Sorting
1. Sorting

- The most basic technique in programming

- So many solutions for it
  - O\(n^2\), O\(n \lg n\), O\(n\)
  - depending on
    - simplicity of mind
    - complexity of insert operation
2. Sorting Algorithms

- Comparison Sort – $O(n^2)$, $O(n \log n)$
  - Insertion sort
  - Merge sort
  - Heap sort
  - Quick sort
  - Counting Sort – $O(n)$
    - radix sort: extended counting sort
  - Bucket Sort – $O(n)$

- Why different complexities?
  - use of smart data structure
  - hide cost operations behind
3. Selection Sort

10  1  9  2  8  3
Swap 10 and 1, 1 is less than 10

1  10  9  2  8  3
Swap 10 and 2, 2 is less than 10

1  2  9  10  8  3
Swap 9 and 3, 3 is less than 9

1  2  3  10  8  9
Swap 10 and 8, 8 is less than 10

1  2  3  8  10  9
Swap 10 and 9, 9 is less than 10

1  2  3  8  9  10
3. Selection Sort – slow

78  84  42  78  56  96  26  38

78  88  42  48  48  48  48  48
3. Selection Sort – the code

- Pseudocode

```plaintext
for i ← 0 to n-2 do
    min ← i
    for j ← (i + 1) to n-1 do
            min ← j
        swap A[i] and A[min]
```

- C Code

```c
selection_sort(int s[], int n) {
    int i,j; /* counters */
    int min; /* index of minimum */

    for (i=0; i<n; i++) {
        min=i;
        for (j=i+1; j<n; j++)
            if (s[j] < s[min]) min=j;
        swap(&s[i],&s[min]);
    }
}
```
4. Insertion Sort

Jth Element

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9

done
4. Insertion Sort – the code

```plaintext
Insertion-Sort (A)
  for each j = 2 to length [A]
    do key = A[j]  //Inserted into Sorted Subarray
      i = j – 1
      while i > 0 & A[i] > key
        do A[i+1] = A[i]
        i = i – 1
      A[i+1] = key
```
5. Quick Sort

- **Divide-and-Conquer**
  - **Divide**
    - the array into two parts
    - the value of x? pivot
  - **Conquer**
    - do the same to each divided subarray
  - **Combine**
5. Quick Sort Example

- pivot: 6
5. Quick Sort Algorithm

\[
\text{QUICKSORT}(A, p, r) \\
\text{if } p < r \\
\quad \text{then } q \leftarrow \text{PARTITION}(A, p, r) \\
\quad \text{QUICKSORT}(A, p, q-1) \\
\quad \text{QUICKSORT}(A, q+1, r)
\]

**Initial call:** \text{QUICKSORT}(A, 1, n)
Details

\textbf{Partition}(A, p, q) \triangleq A[p \ldots q]
\begin{align*}
x &\leftarrow A[p] \quad \triangleleft \text{pivot} = A[p] \\
i &\leftarrow p \\
&\text{for } j \leftarrow p + 1 \text{ to } q \\
&\quad \text{do if } A[j] \leq x \\
&\quad \quad \text{then } i \leftarrow i + 1 \\
&\quad \quad \text{exchange } A[i] \leftrightarrow A[j] \\
&\text{exchange } A[p] \leftrightarrow A[i] \\
&\text{return } i
\end{align*}

Running time = \(O(n)\) for \(n\) elements.
```
quicksort(int s[], int l, int h)
{
    int p;         /* 분할을 위한 인덱스 */
    if ((h-l)>0)) {
        p = partition(s, l, h);
        quicksort(s, l, p-1);
        quicksort(s, p+1, h);
    }
}

int partition(int s[], int l, int h)
{
    int i;            /* counter */
    int p;            /* 피벗 원소의 인덱스 */
    int firsthigh;    /* 피벗을 위한 디바이더 위치 */
    p = l;
    firsthigh = l;
    for (i=l; i<h; i++)
    {
        if (s[i] < s[p]) {
            swap(&s[i], &s[firsthigh]);
            firsthigh++;
        }
    }
    swap(&s[p], &s[firsthigh]);
    return(firsthigh);
}
```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
   int (*compare) (const void *, const void *));

- array: base
- number of elements: nel
- size of each element: width bytes
- you should provide compare function

int mycompare(int *i, int *j)
{
    if (*i > *j) return (1);
    elseif (*i < *j) return (-1);
    else return (0);
}
Library

- bsearch (key, (char *) a, cnt, sizeof(int), intcompare)
  - search the key value in the given sorted array

- C++

```cpp
void sort(RandomAccessIterator bg, RandomAccessIterator end)
void sort(RandomAccessIterator bg, RandomAccessIterator end,
         BinaryPredicate op)
void stable_sort(RandomAccessIterator bg, RandomAccessIterator end)
void stable_sort(RandomAccessIterator bg, RandomAccessIterator end,
                 BinaryPredicate op)
```
Vito’s Family

The input consists of several test cases. The first line contains the number of test cases.

For each test case you will be given the integer number of relatives $r$ ($0 < r < 500$) and the street numbers (also integers) $s_1, s_2, \ldots, s_i, \ldots, s_r$ where they live ($0 < s_i < 30,000$). Note that several relatives might live at the same street number.

For each test case, your program must write the minimal sum of distances from the optimal Vito’s house to each one of his relatives. The distance between two street numbers $s_i$ and $s_j$ is $d_{ij} = |s_i - s_j|$.

Sample Input

2
2 2 4
3 2 4 6

Sample Output

2
4