CS 106B, Lecture 26
Sorting

Plan for Today

- Sorting!
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort (You are not responsible for this one on the exam)

Sorting

- **sorting**: Rearranging the values in a collection into a specific order.
  - *can be solved in many ways:*

Algorithm	Description
bogo sort	shuffle and pray
bubble sort	swap adjacent pairs that are out of order
selection sort	look for the smallest element, move to front
insertion sort	build an increasingly large sorted front portion
merge sort	recursively divide the data in half and sort it
heap sort	place the values into a binary heap then dequeue
quick sort	recursively "partition" data based on a pivot value
bucket sort	cluster elements into smaller groups, sort the groups
radix sort	sort integers by last digit, then 2nd to last, then ...

Selection sort example

- **selection sort**: Repeatedly swap smallest unplaced value to front.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	**-4**	18	12	**22**	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	**2**	12	22	27	30	36	50	7	68	91	56	**18**	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	**7**	22	27	30	36	50	**12**	68	91	56	18	85	42	98	25

Selection sort code

```
// Rearranges elements of v into sorted order.
void selectionSort(Vector<int>& v) {
    for (int i = 0; i < v.size() - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < v.size(); j++) {
            if (v[j] < v[min]) {
                min = j;
            }
        }
        // swap smallest value to proper place, v[i]
        if (i != min) {
            int temp = v[i];
            v[i] = v[min];
            v[min] = temp;
        }
    }
}
```

Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert second item into sorted sublist, shifting first item if needed
  - insert third item into sorted sublist, shifting items 1-2 as needed
  - ...
  - repeat until all values have been inserted into their proper positions

Insertion sort example

- Makes N-1 passes over the array.
- At the end of pass i, the elements that occupied $A[0]...A[i]$ originally are still in those spots and in sorted order.

index	0	1	2	3	4	5	6	7
value	15	2	8	1	17	10	12	5
pass 1	**2**	**15**	8	1	17	10	12	5
pass 2	**2**	**8**	**15**	1	17	10	12	5
pass 3	**1**	**2**	**8**	**15**	17	10	12	5
pass 4	**1**	**2**	**8**	**15**	**17**	10	12	5
pass 5	**1**	**2**	**8**	**10**	**15**	**17**	12	5
pass 6	**1**	**2**	**8**	**10**	**12**	**15**	**17**	5
pass 7	**1**	**2**	**5**	**8**	**10**	**12**	**15**	**17**

Insertion sort code

```
// Rearranges the elements of v into sorted order.
void insertionSort(Vector<int>& v) {
    for (int i = 1; i < v.size(); i++) {
        int temp = v[i];

        // slide elements right to make room for v[i]
        int j = i;
        while (j >= 1 && v[j - 1] > temp) {
            v[j] = v[j - 1];
            j--;
        }
        v[j] = temp;
    }
}
```

Announcements

- Final is **this Friday**, at 12:15PM
  - Practice exam has been released
- Assn. 7 is due Thursday. You can not use late days!
  - Finish it early
- Course Evaluations
  - Please fill out your evaluation for this class. We really appreciate the feedback!

