정렬 알고리즘 3

최소/대 선택 외부정렬

특정 순서 원소 찾기

- 정렬 알고리즘의 응용
- 선택 문제 (selection problem): 임의로 나열된 n개의 데이터에서 크기가 i번째인 것을 찾는 문제
  - i=1 : 최소치 문제
  - i=n : 최대치 문제

최소값 찾기 문제

```
int Minimum (int A[ ], int n ) {
   /*입력 : A : n개의 숫자가 저장되어 있는 배열
       n : 배열 A에 저장되어 있는 숫자의 개수
     출력 : A에 저장된 값 중에서 최소값 */
     int i, Temp ;
     Temp = A[0];
     for (i = 1; i < n; i++) //배열의 원소를 처음부터 끝까지 검사하여 최소인것 찾기
        if (Temp > A[i]) Temp=A[i];
     return Temp;
 }
```

최소값 찾기 문제

- 평균 시간 복잡도
- 최악 시간 복잡도
- 최대값 찾기 문제도 비슷한 방법으로 찾을 수 있음.

최소값 찾기 문제

- 평균 시간 복잡도 = n-1 = O(n)
- 최악 시간 복잡도 = n-1 = O(n)
- 최대값 찾기 문제도 비슷한 방법으로 찾을 수 있음.

최소 최대값 동시 찾기 문제

```
void FindMinMax (int A[ ], int n, int *Minimum, int
*Maximum) {
   /* 입력 : A -n 개의 숫자가 저장되어 있는 배열
         n -배열 A에 저장되어 있는 숫자의 개수
     출력 : *Minimum -A에 저장된 값 중에서 최소값
         *Maximum -A에 저장된 값 중에서 최대값 */
   int  i ;
   *Minimum=A[0]; *Maximum=A[0];
   for (i = 1; i < n-1 ; i+=2) {                          //(n-1)/2
        if  (A[i] < A[i+1]) {Small= A[i]; Large= A[i+1];}    //1
        else {Small = A[i+1]; Large = A[i];}
         if   (Small < *Minimum) *Minimum = Small;           //1
        if   (Large > *Maximum) *Maximum = Large;            //1
   }
   return;
}
```

최소 최대값 동시 찾기 문제

- 최소값 찾기+최대값 찾기 시간 복잡도

- 최소·최대값 동시 찾기 FindMinMax 시간 복잡도

최소 최대값 동시 찾기 문제

- 최소값 찾기+최대값 찾기 시간 복잡도
  = 최소값 찾기 (n-1회 비교) + 최대값 찾기 (n-2회 비교, 최소값 제외)
  = 2n-3 = O(n)
- 최소·최대값 동시 찾기 FindMinMax 시간 복잡도
  ≅ (n-1)/2회 반복×3회 비교
  = 1.5n + 상수= O(n)

선택(Selection) 문제

- 임의로 나열된 n개의 데이터에서 크기가 i번째인 것을 찾는 문제
- 일반적인 경우-정렬 후 i번째 원소 선택
  → O(nlogn)
- 퀵 정렬의 Partition 함수를 이용, 분할정복 방법을 적용 - 퀵정렬의 Partition( ) 함수를 i번째 큰 원소가 나올 때까지 반복해 이용

  → 최악 $O(n^2)$

정렬 알고리즘 시간 복잡도 비교

- 비교기반 정렬 알고리즘

	Worst case	Average case
Selection sort	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$
Quick sort	$O(n^2)$	$O(n \log n)$
Merge sort	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$

선택(Selection) 문제

- 임의로 나열된 n개의 데이터에서 크기가 i번째인 것을 찾는 문제
- 일반적인 경우-정렬 후 i번째 원소 선택
  → O(nlogn)
- 퀵 정렬의 Partition 함수를 이용, 분할정복 방법을 적용 – 퀵정렬의 Partition( ) 함수를 i번째 큰 원소가 나올 때까지 반복해 이용

  → 최악 $O(n^2)$

선택(Selection) 문제

- 퀵 정렬의 분할함수

```
int Partition(int A[ ], int Left, int Right)
     /* 입력:  A[Left] - 분할 원소,              A[Right] - dummy
      출력: A[Left:Right], Right - index   */
 {        int PartElem, Value;
          PartElem  =  Left;
          Value  =  A[PartElem];                //분할원소 지정
          do
             do   while(A[++Left]  <  Value);    // 왼쪽에서 분할원소보다 큰 원소 찾기
             do   while(A[--Right]  >  Value);//오른쪽에서 분할원소보다 작은 원소 찾기
             if (Left  <  Right) Swap(&A[Left], &A[Right]);// 만나기 전이면 서로 교환
             else break;
          while (1);                             //왼쪽 검색과 오른쪽 검색이 만날때 까지 반복
          A[PartElem]  =  A[Right];              //분할원소 만난 자리에 넣기
          A[Right]  =  Value;
          return Right;
 }
```

퀵 정렬 (quick sort)

- 퀵 정렬

```
void QuickSort(int A[ ], int Left, int Right) //순환(recursive) algorithm
    /* 입력: A[Left:Right + 1], A[Right + 1]:  최대값보다 큰 값.
       출력: A[Left:Right],  A[Right + 1]: dummy */
  {     int k;                                   // position of partition
         if (Right  >  Left)
           k =  Partition(A[ ], Left, Right + 1);
            QuickSort(A[ ], Left, k  - 1);     //분할원소의 왼쪽 퀵정렬
            QuickSort(A[ ], k  +  1, Right); //분할원소의 오른쪽 퀵정렬
     }
```

선택(Selection) 문제

```
int Selection_N2(int A[ ], int n, int i) {
/* 입력: A[0..n-1] - 입력 데이터 배열
              i - 찾는 숫자의 순위
   출력: i번째 숫자의 값.   */
   int Left, Right, j;
   A[n] = Infinity; /* 무한대 값 - 더미 원소 */
   Left = 0; Right = n; //초기화
   while (1) {
        j = Partition(A, Left, Right);  //A를 분할 후 j 는 분할원소 들어간곳의 인덱스
           /* 0부터 시작, i 번째 원소 = j+1 */
        if ( i == j+1) return(A[j]); //i 번째 것 찾았으면 리턴
        if ( i < j+1) Right = j; //왼쪽 배열에 있으면 왼쪽 배열 분할을 위해 Right 인덱스 조절
        else Left = j+1; //오른쪽 배열에 있으면 오른쪽 배열 분할을 위해 Left 인덱스 조절
  }
}
```

