University of Waterloo
CS240 Winter 2018
Assignment 2 – Solution Outline

version:
2018-02-01 12:06

Problem 1 [Partial Sort [10 marks]]

Variant 1: Truncated Heapsort

- Idea:
  We use a min-oriented binary heap, as in heapsort, but we stop the sort-down phase after k steps.

- Code:

```
1  procedure partialSort(A)
2     // build min-heap
3     minOrientedHeapify(A)
4     // extract k smallest elements and put them at end of A
5     for i = 0,...,k-1
6        A[n-1-i] = heapDeleteMin(A)
7     // reverse array
8     for i = 0,...,min(k-1,n/2-1)
9        swap(A[i], A[n-1-i])
```

  `minOrientedHeapify` and `heapDeleteMin` are like the corresponding methods from class, but with the reverse ordering of elements in the heap.

- Correctness:
  In line 4, A contains a min-heap (methods from slides correct). In line 7, the suffix of length k of A contains the smallest elements in decreasing order, prefix of A still contains heap elements. By the end of the algorithm we have reversed the suffix of length k into a prefix of length k in increasing order.

- Analysis:
  This algorithm runs in $O(n + k \log n)$ time since it calls heapify once which runs in $O(n)$ time and then does k deleteMins each of which cost $O(logn)$ time; the space usage is constant as we store each deleted node in the back of the given array.

Bonus:

Actually, this algorithm runs in $O(n + k \log k)$ time, as well:

- For $k \leq n/\log n$, we have $k \log n \leq n$, so $n \leq n + k \log n \leq 2n$ and similarly $n \leq n + k \log k \leq n + k \log n \leq 2n$, so the achieved and the required bound both within a constant factor of n, and thus within a constant factor of each other: $\Theta(n + k \log n) = \Theta(n + k \log k)$

- For $k > n/\log n$, we have

$$k \log n \geq k \log k > k \log\left(\frac{n}{\log n}\right) = k \log n - k \log \log n = k \log n \left(1 - \frac{k \log \log n}{k \log n}\right).$$

  Since

$$\frac{k \log \log n}{k \log n} = \frac{\log \log n}{\log n} \to 0, \qquad (n \to \infty),$$

  there is a n_0, so that for all $n \geq n_0$ holds $\frac{\log \log n}{\log n} \leq \frac{1}{2}$. (In fact, $n_0 = 20$ suffices: `http://www.wolframalpha.com/input/?i=plot+log2(log2(n))%2Flog2(n)+for+n+%3D2+to+100`)

  We thus have for $n \geq n_0$:

$$k \log n \geq k \log k \geq \frac{1}{2} k \log n.$$

  So again both terms are within a constant factor of each other.

Together, we find that indeed $\Theta(n + k \log n) = \Theta(n + k \log k)$, our truncated heapsort variant above fulfills the stronger requirement of the bonus problem.

Variant 2: Select, partition and sort prefix

We determine the kth smallest element x using Quickselect (in $O(n)$ time on average). This already leaves the array partitioned into the $k-1$ elements smaller than x and the elements larger than x. Then we sort the prefix of length k in $O(k \log k)$ time by Heapsort.

It uses $O(1)$ extra space, if we implement Quickselect with tail-recursion elimination.

This method overall runs in $O(n + k \log k)$ *expected* time unless we use a worst-case linear-time time selection algorithm. The the running time is achieved also in the worst case, but it is not possible with the methods from class to retain the constant space.

