Practical Numerical Training UKNum: Sorting

Klahr & Bitsch

Max Planck Institute for Astronomy, Heidelberg

Program:

1) Bubble Sort
2) Straight insertion
3) Shell Sort
4) Quick Sort
5) Movie 15 ways to sort…

Sorting

- Key operation to arrange large data-sets (e.g. Google)
- Indispensable to find data in data sets (search algorithm assume sorted data)
- Very different algorithms possible (very slow $\sim N^2$, fast $N \log(N)$)
  - Bubble Sort
  - Straight Insertion
  - Shell Sort
  - Quick Sort
  - Heapsort
  - ...

Bubble Sort

```
/* bubble sort function, sorts elements v[0..n-1] */
void bubble_sort (float v[], int n) {
        int     i,              /* array index */
                swapped;        /* true if we have swapped */

        do {
                /* we have not swapped yet */

                swapped = 0;

                /* go through array, looking for out of order elements */

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

                        /* if v[i-1] and v[i] are out of order... */

                        if (v[i-1] > v[i]) {

                                /* swap them */

                                swap (v, i, i-1);

                                /* and remember to go through the loop again */

                                swapped = 1;
                        }
        } while (swapped);
}
```

loop over entire array !

again if one element was swapped

https://www.youtube.com/watch?v=lyZQPjUT5B4

Bubble Sort

- The *best case*. In the best case, the array is already sorted and `bubble_sort` simply checks that it is sorted and exits. This is n-1 = $\Omega(n)$ comparisons.
- The *worst case*. In the worst case, the element that is supposed to be first is actually last, so that the `do/while` loop must run n times while the value "bubbles" up to the first array index. Each time through the `do/while` loop, n-1 comparisons are done. This is $O(n(n\text{-}1)) = O(n^2)$ comparisons.
- The *average case*. This is the average running time, averaged over all possible initial orderings of the array. The analysis is somewhat more complicated, but the result is still $O(n^2)$ comparisons.

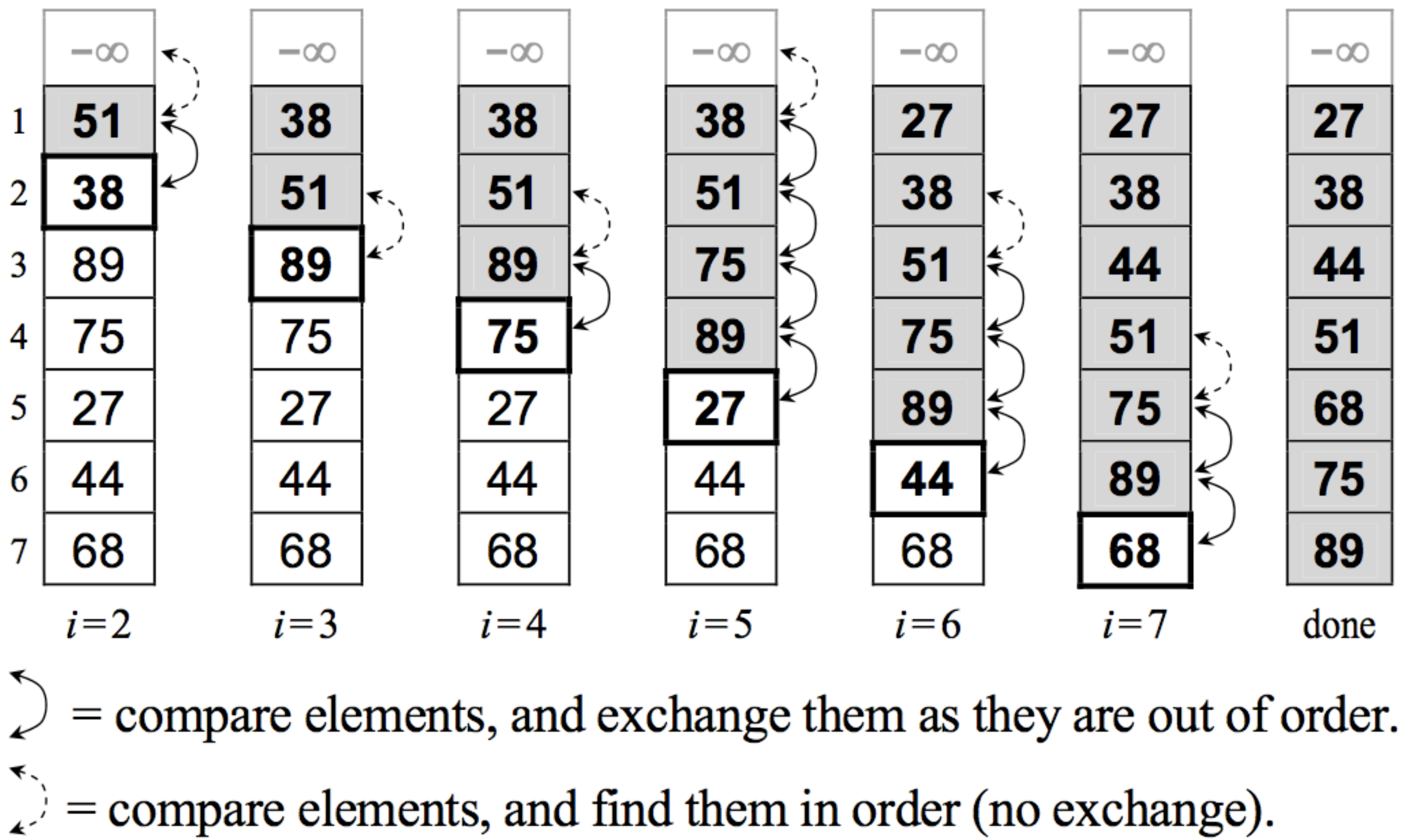
Bubble sort is pretty inefficient because of this quadratic running time; we can do a lot better than $O(n^2)$.