선택(Selection) 문제

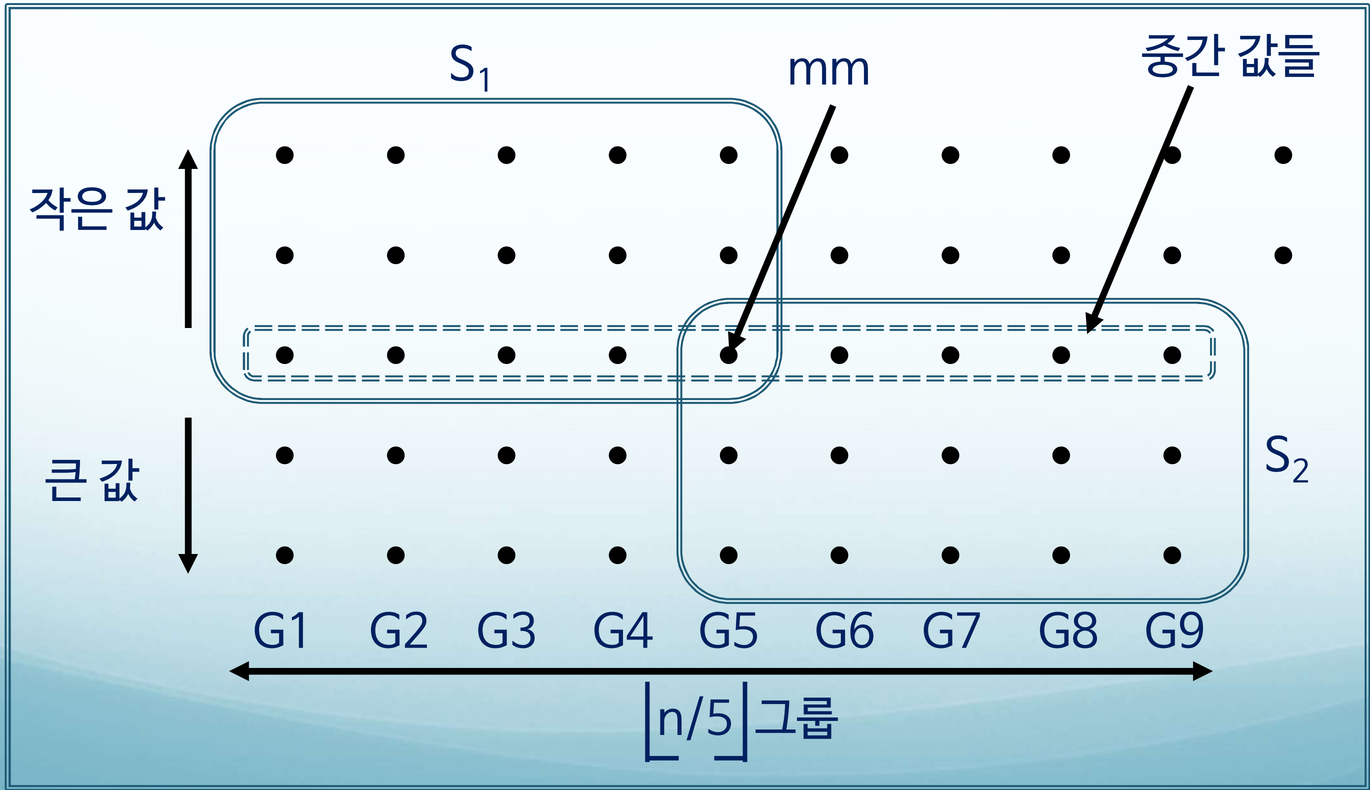
- 최악 시간 복잡도=O(n^2)
  - i=n이고 Partition() 호출 결과 j가 하나씩만 증가하는 경우(분할 원소를 제외한 나머지 전체로 부분배열이 형성되는 경우), 이 경우 while 루프가 n번 반복되며 반복 할 때마다 분할할 배열의 크기가 하나씩 줄어듦.
  - Partition()에 걸리는 시간은 O(m), $\sum_{m=1}^{n} O(m) = O(n^2)$

선택(Selection) 문제

- 최악의 경우 O(n) 인 알고리즘이 가능할까?
- Selection_N2의 최악의 경우는 Partition()에서 항상 부분배열이 하나만 남게 되는 경우이다.
- 이를 피하기 위해서는 항상 두 부분배열로 분할되며 그 비율이 일정 범위 이상임을 보이면 된다.

선택(Selection) 문제

- 해결책 : 항상 일정 비율의 두 부분배열로 Partition( ) 결과가 분할되도록 유도
- 배열의 n개의 원소를 5개의 원소로 구성된 $\lfloor n/5 \rfloor$개의 배열 그룹으로 나눔.
- 그룹마다 그룹 내에 있는 중간값을 찾음 -〉 $\lfloor n/5 \rfloor$개의 중간값.
- 그중에서 중간값(중간값들의 중간값) mm 을 배열을 분할하는 분할 원소로 사용



선택(Selection) 문제

입력데이터

10	58	18	3	29	2	90	26	34	53
11	23	21	4	52	1	36	27	33	57
15	55	22	5	80	14	45	30	88	
17	50	25	6	81	8	40	35	70	
20	12	28	9	89	7	24	60	71	
G_3	G_7	G_4	G_1	G_9	G_2	G_6	G_5	G_8	

선택(Selection) 문제

그룹 내 정렬후

mm

작은 값

10	12	18	3	29	1	24	26	33	53
11	23	21	4	52	2	36	27	34	57
15	50	22	5	80	7	40	30	70	
17	55	25	6	81	8	45	35	71	
20	58	28	9	89	14	90	60	88	
G_3	G_7	G_4	G_1	G_9	G_2	G_6	G_5	G_8	

중간 값들

큰 값

선택(Selection) 문제

- i 번째 원소가 mm 보다 작은 경우, S_2 는 i 번째 원소 후보대상에서 제외
- i 번째 원소가 mm 보다 큰 경우, S_1 는 i 번째 원소 후보대상에서 제외
- 즉, 예시의 경우 매 순환시마다 대략 $3\times\lceil\lceil n/5\rceil/2\rceil$개 원소 배제. ($\lfloor x\rfloor$는 x의 소수 부분 내림, $\lceil x\rceil$는 x의 소수 부분 올림)

선택(Selection) 문제

```
int SELECT (int A[ ], int i, int n) {
/* 배열 A[0.. n-1]에서 i 번째 큰 원소를 찾는다. */

[단계 1] n <= 5 경우는, 배열 A에서 i 번째 큰 원소를 찾아
         복귀시키고, n > 5인 경우는 [단계 2] ~ [단계 6]을 실행한다.
[단계 2] 배열 A를 5개의 원소로 구성된 ⌊n/5⌋개의 그룹으로 나누고,
         남은 원소는 무시한다.
[단계 3] ⌊n/5⌋개의 그룹에서 각각 중간 값을 찾아, 이를
         M={ m_1, m_2, ..., m_⌊n/5⌋}라고 하자.    //삽입정렬 이용
[단계 4] p = SELECT ( M , ⌈⌊n/5⌋/2⌉, ⌊n/5⌋ ) 라고 하자.    //순환호출
[단계 5] p를 분할원소로 하여 배열 A를 분할한다.
         분할 후에 p가 A[j]에 놓였다고 가정하자.
[단계 6]  (1) i=j+1인 경우는 p를 복귀시킨다.
          (2) i<j+1인 경우는 SELECT(A[0.. j-1], i, j)를 계산해 복귀
          (3) i>j인 경우는 SELECT(A[j+1..n-1], i-j-1, n-j-1)을 계산해 복귀
}
```