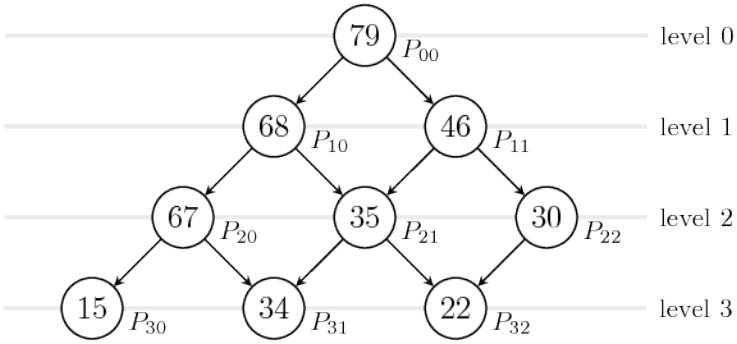
Variant 3: partial Quicksort

A nice alternative of Variant 2 is to use Quicksort and pursue all recursive calls that contain some of the first indices (this always includes the left call, and sometimes additionally the right call).

One can prove that the expected overall running time is essentially the same as Variant 2.[1]

[1] Martínez, Conrado. 'Partial quicksort.' Proceedings of the 1st ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics. 2004.

Problem 2 [Pyramids Theory [6+7+7+(10) = 20 + (10) marks]]



a) For making computations nicer, let us consider the number of *levels* m instead of the *height* h; since a pyramid of height h has the $m = h+1$ levels $0, 1, \ldots, h$, the difference is tiny.

So we look for the smallest m so that the number of nodes that fit in levels $0, 1, \ldots, m-1$ is $\sum_{i=0}^{m-1}(i+1) = \frac{m(m+1)}{2} \geq n$.

Solving for m we find: $m \geq \sqrt{\frac{1}{4}+2n} - \frac{1}{2}$, i.e., $m = \left\lceil \sqrt{\frac{1}{4}+2n} - \frac{1}{2} \right\rceil = \Theta(\sqrt{n})$. Hence also $m - 1 = h = \Theta(\sqrt{n})$

b)

```
1  deleteMax():
2     max = key of root()
3     swap keys of root() and last()
4     removeLast()
5     v = root()
6     while (v is not a leaf)
7        if (key(leftChild(v)) > key(rightChild(v)))
8           u = leftChild(v)
9        else
10          u = rightChild(v)
11       if (key(v) >= key(u)) break
12       swap keys of u and v
13       v = u
14    end while
15    return max
```

The code is literally the same as for binary heaps. All called methods can be implemented with $O(1)$ running time and As shown in class, we follow at most one full path

from the root of the pyramid to a leaf. So the overall running time is $O(h)$ for h the height of the pyramid. By a) this is $O(\sqrt{n})$.

c)

```
1  insert(x):
2     newLast(x)
3     v = last()
4     while (v is not root())
5        if (key(leftParent(v)) > key(rightParent(v)))
6           u = rightParent(v)
7        else
8           u = leftParent(v)
9        if (key(v) <= key(u)) break
10       swap keys of u and v
11       v = u
12    end while
```

The code is very similar to b), but we move up instead of down. For the running time, this does not make a difference: All called methods are $O(1)$ and we follow at most one full path from the last leaf of the pyramid to the root. So the overall running time is $O(h) = O(\sqrt{n})$.

d) For this problem, we assume that we have constant-time access to any node in the pyramid when we know its coordinates; to be specific, we assume $\texttt{get}(i, j) = P_{i,j}$ (whenever i, j is a valid pair of indices). (As Problem 3 shows, this is possible.)

Conceptually, we then do a binary search for each rightmost element in the pyramid on the items lying on the decreasingly ordered chain that we get by following the left-child links. Starting at P_{ll}, we walk down to $P_{ll}, P_{l+1,l}, \ldots, P_{\ell,l}$.

```
1  contains(x):
2     ℓ = maximal level of pyramid
3     for l = 0,...,ℓ
4        lo = l, hi = ℓ
5        while lo <= hi
6           mid = lo/2 + hi/2
7           if      x > key(get(mid,l)) hi = mid - 1
8           else if x < key(get(mid,l)) lo = mid + 1
9           else return true
10       end while
11    end for
12    return false
```

Each binary search takes at most $\log \sqrt{n} + 1$ steps in the worst case, and we do $O(\sqrt{n})$ for them, one for each level. Hence the overall running time is $O(\sqrt{n} \log n)$.

