Searching and sorting

Charlie Li

2019-02-16

Content

- Searching
  - Linear Search
  - Binary Search
  - Ternary Search

- Sorting
  - Selection Sort, Insertion Sort, Bubble Sort
  - Quick Sort, Merge Sort
  - Counting Sort, Radix Sort

Searching

- Usage
  - Generally, it is used to find any root or optimal value of some function f(x).
  - In particular, it can be used to locate a certain object or optimal value in an array

Searching

- 4 types of questions we can ask
- For array,
  - Find the position of certain object in the array
  - Find the optimal (e.g. smallest or largest) element in the array

- For function f(x),
  - Find the root of f(x) = y
  - Find the optimal (e.g. smallest or largest) point of f(x)

Linear Search

- Aka sequential search, dictionary search
- The is one of the naïve ways to locate certain object or to find an optimal element in an array (and even for linked-list (appears in DS(I)))

Linear Search

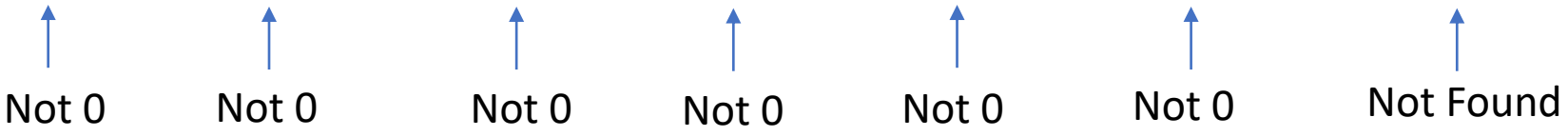
- Example:
- Find any 3 in the following array

Index	**0**	**1**	**2**	**3**	**4**	**5**
Value	1	5	1	3	4	2

Linear Search

- Example:
- Find any 0 in the following array

Index	0	1	2	3	4	5
Value	1	5	1	3	4	2



Linear Search

- For your reference, the code looks like this:

```
for(int i = 0; i <= n; i++) {
    if (i == n) {
        printf("%d is not found in a.\n", x);
        break;
    }
    if (a[i] == x) {
        printf("%d is found at position %d.\n", x, i);
        break;
    }
}
```

- The idea is similar to find the optimal value.

Linear Search

- One can easily see that the above algorithm is always true without any restriction on the array a.

- For the time complexity,
- Best: O(1)
- Worst: O(n)
- Average: O(n)

Linear Search

- Here is one simple optimization problem which can be solved using linear search: HKOJ 01023

Linear Search

- We see that linear search is O(n).
- If we need to repeat the searching for O(n) times, then the overall complexity is O(n^2) which is not fast enough for some problems

Binary Search

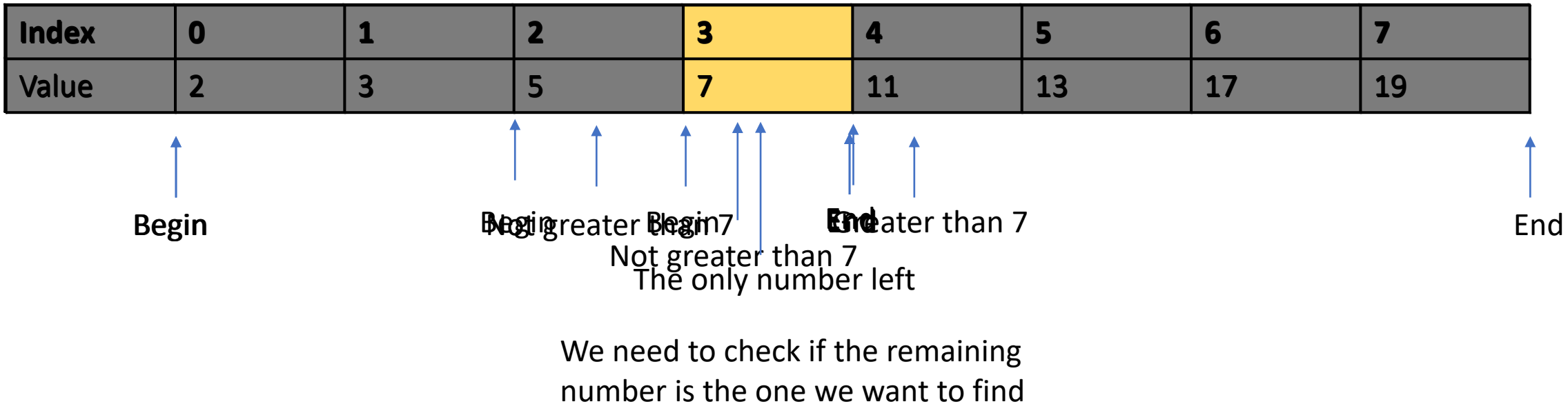
- Actually, we have a faster searching algorithm.
- But this time, we need some restrictions on the array.
- We this time require the array to be sorted.
  - i.e. the elements are in ascending (or descending) order (a[0] <= a[1] <= … <= a[n-1])

Binary Search

- At each iteration, we will check the element at the middle of the range that we are interested in.
- After that, we can know that whether the number we are searching is on the left half or right half of the range.
- So we can reduce the range by half each time.

