Sorting

Today..

- What is sorting?
- How to sort?
- Problems that are solved easily after sorting
  - Minimum distance between points on a line
  - Searching for an element – binary search

Today..

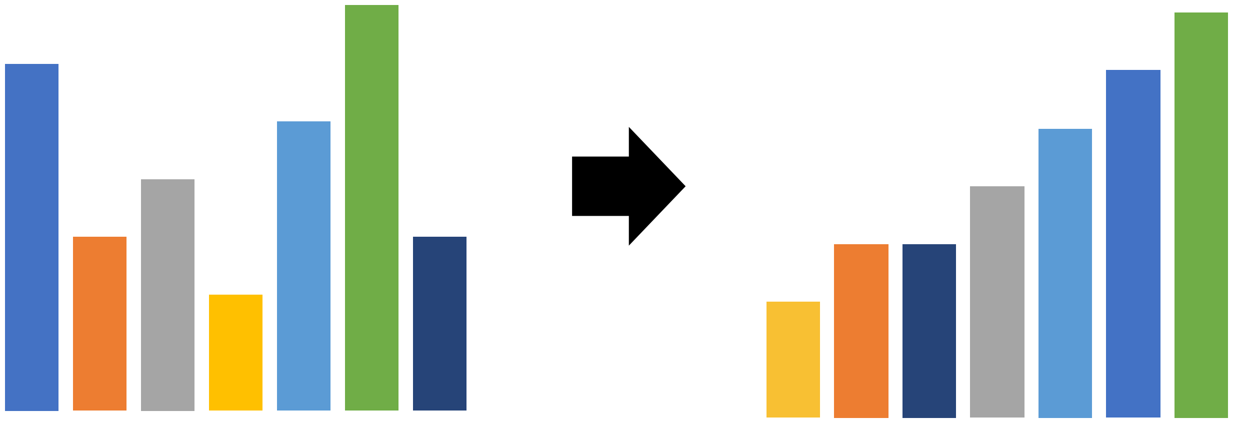
- **What is sorting?**
- How to sort?
- Problems that are solved easily after sorting
  - Minimum distance between points on a line
  - Searching for an element – binary search

What is sorting?

- It is rearranging the given data into a *total order:*

What is sorting? (cont.)

- After sorting an array $a[1], a[2], \cdots, a[n]$,
  - The elements are *permuted:* we only *reorder* the elements of array
  - The elements are *in order*: the element on the left $\leq$ element on the right.



What do we sort?

- As you seen in this picture, we usually sort **numbers** (like integers, doubles, ..)

What do we sort? (cont.)

- However, we can also sort..
- **Strings** in lexicographical order (like in dictionaries)
- **Dates** in chronological order
- **Arrays of numbers**(?!) in the order of their sum
- …
- and lots more.

Requirements of sorting

- To sort $a[1], a[2], \cdots, a[n]$:

1. Elements should be *comparable*.
   - We can sort [1, 5, -9, 4, 2] because numbers are comparable.
   - We can't sort [1, 5, 'hello', 4, 'bye'] because numbers and strings *aren't comparable* (unless we specify)
   - We should be able to compare any $a[i]$ and $a[j]$, and find which one is bigger.

Requirements of sorting (cont.)

- To sort $a[1], a[2], \cdots, a[n]$:

2. All the elements should satisfy..

- **Transitivity**: $x < y$ and $y < z$ implies $x < z$.
  - ex) $1 < 2, 2 < 3$, so we know $1 < 3$
  - Ex2) $rock < paper$, $paper < scissors$ but $rock > scissors$.
  - If transitivity not holds, we cannot sort in an order.

Requirements of sorting (cont.)

- To sort $a[1], a[2], \cdots, a[n]$:

2. All the elements should satisfy..

- Transitivity: $x < y$ and $y < z$ implies $x < z$.
- **Totality**: If $x \neq y$, $x < y$ or $y < x$ should must hold.
  - Ex) $1245 <$ 'bda'? 'bda'< 12345? We don't know unless we define the $\leq$ operator nicely.

Today..

- What is sorting?
- **How to sort?**
- Problems that are solved easily after sorting
  - Minimum distance between points on a line
  - Searching for an element – binary search

We are not interested in *just sorting*

- There are bunch of sorting algorithms in the world, and you will learn about it in any algorithm course.
- Also, sorting is an important problem, so the solution is well-known.
- Therefore, in ICPC, <u>authors don’t ask to just sort the input</u>!
- So we are going to look over some simple algorithms, and introduce the built-in sort function.