Problem 3 [Programming Pyramids]

Full code can be found on the next few pages.

The main ingredients for the various accessor-functions is a translation between coordinates given in the picture for $P_{i,j}$ and the 0-based index in array A. Then we can use the simple rules

- left parent of $P_{i,j}$ is $P_{i-1,j-1}$
- right parent of $P_{i,j}$ is $P_{i-1,j}$
- left child of $P_{i,j}$ is $P_{i+1,j}$
- right child of $P_{i,j}$ is $P_{i+1,j+1}$

The coordinates for a given index k are found by computing the maximal level for a pyramid with size $k+1$ the formula for this was already given in 2a) (but notice that the formula is not valid for $n = 0$). This level is coordinate i. By subtracting from k the number of nodes above the final level – given by $k(k+1)/2$ – we obtain coordinate j.

To translate back, i.e., find the index of a $P_{i,j}$ we reverse this process: The index k is the sum of the elements above level i and j: $k = i(i+1)/2 + j$.

Using simply boundary checks on the limits suffices to detect all cases, in which the newly computed index is not valid.

In the implementation for `deleteMax` and `insert`, we follow the pseudocode from 2b) and 2c), but we have to exclude `key(leftParent(v))` etc. when we try to find the child resp. parent of v with larger resp. smaller key. In our implementation, we set the corresponding value to $-\infty$ resp. $+\infty$ (using the smallest/largest number representable in an `int`).

```
#ifndef PYRAMID_H
#define PYRAMID_H


class pyramid {
public:
    /**
     * Create a new empty max-oriented pyramid with room for up to N items
     * */
    explicit pyramid(int N);

    /**
     * Return the index of the left child of the item at given index
     * or -1 if it does not exist.
     * requires 0 <= index < size.
     * */
    int leftChild(int index);

    /**
     * Return the index of the right child of the item at given index
     * or -1 if it does not exist.
     * requires 0 <= index < size.
     * */
    int rightChild(int index);

    /**
     * Return the index of the left parent of the item at given index
     * or -1 if it does not exist.
     * requires 0 <= index < size.
     * */
    int leftParent(int index);

    /**
     * Return the index of the right parent of the item at given index
     * or -1 if it does not exist.
     * requires 0 <= index < size.
     * */
    int rightParent(int index);

    /**
     * Return the item (its value/priority) stored at the given index.
     */
    int getItem(int index);

    /**
     * Return the number of items stored in this pyramid
     */
    int getSize();

    /**
     * Inserts a new item with value/priority x into the pyramid.
     * May not be called if size() == N.
     */
    void insert(int x);

    /**
```

```
     * Removes and returns an item from the pyramid with maximal priority.
     * (Ties are broken arbitrarily.)
     * May not be called if size() == 0.
     */
    int deleteMax();

    ~pyramid() { delete[] A; }

    void print() {
        int l = maxLevel(size);
        std::cout << "levels: " << l << std::endl;
        for (int i = 0; i <= l; ++i) {
            for (int j = 0; j <= i; ++j) {
                int k = indexFor(i, j);
                if (checkIndex(k)) std::cout << A[k] << " ";
            }
            std::cout << std::endl;
        }

    }

private:

    /** maps P_ij to an index k in A*/
    int indexFor(int i, int j);

    /** maps index k in A to P_ij */
    std::pair<int,int> coordsFor(int index);

    bool checkCoords(int i, int j);

    bool checkIndex(int index);

    /** compute index of lowest level for pyramid of size n */
    int maxLevel(int n);


    int* A;
    int size;
};


#endif //PYRAMID_H
```

```cpp
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <limits>
#include "pyramid.h"

using std::cin;
using std::cout;
using std::cerr;
using std::endl;


pyramid::pyramid(int N)
{
    this->A = new int[N];
}

int pyramid::leftChild(int index)
{
    const std::pair<int, int> coords = coordsFor(index);
    int i = coords.first + 1, j = coords.second;
    int newIndex = indexFor(i, j);
    return checkIndex(index) && checkCoords(i, j) && checkIndex(newIndex) ?
        newIndex : -1;
}

int pyramid::rightChild(int index)
{