Binary Search

- Example:
- Find any 7 in the following array



Binary Search

- For your reference, the code look likes this:

```
int lo = 0, hi = n;
while (lo+1 < hi) { // range: lo <= i < hi
    int mid = (lo + hi) / 2;
    if (a[mid] > x) {
        hi = mid;
    } else {
        lo = mid;
    }
}
if (a[lo] == x) {
    printf("%d is found at position %d.\n", x, lo);
} else {
    printf("%d is not found in a.\n", x);
}
```

- It will be better if you can write your own style of binary search, since this is very basic.

*Be clear with what the range that you are interested in is and beware of infinite loop

Binary Search

- It requires some simple argument to show that the above algorithm is correct, you may work the detail out yourself (and I recommend you to do so).
- Since every time, we reduce the range by half,
  we may want to know 2^?=n, it turns out ? = log_2(n)
- For the time complexity,
- Best: O(1) or O(lg n) (depends on implementation)
- Worst: O(lg n)
- Average: O(lg n)

Binary Search

- Applications
  - Check if an element exists in an array
  - Find its position if it exists
  - If it does not exists, find the position where it should be inserted
  - Find the smallest element >= x or > x
  - Find the largest element <= x or < x
  - The sample code provides two slides before can be used for finding the largest element <= x
    - Try verify this yourself (maybe by looking at some more examples)
    - Try to write the code for the three other cases (by slightly modify the given code)

Binary Search

- In C++, we have some useful functions (in algorithm) which uses binary search.
  - binary_search(begin, end, x)
    - Return whether x appears in the range [begin, end)
  - lower_bound(begin, end, x)
    - Return the pointer to leftmost element >= x
  - upper_bound(begin, end, x)
    - Return the pointer to leftmost element > x
- Be careful, all of the functions only work properly if the array is **sorted**

Binary Search - Binary Search on Answer

- As long as the elements in the array is in ascending order (or descending order), we don’t need to know every element of the array to find x, we only need O(lg n) of them.
- Therefore, sometimes we do not need to construct the whole array but only calculate the value when needed.

Binary Search - Binary Search on Answer

- Example, you are given x <= 10^18, find the smallest integer k such that k^2>=x.
  - It is trivial that the answer is ceil(sqrt(x)), but here we can also use binary search to find the answer
- We can consider this as a virtual array f[k] where
  - f[k] = 0 if k^2 < x
  - f[k] = 1 if k^2 >= x
- Writing out the array (if long enough) gives us the following:

  … 0 … 0 0 0 1 1 1 … 1 …
- The answer to the problem is just the index of leftmost 1

Binary Search - Binary Search on Answer

- Then we can just write a function to determine whether f[k] is 0 or 1 and then we can do the binary search base on this function.
- But again, be careful that the array must be **sorted**
  i.e. All 0's are before all 1's or vice versa

- Also you need to make sure that the initial range is large enough to contain any 1's (the true answer)

Binary Search - Binary Search on Answer

- Sample code for the above problem: (beware of the upper bound)

```
long long lo = 0, hi = 1000000001;
while (lo+1 < hi) { // range: lo <= i < hi
    long long mid = (lo + hi) / 2;
    if (check(mid, x)) {
        lo = mid;
    } else {
        hi = mid;
    }
}
```

- With the following check function

```
bool check (long long mid, long long x) {
    return mid * mid >= x;
}
```

Binary Search - Binary Search on Answer

- Suppose the time complexity of check function is O(f(n))
- Then the time complexity of binary search on answer is just O(f(n) lg (hi - lo))

Binary Search - More

- You may even extend the idea of the “virtual array” to a function from real to real, every things shall be kept the same but we should change the condition from (lo+1 < hi) to (lo+eps < hi) where eps is some very small number like 1e-6.

- There is also an algorithm in math very similar to binary search called bisection method.
  - That one does not require the function to be monotone but continuous, because there is already enough information that tell us whether to choose the left or right side.

Binary Search

- Here are some questions which can be solved using binary search or binary search on answer

HKOJ M1222

HKOJ S144

HKOJ M1714

Ternary Search

- This algorithm is used to find the minimum or maximum point of a **convex function**.

- This may not be very useful for OI contest since the discrete case can be easily done using binary search. (just binary search on the turning point)
  - Even the continuous version can be done using binary search actually
- So I will just talk about it roughly.

Ternary Search

- Suppose we want to find the minimum point
- We will divide the range into three part
- Let m1 be the one-third point, m2 be the two-third point
- The corresponding formula is m1 = lo + (hi-lo)/3, m2 = hi - (hi-lo)/3
- Compare f(m1) and f(m2)
  - If f(m1) > f(m2) then throw away the left one-third
  - If f(m1) < f(m2) then throw away the right one-third
  - If f(m1) = f(m2) then throw away both left and right one-third
- This algorithm only works if **both the increasing and decreasing part are strict**.

Ternary Search

- Time complexity: O(lg n), the same as binary search

- It is not recommended to use this algorithm during contest, try to rewrite it using binary search, there will be less bugs.

