Sorting Algorithm

Outline

1. Introduction

- Sort : 정렬 -> 기존에 있던 데이터들을 다루는 것에 중점을 둔다.

```
data  : 741953
index : 123456 -> 531642
1. 데이터 자체가 아닌 index를 변경하는 경우가 많다.
2. 데이터 자체를 변경하는 경우와 index를 변경하는 경우 서로 구분해서 생각해보자.
```

- stability :
- sorting order :
- sort
  - selection : 위치를 하나 지정(S). 현재 위치 포함하여 이후에 있는 항들 중 가장 작은 값과 S의 값를 바꾼다.
  - insertion : 지정된 위치의 양 옆에 있는 값들과 비교, 자리를 재배치. 주로 index
  - bubble : 몰라도 됨.
  - shell : 일정 간격(F)마다 selection sort를 수행한다. 이후 F값을 줄여 재수행한다.

```
src : 165879324568

1. 1658|7932|4568 -> 1,7,4 비교. 6,9,5 비교. 5,3,6 비교. 8,2,8 비교.
    - 1532|4658|7968
2. 153|246|587|968 -> 1,2,5,9 비교. 5,4,8,6 비교. 3,6,7,8 비교.
    - 143|256|567|988
3. 14|32|56|56|79|88 -> 1,3,5,5,7,8 비교. 4,2,6,6,9,8 비교.
    - 12|34|56|56|78|89
4. 1|2|3|4|5|6|5|6|7|8|8|9 -> 모든 항을 비교
    - 1|2|3|4|5|5|6|6|7|8|8|9
```

```
result : 123455667889
```

- merge :
- heap : queue의 한 종류인 heap을 사용.
- radix :
- quick :

2. Selection Sort

Pseudo Code

1. j = 0 (start index)
2. if j >= N-1 then stop
3. Find the minimum from j to N-1 (store it to the min)
4. exchange j and min
5. j ++ and go to the step 2

```
"945139"
"145939"  ->  "135949"  ->  "134959"  ->  "134599"
```

- 위치 하나를 선정하고, 그 이후에 있는 항들 중 가장 작은(혹은 가장 큰) 값과 선택한 위치를 바꾼다 생각하면 편하다.

code

```
void select_sort(int *a, int N) {
    int min;
    // 가장 작은 값
    int min_idx;
    // 가장 작은 값의 인덱스 번호
    int x, y;
    // 2중 loop를 구현하기 위해 사용하는 변수 두가지

    for (y = 0; y < N - 1; y++) {
    // N-1번째 항의 경우, 뒤에 항이 없으므로 이후 항들 중 최하값이 자기 자신이 될수밖에 없다.
        min_idx = y;
        // 다음 항 중 아무 값도 참조하지 않은 상태이므로 min_index가 y가 된다.
        min = a[y];
        // 같은 맥락으로 가장 작은 값은 a[y]가 됨
        tmp = a[y];
        // min값은 이후 탐색 루프를 돌다 없어질 가능성이 높으므로 후에 위치를 바꿀때를 대비해 하나 저장해둠.
        for (x = y + 1; x < N; x++) {
        // 이후 존재하는 나머지 항들에 대해 가장 작은 값을 가져간다.
            if (min > a[x]) {
                min = a[x];
```

```
                min_idx = x;
            }
        }
        a[min_idx] = a[y];
        a[y] = min;
        // 선택된 값(a[y])와 이후 항들 중 가장 작은 값(a[min_idx])의 위치를 바꿔준다.
    }
}
```

big-O

- 명령어의 개수를 비교하는 방식이라 보면 된다.
- selection sort의 경우, reverse 상태(654321)일때 모든 비교에 대해 교환을 수행해야하므로 가장 복잡도가 높음
- 다만 최고차수만 따진다. 그 외의 요인은 코딩 능력에 달려있다 봄

