**Homework 6\_1 [selection\_sort.cpp]**

First, explain the stable sorting result of the following input.

The difference between stable sorting and unstable sorting is whether the relative order of same key is maintained or not. The reason why following output is unstable is that the relative order in key `30` is changed. Before sorting the key `30` with id `1 preceded the `30`with id `6`. But after sorting, `30`with id `6` is preceding the key `30` , which means relative order between two is changed.

테이블이(가) 표시된 사진

자동 생성된 설명

The stable sorting result of this input is swapping 30` with id `1 and `30`with id `6`.

**Variable analysis**

|  |  |
| --- | --- |
| **data** | |
| **type** | **name** |
| Int \* | id |
| Int \* | score |

id : the sub criteria for sorting

score : the main criteria for sorting

**Function analysis**

1)selection\_sort

The following code is original selection\_sort which is `unstable`. This is because there is an exchange between two distant elements. To make selection\_sort stable, instead of swapping two elements, we can do by just pushing every element one step forward and insert min value in proper place .

void selection\_sort(data \*list, int n)

{

int i, j, least, temp;

for (i = 0; i < n - 1; i++) {

least = i;

for (j = i + 1; j<n; j++)

if (list->score[j]<list->score[least]) least = j;

SWAP(list->score[i], list->score[least], temp);

SWAP(list->id[i], list->id[least], temp);

}

}

2) selection\_sort\_stable

: The purpose of this code is to sort given list in asceding order in stable manner.

1. Find minimum value
2. Push every element one step backward
3. 3.Insert minimum value at the beginning of a place that's not aligned yet. (i)

void selection\_sort\_stable(data \*list, int n)

{

int i, j, k, least, min\_score, min\_id;

for (i = 0; i < n - 1; i++) {

least = i;

//1. find minimum element

for (j = i + 1; j < n; j++)

if (list->score[j] < list->score[least]) least = j;

// keep min data

min\_score = list->score[least];

min\_id = list->id[least];

//2. instead of swapping.push element one step ward( right shift )

for (k = least; k >i; k--) {

list->score[k] = list->score[k - 1];

list->id[k] = list->id[k - 1];

}

//3 . Insert minimum value in its correct place

list->score[i] = min\_score;

list->id[i] = min\_id;

}

}

**Simulation**

**[i=0]**

1. Find min value ‘2’ and keep in variable min\_score and min\_id.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **5** | **8** | **3** | **2** | **9** | **7** |

2. Push element one step backward

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **5** | **5** | **8** | **3** | **9** | **7** |

3.Insert minimum(keep) value at the beginning of a place that's not aligned yet. (i)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2** | **5** | **8** | **3** | **9** | **7** |

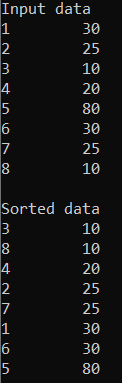
**By keeping this process, the relative order in same key does not change.**

**…**

**[i=5]**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2** | **3** | **5** | **8** | **7** | **9** |

**[result]**

****

**Homework 6\_1 [radixsort.cpp]**

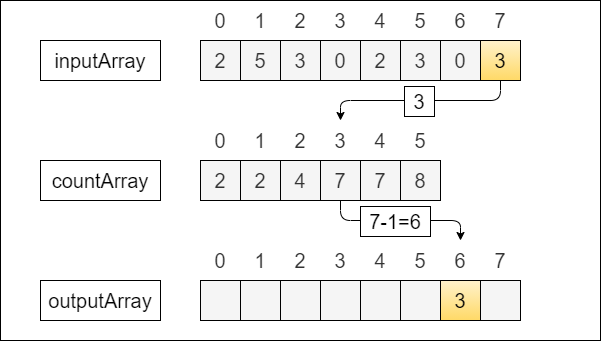
**Variable analysis**

#define INPUT 10 : number of input data

#define BUCKETS 64 : the range of each digit. 24 bit / 4 = 6-bit. 2^(6) = 64.

#define DIGITS 4 : the number of digit.