Conclusion - Searching

- Use different searching algorithm in different situations.
- Don’t need to repeat the search for many times -> Linear search (Sometimes, the problem may be optimized using sliding windows (appears in Optimizations) to reduce a linear factor)
- If the check function is some kind of “sorted array” -> Binary search

Short break

Sorting - introduction

- Rearrange the elements in an array to some order, usually ascending
- Sometimes, sorting is required as a preprocess
  e.g. binary search, greedy

- There are many sorting algorithms
  comparison based vs non-comparison based
  fast vs slow
  simple vs complicated

Selection Sort

- Always choose the smallest element from the unsorted part and put it at the back of the sorted part

- It is named because at every iteration, we are selecting the next element to insert.

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Selection Sort

- For your reference, the code looks like this:

```
for (int i = 0; i < n; i++) {
    int m = i;
    for (int j = i; j < n; j++) {
        if (a[j] < a[m]) {
            m = j;
        }
    }
    swap(i, m);
}
```

Selection Sort

- Worst Case Performance:
  - Swaps: O(n)
  - Comparisons: O(n^2)
  - Time complexity: O(n^2)

- Average Performance:
  - Swaps: O(n)
  - Comparisons: O(n^2)
  - Time complexity: O(n^2)

Selection Sort

- Advantage
  - It do the least number of swaps possible (O(n)), even less then the fast sorting algorithms.
  - O(1) extra space

- Disadvantage
  - Slow (O(n^2))

Insertion Sort

- We are maintaining a sorted prefix of the array.
- To add a new element into the sorted prefix of the array, we insert it into the place it should be.

- It is named because at every iteration, we are inserting the next element to the right place.

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Insertion Sort

- For your reference, the code looks like this:

```
for (int i = 0; i < n; i++) {
    for (int j = i; j > 0; j--) {
        if (a[j] < a[j - 1]) {
            swap(a[j], a[j - 1]);
        } else {
            break;
        }
    }
}
```

- Fun fact: number of swaps done = inversion of the original array.

Insertion Sort

- Worst Case Performance:
  - Swaps: O(n^2)
  - Comparisons: O(n^2)
  - Time complexity: O(n^2)

- Average Performance:
  - Swaps: O(n^2)
  - Comparisons: O(n^2)
  - Time complexity: O(n^2)

Insertion Sort

- Advantage
  - We can get the sorted version for every prefix
  - Fast (O(n)) if the initial array is almost sorted
  - O(1) extra space
- Disadvantage
  - Slow (O(n^2))

Bubble Sort

- Smaller elements will push larger elements to the end of the array so that the suffix will get the largest elements.

- I don’t quite know why it is called bubble sort.

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Bubble Sort

- For your reference, the code looks like this:

```
for (int i = 0; i < n; i++) {
    for (int j = 1; j + i < n; j++) {
        if (a[j] < a[j - 1]) {
            swap(a[j], a[j - 1]);
        }
    }
}
```

- Fun fact: number of swaps done = inversion of the original array.

Bubble Sort

- Worst Case Performance:
  - Swaps: O(n^2)
  - Comparisons: O(n^2)
  - Time complexity: O(n^2)

- Average Performance:
  - Swaps: O(n^2)
  - Comparisons: O(n^2)
  - Time complexity: O(n^2)

Bubble Sort

- Advantage
  - Easy to code
  - O(1) extra space
- Disadvantage
  - Slow (O(n^2))

```
for (int i = 0; i < n; i++) {
    for (int j = 1; j < n; j++) {
        if (a[j] < a[j - 1]) {
            swap(a[j], a[j - 1]);
        }
    }
}
```

- Usually bubble sort will be written like the above one instead of the one shown on previous page

Quick Sort

- This algorithm use the concept of Divide and Conquer.

- Let’s first assume that we magically have a function
  void quick_sort(int begin, int end)
  which can sort the elements in the array for range [begin, end)

- And then we need to implement our own quick_sort using the magic quick_sort with the restriction that we cannot pass our own parameter into the magic one. (this is like cheating)