w/ void pointer and function pointer

- 나중에 배열과 void 포인터 활용한 것을 서로 비교할것

code

```
void gen_select_sort(void *base, size_t nelem, size_t width,
int (*fcmp)(const void *, const void *))
// base <-> a, nelem <-> N
// width : 자료형의 크기
// fcmp : 지정된 메모리 위치 내의 크기를 특정 상수와 비교하는 함수. intcmp의 포인터를
받는다.
{
    void *min;
    // min : 최소값. 다만 메모리 사용량을 줄이기 위해 포인터로 선언
    int min_idx;
    // 마찬가지로 최소값의 index 번호
    int x, y;
    //루프를 돌때 사용하는 변수
    min = malloc(width);
    // min에 자료형의 크기만큼 할당해줌
    for(y = 0; y<nelem-1; y++)
    // 마지막에서 두번째 항까지
    {
        min_idx = y;
        memcpy(min, (char *)base+y*width, width);
        // a[y] = (char*)base + y*width,width
        // y를 가장 작은 값으로 설정한다.
        for(x = y+1; x<nelem; x++)
        // y 다음항부터 끝까지
        {
            if(fcmp(min, (char *)base + x*width))
            // min과 a[x]의 값을 비교한다.
            {
```

```
                memcpy(min, (char *)base + x*width, width);
                // 만약 결과가 0보다 크면 min값을 변경
                min_idx = x;
                // 최소값의 인덱스값도 변경
            }
        }
        memcpy((char*)base + y*width,width, (char *)base + min_idx*width, width);
        memcpy((char *)base + x*width, min, width);
        // 위치를 서로 바꿔준다.
    }
    free(t);
}

int intcmp(const void *a, const void *b)
{
        return (*(int *)a > *(int *)b);
}
```

3. Insertion Sort

- Pseudo code
    1. i = 1;
    2. j = i; t = a[i];
    3. while a[j-1] > t && j > 0 -> a[j] = a[j-1]; j--;
    4. a[j] = t;
    5. i++ and go to the step 2)
- 평균적으론 더 낫다.
- worst case(역순)일땐 selection sort 보단 안좋음

```
default
9451399 -> 4951399 -> 4591399 -> 1459399 -> 1345999
```

code

```
void insert_sort(int *a, int N) {
    int i, j, t;
    // i -> 루프를 돌때 사용, j -> min_index 저장할때 사용, t -> a[i]를 임시 저장할때
사용.
    for (i = 1; i < N; i++) {
        t = a[i];
        j = i;
        // 초기값을 min_value라 가정, index와 value를 저장한다.
        while (j > 0 && a[j - 1] > t) {
        // j가 0보다 크고, a[j-1]이 기준 값보다 크다면
            a[j] = a[j - 1];
            j--;
            // index를 뒤로 한칸씩 밀고 비교 대상을 바꾼다. 이때, 해당 구문을 한번이라
도 실행하면 a[i]는 덮여서 사라진다.
```

```
            }
            a[j] = t;
            // 탐색된 위치에 이전에 저장해뒀던 a[i]를 대입한다.
        }
    }
```

w/ void pointer and function pointer

```
void insert_sort(void *a, size_t nelem, size_t width, int(*fcmp)(const void *,
const void *)) {
        void *t;
        int i, j;
        t = malloc(width);
        for (i = 1; i < nelem; i++) {
                memcpy(t, (char*)a + i * width, width);
                j = i;
                // 초기값을 min_value라 가정, index와 value를 저장한다.
                while (j > 0 && fcmp((char*)a + (j - 1)*width,t)) {
                        memcpy((char*)a + j * width, (char*)a + (j - 1)*width,
width);
                        j--;
                }
                memcpy((char*)a + j * width, (char*)t, width);
        }
        free(t);
}

int intcmp(const void *a, const void *b)
{
        return (*(int *)a > *(int *)b);
}
```

Optimization

- 원본을 위한 배열을 하나 더 만든다.
- 원본 자체를 sorting 해버리면 원본이 사라짐. 따라서 index라는 값을 통해 간접적으로 원본의 값을 참조한다.

```
void insert_sort(int *a, int N) {
    int i, j, t;
    for (i = 0; i < N; i++)
        index[i] = i;
    for (i = 1; i < N; i++)
    {
        t = index[i];
        j = i;
        while (j > 0 && a[index[j-1]] > a[t])
        {
```

```
                index[j] = index[j - 1];
                j--;
            }
            index[j] = t;
        }
    }
```

```
default : 9451399
value : 9451399 -> 9451399 -> 9451399 -> 9451399 -> 9451399 -> 9451399 ->
9451399
index : XXXXXXX -> 10XXXXX -> 120XXXX -> 3120XXX -> 34120XX -> 341205X ->
3412056
        0123456 -> 1023456 -> 1203456 -> 3120456 -> 3412056 -> 3412056 ->
3412056
```