Merge sort

- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

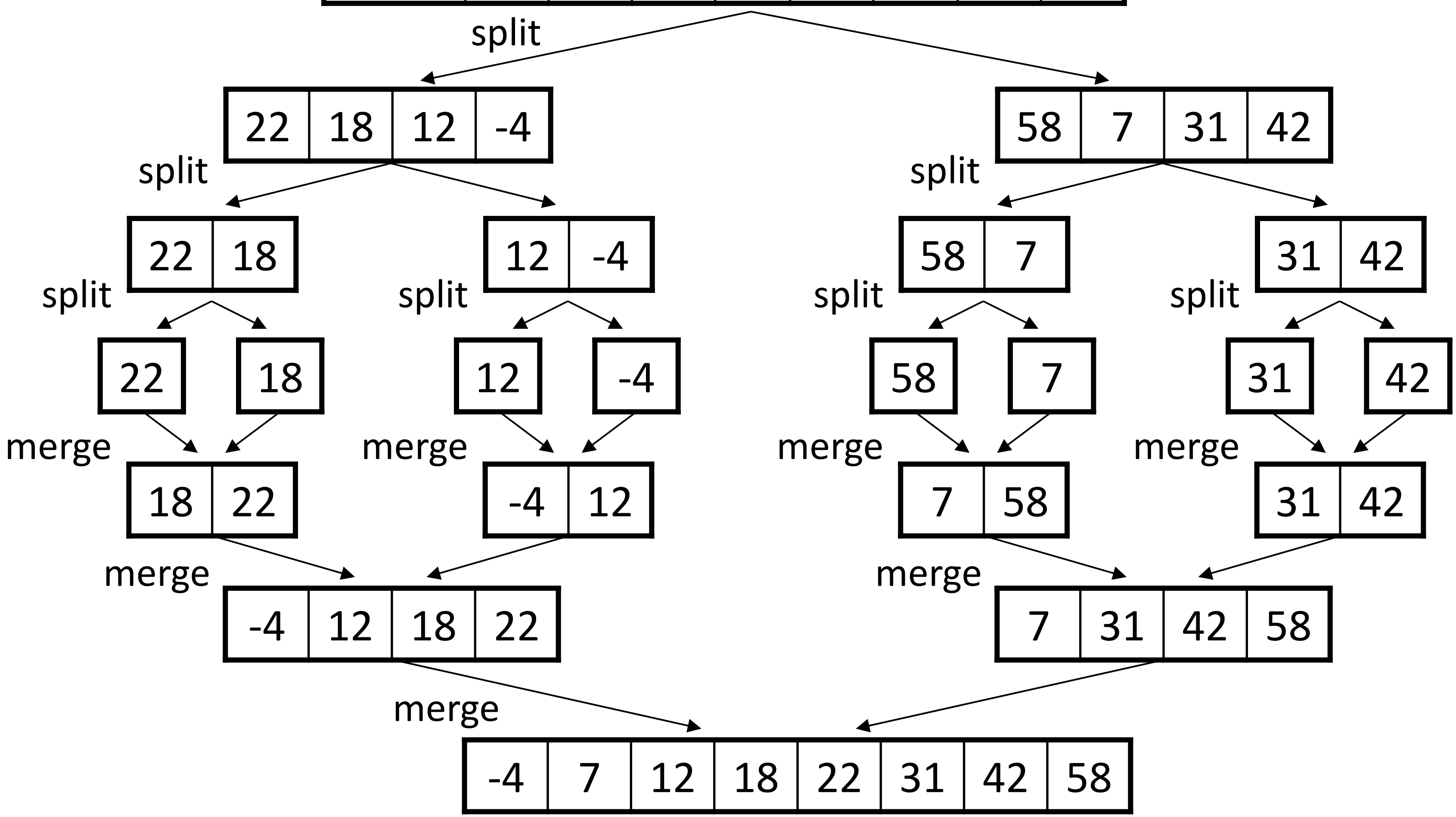
  The algorithm:
  - Divide the list into two roughly equal halves.
  - Sort the left half.
  - Sort the right half.
  - Merge the two sorted halves into one sorted list.

  - Often implemented recursively.
  - An example of a "divide and conquer" algorithm.
    - Invented by John von Neumann in 1945