    const std::pair<int, int> coords = coordsFor(index);
    int i = coords.first + 1, j = coords.second + 1;
    int newIndex = indexFor(i, j);
    return checkIndex(index) && checkCoords(i, j) && checkIndex(newIndex) ?
        newIndex : -1;
}

int pyramid::leftParent(int index)
{
    const std::pair<int, int> coords = coordsFor(index);
    int i = coords.first - 1, j = coords.second - 1;
    int newIndex = indexFor(i, j);
    return checkIndex(index) && checkCoords(i, j) && checkIndex(newIndex) ?
        newIndex : -1;
}

int pyramid::rightParent(int index)
{
    const std::pair<int, int> coords = coordsFor(index);
    int i = coords.first - 1, j = coords.second;
    int newIndex = indexFor(i, j);
    return checkIndex(index) && checkCoords(i, j) && checkIndex(newIndex) ?
        newIndex : -1;
}

const int negInf = std::numeric_limits<int>::min();
const int posInf = std::numeric_limits<int>::max();
```

```
void pyramid::insert(int x) {
    A[size++] = x;
    int v = size-1, u;
    while (v != 0) {
        int leftParentKey = leftParent(v) == -1 ? posInf : A[leftParent(v)];
        int rightParentKey = rightParent(v) == -1 ? posInf : A[rightParent(v)];
        if (leftParentKey > rightParentKey)
            u = rightParent(v);
        else
            u = leftParent(v);
        if (A[v] <= A[u]) break;
        std::swap(A[u],A[v]);
        v = u;
    }
}

int pyramid::deleteMax() {
    int max = A[0];
    std::swap(A[0],A[size-1]);
    --size;
    int l = maxLevel(size);
    int v = 0, u;
    while (coordsFor(v).first < l) {
        int leftChildKey = leftChild(v) == -1 ? negInf : A[leftChild(v)];
        int rightChildKey = rightChild(v) == -1 ? negInf : A[rightChild(v)];
        if (leftChildKey > rightChildKey)
            u = leftChild(v);
        else
            u = rightChild(v);
        if (A[v] >= A[u]) break;
        std::swap(A[u],A[v]);
        v = u;
    }
    return max;
}

int pyramid::getSize()
{
    return size;
}

int pyramid::getItem(int index) {
    return A[index];
}

int pyramid::indexFor(int i, int j) {
    int nodesAboveI = i*(i+1)/2;
    return nodesAboveI + j;
}

std::pair<int, int> pyramid::coordsFor(int index) {
    int i = maxLevel(index + 1);
    int nodesAboveI = i*(i+1)/2;
    return {i, index - nodesAboveI};
}
```

```
int pyramid::maxLevel(int n) {
    return n == 0 ? 0 : ceil(sqrt(0.25 + 2 * n) - 0.5 ) - 1;
}

bool pyramid::checkCoords(int i, int j) {
    int l = maxLevel(size);
    return 0 <= i && i <= l && 0 <= j && j <= i;
}

bool pyramid::checkIndex(int index) {
    return 0 <= index && index < size;
}


int main (int argc, char* argv[])
{

    char op=' ';
    pyramid* P = 0;
    while (op != 'x') {
        cin >> op;
        int i, x, N;
        switch (op)
        {
            case 'n': // new pyramid
                delete P;
                cin >> N;
                P = new pyramid(N);
                break;
            case 's': // getSize
                cout << P->getSize() << endl;
                break;
            case 'S': //print
                P->print();
                break;
            case 'i': // insert
                cin >> x;
                P->insert(x);
                break;
            case 'g': // get
                cin >> i;
                x = P->getItem(i);
                cout << x << endl;
                break;
            case 'm': // deleteMax
                x = P->deleteMax();
                cout << x << endl;
                break;
            case 'l': // leftChild
                cin >> i;
                x = P->leftChild(i);
                cout << x << endl;
                break;
            case 'r': // rightChild
                cin >> i;
                x = P->rightChild(i);
```

```
                cout << x << endl;
                break;
            case 'p': // leftParent
                cin >> i;
                x = P->leftParent(i);
                cout << x << endl;
                break;
            case 'q': // rightParent
                cin >> i;
                x = P->rightParent(i);
                cout << x << endl;
                break;
            case 'x':
                delete P;
                std::exit(0);
            default:
                cerr << "Wrong action selected" << endl;
        }
    }
}
```