w/ void pointer and function pointer

```
void op_insert_sort(void *a, size_t nelem, size_t width, int(*fcmp)(const void *,
const void *)) {
        int i, j;
        void *t = malloc(width);
        for (i = 0; i < nelem; i++)
                index[i] = i;
        for (i = 1; i < nelem; i++)
        {
                memcpy(t, (char*)index + i * width, width);
                j = i;
                while (j > 0 && fcmp((char*)a + *((index) + (j - 1))*width,
(char*)a + (*(int*)t) * width))
                {
                        memcpy((char*)index + j * width, (char*)index + (j - 1) *
width, width);
                        j--;
                }
                memcpy((char*)index + j * width, (char*)t, width);
        }
}

int intcmp(const void *a, const void *b)
{
        return (*(int *)a > *(int *)b);
}
```

4. Shell Sort

- 데이터를 일정 간격으로 비교하여 각각 정렬, 이후 간격을 줄여 재정렬한다.

- interval : 얼마의 간격을 두고 데이터를 자를 것인가?

code

```
void shell_sort(int a*, int N)
{
    int i, j, k, h, v;
    for(h = N/2; h>0; h /= 2){
        for(i = 0; i<h; i++){
            for(j = i+h; j<N; j += h){
                v = a[j];
                k = j;
                while(k > h-1 && a[k-h] > v){
                    a[k] = a[k-h];
                k -= h;
                }
                a[k] = v;
            }
        }
    }
}
```

w/ void pointer and function pointer

```
void shell_sort(int *a, size_t nelem, size_t width, int(*fcmp)(const void *, const
void *))
{
    int i, j, k, h;
    void *v = malloc(width);
    for(h = N/2; h>0; h /= 2){
        for(i = 0; i<h; i++){
            for(j = i+h; j<N; j += h){
                memcpy(v,(char*)a + j*width);
                k = j;
                while(k > h-1 && fcmp((char*)a + (k-h)*width, (char*)v)){
                    a[k] = a[k-h];
                    k -= h;
                }
                memcpy((char*)a + k*width, v, width);
            }
        }
    }
}
```

Optimization

- interval의 변화를 변경한다
- h_cur = 3*h_next + 1

5. Trouble Shooting

1. Generalized ver.을 사용해도 이전 버전보다 더 느린 경우가 있다.
    - 함수 호출이 이전보다 더 많아져 size가 더 작은 data면 그럴 수도 있음
    - 일반적으로 generalized ver.은 하나의 함수로 모든 데이터 타입에 대하여 활용할 수 있다는 점에서 메리트가 있다.

6. Quick Sort

- O(N*logN)
- resucsive!
- devide & conquer 개념이 매우 잘 녹아있는 방법
- Pseudo code :
    1. If n>1,
        1. Partition(divide) array a[] whose size is N, and Exchange the positiion of pivot to "mid"
        2. Quick Sort(a,mid);
        3. Quick Sort(a+mid+), N-mid-1
- There ar tow index, i and j for moving left -> right and right -> left.
- If i >= j, then exchange the pos. of pivot to i.

```
src   : 321568756

pivot : 6
1. i = 3 : i < pivot -> 문제 없음
   j = 5 : j < pivot -> 이후 바꿔야됨
2. i = 2 : i < pivot -> 문제 없음
3. i = 1 : i < pivot -> 문제 없음
4. i = 5 : i < pivot -> 문제 없음
5. i = 6 : i = pivot -> 문제 없음
6. i = 8 : i > pivot -> 바꿔야됨
    - i와 j의 위치를 바꾼다.(321568756 -> 321565786)
7. i = 7 = j
    - pivot과 i의 위치를 바꾼다.(321565786 -> 321565|687)
8. 각각 321565786, 687에 대해 새로 quick sort를 실행
```

- 2개의 index가 필요.