If you know what bubble sort is, wipe it from your mind;
if you don't know, make a point of never finding out!
Press et al., Numerical Recipes

from http://www.cs.utexas.edu/users/djimenez/utsa/cs1723/lecture12.html

Straight insertion

“card players method”



from http://www.math.uic.edu/~leon/cs-mcs401-s08

Straight insertion

Input: An array A with element type T, and integers p and r with $lowerbound(A) \le p \le r \le upperbound(A)$.

Output: The array A, with $A[p..r]$ sorted, and any remaining elements of A unchanged.

Algorithm (implemented using exchanges)

```
void straight_insertion_sort( T[] A, Integer p, Integer r)
    for ( i = p+1, p+2, ..., r )                // Insert A[i] into already
        j = i;                                  //  sorted subarray A[p..i–1].
        while ( j > p and A[j–1] > A[j] ) look for the place to insert A[i]
            swap( A[j–1], A[j]);
            j = j–1;
```

from http://www.math.uic.edu/~leon/cs-mcs401-s08

- `#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;`
-

Straight insertion

Scaling: N^2

quite slow!

only practicable for small N < 20

but scales as ~ N for sorted lists

Shell* Sort

- first sort numbers spaced by *increment* d using the straight sort algorithm
- reduce increment d until $\mathrm{d} = 1$ (*diminishing increment method*)

Advantage: straight insertion gets pre-sorted lists

*by Donald L. Shell

Shell Sort

Example: $N=16$ numbers $(n_1, n_2, \ldots n_{16})$
initial increment $d=N/2 = 8$

1. sort 8 lists of 2: $(n_1,n_9), (n_2,n_{10}), \ldots, (n_8,n_{16})$
2. sort 4 lists of 4: (halve increment)
$(n_1,n_5,n_9,n_{13}), (n_2,n_6,n_{10},n_{14}), \ldots, (n_4,n_8,n_{12},n_{16})$
3. sort 2 lists of 8: $(n_1,n_3,n_5, n_7,n_9, n_{11},n_{13},n_{15}), \ldots$
4. sort last list of 16: very much pre-ordered