Problem 4 [Holy-Grail of Sorting [9+1]]

a)

Algorithm	**inputs** e. g. comparison based, numbers in $\{0, \ldots, R^m - 1\}$	**worst case time** order of growth	**in-place?** order of growth of space usage in $O(1)$?	**stable?**
PQ-Sort (heaps)	comparison-based	$\Theta(n \log n)$	no	no
Mergesort	comparison-based	$\Theta(n \log n)$	no	yes
Heapsort	comparison-based	$\Theta(n \log n)$	yes	no
Quicksort	comparison-based	$\Theta(n^2)$, but expected $\Theta(n \log n)$	no	no
Bucketsort	$\{0, \ldots, R\}$	$\Theta(n + R)$	no	yes
Countsort	$\{0, \ldots, R\}$	$\Theta(n + R)$	no	yes
MSD-Radix-Sort	$\{0, \ldots, R^m - 1\}$	$\Theta(m(n + R))$	no	yes
LSD-Radix-Sort	$\{0, \ldots, R^m - 1\}$	$\Theta(m(n + R))$	no	yes

A counter-example to stability for Heapsort is $1a, 1b, 1c, 2$ (order by numbers) We will build the heap in-place by heapify, which leaves the array in order $2, 1a, 1c, 1b$. Then we extract the unique max 2, swap it with $1b$ do bubble-down (no changes). The next extract Max gets $1b$ and puts it in front of 2. Thereby $1c$ will be before $1b$ in the output.

For Quicksort, we use $1a, 1b, 1c$. We will now choose one of the elements as pivot and swap it with $1a$. The partitioning process then also swaps the elements at the last two positions, and finally puts the pivot in place, which always means swapping the first and last element. Observe that $1c$ can never end up in the last position in this process, so the sort is not stable.

b) no

Problem 4.1 Further Remarks

The above is a sensible combination to ask for which is not met by the typical textbook algorithms; a quick comment on the theoretical question whether this is possible might be in order.

Holy-grail sorting methods do exist, but the algorithms are rather complicated. A summary is given here: `https://cs.stackexchange.com/q/2569`. In the literature, even the additional requirement of $O(n)$ element moves is added, and a corresponding algorithm was presented at STACS 2005 (full paper doi: 10.1007/s00224-006-1311-1). These methods do not seem to be used in practice though.

There are elementary variations of Mergesort that run in-place, see e.g. the concise explanation here: `https://stackoverflow.com/a/15657134`. However, this method is not stable.

`std::stable_sort` uses a stable in-place mergesort if not enough memory is available, which has runtime $O(n \log^2 n)$.

Problem 5 [Sorting with Stacks [5 + 10+(5) + 5]]

This is known as stack-sortable permutations and rather well-studied under this name in combinatorics. (Warning: There are different notions of "k-stack sortability" in the literature.)

For working in the model, we program *drivers* that only use the public interface of S, I and O, i. e, the stack and queue ADT operations.

a) There are several possible arguments.

- A simple counting argument.