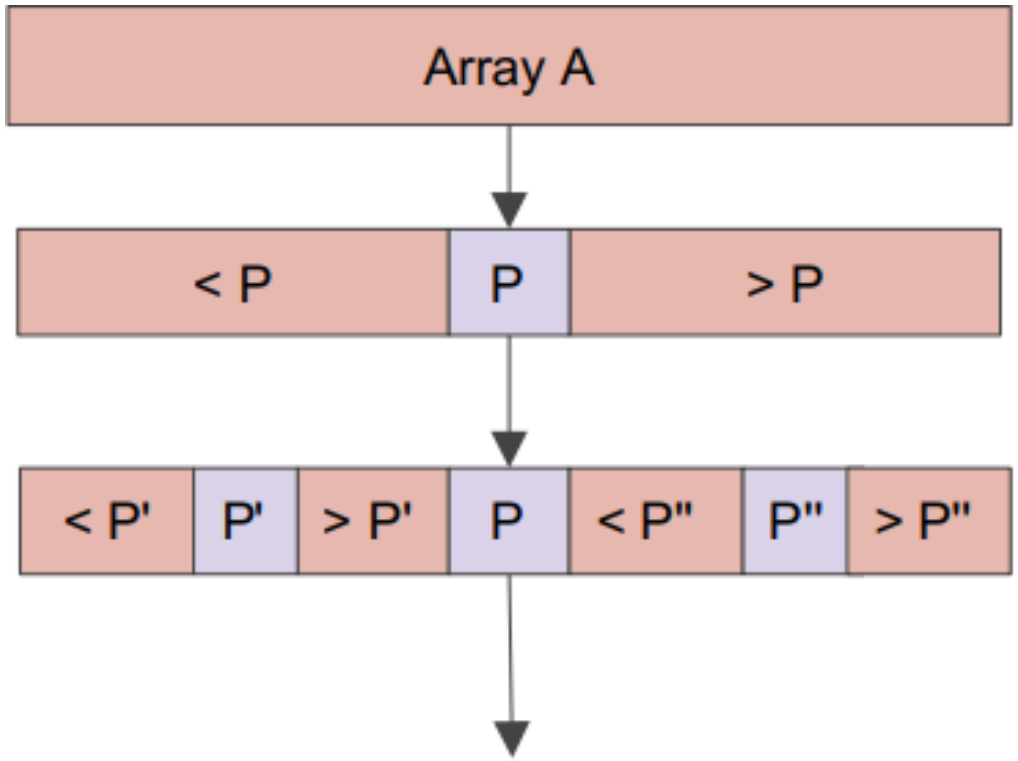
Quick Sort

- If begin + 1 = end, then there is only one element in the range which is automatically sorted, so without using the magical quick_sort, we can immediately return as the range is sorted already.

- Otherwise, we just arbitrarily pick a pivot (e.g. a[begin]).
- Then we put everything less than the pivot to its left, everything greater than or equal to the pivot to its right
- Sort both the left part and the right part using the magical quick_sort.
- Now we claim that we have done our job.

Quick Sort

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html



Quick Sort

- Here is my implementation: (you can and should come up with your own)

```
void quick_sort(int begin, int end) {
    if (begin + 1 == end) {
        return;
    }
    swap(a[begin], a[choose_pivot(begin, end)]);
    int l = 0, p = a[begin];
    for (int i = begin + 1; i < end; i++) {
        if (a[i] < p) {
            swap(a[i], a[begin + l + 1]);
            l++;
        }
    }
    swap(a[begin], a[begin + l]);
    quick_sort(begin, begin + l);
    quick_sort(begin + l + 2, end);
}
```

Quick Sort

- Worst Case Performance:
  - Swaps: O(n^2)
  - Comparisons: O(n^2)
  - Time complexity: O(n^2)

- Average Performance:
  - Swaps: O(n lg n)
  - Comparisons: O(n lg n)
  - Time complexity: O(n lg n)

Quick Sort

- **Worst Case Performance:**
  - **Swaps: O(n^2)**
  - **Comparisons: O(n^2)**
  - **Time complexity: O(n^2)**

- Average Performance:
  - Swaps: O(n lg n)
  - Comparisons: O(n lg n)
  - Time complexity: O(n lg n)

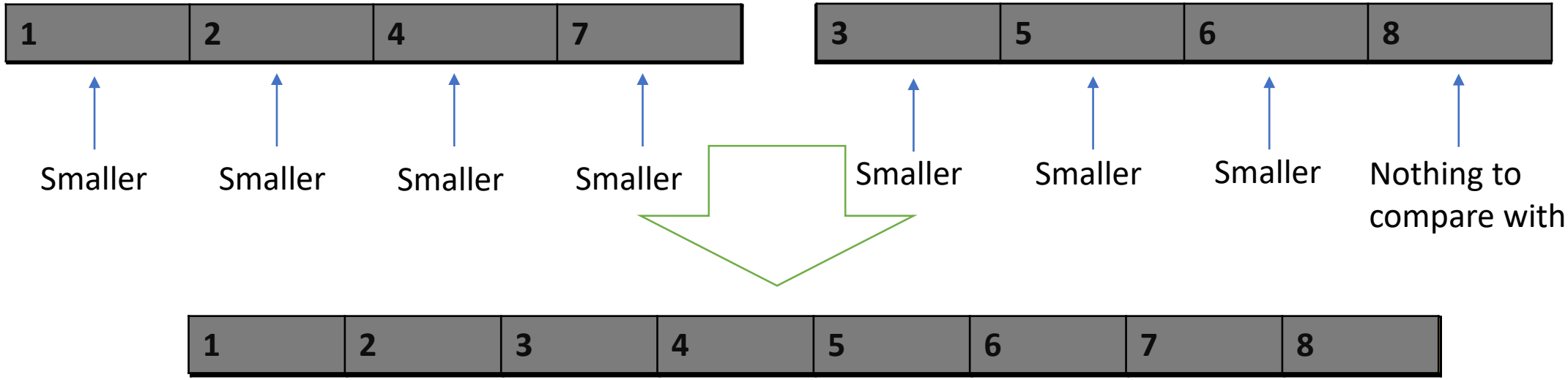
Merge Sort

- Before talking about merge sort, we need to think about how tow merge two sorted arrays into a single sorted array efficiently (in linear time).

- More simple question, if we are given two sorted arrays, how can we select the next smallest element to be inserted into the sorted array?

Merge Sort

- Consider the following example:



Merge Sort

- This algorithm again use the concept of Divide and Conquer.