Shell Sort

- Choice of increments determines speed of method
- Best choice not known
- Much better choice of increments (rather than N/2, ..., 8, 4, 2, 1) is

$$(3^k-1)/2, ..., 40, 13, 4, 1$$

guarantees order $N^{3/2}$ scaling in the worst case and $N^{1.25}$ scaling on average *(Knuth, The Art of Computer Programming)*

Shell Sort

```
void shell(unsigned long n, float a[])
```

Sorts an array `a[1..n]` into ascending numerical order by Shell's method (diminishing increment sort). **n** is input; **a** is replaced on output by its sorted rearrangement.

```
{
    unsigned long i,j,inc;
    float v;
    inc=1;                          Determine the starting increment.
    do {
        inc *= 3;
        inc++;
    } while (inc <= n);
    do {                            Loop over the partial sorts.
        inc /= 3;
        for (i=inc+1;i<=n;i++) {    Outer loop of straight insertion.
            v=a[i];
            j=i;
            while (a[j-inc] > v) {  Inner loop of straight insertion.
                a[j]=a[j-inc];
                j -= inc;
                if (j <= inc) break;
            }
            a[j]=v;
        }
    } while (inc > 1);
}
```

Press et al., Numerical Recipes

Quicksort

by C.A.R. Hoare

on *average* the fastest sorting algorithm

partition-exchange sorting method

1. select a partition (pivot) element **a** is from the list
2. pairwise exchange of elements lead to two lists
   a is in its final position in the list
   all elements in the left sub-list are ≤ **a**
   all elements in the right sub-list are ≥ **a**
3. repeat partitioning on both lists independently

Quicksort

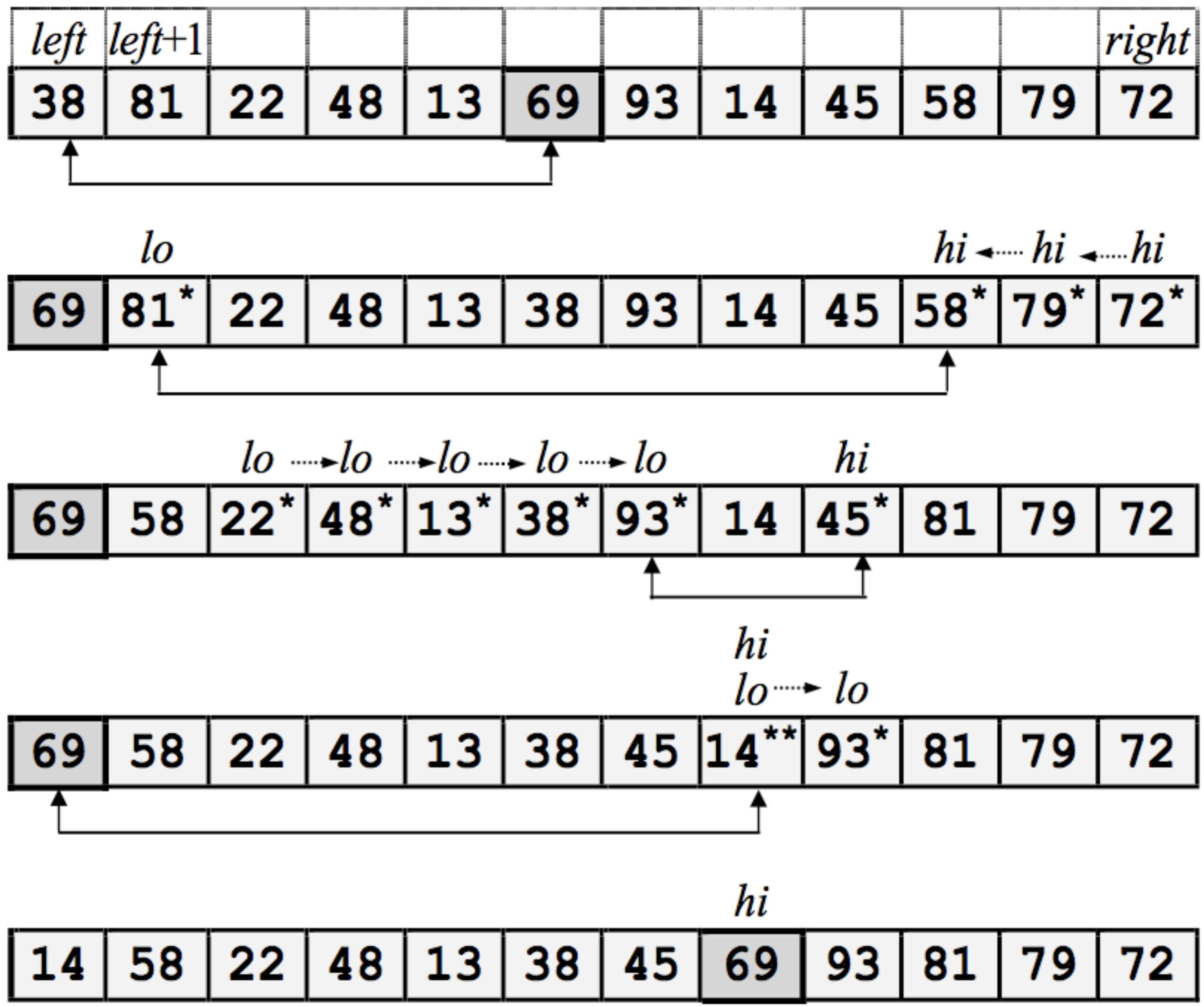
Swap pivot element with leftmost element. *lo*=*left*+1; *hi*=*right*;