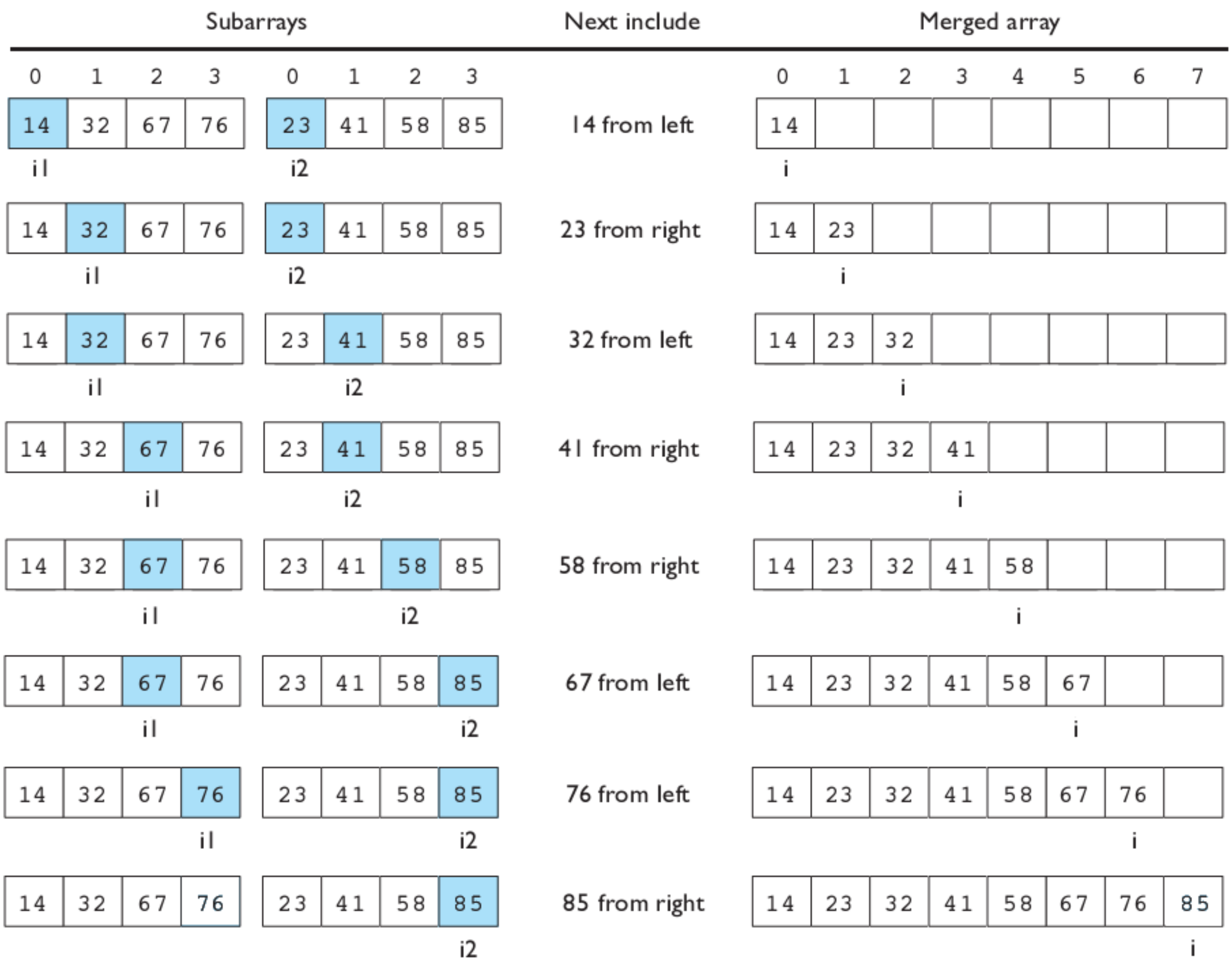
  - Runtime: **O(*N* log *N*)**. Somewhat faster for asc/descending input.

Merge sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



Merging sorted halves



Merge sort code

```
// Rearranges the elements of v into sorted order using
// the merge sort algorithm.
void mergeSort(Vector<int>& v) {
    if (v.size() >= 2) {
        // split vector into two halves
        Vector<int> left = v.subList(0, v.size() / 2);
        Vector<int> right = v.subList(v.size() / 2 + 1);

        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        v.clear();
        merge(v, left, right);
    }
}
```