- Let’s first assume that we magically have a function
  void merge_sort(int begin, int end)
  which can sort the elements in the array for range [begin, end)

- And then we need to implement our own merge_sort using the magic merge_sort with the restriction that we cannot pass our own parameter into the magic one. (this is like cheating)

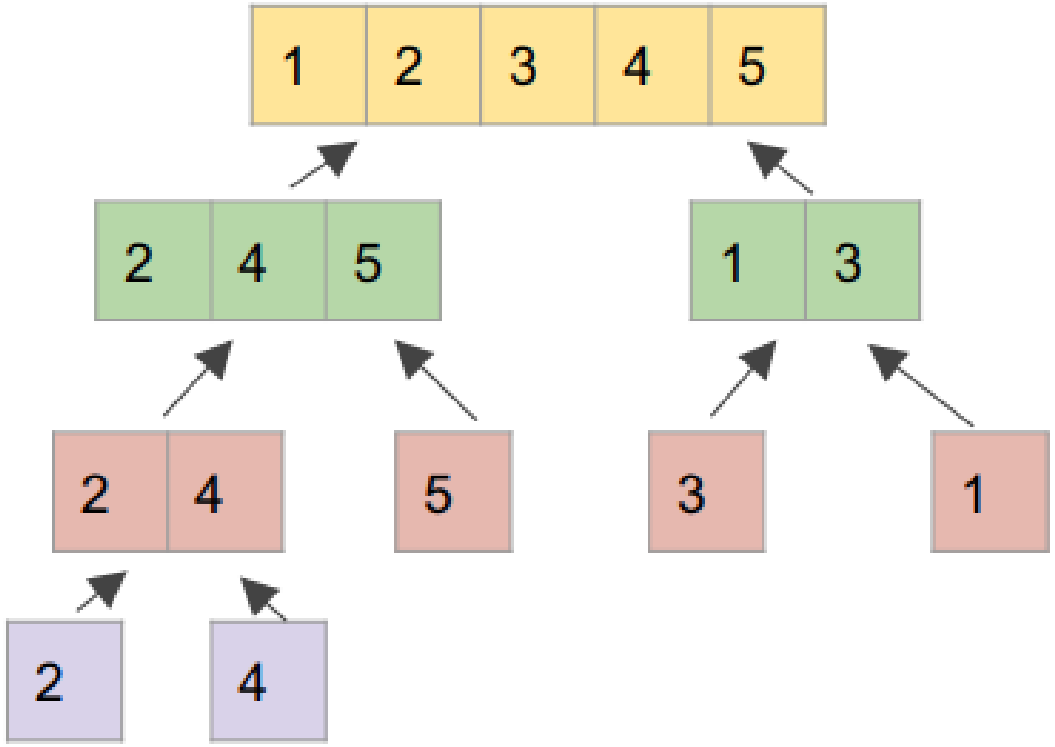
Merge Sort

- If begin + 1 = end, then there is only one element in the range which is automatically sorted, so without using the magical merge_sort, we can immediately return as the range is sorted already.

- Otherwise, we just split the array into two half
- Sort both halves using the magical merge_sort.
- After that, perform the merging as described before.
- Now we claim that we have done our job.

Merge Sort

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html



Merge Sort

- Here is my implementation: (you can and should come up with your own)

```
void merge_sort(int begin, int end) {
    if (begin + 1 == end) {
        return;
    }
    int mid = (begin + end) / 2;
    merge_sort(begin, mid);
    merge_sort(mid, end);
    int l = begin, r = mid;
    for (int i = begin; i < end; i++) {
        if (l == mid) {
            tmp[i] = a[r++];
        } else if (r == end) {
            tmp[i] = a[l++];
        } else if (a[l] < a[r]) {
            tmp[i] = a[l++];
        } else {
            tmp[i] = a[r++];
        }
    }
    for (int i = begin; i < end; i++) {
        a[i] = tmp[i];
    }
}
```

Merge Sort

- Worst Case Performance:
  - Swaps: O(n lg n)
  - Comparisons: O(n lg n)
  - Time complexity: O(n lg n)

- Average Performance:
  - Swaps: O(n lg n)
  - Comparisons: O(n lg n)
  - Time complexity: O(n lg n)

Inversions

- Given any permutation, we can define the inversions of the permutation to be the number of ordered tuple (i, j) such that i < j but a[i] > a[j]

- Your task:
  - Medium: To find the number of inversions using each of the 3 O(n^2) sorting algorithms
  - Difficult: To find the number of inversions using each of the 2 O(n lg n) sorting algorithms

Comparison Based Sorting

- It can been shown that O(n lg n) is a lower bound for a comparison based sorting algorithm, so we cannot find any comparison based sorting algorithm faster than O(n lg n)
  - You may have a look here if you are interested

- All the above 5 mentioned sorting algorithms are all comparison based, i.e. the comparison operators (< or >) is involved.
- We have some sorting algorithm that does not do any comparison, they are called non-comparison based sortings

Short break

Counting Sort

- This algorithm is useful when we have a lot of elements in an array but the possible types of elements are not many (e.g. small range for integers)

- We are actually not doing the sorting but creating several buckets and put the elements into the corresponding buckets.
- We are converting an array into a frequency table.
- And then go through the frequency table once and report form the smallest element to largest element according to their frequency.