Move *hi* left and *lo* right as far as we can; then swap *A*[*lo*] and *A*[*hi*], and move *hi* and *lo* one more position.

Repeat above

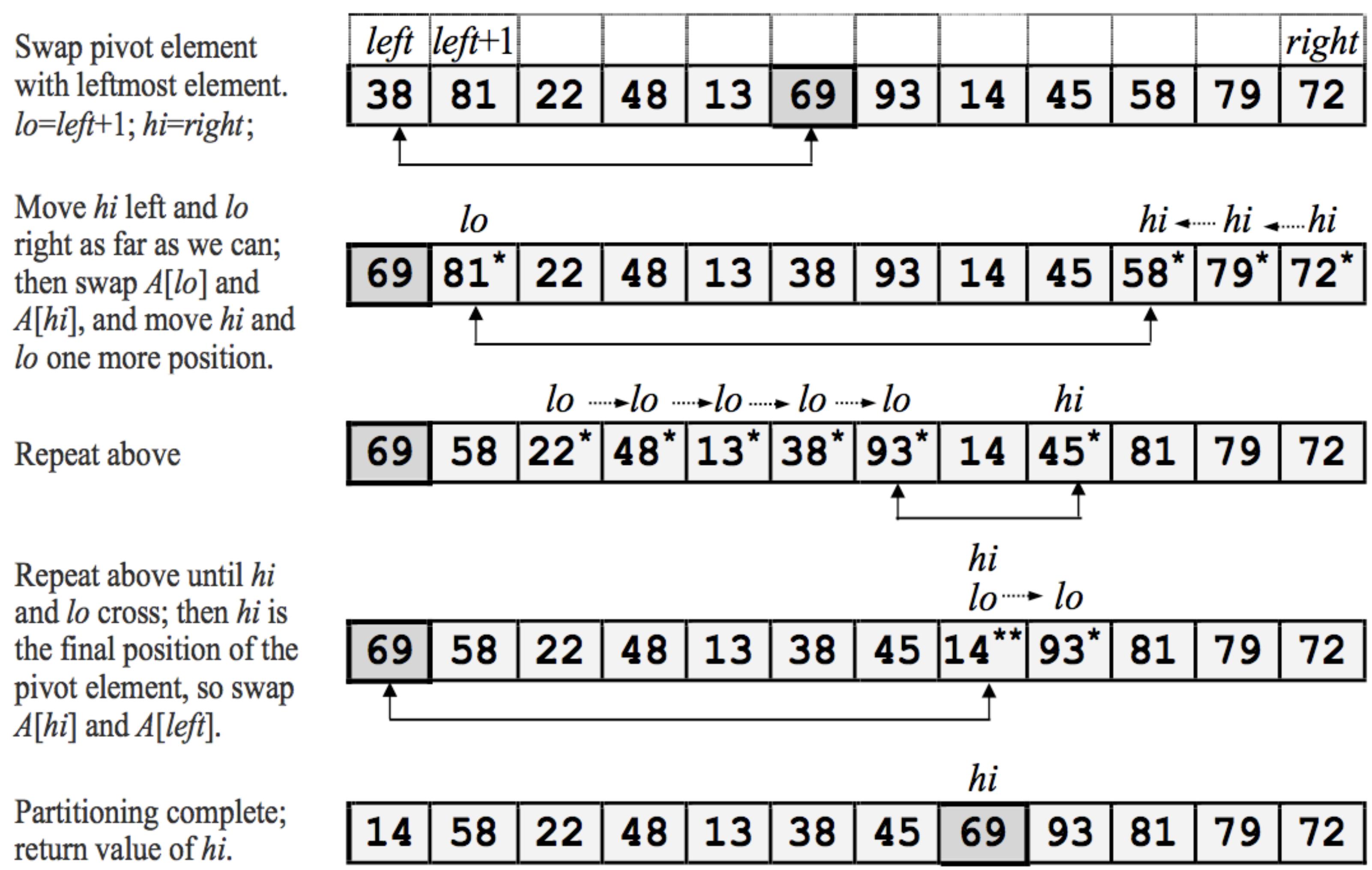
Repeat above until *hi* and *lo* cross; then *hi* is the final position of the pivot element, so swap *A*[*hi*] and *A*[*left*].

Partitioning complete; return value of *hi*.



from http://www.math.uic.edu/~leon/cs-mcs401-s08

Quicksort



divides the input list into two sub-arrays where

$$A[\text{list1}] \leq a \leq A[\text{list2}]$$

a = pivot element

from http://www.math.uic.edu/~leon/cs-mcs401-s08

Quicksort

```
Integer partition( T[] A, Integer left, Integer right)
    m = ⌊left+right⌋ / 2;
    swap( A[left], A[m]);
    pivot = A[left];
    lo = left+1; hi = right;
    while ( lo ≤ hi )
        while ( A[hi] > pivot )
            hi = hi – 1;
        while ( lo ≤ hi and A[lo] ≲ pivot )
            lo = lo + 1;
        if ( lo ≤ hi )
            swap( A[lo], A[hi]);
            lo = lo + 1; hi = hi – 1;
    swap( A[left], A[hi]);
    return hi
```

```
void quicksort( T[] A, Integer left, Integer right)
    if ( left < right )
        q = partition( A, left, right);
        quicksort( A, left, q–1);
        quicksort( A, q+1, right);
```