  Ultimately, the stack-machine applies a certain permutation to the input. For this result, we can restrict our attention to the two operations $I \to S$ and $S \to O$. So in each step, we either push an element form I onto S, or we pop an element from S and put it into O. In every such step, either the size of I shrinks by one or the size of O grows by one, so we need exactly $2n$ steps to pass all n elements through the stack.

  Overall, this gives at most $2^{2n} = 4^n$ different possible executions (permutations that are applied to the input), but there are $n!$ different input permutations. For $n \geq 9$ when $4^n < n!$, some permutations thus cannot be sorted because they are different, but receive the same treatment.

- Counterexample.

  The shortest permutation that cannot be sorted using a stack is $2, 3, 1$; no matter how we use the stack, we cannot output 1 first without getting 2 and 3 out of order.

- Sorting lower bound.

  Similar to the first argument, we observe that we can do at most $2n$ non-redundant comparisons during one execution since we have to move some elements after each comparison before we can learn more about the input. Since for $n \geq 9$ we have $2n < \log_2(n!)$, the lower bound for comparison-based sorting, we cannot sort all inputs correctly.

b) There are several solutions. The first one does never obtain any actual value from the stack-machine, but only uses direct comparisons between $S.top()$ and $I.front()$. It uses constant extra space and constant time between any two operations send to the stack-machine. The given code uses n to control the code. This avoids explicit checks for corner cases and thus makes the code more readable; it is not required though and

could be replaced by empty-checks on S and I. This algorithm hence effectively allows to operate blindly.

- Idea:
  The stack can be used to merge a run on the stack with one still in I. We can thus simulate (bottom-up) Mergesort.
  How to merge:
  If we have two runs in I, the first in descending order, the second in ascending order, we can push the first onto S, thereby reversing its order. Then we can alternatingly pop resp. take from I and merge the two runs into one large ascending run.
  Since the run gets reversed while being put into S, we need every other in descending order. To obtain runs alternatingly in ascending and descending order, we spend a second round after merging, in which we reverse every other run.
- Code:
  We assume that O and I are given as one connected queue I. The following code uses the typical ADT interface for queue and stack to sort using I and S.

```
1  mergesortDriver(I, S, n)
2     for (len = 1; len < n; len *= 2) // 2 rounds each
3        // reverse every other run
4        for (offset = 0; offset < n; offset += 2*len)
5           len1 = min { len, n - offset }
6           len2 = min { len, n - offset - len, 0 }
7           // reverse first run
8           for (i = 0; i < len1; ++i) S.push(I.dequeue) // I → S
9           for (i = 0; i < len1; ++i) I.enqueue(S.pop()) // S → O
10          // copy second run
11          for (i = 0; i < len2; ++i)
12             I.enqueue(I.dequeue()) // I → S, S → O
13       // first round finished, now merge runs
14       for (offset = 0; offset < n; offset += 2*len)
15          len1 = min { len, n - offset }
16          len2 = min { len, n - offset - len, 0 }
17          for (i = 0; i < len1; ++i)
18             S.push(in.dequeue()); // S → I
19          int i1 = 0, i2 = 0
20          while (i1 < len1 || i2 < len2)
21             if (i1 == len1)
22                ++i2; I.enqueue(I.dequeue()) // I → S, S → O
23             else if (i2 == len2)
24                ++i1; I.enqueue(S.pop())  // S → O
25             else if (S.top() <= I.front()) // compare(S, I)
26                ++i1; I.enqueue(S.pop()) // S → O
27             else
```

```
28                ++i2; I.enqueue(I.dequeue()) // I → S, S → O
29            end while // one merge finished
30        end for
31        // second round finished
32    end for
```

- Correctness:
  The overall structure is bottom-up mergesort as in Program 8.5 of [Sedgewick]. In the first round, we make sure that the first run in every pair of runs is reversed. Hence in the second round, merges can be done by pushing the first run into S thereby reversing it again. So the correctness follow from the correctness of bottom-up Mergesort and merge.
- Analysis:
  The outer loop (line 2) runs $\lceil \log_2 n \rceil$ times, and each iteration executes 2 rounds of running all n elements from I through S back into I (resp. O). Hence we need $k = 2\lceil \log_2 n \rceil = O(\log n)$ rounds.
  Note: If n is not known up front, we have to spend another round to count the elements.