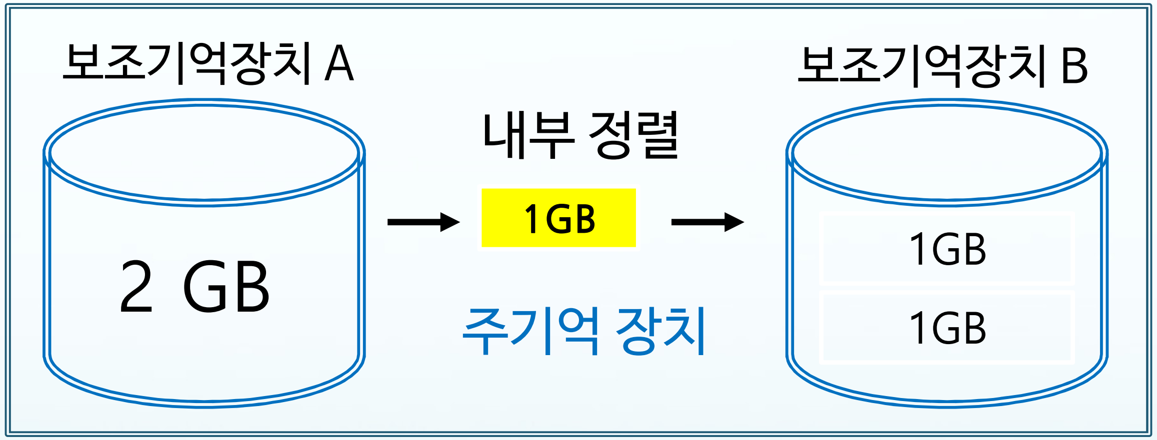
- 단계3: 5개 원소에 대한 삽입정렬은 O(1) 시간이 걸리는 것으로 볼 수 있고, 그룹은 O(n)개 이므로 O(n) 시간이 걸림
- 단계 4: 개 원소의 중간 값들 집합 M에서 중간 값, 즉 $\left\lceil \left\lfloor n/5 \right\rfloor /2 \right\rceil$ 번째 원소 찾기
- 단계 6: 매 순환시마다 대략 $3\times\left\lceil \left\lfloor n/5 \right\rfloor /2 \right\rceil$ 개 원소 배제
- 최악 시간 복잡도=O(n)

외부정렬

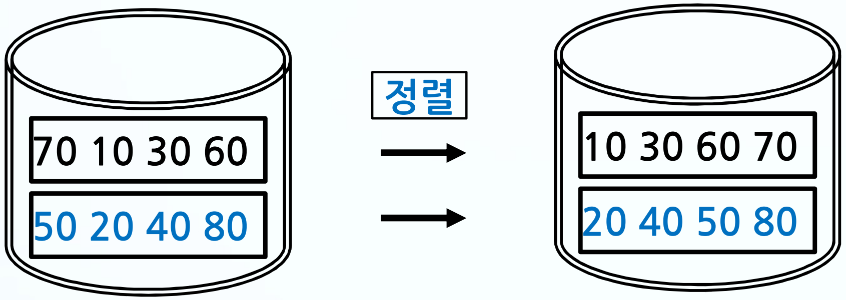
- 자료가 정렬 당시에 어디에 저장되어 있는가에 따라 정렬 알고리즘은 내부정렬(internal sort)와 외부 정렬(external sort)로 나뉘어짐.
- 내부정렬 – 정렬할 자료의 양이 적어서 자료 전체가 주기억장치에 저장될 수 있는 경우에 내부정렬을 사용하여 자료를 정렬함.
- 외부정렬
  - 자료의 양이 많을 때는 속도가 느리고 접근 방식이 제한적인 보조기억장치에 전체 자료를 두고 자료의 일부분을 한번에 조금씩 주기억장치에 옮겨와서 정렬을 함.
  - 자료가 주기억장치와 보조 기억장치 사이를 오고하는 데 드는 시간은 주기억장치에 저장되어 있는 자료들을 서로 비교하는 데 드는 시간보다 상대가 안될 정도로 길.
  - 외부 정렬의 실행시간에 크게 영향을 미치는 요인은 데이터간의 비교 횟수라기 보다는 데이터들이 주기억장치와 보조 기억장치 사이를 오고가는 횟수이다.
  - 보조 기억장치에 있는 데이터를 읽거나 쓰는 횟수가 외부 정렬의 실행시간을 결정하는 주요인이므로 이를 줄이는 방향으로 외부 정렬 알고리즘을 고안해야함.
- 디스크나 테이프 등 보조기억장치에 저장된 대량의 데이터를 정렬하는 방법
  - 주기억 장치의 용량 만큼 보조기억장치에서 데이터를 읽어 들여 정렬
  - 보조기억장치 특성에 따라 (Ex. 테이프) 데이터에 순차적 접근만 가능

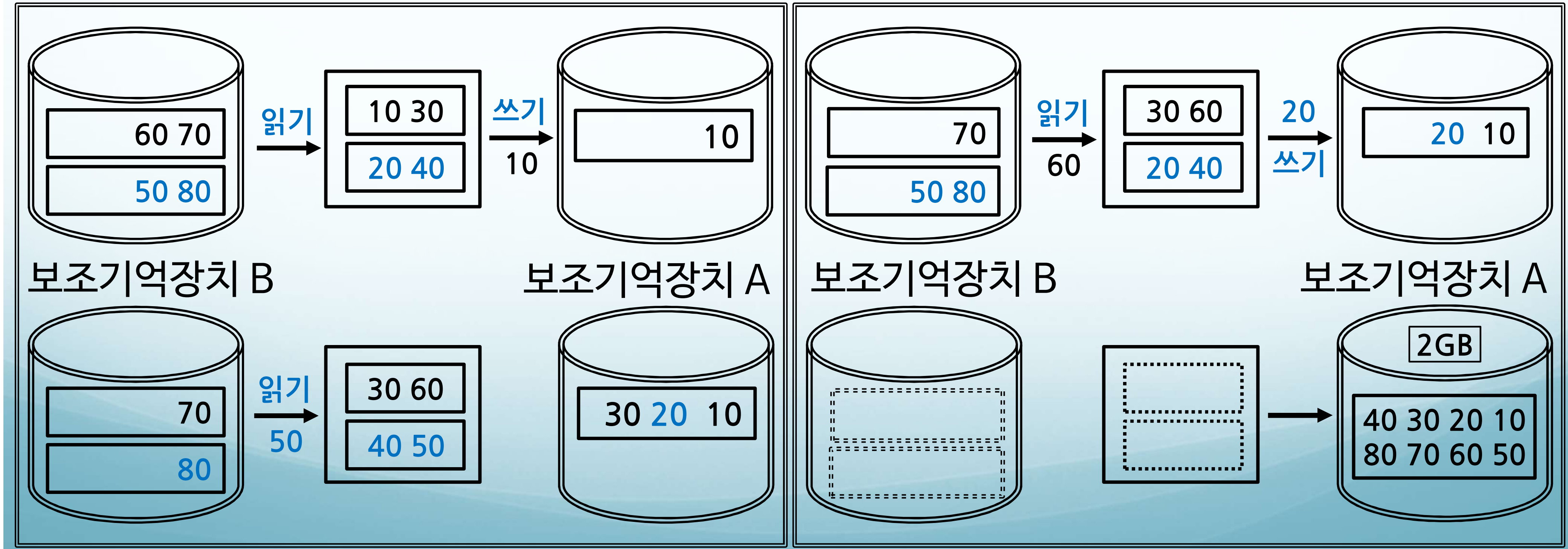
외부정렬

- Q. 주기억 장치의 크기가 1GB인 경우 보조기억장치 A에 있는
  2GB 크기의 블록을 어떻게 정렬시킬 수 있을까?