Counting Sort

- Consider the following example:

Array A

2	4	5	1	3	1

Frequency Table

Index	1	2	3	4	5
Count	2	1	1	1	1

Sorted Array A

1	1	2	3	4	5

Counting Sort

- Implementation 1

```
for (int i = 0; i < n; i++) {
    cnt[a[i]]++;
}
for (int i = lo, j = 0; i < hi; i++) {
    for (int k = 0; k < cnt[i]; k++) {
        a[j++] = i;
    }
}
```

Counting Sort

- Implementation 2

```
for (int i = 0; i < n; i++) {
    cnt[a[i]]++;
}
for (int i = lo; i < hi; i++) {
    cnt[i] += cnt[i-1];
}
for (int i = 0; i < n; i++) {
    tmp[--cnt[a[i]]] = a[i];
}
for (int i = 0; i < n; i++) {
    a[i] = tmp[i];
}
```

Counting Sort

- Suppose the array size is N and the number of possible elements are M.
- Then the time complexity of counting sort is just O(N+M)
- The space complexity of counting sort is also O(N+M)

- So this is only applicable when M is small.
- If M is too large, then counting sort will be slow and use a lot of memory.

Radix Sort

- This can be used for sorting integers.
- This is like multiple rounds of counting sort

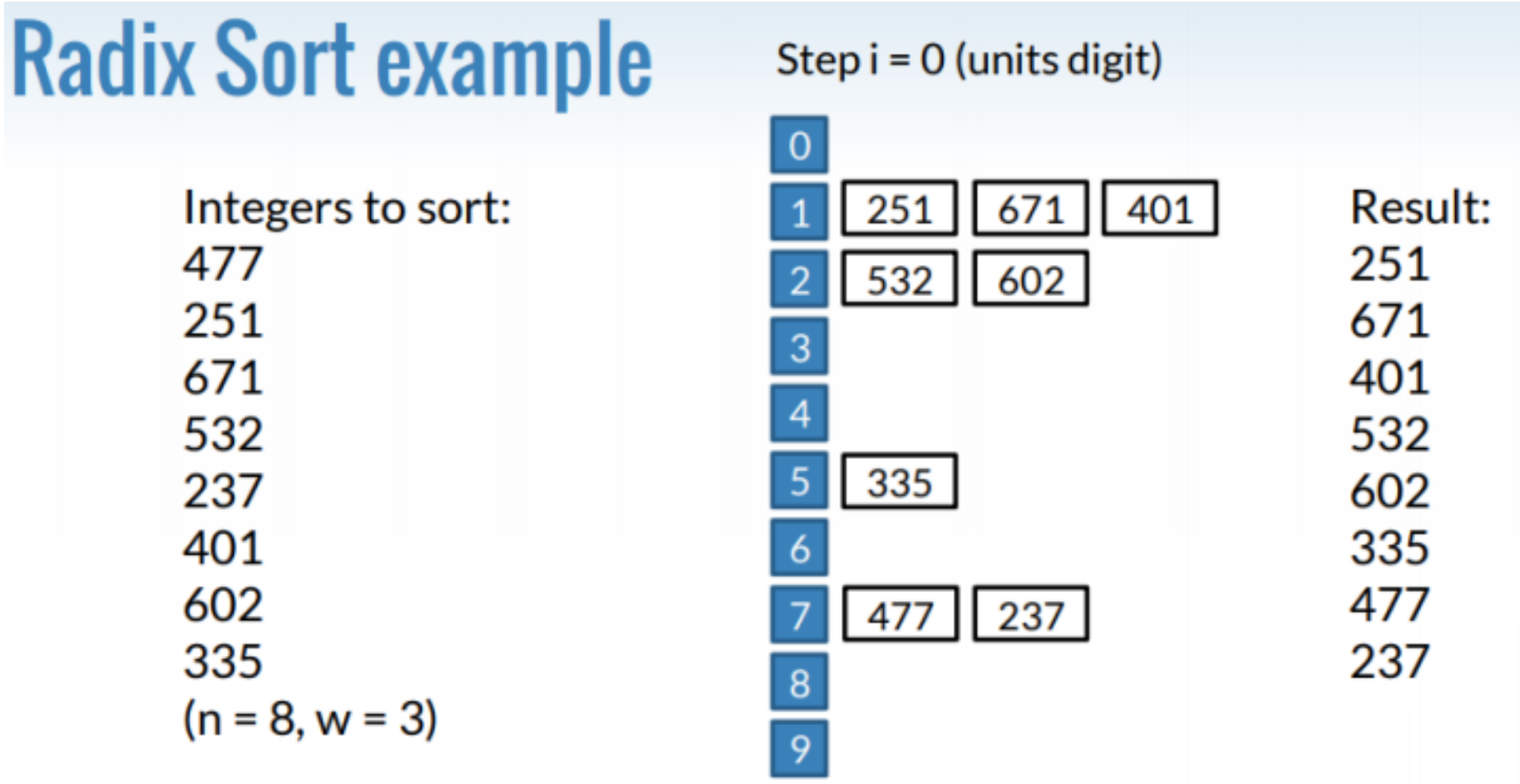
  the first round is on ones-digit
  the second round is on tens-digit
  …