Selection sort

```
for(int i = 1; i <= n; i++) {
   // 1. find smallest value a[x] in a[i..n]
   int x = i;
   for(int j = i+1; j <= n; j++)
      if(a[x] > a[j]) x = j;

   // 2. change a[x] and a[i].
   // Now a[i] is the smallest among a[i..n]
   swap(a[x], a[i]);
}
```

Insertion sort

```
for(int i = 1; i <= n; i++) {
   // We suppose a[1..(i-1)] is already sorted
   // Our goal is to insert a[i] into a[1..(i-1)],
   // so that a[1..i] is sorted.

   for(int j = i-1; j >= 1; j--) {
      if(a[j] < a[j+1]) break;
      swap(a[j], a[j+1]);
   }
}
```

Sort function in STL

- There is a built-in function of sorting:
- Suppose we are trying to sort an array defined like:

`type arr[10000];`

- From `arr[0]` to `arr[n-1]`.
- We can sort this array by:
  - `#include <algorithm>`
  - `std::sort(arr, arr + n);`
- Time complexity is $O(n \log n)$, so you can use this function to sort large inputs (like $n \leq 10^6$).

Sort function in STL (cont.)

- Additionally, to sort a `vector<type> a`, we can write like
  - `#include <algorithm>`
  - `std::sort(a.begin(), a.end());`
- You can use either a pointer or an iterator as a argument to the built-in sort function.

Sort function in STL (cont.)

- We can also define a *comparator* by ourselves, like for example:

```
bool compare (const type &i, const type &j) {
    return f(i) < f(j);
}
```

- The `compare` function should..
  - return `true` if $i < j$, and `false` otherwise.
  - satisfy transitivity: $x < y$ and $y < z$, then $x < z$.
  - `compare(x, y) == false` and `compare(y, x) == false` if and only if $x = y$.
  - There should be no case that `compare(x, y) == true` and `compare(y, x) == true`

Sort function in STL (cont.)

- If we define a comparator nicely, we can use the built-in sort function like:

```
sort(a, a+n, compare);
```

- Of course, the type of i and j must be the same as the type of the array `a`.

Time complexity of comparison-based sorting can't be better than $O(n \log n)$

- Actually, the built-in sort function is *asymptotically optimal* for sorting (if we only use comparisons to find order)
- Sketch of proof:
  - We have to pick one of the $n!$ permutations of the array.
  - For each comparison, the number of possible permutations can be reduced at most half.
  - So we need at least $\log_2(n!)$ comparisons, which is $O(n \log n)$. (we omit proof of this)

Today..

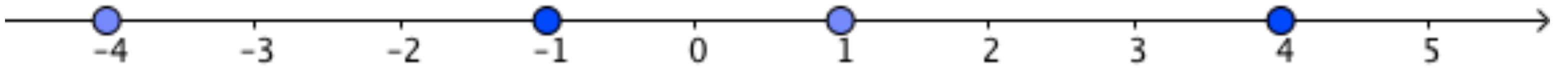
- What is sorting?
- How to sort?
- **Problems that are solved easily after sorting**
  - Minimum distance between points on a line
  - Searching for an element – binary search

Today..

- What is sorting?
- How to sort?
- Problems that are solved easily after sorting
  - **Minimum distance between points on a line**
  - Searching for an element – binary search

Minimum distance between points on a line

- There are points $P_1, P_2, \cdots, P_n$ with x-coordinates $x_1, x_2, \cdots, x_n$.



- Ex) $x_1 = -1, x_2 = 4, x_3 = 1, x_4 = -4$.
- Find the minimum distance between these points.

Minimum distance between points on a line (cont.)

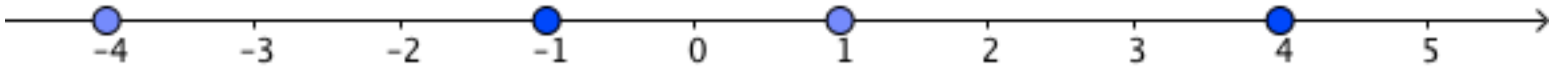
- Simplest algorithm: We are asked to find $\min_{1 \le i < j \le n} |x_i - x_j|$. So..

```
int answer = INT_MAX;
for(int i = 1; i <= n; i++) {
   for(int j = i+1; j <= n; j++) {
      int d = abs(x[i] - x[j]);
      if(answer > d) answer = d;
   }
}
```

- The time complexity is $O(n^2)$.

Minimum distance between points on a line (cont.)

- Faster algorithm: sort the array $x_{1..n}$ first. Then..



- $x_1 = -4, x_2 = -1, x_3 = 1, x_4 = 4.$
- We can only consider *adjacent points* to get the minimum distance!
- (If you are given the picture and suppose to get the minimum distance, you would only consider adjacent points)

Minimum distance between points on a line (cont.)