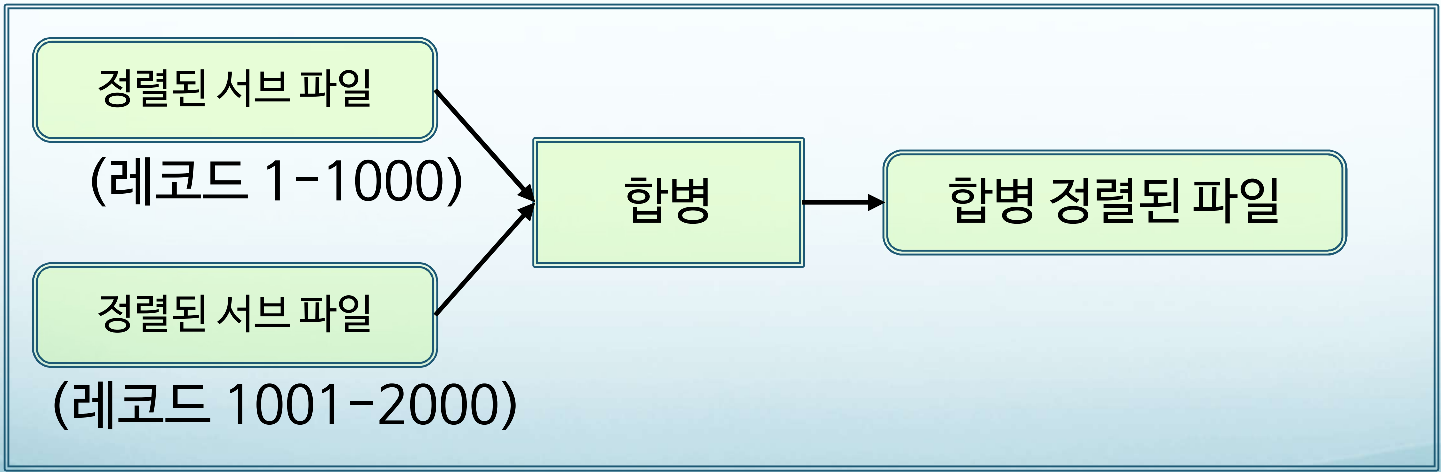
외부정렬


70 10 30 60
50 20 40 80
정렬
10 30 60 70
20 40 50 80


60 70
50 80
읽기
10 30
20 40
쓰기
10
10
보조기억장치 B
보조기억장치 A
70
80
읽기
50
30 60
40 50
30 20 10
70
50 80
읽기
60
30 60
20 40
20
쓰기
20 10
보조기억장치 B
보조기억장치 A
2GB
40 30 20 10
80 70 60 50

외부정렬

- 런(run) : 하나의 파일을 여러 개의 서브파일(subfile)로 나누어 내부 정렬 기법을 사용하여 정렬시킨 파일
- Ex. 2,000개의 레코드를 가진 파일을 정렬하는 문제에서 주기억 장치의 용량이1,000개의 레코드까지만 허용한다면, 2개의 런이 사용됨

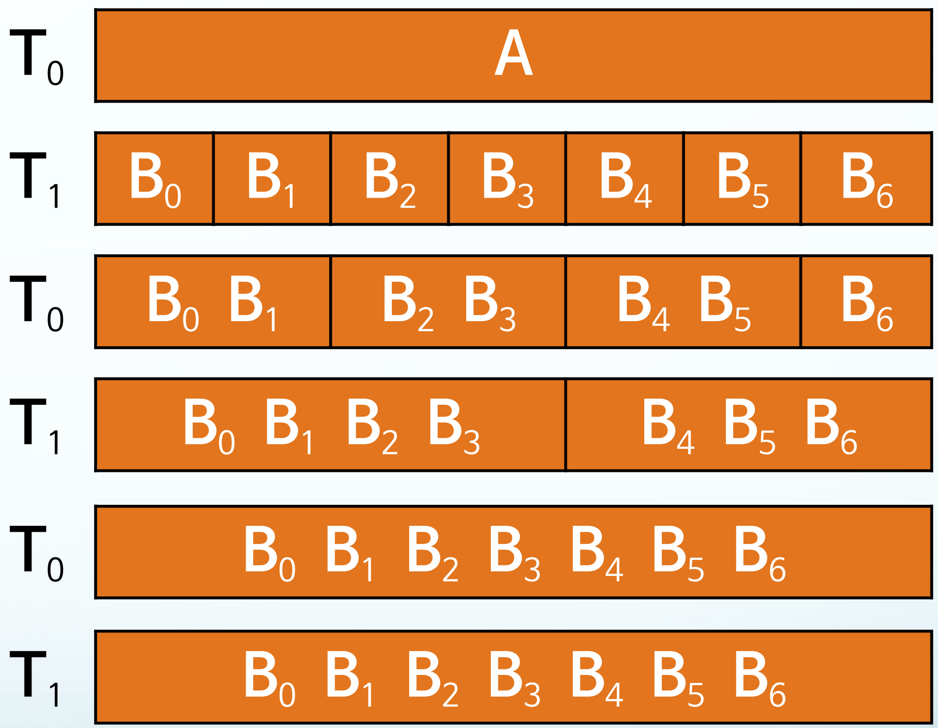


외부정렬

- 합병 정렬 알고리즘이 기본
  - 정렬 단계(sort phase)
    - ✓ 정렬할 파일의 레코드들을 지정된 길이의 서브파일로 분할하고, 이를 정렬하여 런(run)을 만든 뒤 입력 파일로 분배하는 단계
  - 합병 단계
    - ✓ 정렬된 런들을 합병해서 보다 큰 런으로 만들고, 이것들을 다시 입력 파일로 재분배하여 합병하는 방식으로 모든 레코드들이 하나의 런에 포함되도록 만드는 단계

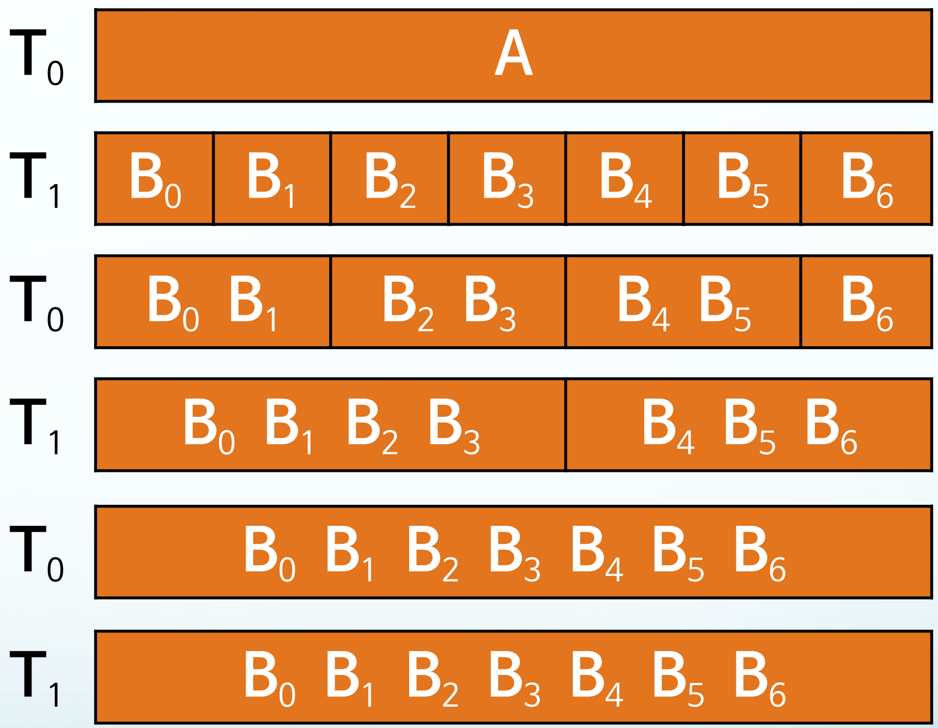
합병 정렬

- 보조기억장치에 있는 파일을 주기억장치에 옮겨올 수 있도록 작은 크기의 블록으로 나눔.
- 블록들을 한 개씩 주기억장치에 읽어 들여서 내부 정렬 시킨 후 정렬된 작은 블록들을 보조기억장치에 분산시켜 저장.
- 정렬된 블록들을 합병해나감. 합병할 수록 블록 수는 줄고 각 블록은 커지게 됨.
- 하나의 블록만 남게 되면 정렬이 완료.