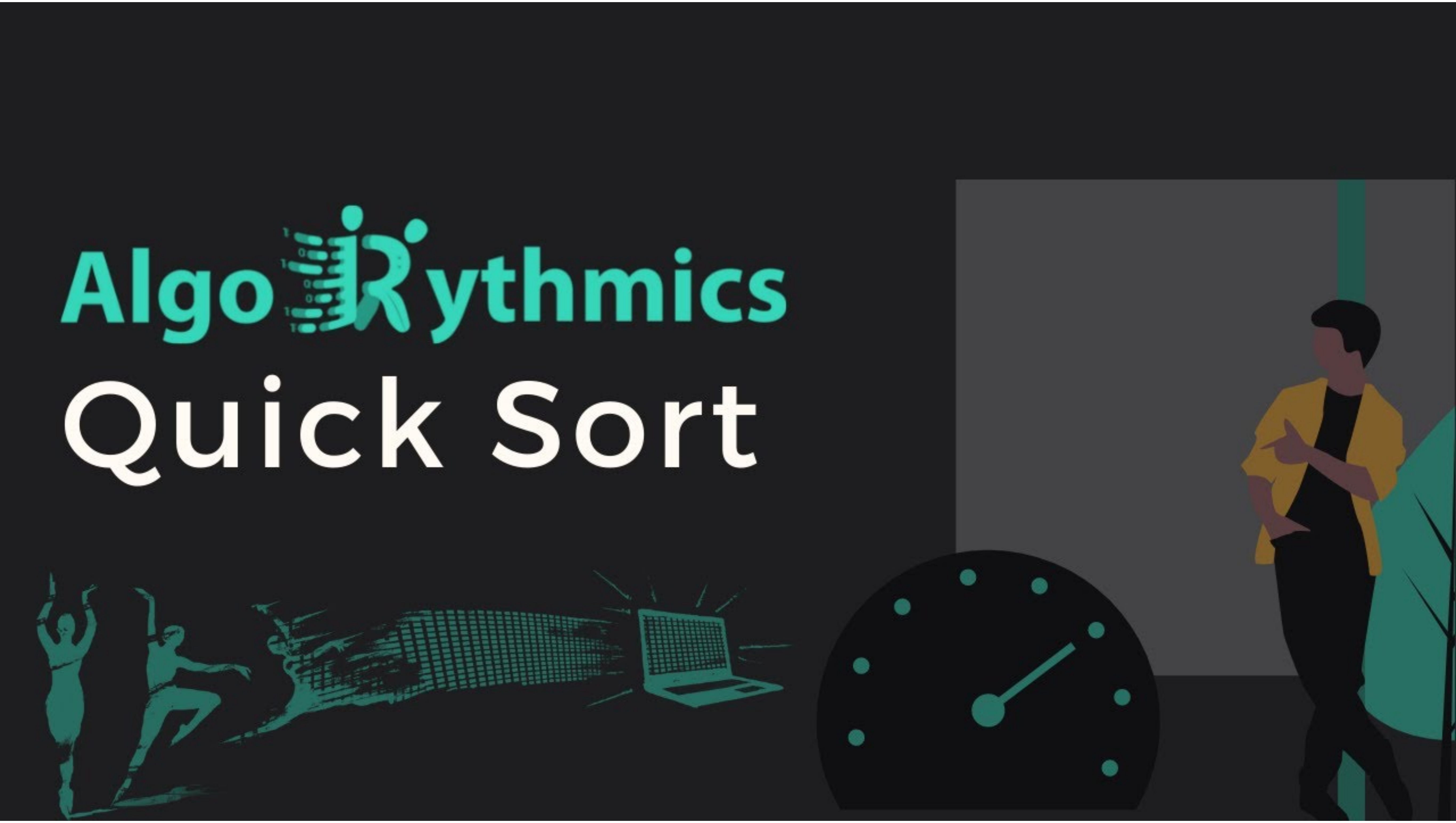
quicksort(A, left, right) sorts *A[left, ..., right]* by using *partition()* to partition *A*, and then calling itself recursively twice to sort the two sub-arrays.

