< 정렬 알고리즘 1 >

선택정렬 (Selection Sort)

1. 주어진 리스트 중에 최소값을 찾는다.
2. 그 값을 맨 앞에 위치한 값과 교체한다. (패스(PASS))
3. 맨 처음 위치를 뺀 나머지 리스트를 같은 방법으로 교체한다.

패스	테이블	최솟값
0	9 1 6 8 4 3 2 0	0
1	0 1 6 8 4 3 2 9	1
2	0 1 6 8 4 3 2 9	2
3	0 1 2 8 4 3 6 9	3
4	0 1 2 3 4 8 6 9	4
5	0 1 2 3 4 8 6 9	6
6	0 1 2 3 4 6 8 9	8

$$\sum_{i=1}^{N-1} N-i = \frac{N(N-1)}{2} = O(n^2)$$

```
void selectionSort(int *list, const int n)
{
    int i, j, indexMin, temp;

    for (i = 0; i < n - 1; i++)
    {
        indexMin = i;
        for (j = i + 1; j < n; j++)
        {
            if (list[j] < list[indexMin])
            {
                indexMin = j;
            }
        }
        temp = list[indexMin];
        list[indexMin] = list[i];
        list[i] = temp;
    }
}
```

거품 정렬 (bubble Sort)

두 인접한 원소를 검사하여 정렬하는 방법이다.

55, 07, 78, 12, 42 초기값
07, 55, 78, 12, 42 첫 번째 패스
07, 55, 78, 12, 42
07, 55, 12, 78, 42
07, 55, 12, 42, 78 두 번째 패스
07, 55, 12, 42, 78
07, 12, 55, 42, 78
07, 12, 42, 55, 78 세 번째 패스
07, 12, 42, 55, 78 네 번째 패스
07, 12, 42, 55, 78 다섯 번째 패스
07, 12, 42, 55, 78 여섯 번째 패스
07, 12, 42, 55, 78 일곱 번째 패스 정렬 끝

```
int* bubble_sort(int arr[], int n) {
    int i, j, temp;
    for (i=n-1; i>0; i--) {
        for (j=0; j<i; j++) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    return arr;

}
```

삽입 정렬 (Insertion Sort)

자료 배열의 모든 요소를 앞에서부터 차례대로 이미 정렬된 배열 부분과 비교하여, 자신의 위치를 찾아 삽입함으로써 정렬을 완성하는 알고리즘이다.

31, 25, 12, 22, 11 — 처음 상태
31, 25, 12, 22, 11 — 두 번째 원소를 부분 리스트에서 적절한 위치에 삽입한다.
25, 31, 12, 22, 11 — 세 번째 원소를 부분 리스트에서 적절한 위치에 삽입한다.
12, 25, 31, 22, 11 — 네 번째 원소를 부분 리스트에서 적절한 위치에 삽입한다.
12, 22, 25, 31, 11 — 마지막 원소를 부분 리스트에서 적절한 위치에 삽입한다.
11, 12, 22, 25, 31 — 종료

```
void insertion_sort ( int *data, int n )
{
  int i, j, remember;
  for ( i = 1; i < n; i++ )
  {
    remember = data[(j=i)];
    while ( --j >= 0 && remember < data[j] ){
        data[j+1] = data[j];
        data[j] = remember;
    }
  }
}
```

퀵 정렬 (Quick Sort)

1. 리스트 가운데서 하나의 원소를 고른다. 이렇게 고른 원소를 피벗(Pivot)이라고 한다.
2. 피벗 앞에는 피벗보다 작은 모든 원소들이 오고, 피벗 뒤에는 피벗 보다 값이 큰 모든 원소들이 오도록 피벗을 기준으로 리스트를 둘로 나눈다.
   이렇게 리스트를 둘로 나누는 것을 분할이라고 한다. 분할을 마친 뒤에 피벗은 더 이상 움직이지 않는다.
3. 분할된 두 개의 작은 리스트에 대해 재귀(Recursion)적으로 이 과정을 반복한다. 재귀는 리스트의 크기가 0이나 1이 될 때까지 반복된다.

재귀 호출이 한 번 진행될 때마다 최소한 하나의 원소는 최종적으로 위치가 정해지므로, 이 알고리즘은 반드시 끝난다는 것을 보장할 수 있다.