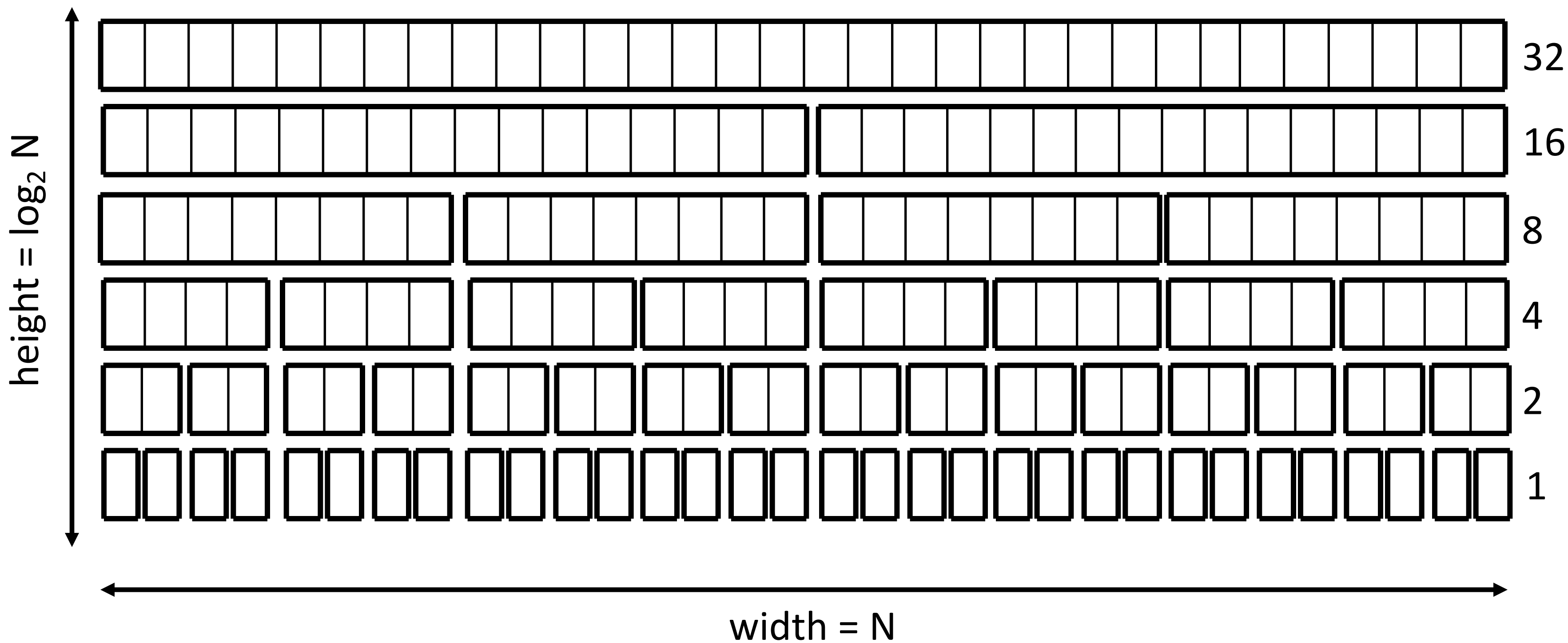
Merge halves code

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
void merge(Vector<int>& result,
           Vector<int>& left, Vector<int>& right) {
    int leftIndex = 0;
    int rightIndex = 0;