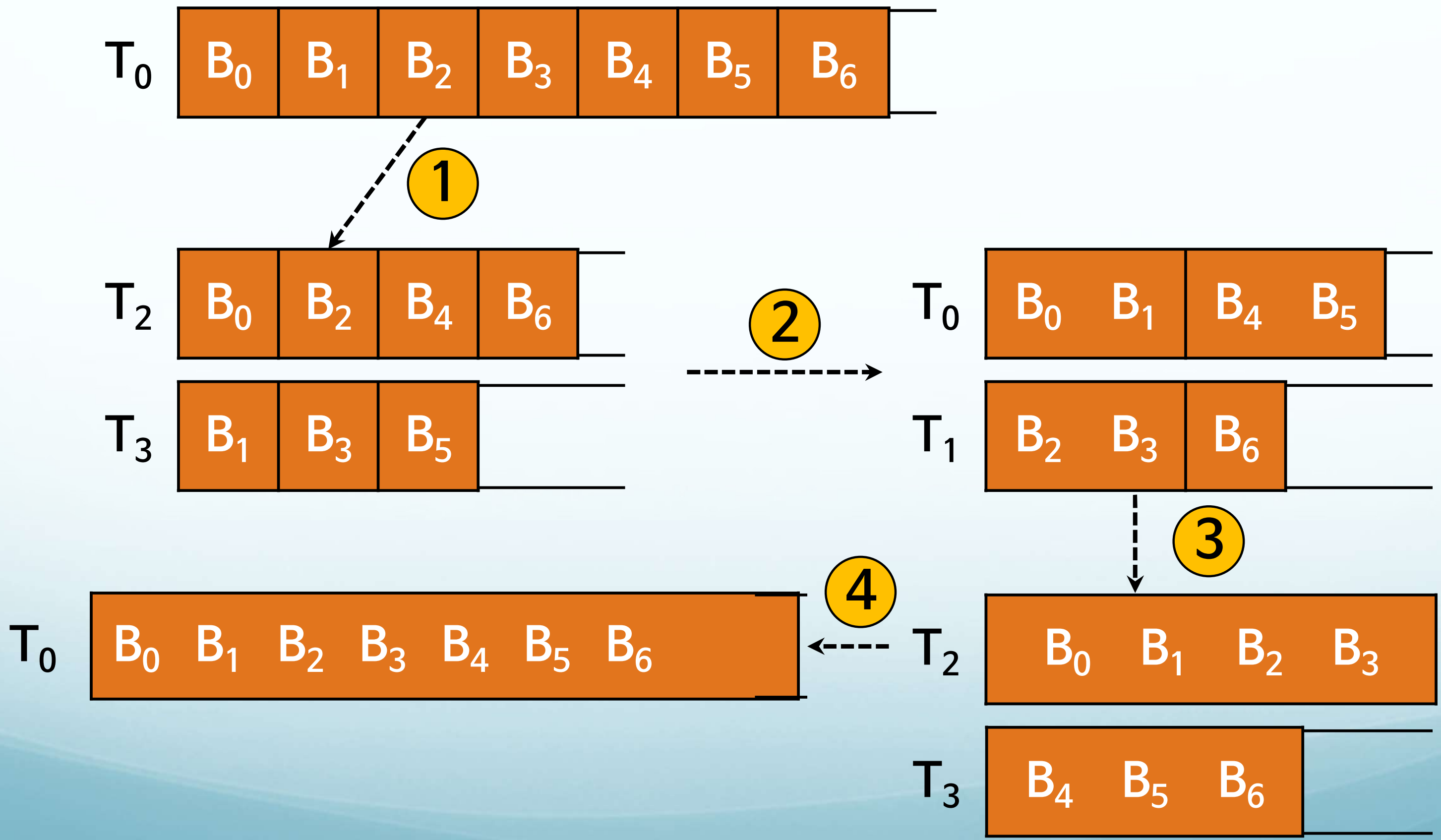
합병 정렬

- 입력 T_0 : 입력 데이터가 저장된 장치
  출력 T_1: 정렬된 데이터가 저장될 장치
  A: 미 정렬된 데이터
  B_i: 정렬된 런



균형적 다방향 합병정렬
Balanced Multiway Merge Sort

- 원소수: n, 주기억장치에 읽을 수 있는 수: b, 4개의 테이프 T_{0}, T_{1}, T_{2}, T_{3} 를 이용하여 정렬.



균형적 다방향 합병정렬
Balanced Multiway Merge Sort

- 각 원소의 테이프간 이동 횟수:
  n: 입력크기, b: 메모리크기, t: 사용한 테이프 수

$$\left\lceil \log_{t/2}(n/b) \right\rceil + 1$$

대치 선택
replacement selection

- 외부정렬 = 정렬단계 + 합병단계
  - 보조기억장치에 있는 파일을 정렬된 작은 블록으로 나누어 정렬 후 보조기억장치에 분산.
  - 두개씩 순환적으로 합병.

- 효율성 높이기 - 보조기억장치와 주기억장치 사이의 데이터 이동 횟수 최소화

  - 전략1: 런 하나 크기를 가능한 크게 만들어 전체 합병 횟수 줄이기 (정렬된 블록이 크면 클수록 합병 횟수가 줄게 된다.)
    - ✓ 정렬단계에서의 대치 선택 (Replacement Selection)

대치 선택
replacement selection

- 주기억장치에서 내부 정렬할 수 있는 블록의 최대 크기가 b 일 때, b 보다 큰 사이즈의 런(Run)을 만들어낼 수는 없을까?
  - 런을 만드는 과정에서 주기억장치에 항상 b개의 레코드를 상주시키고, 이를 최소값 힙(min heap) 구조로 운영
  - 마지막으로 출력된 키 값보다 큰 키 값을 가진 레코드가 출현할 경우 이를 별도로 마크(*, frozen)시키고, 마크된 레코드의 키 값은 ∞로 취급
  - 주기억장치 내 모든 레코드가 마크된 경우 새로운 런 생성 시작

대치 선택

replacement selection

2	3	10	9	12	5	7	6	11	8	2	4

(주기억장치 크기 b = 4 인 경우)

〈정렬된 블록을 만드는 과정〉

단계	생성된 블록(런)	주기억장치에 있는 키 (최소힙 구조로 저장됨)
0		{2, 3, 10, 9}
1	2	{3, 10, 9, 12}
2	2 3	{10, 9, 12, 5}
3	2 3 5	{10, 9, 12, 7}
4	2 3 5 7	{10, 9, 12, *6}
5	2 3 5 7 9	{10, 12, *6, 11}
6	2 3 5 7 9 10	{12, *6, 11, *8}
7	2 3 5 7 9 10 11	{12, *6, *8, *2}
8	2 3 5 7 9 10 11 12	{*6, *8, *2, *4}

6이 마지막 출력 원소보다 작으므로 다음에 만들어질 블록에 속하도록 마크

현재 블록에 속할 수 있는 레코드가 더 이상 없으면 블록 만들기를 마치고 다음 블록 시작.