- The algorithm will sort from the least significant digit to the most significant digit

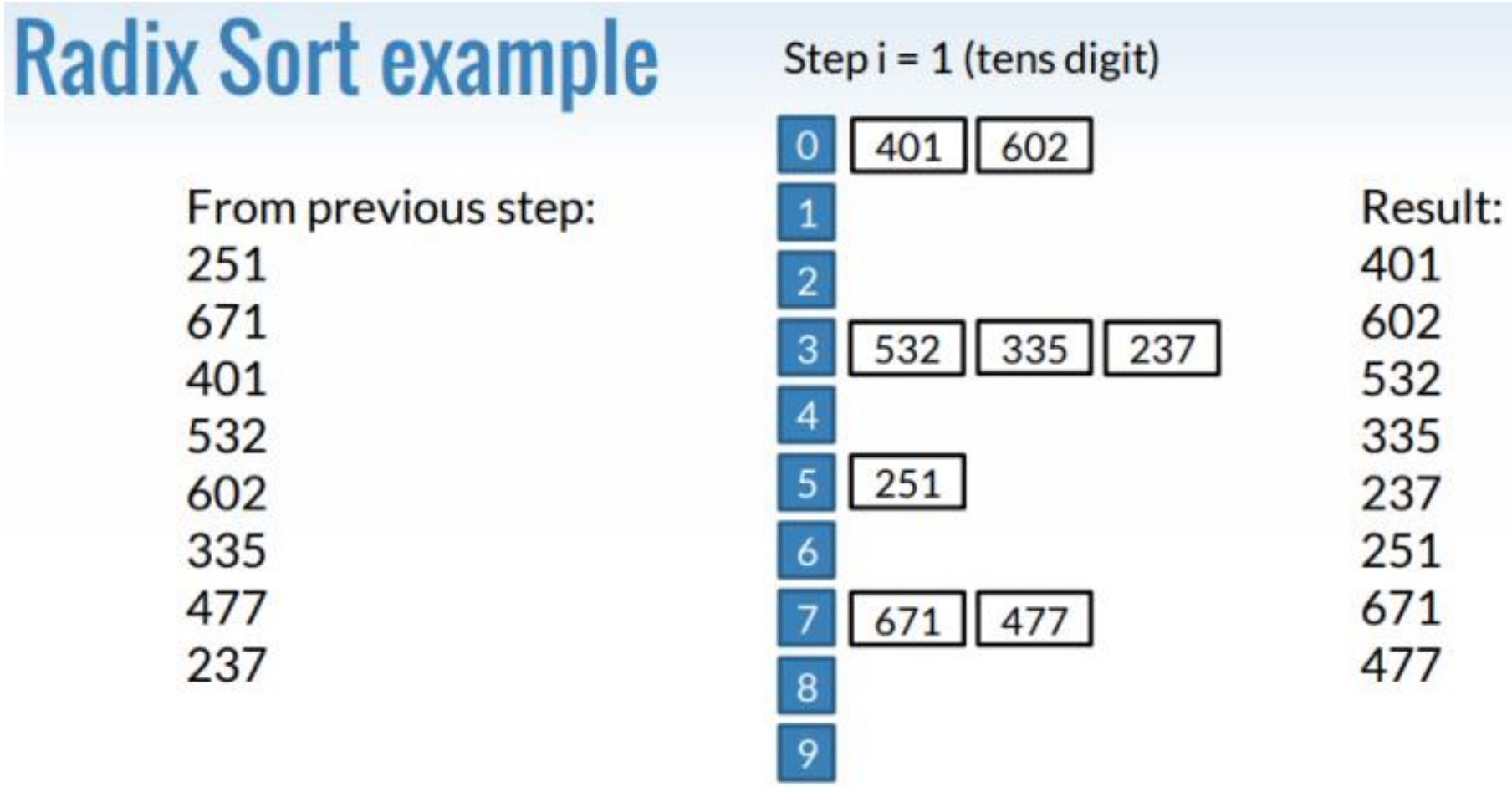
Radix Sort

- At the i-th round of iteration, we are doing counting sort for the i-th least significant digit.
- After that, we get the result from the bucket one by one adopting first-in-first-out principle (aka. queue).
- Repeat this process and we can sort the numbers.