Apart from the reversal needed for the stack, the above code borrows tricks from the setting of sorting data stored on tape drives which essentially behave like large queues. Unlike in our model, having additional tapes is allowed and can be used to speed up sorting. Tape sorting methods typically work without knowing n up front (and without using an expensive extra round to determine it).

Bonus:

- Idea:
  We can indeed simulate a version of bottom-up mergesort using $\lceil \log n \rceil$ rounds by figuring out the correct order for runs up front.
- Code:
  Below is Java code illustrating the idea. It take $I = O$ as input and sorts the elements therein using a stack. Internally, we move elements in each round from `in` (I) into another queue `out` (O) and then copy them back to I for the next round. we could alternatively put elements directly back into I and use counter to determine the end of the round.
  The code allocates an array to store the direction of runs. At the expense of more complicated code, we could get rid of that and use that the sequence of run directions forms the *Thue-Morse sequence*, whose elements can be computed explicitly (`https://en.wikipedia.org/wiki/Thue%E2%80%93Morse_sequence`).

```
1  public static <Elem> void stackMergesort(Queue<Elem> in, Comparator<↩
       Elem> c) {
2    int n = in.size();
```

```
3     Deque<Elem> stack = new ArrayDeque<>(n);
4     Queue<Elem> out = new ArrayDeque<>(n);
5     boolean ascStart = (ceilLog2(n) & 1) != 0;
6     for (int len = 1; len < n; len *= 2) {
7        boolean[] asc = runDirection(n / (2 * len) + 1, ascStart);
8        for (int merge = 0; merge <= n / (2*len); ++merge) {
9           for (int i = 0; i < len && !in.isEmpty(); ++i)
10             stack.push(in.poll());
11          int inLen = min(in.size(), len);
12          for (int i = 0; i < 2*len; ++i) {
13             if (stack.isEmpty() && in.isEmpty()) break;
14             if (stack.isEmpty()) { out.add(in.poll()); --inLen; }
15             else if (inLen == 0) out.add(stack.pop());
16             else {
17                Elem x = stack.peek(), y = in.peek();
18                if (asc[merge] && less(x,y,c)
19                      || !asc[merge] && less(y,x,c))
20                   out.add(stack.pop());
21                else
22                   { out.add(in.poll()); --inLen; }
23             }
24          }
25       }
26       assert in.isEmpty();
27       for (int i = 0; i < n; ++i) in.add(out.poll());
28       ascStart = !ascStart;
29    }
30 }

32 public static boolean[] runDirection(int n, boolean first) {
33    boolean[] res = new boolean[n];
34    res[0] = first;
35    for (int l = 1; l < n; l *= 2)
36       for (int i = l; i < 2 * l && i < n; ++i)
37          res[i] = !res[i - l];
38    return res;
39 }

41 public static int ceilLog2(int n) {
42    if (n <= 0) throw new IllegalArgumentException();
43    int i = 0, twoToI = 1;
44    while (twoToI < n) { ++i; twoToI <<= 1; }
45    return i;
46 }
```

- Analysis:

We use one round for each level / run-length, so we use $\lceil \log_2 n \rceil$ rounds.

Quicksort:
If we allow storing and processing the elements outside of the stack-machine, we can also simulate Quicksort:

We select the exact median and use this as "pivot". Then in the next round, we put all elements greater than the pivot into S, all others into O. Once all elements have been taken out of I, we empty S into O. For the next round, we select the first and third quartile and use those as pivots for the first resp. second half of the array, i. e., the two segments of the first partitioning step. Repeating for $\lceil \log(n) \rceil$ rounds will sort the elements.

c) Again, there are several options.