***partition*(*A*, *left*, *right*)** rarranges $A[left..right]$ and finds and returns an integer q, such that

$$A[left], ..., A[q-1] \lesssim pivot, \quad A[q] = pivot, \quad A[q+1], ..., A[right] > pivot,$$

where *pivot* is the middle element of $a[left..right]$, before partitioning. (To choose the pivot element differently, simply modify the assignment to m.)

from http://www.math.uic.edu/~leon/cs-mcs401-s08



https://www.youtube.com/watch?v=ywWBy6J5gz8

Quicksort

```
#define M 7
#define NSTACK 50
```

Here M is the size of subarrays sorted by straight insertion and NSTACK is the required auxiliary storage.

```
void sort(unsigned long n, float arr[])
```

Sorts an array arr[1..n] into ascending numerical order using the Quicksort algorithm. n is input; arr is replaced on output by its sorted rearrangement.

```
{
    unsigned long i,ir=n,j,k,l=1,*istack;
    int jstack=0;
    float a,temp;

    istack=lvector(1,NSTACK);
    for (;;) {                          Insertion sort when subarray small enough.
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                a=arr[j];
                for (i=j-1;i>=l;i--) {
                    if (arr[i] <= a) break;
                    arr[i+1]=arr[i];
                }
                arr[i+1]=a;
            }
            if (jstack == 0) break;
            ir=istack[jstack--];        Pop stack and begin a new round of parti-
            l=istack[jstack--];             tioning.
        } else {
```

Press et al., Numerical Recipes

Quicksort cont.

```
} else {
    k=(l+ir) >> 1;                       Choose median of left, center, and right el-
    SWAP(arr[k],arr[l+1])                   ements as partitioning element a. Also
    if (arr[l] > arr[ir]) {                 rearrange so that a[l] ≤ a[l+1] ≤ a[ir].
        SWAP(arr[l],arr[ir])
    }
    if (arr[l+1] > arr[ir]) {
        SWAP(arr[l+1],arr[ir])
    }
    if (arr[l] > arr[l+1]) {
        SWAP(arr[l],arr[l+1])
    }
    i=l+1;                               Initialize pointers for partitioning.
    j=ir;
    a=arr[l+1];                          Partitioning element.
    for (;;) {                           Beginning of innermost loop.
        do i++; while (arr[i] < a);          Scan up to find element > a.
        do j--; while (arr[j] > a);          Scan down to find element < a.
        if (j < i) break;                Pointers crossed. Partitioning complete.
        SWAP(arr[i],arr[j]);             Exchange elements.
    }                                    End of innermost loop.
    arr[l+1]=arr[j];                     Insert partitioning element.
    arr[j]=a;
    jstack += 2;
    Push pointers to larger subarray on stack, process smaller subarray immediately.
    if (jstack > NSTACK) nrerror("NSTACK too small in sort.");
    if (ir-i+1 >= j-l) {
        istack[jstack]=ir;
        istack[jstack-1]=i;
        ir=j-1;
    } else {
        istack[jstack]=j-1;
        istack[jstack-1]=l;
        l=i;
    }
  }
 }
 free_lvector(istack,1,NSTACK);
}
```

Press et al., Numerical Recipes

Quicksort

on *average* the fastest sorting algorithm

- scales as ~ $N \log(N)$ on average
- but ~ N^2 in the worst case: on **sorted** lists

Return Feb. 28, 9:15 a.m.

Free Training

- Write routines using the
  1. straight insertion
  2. Shell's
  3. quicksort

  algorithms presented in the lecture[1]. Use the following array to test your routines