대치 선택

- 런 생성 방법

  1. 입력 파일에서 버퍼로 b개 레코드 읽기

  2. 버퍼를 최소값 힙으로 재구성하고, 키 값이 가장 작은 레코드를 선택해 출력한 뒤 버퍼에서 삭제

  3. 입력 파일에서 다음 레코드를 읽어 버퍼에 포함시킨 뒤 출력된 레코드와 비교(모든 레코드가 동결되어 다음 레코드를 읽어 들일 공간이 없다면 단계 4로 진행)

     - if (입력 레코드의 키 값 < 출력된 레코드의 키 값) then 레코드에 "동결(frozen)" 마크 표시
     - 동결된 레코드는 단계 2의 선택에서 제외
     - 동결되지 않은 레코드는 단계 2로 돌아간다.

  4. 동결된 레코드들을 모두 동결 해제하고, 단계 2로 돌아가 새로운 런 생성

대치 선택

- 런의 평균 길이 = 2b

  정렬하고자 하는 파일의 레코드가 임의 순서로 나열되어 있고 또한 그러한 순서가 될 확률이 다른 순서가 될 확률과 동일하다면, 평균적으로 대치 선택의 블록 크기는 2b라는 것이 증명됨

다단계 합병 정렬
Polyphase Merge Sort

- 외부정렬 = 정렬단계 + 합병단계
  - 보조기억장치에 있는 파일을 정렬된 작은 블록으로 나누어 정렬 후 보조기억장치에 분산.
  - 두개씩 순환적으로 합병.
- 효율성 높이기 - 보조기억장치와 주기억장치 사이의 데이터 이동 횟수 최소화

➢ 전략1: 런 하나 크기를 가능한 크게 만들어 전체 합병 횟수 줄이기 (정렬된 블록이 크면 클수록 합병 횟수가 줄게 된다.)

✓ 정렬단계에서의 대치 선택 (Replacement Selection)

➢ 전략2: 합병 전 각 장치에 저장할 런 개수를 조정해 합병과정에서 만들어지는 불필요한 이동 및 복사횟수 줄이기

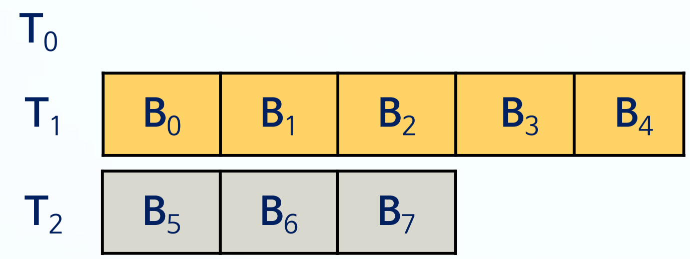
✓ 합병단계에서의 다단계 합병정렬 (Polyphase Merge Sort)

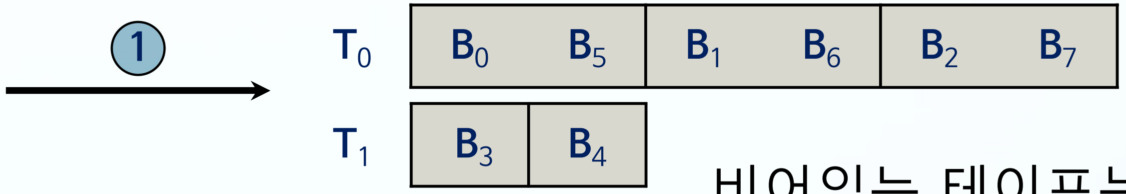
다단계 합병 정렬
Polyphase Merge Sort

- 합병 전 각 장치에 저장할 런 개수를 조정해 합병과정에서 만들어지는 불필요한 이동 및 복사회수를 줄일 수 있을까?
  - 합병 전 블록들을 t-1 개의 장치에 분산시키고 또한 각 장치가 갖는 블록의 수를 적절히 다르게 배분한다. (t-테이프 갯수)

