< 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);
}
```