$$[7, 5, 3, 1, 9, 6, 10, 2, 8, 4] \, .$$

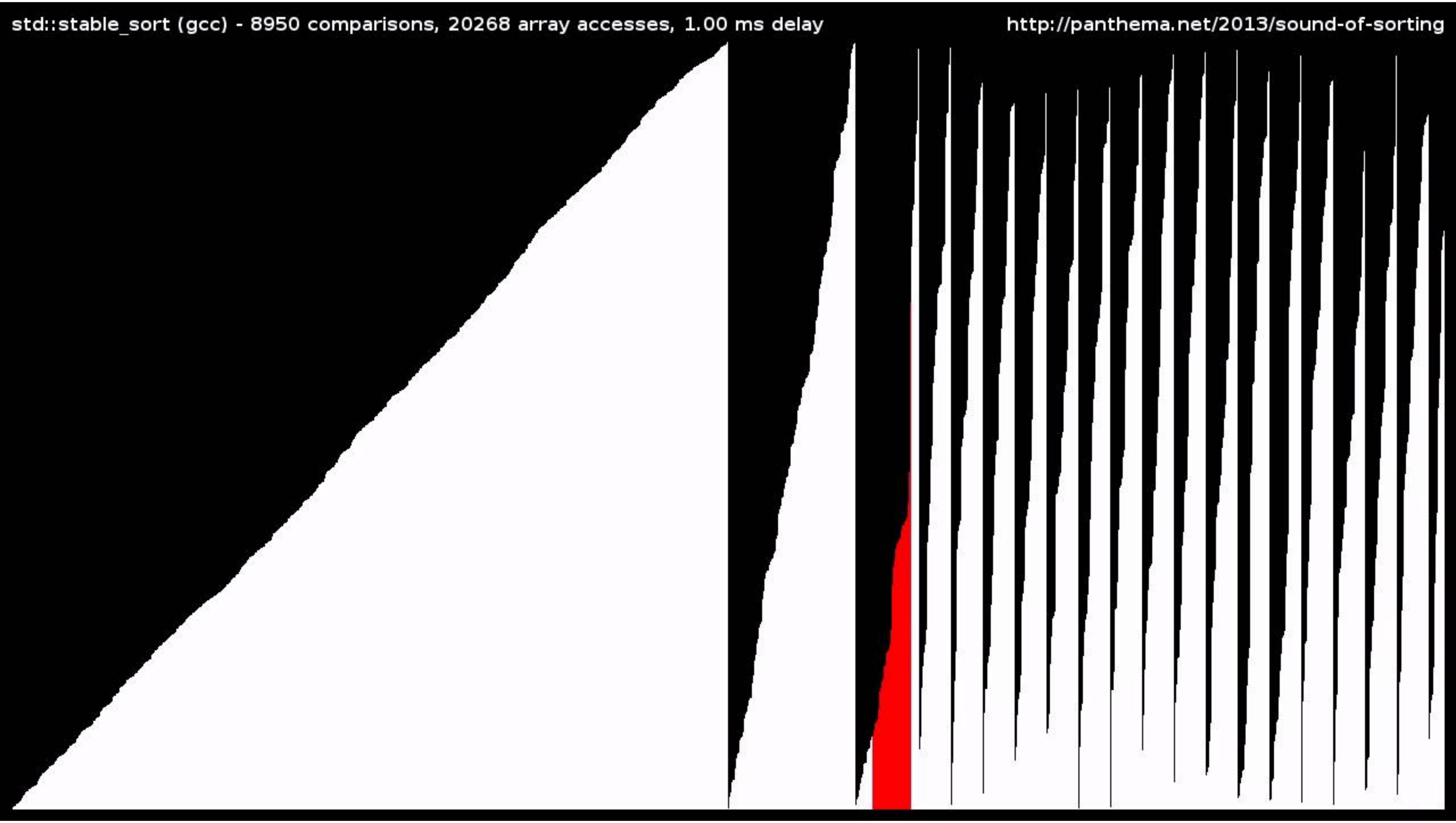
Assignment for Afternoon/Home Work, 20 Points

- **Exercise 6.1, 5 points**: Varification
  Varify that your algorithms work using the above list and another list of 10 random number. Print out the lists before and after sorting.

- **Exercise 6.2, 10 points**: Timing on *unsorted* lists
  Measure the runtime[2] of your algorithms for unsorted lists[3] of the length $N = 10^n$ with $n = [2, 3, ..., 8]$, if feasible. Discuss the occured and possible problems. Plot the results in a double-logarithmic diagram. What are the scaling properties?

- **Exercise 6.3, 5 points**: Timing on *sorted* lists
  Do the same (Ex. 6.2) for perfectly sorted lists (i.e. $A = [1, 2, 3, ..., N]$).

15 Sorting Algorithms in 6 Minutes… turn off sound?



https://www.youtube.com/watch?v=kPRA0W1kECg