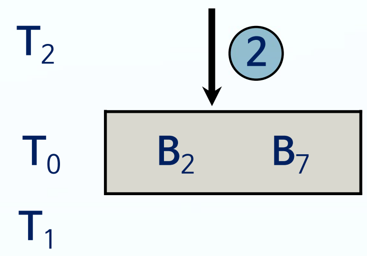
다단계 합병 정렬
Polyphase Merge Sort

테이프 3개를 사용하는 다단계 합병 정렬의 예

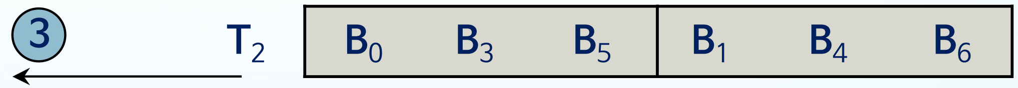


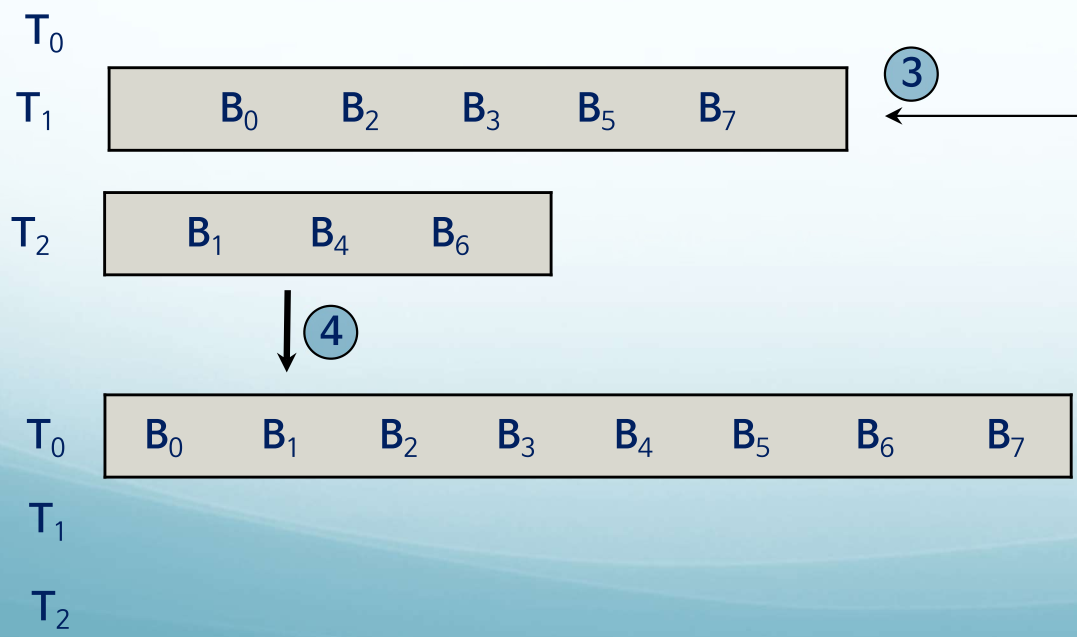


비어있는 테이프는
출력용으로 사용



시작시 T_1 에 정렬된 5개 런, T_2 에 3개의 런



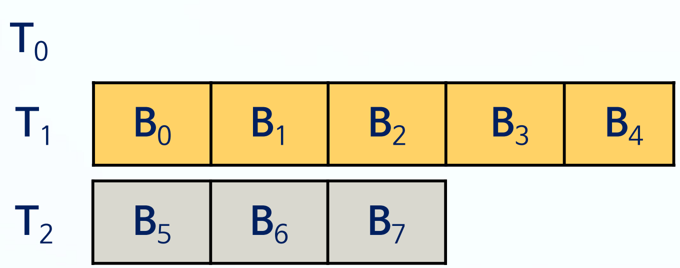


B_0 의 테이프간 이동 횟수 :
T_1->T_0->T_2->T_1->T_0
B_4 의 테이프간 이동 횟수 :
T_1->T_2->T_0

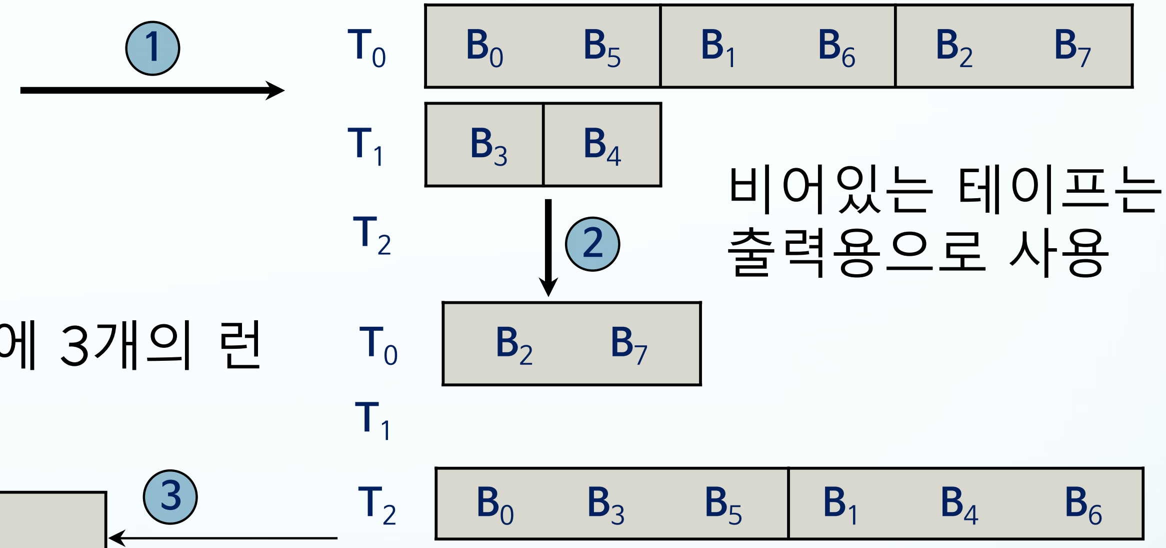
사용 테이프 수에 따라 각 블럭
이동 수가 달라짐

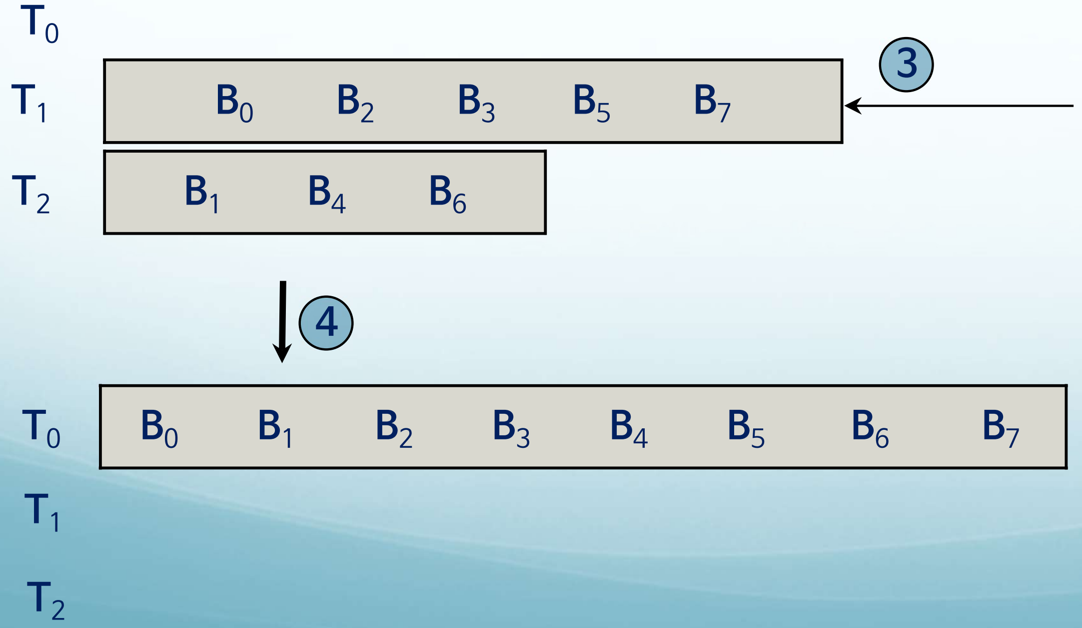
다단계 합병 정렬
Polyphase Merge Sort

테이프 3개를 사용하는 다단계 합병 정렬의 예



시작시 T_1 에 정렬된 5개 런, T_2 에 3개의 런





테이프 \ 단계	0	1	2	3	4
T_0	0	3	1	0	1
T_1	5	2	0	1	0
T_2	3	0	2	1	0

단계별 테이프의 블록 수

다단계 합병 정렬
Polyphase Merge Sort

- 블록 수 정하는 방식 – 피보나치(Fibonacci) 수열을 따름

$F_i = F_{i-1} + F_{i-2}$

$F_1 = 0, F_2 = 1, F_3 = 1, F_4 = 2, F_5 = 3, F_6 = 5, F_7 = 8, \cdots$

- 마지막으로 부터 i번째 합병 단계에서 비어있지 않은 두 테이프가 가지는 블록의 수는 F_i, $F_{i+1} = F_i + F_{i-1}$ 개임.

역순	5	4	3	2	1
단계 / 테이프	**0**	**1**	**2**	**3**	**4**
T_0	0	3	1	0	1
T_1	5	2	0	1	0
T_2	3	0	2	1	0

단계별 테이프의 블록 수

다단계 합병 정렬
Polyphase Merge Sort

- 블록 수 정하는 방식 – 피보나치(Fibonacci) 수열을 따름

$F_i = F_{i-1} + F_{i-2}$

$F_1 = 0, F_2 = 1, F_3 = 1, F_4 = 2, F_5 = 3, F_6 = 5, F_7 = 8, \cdots$

- 마지막으로 부터 i번째 합병 단계에서 비어있지 않은 두 테이프가 가지는 블록의 수는 $F_{i,}$, $F_{i+1} = F_i + F_{i-1}$ 개 임.

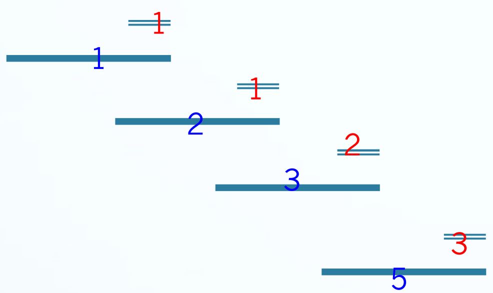
역순	5	4	3	2	1
단계 / 테이프	**1**	**2**	**3**	**4**	**5**
T_0	0	3 $F_4+F_3=F_5$	1 F_3	0	1
T_1	5 $F_5+F_4=F_6$	2 F_4	0	1 $F_2+F_1=F_3$	0
T_2	3 F_5	0	2 $F_3+F_2=F_4$	1 F_2	0

단계별 테이프의 블록 수

다단계 합병 정렬
Polyphase Merge Sort

- 블록 수 정하는 방식 – 피보나치(Fibonacci) 수열을 따름

0 1 1 2 3 5 8 13 ….