    for (int i = 0; i < left.size() + right.size(); i++) {
        if (rightIndex >= right.size() ||
           (leftIndex < left.size() &&
            left[leftIndex] <= right[rightIndex])) {
            result += left[leftIndex];    // take from left
            leftIndex++;
        } else {
            result += right[rightIndex];   // take from right
            rightIndex++;
        }
    }
}
```

Runtime intuition

- Merge sort performs O(N) operations on each level. *(width)*
  - Each level splits the data in 2, so there are $\log_2 N$ levels. *(height)*
  - Product of these = $N * \log_2 N = O(N \log N)$. *(area)*
  - Example: $N = 32$. Performs ~ $\log_2 32 = 5$ levels of N operations each:



Quick sort

- **quick sort**: Orders a list of values by partitioning the list around one element called a *pivot*, then sorting each partition.
  - invented by British computer scientist C.A.R. Hoare in 1960

- Quick sort is another divide and conquer algorithm:
  - Choose one element in the list to be the pivot.
  - *Divide* the elements so that all elements less than the pivot are to its left and all greater (or equal) are to its right.
  - *Conquer* by applying quick sort (recursively) to both partitions.

- Runtime: **O(*N* log *N*)** average, but O(N^2) worst case.
  - Generally somewhat faster than merge sort.

Choosing a "pivot"

- The algorithm will work correctly no matter which element you choose as the pivot.
  - A simple implementation can just use the first element.

- But for efficiency, it is better if the pivot divides up the array into roughly equal partitions.
  - What kind of value would be a good pivot? A bad one?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

Partitioning an array

- Swap the pivot to the last array slot, temporarily.
- Repeat until done partitioning (until *i*,*j* meet):
  - Starting from *i* = 0, find an element a[*i*] ≥ pivot.
  - Starting from *j* = *N*-1, find an element a[*j*] ≤ pivot.
  - These elements are out of order, so swap a[*i*] and a[*j*].
- Swap the pivot back to index *i* to place it between the partitions.

index	0	1	2	3	4	5	6	7	8	9
value	6	1	4	9	0	3	5	2	7	8
	8 i							←	j	6
	2	i	→	→			j	8		
				5	i	→	9			
							6			9
	2	1	4	5	0	3	6	8	7	9

Quick sort example

index	0	1	2	3	4	5	6	7	8	9	
value	**65**	23	81	43	92	39	57	16	75	32	choose pivot=65
	32	23	81	43	92	39	57	16	75	**65**	swap pivot (65) to end
	32	23	**16**	43	92	39	57	**81**	75	65	swap 81, 16
	32	23	16	43	**57**	39	**92**	81	75	65	swap 57, 92
	32	23	16	43	57	39	92	81	75	65	
	32	23	16	43	57	39	**65**	81	75	**92**	swap pivot back in

recursively quicksort each half

32	23	16	43	57	39	pivot=32
39	23	16	43	57	**32**	swap to end
16	23	**39**	43	57	32	swap 39, 16
16	23	**32**	43	57	**39**	swap 32 back in

81	75	92	pivot=81
92	75	**81**	swap to end
75	92	81	swap 92, 75
75	**81**	**92**	swap 81 back in

...

...

Quick sort code

```
void quickSort(Vector<int>& v) {
    quickSortHelper(v, 0, v.size() - 1);
}

void quickSortHelper(Vector<int>& v, int min, int max) {
    if (min >= max) {  // base case; no need to sort
        return;
    }

    // choose pivot; we'll use the first element (might be bad!)
    int pivot = v[min];
    swap(v, min, max);      // move pivot to end

    // partition the two sides of the array
    int middle = partition(v, min, max - 1, pivot);

    swap(v, middle, max);   // restore pivot to proper location

    // recursively sort the left and right partitions
    quickSortHelper(v, min, middle - 1);
    quickSortHelper(v, middle + 1, max);
}
```

Partition code

```
// Partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
int partition(Vector<int>& v, int i, int j, int pivot) {
    while (i <= j) {
        // move index markers i,j toward center
        // until we find a pair of out-of-order elements
        while (i <= j && v[i] < pivot) { i++; }
        while (i <= j && v[j] > pivot) { j--; }

        if (i <= j) {
            swap(v, i++, j--);
        }
    }
    return i;
}

// Moves the value at index i to index j, and vice versa.
void swap(Vector<int>& v, int i, int j) {
    int temp = v[i];  v[i] = v[j];  v[j] = temp;
}
```

Choosing a better pivot

- Choosing the first element as the pivot leads to very poor performance on certain inputs (ascending, descending)
  - does not partition the array into roughly-equal size chunks

- Alternative methods of picking a pivot:
  - *random:* Pick a random index from [*min* .. *max*]
  - *median-of-3:* look at left/middle/right elements and pick the one with the medium value of the three:
    - `v[min]`, `v[(max+min)/2]`, and `v[max]`
    - better performance than picking random numbers every time
    - provides near-optimal runtime for almost all input orderings

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	91	-4	27	30	86	50	65	78	5	56	2	25	42	98	**31**