Code

- recursive와 nr 두개 존재
    - recursive : 빠르지만 복잡하고 메모리 많이 잡아먹음

```
void quick_sort(int *base, int N) {
    int *pivot, temp;
    int i = -1, j = N-1;
    while (N > 1) {
        pivot = base + N - 1;
        while (base[++i] > *pivot);
```

```
        while (base[--j] < *pivot);
        if (i >= j) break;
        temp = base[i];
        base[i] = base[j];
        base[j] = base[i];
    }
    temp = base[i];
    base[i] = *pivot;
    *pivot = temp;

    quick_sort(base, i);
    quick_sort(base + i + 1, N);
}
```

- non-recursive : 느리지만 단순하며 메모리를 적게 잡아먹음

```
void nr_quick_sort(int *base, int N) {
    int pivot, temp;
    int bottom = 0, ceiling = N - 1;
    // 각각 while문 내에서 바닥과 천장을 알려주는 역할
    int up2down, down2up;
    // 위에서 아래로 내려가는 index, 아래에서 위로 올라가는 index
    init_stack();

    push(bottom);
    push(ceiling);

    while (!stack_empty()) {
        ceiling = pop();
        bottom = pop();
        // 바닥과 천장을 pop
        if (ceiling - bottom > 0) {
            // 바닥과 천장이 같으면 해당 배열의 크기는 1이다. 따라서 정렬을
해줄 필요가 없다.
            pivot = base[ceiling];
            // 마지막 항을 pivot이라 설정
            down2up = bottom - 1;
            up2down = ceiling;
            // 양 끝에 index 설정. pre-increase system이므로 1씩 빼준다.
            while (1) {
                while (base[++down2up] < pivot);
                while (base[--up2down] > pivot);
                if (down2up > up2down) break;
                temp = base[down2up];
                base[down2up] = base[up2down];
                base[up2down] = temp;
            }
            temp = base[ceiling];
            base[ceiling] = base[down2up];
            base[down2up] = temp;
            // re와 알고리즘 동일
```

```
            push(down2up);
            push(ceiling);
            // pivot보다 숫자가 큰 부분의 시작과 끝 index를 push해준다.
            push(bottom);
            push(down2up - 1);
            // pivot보다 숫자가 작은 부분의 시작과 끝 index를 push해준다.
        }
    }
}
```

- generalization : int가 아닌 다른 자료형을 활용해도 같은 함수를 쓸 수 있도록 일반화
  - recursive

```
void gen_re_quick_sort(void *base, size_t N, size_t width, int(*fcmp)
(const void *, const void *)) {
    void *pivot = malloc(width);
    void *temp = malloc(width);
    // value는 배열 내 원소의 크기를 따라가므로 void형 포인터로 선언
    int i = -1, j = N - 1;
    // index는 정수이므로 int형으로 선언
    if (N > 1) {
        memcpy(pivot, (char*)base + (N - 1)*width, width);
        while (1) {
            while (fcmp(pivot, (char*)base + ++i*width));
            // pivot보다 큰 바닥에서 제일 가까운 항을 찾는다.
            while (fcmp((char*)base + --j*width, pivot));
            // pivot보다 작은 천장에서 가장 가까운 항을 찾는다.
            if (i >= j) break;
            // 정지 조건 : 아래에서 올라오는 index와 위에서 내려오는 index
가 서로 마주쳤을 경우
            memcpy(temp, (char*)base + i * width, width);
            memcpy((char*)base + i*width, (char*)base + j * width,
width);
            memcpy((char*)base + j*width, temp, width);
            // 정지 조건이 아닐 경우, 두 항의 위치를 바꿔준다.
        }
        memcpy(temp, (char*)base + i*width, width);
        memcpy((char*)base + i*width, (char*)base + (N - 1)*width,
width);
        memcpy((char*)base + (N - 1)*width, temp, width);
        // pivot을 항 중간에 끼워넣는다
        // i 이하 = pivot보다 작은 항들의 배열, j 이상 -> pivot보다 큰 항
들의 배열
        // 따라서 i 다음 항에 pivot을 끼워넣음

        //base[N : i : 0] -> base[N : i] -> pivot보다 작거나 같은 항들의
모임
        //123496785                                          -> pivot보다 크
거나 같은 항들의 모임
        gen_re_quick_sort(base, i, width, fcmp);
        // base -> 1234 정렬
```

```
        gen_re_quick_sort((char*)base + (i + 1)*width, N - i - 1,width,
fcmp);
        // base -> 97858 정렬
    }
}
```