단계 / 테이프	1	2	3	4	5
T_0	0	3	1	0	1
T_1	5	2	0	1	0
T_2	3	0	2	1	0

다단계 합병 정렬

- 테이프 수가 4개 이상인 경우?
- $S_i = S_{i-1} + S_{i-2} + S_{i-3} + \cdots + S_{i-t+1}$, $i \geq 1$, (t: 테이프 수)

- 테이프 수 6개 예제

$F_1 = 0, F_2 = 0, F_3 = 0, F_4 = 0, F_5 = 1$

$F_i = F_{i-1} + F_{i-2} + F_{i-3} + F_{i-4} + F_{i-5}$, $i \geq 6$

0 0 0 0 1 1 2 4 8 16 31 61 …

역단계 / 테이프	9	8	7	6	5	4	3	2	1
T_0									1
T_1									0
T_2									0
T_3									0
T_4									0
T_5									0

- 마지막으로부터 i번째 단계에서 5개의 테이프가 가지는 블록 수

S_{i+4}

$S_{i+4} + S_{i+3}$

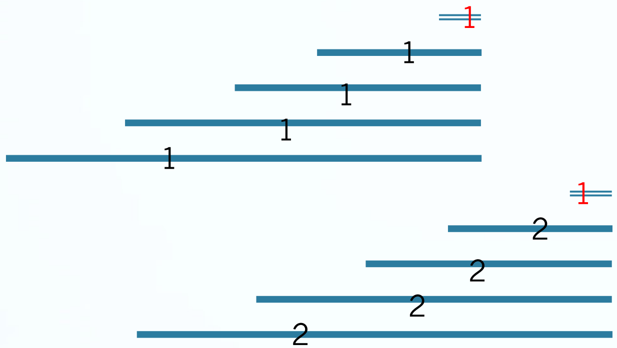
$S_{i+4} + S_{i+3} + S_{i+2}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1+} S_i$

다단계 합병 정렬

0 0 0 0 1 1 2 4 8 16 31 61 …



역단계 / 테이프	9	8	7	6	5	4	3	2	1
T_0							1	0	1
T_1							0	1	0
T_2							2	1	0
T_3							2	1	0
T_4							2	1	0
T_5							2	1	0

- 마지막으로부터 i번째 단계에서 5개의 테이프가 가지는 블록 수

S_{i+4}

$S_{i+4} + S_{i+3}$

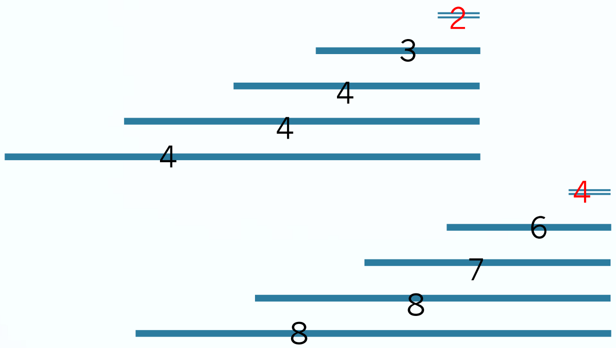
$S_{i+4} + S_{i+3} + S_{i+2}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1+}S_i$

다단계 합병 정렬

0 0 0 0 1 1 2 4 8 16 31 61 …



테이프 \ 역단계	9	8	7	6	5	4	3	2	1
T_0					7	3	1	0	1
T_1					6	2	0	1	0
T_2					4	0	2	1	0
T_3					0	4	2	1	0
T_4					8	4	2	1	0
T_5					8	4	2	1	0

- 마지막으로부터 i번째 단계에서 5개의 테이프가 가지는 블록 수

S_{i+4}

$S_{i+4} + S_{i+3}$

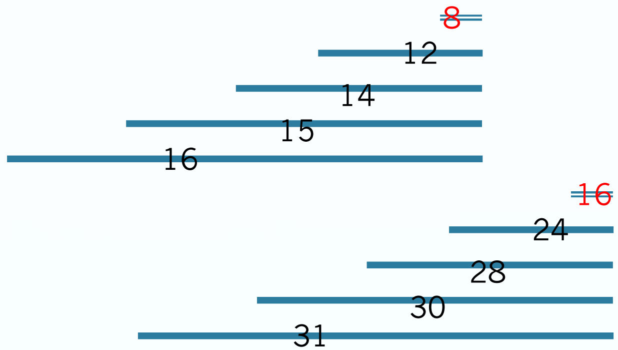
$S_{i+4} + S_{i+3} + S_{i+2}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1} + S_i$

다단계 합병 정렬

0 0 0 0 1 1 2 4 8 16 31 61 …



역단계 / 테이프	9	8	7	6	5	4	3	2	1
T_0			31	15	7	3	1	0	1
T_1			30	14	6	2	0	1	0
T_2			28	12	4	0	2	1	0
T_3			24	8	0	4	2	1	0
T_4			16	0	8	4	2	1	0
T_5			0	16	8	4	2	1	0

- 마지막으로부터 i번째 단계에서 5개의 테이프가 가지는 블록 수

S_{i+4}

$S_{i+4} + S_{i+3}$

$S_{i+4} + S_{i+3} + S_{i+2}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1+}S_i$

다단계 합병 정렬

- 테이프 수가 4개 이상인 경우?
- $S_i = S_{i-1} + S_{i-2} + S_{i-3} + \cdots + S_{i-t+1}, i \geq 1$, (t: 테이프 수)
- 테이프 수 6개 예제

$F_1 = 0, F_2 = 0, F_3 = 0, F_4 = 0, F_5 = 1$

$F_i = F_{i-1} + F_{i-2} + F_{i-3} + F_{i-4} + F_{i-5}$, $i \geq 6$

0 0 0 0 1 1 2 4 8 16 31 61 …

역단계 / 테이프	9	8	7	6	5	4	3	2	1
T_0	61	0	31	15	7	3	1	0	1
T_1	0	61	30	14	6	2	0	1	0
T_2	120	59	28	12	4	0	2	1	0
T_3	116	55	24	8	0	4	2	1	0
T_4	108	47	16	0	8	4	2	1	0
T_5	92	31	0	16	8	4	2	1	0

- 마지막으로부터 i번째 단계에서 5개의 테이프가 가지는 블록 수

S_{i+4}

$S_{i+4} + S_{i+3}$

$S_{i+4} + S_{i+3} + S_{i+2}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1}$

$S_{i+4} + S_{i+3} + S_{i+2} + S_{i+1} + S_i$

다단계 합병 정렬

- 테이프를 3개만 사용하면 균형적 다방향 합병정렬이 다단계 합병 정렬 보다 효율이 좋으나 테이프 수가 4, 5 개로 증가하면 균형적 다방향 합병 정렬보다 다단계 합병 정렬이 더 효율적인 정렬이 된다.
- 테이프를 t 개 사용하고 초기 블록의 개수가 x일 때, 균형적 다방향 합병 정렬에서는 각 블록이 $\log_{t/2} x+1$ 회만큼의 합병 단계를 거치게 된다. 다단계 합병 정렬의 경우는 평균적으로 테이프의 개수가 3일 때 $1.04\log_2 x+1$단계를 , 테이프가 5개일 때 $0.7\log_2 x+1$단계를 거친다. 이보다 더 좋은 성능을 얻기 위하여는 테이프의 개수를 상당히 많이 늘여야 한다.