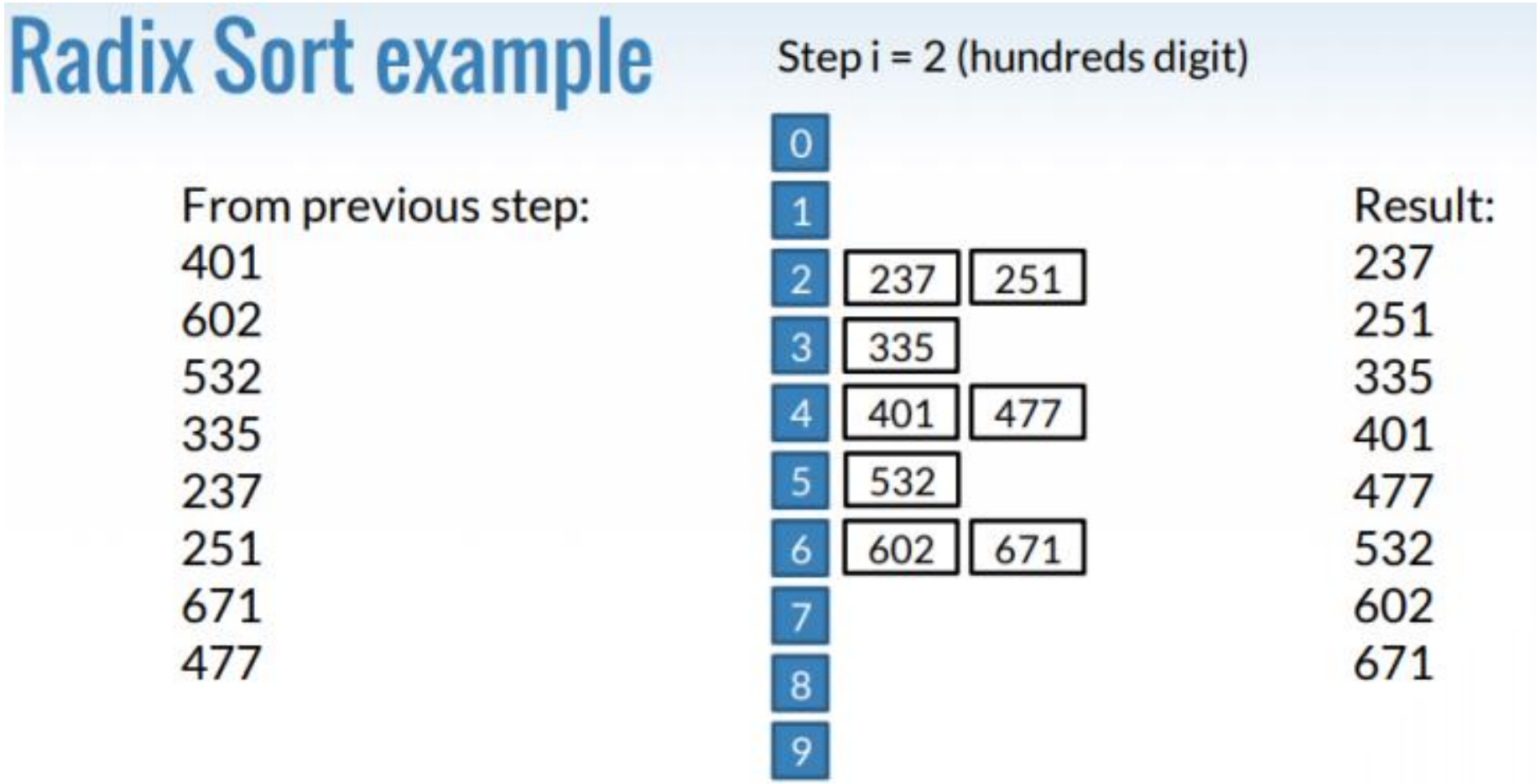
Radix Sort


Radix Sort example
Integers to sort:
477
251
671
532
237
401
602
335
(n = 8, w = 3)
Step i = 0 (units digit)
0
1
251
671
401
2
532
602
3
4
5
335
6
7
477
237
8
9
Result:
251
671
401
532
602
335
477
237

Radix Sort


Radix Sort example
Step i = 1 (tens digit)
From previous step:
251
671
401
532
602
335
477
237
0
401
602
1
2
3
532
335
237
4
5
251
6
7
671
477
8
9
Result:
401
602
532
335
237
251
671
477

Radix Sort


Radix Sort example
Step i = 2 (hundreds digit)
From previous step:
401
602
532
335
237
251
671
477
0
1
2
237
251
3
335
4
401
477
5
532
6
602
671
7
8
9
Result:
237
251
335
401
477
532
602
671

Radix Sort

- Implementation 1

```
for (int i = 0; i < w; i++) {
    for (int j = 0; j < m; j++) {
        buckets[m].clear() = 0;
    }
    for (int j = 0; j < n; j++) {
        buckets[digit(a[j], i)].push_back(a[j]);
    }
    for (int j = 0, k = 0; j < m; j++) {
        for (auto x: buckets[j]) {
            a[k++] = x;
        }
    }
}
```

Radix Sort

- Implementation 2

```
for (int i = 0; i < w; i++) {
    for (int j = 0; j < m; j++) {
        cnt[m] = 0;
    }
    for (int j = 0; j < n; j++) {
        cnt[digit(a[j], i)]++;
    }
    for (int j = 1; j < m; j++) {
        cnt[j] += cnt[j-1];
    }
    for (int j = n - 1; j >= 0; j--) {
        tmp[--cnt[digit(a[j], i)]] = a[j];
    }
    for (int j = 0; j < n; j++) {
        a[j] = tmp[j];
    }
}
```

STL Sort

- We can use the sort provided by the library algorithm.
- In which we can write a customized compare function to pass.
- To sort the array in descending order, you just need to define the following function and call sort(a, a + n, cmp);

```
bool cmp (int x, int y) {
    return x > y;
}
```

- After C++11, it even support lambda functions, so now you can write

```
sort(a, a + n, [] (int x, int y) {
    return x > y;
});
```