- So the solution goes like:

```
sort(x+1, x+n+1);

int answer = INT_MAX;
for(int i = 2; i <= n; i++) {
   int d = x[i] - x[i-1];
   if(answer > d) answer = d;
}
```

- The time complexity is $O(n \log n)$, since sort takes $O(n \log n)$ and the loop takes $O(n)$.

Today..

- What is sorting?
- How to sort?
- Problems that are solved easily after sorting
  - Minimum distance between points on a line
  - **Searching for an element – binary search**

Searching for an element

- Suppose you have an array $a[0..(n-1)]$ and you are trying to find whether x is in this array or not.
- We could do like this:

```
for(int i = 0; i < n; i++) {
    if(a[i] == x) {
        printf("We found %d in position %d\n", x, i);
        break;
    }
}
```

- Which takes $O(n)$ time.

What if we have to search lots of times?

- Suppose you have an array $a[0..(n-1)]$ and you are trying to find whether $x_0, x_1, \cdots, x_{m-1}$ is in this array or not.
- We can also do the same:

```
for(int j = 0; j < m; j++) {
    bool found = false;
    for(int i = 0; i < n; i++) {
        if(a[i] == x[j]) found = true;
    }
    if(found) printf("We found %d in position %d\n", x[j], i);
    else printf("We couldn't find %d in position %d\n", x[j], i);
}
```

- It takes $O(nm)$ time. Can we improve more?

Searching for an element in a sorted array

- That's where the sorting pops out! Now, suppose the array $a[0..(n-1)]$ is sorted.

0	1	2	3	4	5	6	7
1	3	5	6	9	13	14	17

- Let's try to check whether 4 is in the array or not.

Searching for an element in a sorted array (cont.)

- The idea is this. Let’s pick any element in the array, for example $a[5]$.

0	1	2	3	4	5	6	7
1	3	5	6	9	**13**	14	17

- Compare 2 with 13. Obviously 2 < 13.
- Since 2 < 13, obviously **2** < 13 < **14**, **2** < 13 < **17** too.
- The array is sorted, so now we know that **if 2 is present, it must be in $\boldsymbol{a[0..4]}$!**

Searching for an element in a sorted array (cont.)

- This time, let's pick $a[3]$. We know that 2 < 6, and 2 < 6 < 9 also holds too.

0	1	2	3	4
1	3	5	**6**	9

- Therefore we can be sure **<u>if 2 is present, it must be in $\boldsymbol{a[0..2]}$!</u>**

Searching for an element in a sorted array (cont.)

- This time, let's pick $a[0]$. We know that 2 > 1.

0	1	2
1	3	5

- Therefore we can be sure **if 2 is present, it must be in $\boldsymbol{a[1..2]}$!**

Searching for an element in a sorted array (cont.)

- This time, let's pick $a[1]$. We know that 2 < 3, and 2 < 3 < 5 also holds too.

1	**2**
3	5

- Now, we can be sure that 2 is not present in the array.

Searching for an element in a sorted array (cont.)

- In this example, we just randomly took any element to narrow the possible candidates of positions of x.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	3	5	6	9	**13**	14	17

- For example, we chose $a[5]$ in the above array, and the # of candidates decreased from 8 to 5.
- If we chose $a[7]$, the number of possible candidates would decrease from 8 to 7, which is really big..

Searching for an element in a sorted array (cont.)

- To avoid this situation, we are going to take the *midpoint* of the interval. (If number of elements are even, take any.)

0	1	2	3	4	5	6	7
1	3	5	**6**	9	13	14	17

- If $x < 6$, the number of candidates becomes 3.
- If $x = 6$, everything is over.
- If $x > 6$, the number of candidates becomes 4.
- So for any case, the number of candidates becomes at most half the original number of candidates.

Binary search

- This algorithm is called *binary search*.
- The time complexity is $O(\log_2 n) = O(\log n)$.
    - .. since for each comparison (midpoint with x),
    - the number of possible candidates becomes at most half.

Binary search (cont.)

- If we try to find $x = 2$ by binary search, we go like..

0	1	2	3	4	5	6	7
1	3	5	**6**	9	13	14	17

0	1	2
1	**3**	5

0
1

Binary search (cont.)

```
int l = 0, r = n-1;
// If x is in the array, it must be inside a[l..r]
while(l <= r) {
    int m = (l + r) / 2;
    if(x < a[m]) {
        r = m - 1;
    } else if(x > a[m]) {
        l = m + 1;
    }else { // we found it!
        printf("we found %d in position %d\n", x, m);
        break;
    }
}
```

There are lots of other problems..

- ..that can be solved easily after sorting the input.
- Please enjoy!