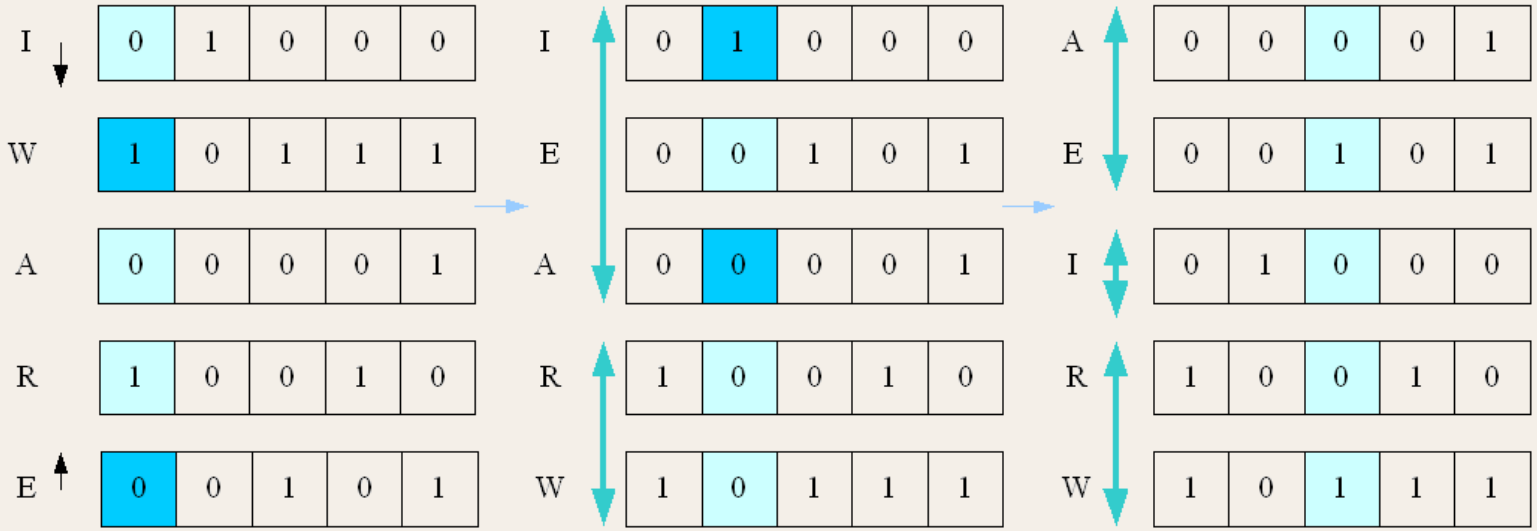
- 문자 단위의 정렬에 셈 정렬 대신 쾌속정렬 사용
- 별도 메모리 대신 스와핑 사용
- 3-way 파티션
  - 피벗 b와 일치하지 않는 키(add, ace 그룹, dad, fee, cab 그룹)는 다시 첫 문자를 기분으로 재귀적으로 정렬해야 함.
  - 피벗 b와 일치하는 키(bed, bad, bee)는 두번째 문자를 대상으로 정렬을 진행

0	a	d	d
1	c	a	b
2	f	e	e
3	b	a	d
4	d	a	d
5	b	e	e
6	b	e	d
7	a	c	e
	⇧		

0	a	d	d
1	a	c	e
2	b	e	d
3	b	a	d
4	b	e	e
5	d	a	d
6	f	e	e
7	c	a	b
		⇧	

비트단위의 기수교환정렬

- 파티션
  - 파티션 결과 0인 그룹과 1인 그룹으로 양분
  - 첫문자에 재귀적으로 파티션할 필요가 없음
  - 비트단위의 쾌속정렬
  - 비트 열이 유일(Uniqueness)해지면 더 이상 비교대상에서 제외시킴



비트 단위의 기수교환 정렬

유일한 비트열

- 파티션이 가해질 때마다 N개의 비트 값이 비교
- 데이터 N 개라면 대략적으로 $\log_2 N$ 비트 만에 모든 비트 열이 유일해 짐.
- 기수교환 정렬의 효율은 $O(N\log_2 N)$

통계적 분석

- 통계적으로 볼 때, 어떤 자리수가 0일 확률과 1일 확률이 1/2로서 동일
- 비트 단위의 파티션을 가하면 데이터의 반이 0 그룹, 나머지 반이 1 그룹
- 균형적 파티션이므로 단계의 수는 쾌속 정렬에서 지속적으로 균형적인 파티션이 일어날 때의 단계 수인 $\log_2 N$과 동일
- 균형이 일어날 확률이 높음으로 인해 쾌속 정렬보다 $O(N\log_2 N)$에 근접

효율

- 소요되는 시간 면에서 비트 단위의 기수교환 정렬은 쾌속 정렬과 큰 차이
- 기수교환 정렬에서 $O(N\log_2 N)$이라고 할 때에는 비트 단위의 비교회수
- 쾌속 정렬에서 $O(N\log_2 N)$라고 할 때에는 문자열 단위의 비교회수