**Counting sort**



: Radix\_sort can be implemented by applying `counting sort` . The reason why counting sort can be subroutine of radix sort is that it is `stable` sort. (Unstable sort can not be used in sorting each digit. ) [Counting sort](http://en.wikipedia.org/wiki/Counting_sort) is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence

For simplicity, consider the data in the range 0 to 9. (BUCKETS=10)

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2) Modify the count array such that each element at each index

stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

3) Starting from end of the Input array, find proper place to insert using cumulative histogram.

What the accumulative sum upto `2` is `4` means there are `3` items less or equal than `2`, so the proper place to insert `2` is 4th . and for next equal item to be sorted in front of this item, the count decreases by one.

Input data: 1, 4, 1, 2, 7, 5, 2

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

Sorted: X X X 2 X X X X

Input data: 1, 4, 1, 2, 7, 5, ~~2~~

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 3 4 5 6 6 7 7 7

Sorted: X X X 2 X X 5 X

…

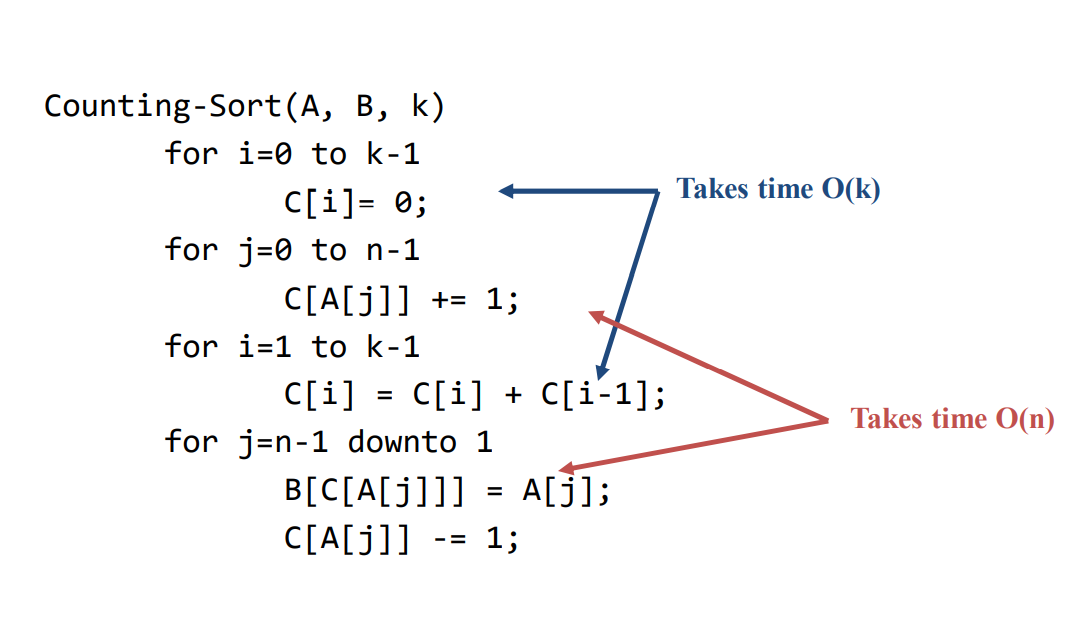
Input data: 1, 4, 1, 2, 7, 5, 2

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 1 2 4 4 5 6 6 7 7

Sorted: 1 1 2 2 4 5 7

The pseudo code is as below.

****

**Function analysis**

1)generate\_random

: To test sorting algorithm, we have to generate random number. To generate random number in specified range (0~2^(24)), it is not enough to call a rand() once. Because the maximum value in rand() is 32767, which is smaller than 2^(24). So, to make big random number over 34767, we have to use rand() more than once and multiply the two.

void generate\_random(int A[]) {

srand((unsigned)time(NULL));

for (int i = 1; i <= INPUT; i++) {

int n1 = rand() % 4096 + 1; // 1 ~ 2^12

int n2 = rand() % 4096 + 1; // 1 ~ 2^12

int r = n1 \* n2 - 1; // 0 ~ 2^24-1

A[i] = r;