5, 3, 7, 6, 2, 1, 4 — 처음 상태. P= 피벗(Pivot)
(P: 4)

5, 3, 7, 6, 2, 1, 4 — i 값이 피벗 값보다 크고, j 값은 피벗 값보다 작으므로 둘을 교환.
(i: 5, j: 1, P: 4)

1, 3, 7, 6, 2, 5, 4
(i: 1, j: 5, P: 4)

1, 3, 7, 6, 2, 5, 4 — 둘 다 피벗보다 작으므로 교환하지 않음.
(i: 3, j: 2, P: 4)

1, 3, 7, 6, 2, 5, 4 — i 값이 피벗 값보다 크고, j 값은 피벗 값보다 작으므로 둘을 교환.
(i: 7, j: 2, P: 4)

1, 3, 2, 6, 7, 5, 4
(i: 7, j: 2, P: 4)

1, 3, 2, 4, 7, 5, 6 — 초록색 그룹에서 다시 해줌(재귀)
(i: 2, P: 4, j: 7)

```
void quickSort(int arr[], int left, int right) {
      int i = left, j = right;
      int pivot = arr[(left + right) / 2];
      int temp;
      do
      {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i<= j)
        {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j--;
        }
      } while (i<= j);

    /* recursion */
    if (left < j)
        quickSort(arr, left, j);

    if (i < right)
        quickSort(arr, i, right);
}
```

힙 정렬 (heap Sort)

최대 힙 트리나 최소 힙 트리를 구성해 정렬을 하는 방법으로 내림차순 정렬을 위해서는 최대 힙을 구성하고, 오름차순 정렬을 위해서는 최소 힙을 구성하면 된다.

1. n 개의 노드에 대한 완전 이진 트리를 구성한다. 이때 루트노드부터 부모 노드, 왼쪽 자식노드, 오른쪽 자식 노드 순으로 구성한다.
2. 최대 힙을 구성한다. 최대 힙이란 부모노드가 자식노드보다 큰 트리를 말하는데, 단말노드를 자식 노드로 가진 부모노드부터 구성하며 아래 부터 루트 까지 올라오며 순차적으로 만들어 갈 수 있다.
3. 가장 큰 수(루트에 위치)를 가장 작은 수와 교환한다.
4. 2와 3을 반복한다.

힙	새로 추가된 요소	요소 교체
null	6	
6	5	
6, 5	3	
6, 5, 3	1	
6, 5, 3, 1	8	
6, **5**, 3, 1, **8**		5, 8
6, **8**, 3, 1, 5		6, 8
8, 6, 3, 1, 5	7	
8, 6, **3**, 1, 5, **7**		3, 7
8, 6, 7, 1, 5, 3	2	
8, 6, 7, 1, 5, 3, 2	4	
8, 6, 7, **1**, 5, 3, 2, **4**		1, 4
8, 6, 7, 4, 5, 3, 2, 1		

힙	요소 교체	요소 삭제	요소 정렬
8, 6, 7, 4, 5, 3, 2, **1**	8, 1		
1, 6, 7, 4, 5, 3, 2, **8**		8	
1, 6, **7**, 4, 5, 3, 2	1, 7		8
7, 6, **1**, 4, 5, **3**, 2	1, 3		8
7, 6, 3, 4, 5, 1, **2**	7, 2		8
2, 6, 3, 4, 5, 1, **7**		7	8
2, **6**, 3, 4, 5, 1	2, 6		7, 8
6, **2**, 3, 4, **5**, 1	2, 5		7, 8
6, 5, 3, 4, 2, **1**	6, 1		7, 8
1, 5, 3, 4, 2, **6**		6	7, 8
1, **5**, 3, 4, 2	1, 5		6, 7, 8
5, **1**, 3, **4**, 2	1, 4		6, 7, 8
5, 4, 3, 1, **2**	5, 2		6, 7, 8
2, 4, 3, 1, **5**		5	6, 7, 8
2, **4**, 3, 1	2, 4		5, 6, 7, 8
4, 2, 3, **1**	4, 1		5, 6, 7, 8
1, 2, 3, **4**		4	5, 6, 7, 8
1, 2, **3**	1, 3		4, 5, 6, 7, 8
3, 2, **1**	3, 1		4, 5, 6, 7, 8
1, 2, **3**		3	4, 5, 6, 7, 8
1, **2**	1, 2		3, 4, 5, 6, 7, 8
2, **1**	2, 1		3, 4, 5, 6, 7, 8
1, **2**		2	3, 4, 5, 6, 7, 8
1		1	2, 3, 4, 5, 6, 7, 8
			1, 2, 3, 4, 5, 6, 7, 8

```
void downheap(int cur, int k)
{
  int left, right, p;
    while(cur < k) {
      left = cur * 2 + 1;
      right = cur * 2 + 2;

      if (left >= k && right >= k) break;

      p = cur;
      if (left < k && data[p] < data[left]) {
        p = left;
      }
      if (right < k && data[p] < data[right]) {
        p = right;
      }
      if (p == cur) break;

      swap(&data[cur],&data[p]);
      cur=p;
    }
}

void heapify(int n)
{
  int i,p;
  for(i = (n-1)/2; i >= 0; i--){
    downheap(i,n);
  }
  //for(i=0;i<size;++i)printf("%d ",data[i]);
  //printf("\n");
}

void heap()
{
  int k;
  heapify(size);
  for(k = size-1; k > 0; ){
    swap(&data[0],&data[k]);
    //k--;
    downheap(0,k);
    k--;
  }
}
```