- non-recursive

```
void gen_nr_quick_sort(void *base, size_t N, size_t width, int(*fcmp)
(const void *, const void *)) {
    void *pivot = malloc(width);
    void *temp = malloc(width);
    int bottom = 0, ceiling = N - 1;
    int up2down, down2up;
    init_stack();
    push(bottom);
    push(ceiling);
    while (!stack_empty()) {
        ceiling = pop();
        bottom = pop();
        if (ceiling - bottom > 0) {
            memcpy(pivot, (char*)base + ceiling * width, width);
            down2up = bottom - 1;
            up2down = ceiling;
            while (1) {
                while (fcmp(pivot, (char*) base + (++down2up)*width));
                while (fcmp((char*)base + (--up2down)*width, pivot));
                if (down2up > up2down) break;
                memcpy(temp, (char*)base + down2up*width, width);
                memcpy((char*)base + down2up*width, (char*)base +
up2down*width, width);
                memcpy((char*)base + up2down*width, temp, width);
            }

            memcpy(temp, (char*)base + ceiling*width, width);
            memcpy((char*)base + ceiling*width, (char*)base +
down2up*width, width);
            memcpy((char*)base + down2up*width, temp, width);
            push(down2up);
            push(ceiling);
            push(bottom);
            push(down2up - 1);
        }
    }
}
```

7. Median Filtering

- 같이 제공된 txt 파일을 통해 median filtering 수행해보자

- median filter : 하나의 픽셀 주변에 있는 픽셀의 데이터 정보를 가져와, 해당 픽셀 내의 값을 주변 데이터의 평균값으로 바꾼다.
    - 평균값을 대입해 noise 현상을 제거 가능.

Code

```
// 일렬로 늘어선 3개의 픽셀을 통해 median filter 실행
void medianFiltering_line(BYTE *src, BYTE *dst, int height, int width) {
        for (int y = 0; y < height; y++) {
                for (int x = 1; x < width-1; x++) {
                        lineFilter[0] = src[WEST(x, y, width)];
                        lineFilter[1] = src[MIDDLE(x, y, width)];
                        lineFilter[2] = src[EAST(x, y, width)];
                        dst[MIDDLE(x, y, width)] = sorting(lineFilter,
sizeof(lineFilter));
                }
        }
}

// 중심 픽셀과 4방위각에 있는 픽셀에 대해 median filter을 실행
void medianFiltering_4azimuth(BYTE *src, BYTE *dst, int height, int width) {
        for (int y = 1; y < height-1; y++) {
                for (int x = 1; x < width - 1; x++) {
                        azimuth4Filter[0] = src[WEST(x, y, width)];
                        azimuth4Filter[1] = src[MIDDLE(x, y, width)];
                        azimuth4Filter[2] = src[EAST(x, y, width)];
                        azimuth4Filter[3] = src[SOUTH(x, y, width)];
                        azimuth4Filter[4] = src[NORTH(x, y, width)];

                        dst[MIDDLE(x, y, width)] = sorting(azimuth4Filter,
sizeof(azimuth4Filter));
                }
        }
}

// 중심 픽셀과 8방위 픽셀에 대해 median sorting을 실행.
void medianFiltering_8azimuth(BYTE *src, BYTE *dst, int height, int width) {
        for (int y = 1; y < height-1; y++) {
                for (int x = 1; x < width - 1; x++) {
                        azimuth8Filter[0] = src[WEST(x, y, width)];
                        azimuth8Filter[1] = src[MIDDLE(x, y, width)];
                        azimuth8Filter[2] = src[EAST(x, y, width)];
                        azimuth8Filter[3] = src[SOUTH(x, y, width)];
                        azimuth8Filter[4] = src[NORTH(x, y, width)];
                        azimuth8Filter[5] = src[SOUTHEAST(x, y, width)];
                        azimuth8Filter[6] = src[SOUTHWEST(x, y, width)];
                        azimuth8Filter[7] = src[NORTHEAST(x, y, width)];
                        azimuth8Filter[8] = src[NORTHWEST(x, y, width)];

                        dst[MIDDLE(x, y, width)] = sorting(azimuth8Filter,
sizeof(azimuth8Filter));
                }
```

```
        }
}
```

8. Before Start

- Quick Sort
  - worst case : sorted case, worst case에서 '매우' 비효율적이다.
  - 단, random한 경우에는 다른 sorting에 비해 효율적임

9. Improve Quick Sort

1. pivot을 랜덤하게 설정해준다.