}

}

**2)** radix\_sort

: **radix sort** is a non-[comparative](https://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm). It avoids comparison by creating and [distributing](https://en.wikipedia.org/wiki/Distribution_sort) elements into buckets according to their [radix](https://en.wikipedia.org/wiki/Radix). For elements with more than one [significant digit](https://en.wikipedia.org/wiki/Significant_digit), this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered.  We divides 24-bit number into 4 digits (=segments), so the range of each digit can be between 0 and 2^(6)-1. So the BUCKET size should be 2^(6)=64. Sorting values at each digit is only possible from LSD to MSD. Because the MSB is more influential criteria for sorting. Than, we can apply counting sort in each digit. Just as we do n/10^(n)%10 to find the n th digit in decimal (radix = 0~9), we use this same formula n/64^(n)%64 to find the n th digit (radix = 0~63) The variable `factor` takes charge of dividing each digit. For every loop, factor is multiplied by the size of BUCKETS.

void radix\_sort(int A[], int n) {

int C[BUCKETS], B[INPUT+1] = { 0, };

int factor = 1; //

int i,j;

for ( i = 1; i <= DIGITS; i++) {

//counting sort

//1. initialize

for (j = 0; j < BUCKETS; j++) {

C[j] = 0;

}

// 2. count appearances of each number.

for (j = 1; j <= n; j++)

C[(A[j] / factor) % BUCKETS] += 1;

// 3. sum accumulately

for (j = 1; j < BUCKETS; j++)

C[j] = C[j] + C[j - 1];

// 4. find proper place to insert ` A[j]` using 'C(cumulative histogram)'

for (j = n ; j > 0; j--) {

B[C[(A[j] / factor) % BUCKETS]] = A[j];

C[(A[j] / factor) % BUCKETS] -= 1;

}

//copy to original array.

for (j = 1; j <= n; j++)

A[j] = B[j];

factor \*= BUCKETS;

}

}

Simulation

This example supposes that n=4, b=24, r=6, b/r=4 (# of digits)

[The inintial state]

: each digit can varies from 0 to 2^(6)-1. Using counting sort in each digit.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Decimal Digit | 4 | 3 | 2 | 1 |
| 16777215 | 111111 | 111111 | 111111 | 111111 |
| 1359253 | 000101 | 001011 | 110110 | 010101 |
| 13462144 | 110011 | 010110 | 101010 | 000000 |
| 1359298 | 000101 | 001011 | 110110 | 000010 |

When i=1 (digit =1),

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Decimal Digit | 4 | 3 | 2 | 1 |
| 16777215 | 111111 | 111111 | 111111 | 111111 |
| 1359253 | 000101 | 001011 | 110110 | 010101 |
| 1359298 | 000101 | 001011 | 110110 | 000010 |
| 13462144 | 110011 | 010110 | 101011 | 000000 |

i=2 (digit =2),

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Decimal Digit | 4 | 3 | 2 | 1 |
| 16777215 | 111111 | 111111 | 111111 | 111111 |
| 1359298 | 000101 | 001011 | 110111 | 000010 |
| 1359253 | 000101 | 001011 | 110110 | 010101 |
| 13462144 | 110011 | 010110 | 101010 | 000000 |

i=3 (digit =3),

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Decimal Digit | 4 | 3 | 2 | 1 |
| 16777215 | 111111 | 111111 | 111111 | 111111 |
| 13462144 | 110011 | 010110 | 101010 | 000000 |
| 1359298 | 000101 | 001011 | 110111 | 000010 |
| 1359253 | 000101 | 001011 | 110110 | 010101 |

: even though there are same values in processing this digit, the order is already determined. This is the reason why the only stable sorting is allowed in radix sort.

i=4 (digit =4),

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Decimal Digit | 4 | 3 | 2 | 1 |
| 16777215 | 111111 | 111111 | 111111 | 111111 |
| 13462144 | 110011 | 010110 | 101010 | 000000 |
| 10672898 | 000101 | 001011 | 110111 | 000010 |
| 10672917 | 000101 | 001011 | 110110 | 010101 |

**3)** check

: it is a function that check if the sorting in `acending order` is implemented correctly.

It travels all items in a given array `A` and compare its size with before. To be sorted in ascending order correctly, the before node is always smaller than the current.

bool check(int A[], int n) {

for (int i = 2; i <= n; i++) {

if (A[i] < A[i - 1]) // if the before item is larger than current

return false;

}

return true;

}

**[result]**

: this is the result when the `INPUT = 10`

**텍스트이(가) 표시된 사진

자동 생성된 설명**