합병 정렬 (Merge Sort)

2개 이상의 자료를 오름차순이나 내림차순으로 재배열하는 것이다. 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 정렬하는 것으로써 부분집합으로 분할(devide)하고, 각 부분집합에 대해서 정렬 작업을 완성(conquer)한 후에 정렬된 부분 집합들을 다시 결합(combine)하는 분할 정복(divide and conquer) 기법을 사용한다.

n-way 합병 정렬의 개념은 다음과 같다.

1. 정렬되지 않은 리스트를 각각 하나의 원소만 포함하는 n개의 부분리스트로 분할한다.
2. 부분리스트가 하나만 남을 때 까지 반복해서 병합하며 정렬된 부분리스트를 생성한다. 마지막에 남은 부분리스트가 정렬된 리스트이다.

흔히 쓰는 하향식 2-way 합병 정렬은 다음과 같이 작동한다.

1. 리스트의 길이가 1이하이면 이미 정렬된 것으로 본다. 그렇지 않은 경우에는
2. 분할(divide) : 정렬되지 않은 리스트를 절반으로 잘라 비슷한 크기의 두 부분 리스트로 나눈다.
3. 정복(conquer) : 두 부분 리스트를 합병 정렬을 이용해 정렬한다.
4. 결합(combine) : 두 부분 리스트를 다시 하나의 정렬된 리스트로 합병한다. 이때 정렬 결과가 임시 배열에 저장된다.
5. 복사(copy) : 임시 배열에 저장된 결과를 원래 배열에 복사한다.

```
/// merge sort range : [low ~ high]
void mergeSort(int A[], int low, int high, int B[]){
    // 1. base condition
    if(low >= high) return;

    // 2. divide
    int mid = (low + high) / 2;

    // 3. conquer
    mergeSort(A, low, mid, B);
    mergeSort(A, mid+1, high, B);

    // 4. combine
    int i=low, j=mid+1, k=low;
    for(;k<=high;++k){
        if(j > high ) B[k] = A[i++];
        else if(i > mid) B[k] = A[j++];
        else if(A[i] <= A[j]) B[k] = A[i++];
        else B[k] = A[j++];
    }

    // 5. copy
    for(i=low;i<=high;++i) A[i] = B[i];
}
```

```
// array A[] has the items to sort; array B[] is a work array
void BottomUpMergeSort(A[], B[], n)
{
    // Each 1-element run in A is already "sorted".
    // Make successively longer sorted runs of length 2, 4, 8, 16... until whole array is sorted.
    for (width = 1; width < n; width = 2 * width)
    {
        // Array A is full of runs of length width.
        for (i = 0; i < n; i = i + 2 * width)
        {
            // Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[]
            // or copy A[i:n-1] to B[] ( if(i+width >= n) )
            BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B);
        }
        // Now work array B is full of runs of length 2*width.
        // Copy array B to array A for next iteration.
        // A more efficient implementation would swap the roles of A and B.
        CopyArray(B, A, n);
        // Now array A is full of runs of length 2*width.
    }
}

//  Left run is A[iLeft :iRight-1].
// Right run is A[iRight:iEnd-1  ].
void BottomUpMerge(A[], iLeft, iRight, iEnd, B[])
{
    i = iLeft, j = iRight;
    // While there are elements in the left or right runs...
    for (k = iLeft; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iRight && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}

void CopyArray(B[], A[], n)
{
    for(i = 0; i < n; i++)
        A[i] = B[i];
}
```