Code

```
void sorting(int *base, int size) {
        int pivot, temp;
        int bottom = 0, ceiling = size - 1;
        // 각각 while문 내에서 바닥과 천장을 알려주는 역할
        int up2down, down2up;
        // 위에서 아래로 내려가는 index, 아래에서 위로 올라가는 index
        init_stack();

        push(bottom);
        push(ceiling);

        while (!stack_empty()) {
                ceiling = pop();
                bottom = pop();
                // 바닥과 천장을 pop
                if (ceiling - bottom > 0) {
                        // 바닥과 천장이 같으면 해당 배열의 크기는 1이다. 따라서 정렬
을 해줄 필요가 없다.
                        pivot = rand()%(ceiling-bottom+1);
                        // 마지막 항을 pivot이라 설정
                        down2up = bottom - 1;
                        up2down = ceiling;
                        // 양 끝에 index 설정. pre-increase system이므로 1씩 빼준다.
                        while (1) {
                                while (base[++down2up] < pivot);
                                while (base[--up2down] > pivot);
                                if (down2up > up2down) break;
                                temp = base[down2up];
                                base[down2up] = base[up2down];
                                base[up2down] = temp;
                        }
                        temp = base[ceiling];
                        base[ceiling] = base[down2up];
                        base[down2up] = temp;
                        // re와 알고리즘 동일
```

```
                        push(down2up + 1);
                        push(ceiling);
                        // pivot보다 숫자가 큰 부분의 시작과 끝 index를 push해준다.
                        push(bottom);
                        push(down2up - 1);
                        // pivot보다 숫자가 작은 부분의 시작과 끝 index를 push해준다.
                }
        }
}
```

2. 변수 개수에 따라 다른 sorting을 사용 -> interval size가 N 이하일때 insert sort, 그 외엔 quick sort 실행한다.

Code

```
void quick_sort3(int *a, int N)
{
    int v, t;
    int i, j;
    int l, r;
    init_stack();
    l = 0;
    r = N-1;
    push(r);
    push(l);
    while(!is_stack_empty()){
        l = pop();
        r = pop();
        if(r-l+1 > 200) // quick sort
        {
            t = rand()%(r-l+1);
            v = a[t];
            a[t] = a[r];
            a[r] = v;
            i = l-1;
            j = r;
            while(1){
                while(a[++i] < v);
                while(a[ j--] > v);
                if(I >= j) break;
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
            t = a[i];
            a[i] = a[r];
            a[r] = t;
            push(r);
            push(i + 1);
            push(i - 1);
            push(l);
```

```
        }
        else
            insert_sort(a+l, r-l+1 );
    }
}
```

10. Heap Sort

- Priority First Sort(Priority : Value)
- Made by Tree structure
- not require any addition memory
- Complexity : O(N*logN)
- Root : Maximum & Leaf : Minimum
- Ground Rule : 부모의 값보다 자식의 값이 작거나 같아야한다.
- **uphead** & **downheap** : 각각의 역할이 뭔지 잘 알고 있어야 한다.
- index를 0이 아닌 1부터 시작 -> child와 parent 관계를 보다 편하게 계산 가능하다.
    - a[1]의 자식 -> a[2], a[3] = N * 2, N * 2 + 1
    - a[2]의 자식 -> a[4], a[5] = N * 2, N * 2 + 1
    - a[3]의 자식 -> a[6], a[7] = N * 2, N * 2 + 1
    - a[4]의 자식 -> a[8], a[9] = N * 2, N * 2 + 1
    - a[5]의 자식 -> a[10], a[11] = N * 2, N * 2 + 1

Code

- A : heap에 순서대로 집어넣는다(upheap) -> 이후 root를 뺄때 downheap을 통해 재정렬

```
void heap_sorting_A(int *a, int N)
{
    int i;
    pq_init();
    for (i = 0; i < N; i++)
        insert(h, &nheap, a[i]);
    for (i = nheap; i > 0; i--) {
        printf("%2d", extract(h));
    }
}
```

- B : 무작위로 집어넣어져 있는 heap에 downheap을 통해 정렬 -> 이후 root를 빼고 downheap을 통해 재정렬

```
void heap_sorting_B(int *a, int N) {
    int k, t;
    a[0] = INT_MAX;
    nheap = N;
    for (k = N / 2; k > 0; k--)
        downheap(a, k);
    while (nheap > 0) {
```

```
            printf("%2d", t = a[1]);
            a[1] = a[nheap];
            a[nheap--] = t;
            downheap(a, 1);
        }
    }
```

Summary

- Selection Sort : 선택된 항 앞자리를 탐색, 들어가야하는 자리에 집어넣는다.
    - result

```
src : 546878953
result : 546878953 -> 346878955 -> 346878955 -> 345878965 -> 345578968
-> 345568978 -> 345567988 -> 345567898 -> 345567889
```

    - code

```
void gen_select_sort(void *base, size_t nelem, size_t width, int(*fcmp)
(const void *, const void *))
// base <-> a, nelem <-> N
// width : 자료형의 크기
// fcmp : 지정된 메모리 위치 내의 크기를 특정 상수와 비교하는 함수
{
    void *min;
    // min : 최소값. 다만 메모리 축소를 위해 포인터로 선언
    int min_idx;
    int x, y;
    min = malloc(width);
    // min에 자료형만큼 할당해줌
    for (y = 0; (unsigned int)y < nelem - 1; y++)
    {
        min_idx = y;
        memcpy(min, (char *)base + y * width, width);
        // a[y] = (char*)base + y*width,width
        // 처음엔 중심 항을 최소값으로 간주한다.

        for (x = y + 1; (unsigned int)x < nelem; x++)
        {
            if (fcmp(min, (char *)base + x * width))
            // min과 a[x]의 값을 비교한다.
            {
                memcpy(min, (char *)base + x * width, width);
                // 만약 결과가 0보다 크면 min값을 변경
                min_idx = x;
                // 최소값의 인덱스값도 변경
            }
        }
        memcpy((char *)base + min_idx * width, (char*)base + y * width,
```

```
width);
        // base[min_idx] = base[y];
        memcpy((char *)base + y * width, min, width);
        // base[y] = min;
    }
    free(min);
}
```

- Insertion Sort : 선택된 항과 양 옆자리를 비교해 값을 서로 바꾼다.
  - 어느정도 정렬이 되어있는 자료를 빠르게 sorting 가능. -> shell sort로 발전
  - result

```
src : 546878953
rst : 456878953 -> 456878953 -> 456878953 -> 456788953 -> 456788953 ->
456878953 -> 455678893 -> 345567889
```

  - code

```
void gen_insert_sort(void *a, size_t nelem, size_t width, int(*fcmp)
(const void *, const void *)) {
    void *t;
    int i, j;
    t = malloc(width);
    for (i = 1; (unsigned int)i < nelem; i++) {
    // i = 0 일때, 비교할 것이 없으므로 굳이 따질 필요 없다.
        memcpy(t, (char*)a + i * width, width);
        j = i;
        // 초기값을 min_value라 가정, index와 value를 저장한다.
        while (j > 0 && fcmp((char*)a + (j - 1)*width, t)) {
            memcpy((char*)a + j * width, (char*)a + (j - 1)*width,
width);
            j--;
        }
        memcpy((char*)a + j * width, (char*)t, width);
    }
    free(t);
}
```

- Shell Sort : 특정 interval마다 insertion sort를 실행. 최종적으로 interval이 1이 되면 sorting이 종료된것이다.
  - 자료를 점차 좁은 간격으로 sorting해나간다.
  - result

```
src : 546878953153
rst : |543153|956878| -> |5331|5495|6878| -> |133|554|856|978| ->
|13|34|55|65|78|89| -> |1|3|3|4|5|5|5|6|7|8|8|9|
```

- code

```
void shell_sort(int *a, size_t nelem, size_t width, int(*fcmp)(const
void *, const void *))
{
    int i, j, k, h;
    void *v = malloc(width);
    for(h = N/2; h>0; h /= 2){
        for(i = 0; i<h; i++){
            for(j = i+h; j<N; j += h){
                memcpy(v,(char*)a + j*width);
                k = j;
                while(k > h-1 && fcmp((char*)a + (k-h)*width,
(char*)v)){
                    a[k] = a[k-h];
                    k -= h;
                }
                memcpy((char*)a + k*width, v, width);
            }
        }
    }
}
```

- Quick Sort
- Improve Quick Sort
- Heap Sort