- Counting argument:
  The k rounds perform independent movements of the elements, so we can upper bound the number of different runs by $(4^n)^k$ (recall from a): 4^n is the number of different runs for one round. We thus need $4^{nk} \geq n!$ to be able to sort, which implies $k \geq \log_4(n!)/n \leq \frac{1}{2}\frac{n \log_2(n)}{n}$.
  This bound might not be tight, but the actual number of rounds needed seems to be an open problem.
  Note that this bound even holds if we allow the algorithm the superpower to "know" the permutation up front, so that it only has to figure out the shortest sequence of movements to transform it into the sorted permutation.
- Sorting Lower Bound:
  This applies only to the strict version of the model. Since we have to do (at least) one move between any pair of non-redundant comparisons, we get at most $2n$ comparisons per round (see also a)) and $\leq k \cdot (2n)$ comparisons from k rounds. As the lower bound for comparison based sorting dictates $k(2n) \geq \log_2(n!)$, we find $k \geq \frac{1}{2}\frac{\log_2(n!)}{n} \leq \frac{1}{2}\log_2 n$.

Problem 6 [Goofy-Sort [5+5+10+(10)]]

a) Goofy-Sort iteratively removes a randomly chosen element from the list and re-inserts it at the position found by binary searching it in the remaining array, until the whole array is sorted.

A precondition of binary search to return the correct insertion position is that the searched array is *sorted*. Since the array is *not sorted*, the binary search can essentially return any index. We thus apply a method outside of its intended usage, which in general can lead to arbitrary results.

If Goofy-Sort terminates it will have sorted the array successfully, but given the improper use of binary search, it is by no means obvious if it ever terminates.

b) Binary insertion sort:

1. For $i = 1, \ldots, n-1$

1.1. Let $x := A[i]$.

1.2. Let j be the index returned by binary search on $A[0..i-1]$ with key x.

1.3. For $k := i, i-1, \ldots, j+1$ do $A[k] := A[k-1]$.

1.4. $A[j] := x$.

The number of comparisons is then

$$\begin{aligned}\sum_{i=1}^{n-1} \text{cost of binary search on size-}n\text{ array} &= \sum_{i=1}^{n-1}(\lfloor \log_2(i) \rfloor + 1) \\ &= n - 2 + \sum_{i=1}^{n-1} \log_2(i) \\ &\leq (n-1)\log_2(n-1) + n - 2 \\ &= \Theta(n \log n),\end{aligned}$$

which is optimal (even including the constant in the leading term). However, the number of element moves in the worst case (when $j = 0$) is $\sum_{i=1}^{n-1} i = \Theta(n^2)$, so binary insertion sort has overall quadratic running time and is not in general recommendable.

c) If we (randomly) select the smallest or largest element as x, it will certainly end up at its correct position in the array, since it compares the same with any other element. For example for the max, binary search will always branch to the right, and eventually return $n-1$. So max is always inserted in the correct position when it is selected, The same is true for min.

Similarly, assume now that the k smallest and l largest elements have been successfully put to their correct positions at the very left resp. right end of the array. That means the array is partially sorted (in the sence of **??**) in the prefix of length k and the suffix of length l. If we now select the $(k+1)$st smallest or $(l+1)$st largest element, it will be inserted correctly: A comparison of, say, the $(l+1)$st largest element with any element left of (incl.) its correct position will direct the search to the right, and a comparison with any element to the right will direct the search left. The same is true for the $(k+1)$st smallest element.

This means that at any point in time we have a $2/n$ chance of making progress. Let us call $m = n - (k+l)$ the number of elements between the sorted prefix and suffix, i. e., the number of elements for which we do *not yet know* for sure that they are at their final position. Initially, we know nothing and $m = n$. When $m = 0$, the array is completely sorted and goofy-sort terminates. A round in which an extreme element is selected (i. e. the $(k+1)$st smallest or $(l+1)$st largest element when the prefix and

suffix have length k and l, respectively), either k or l increases by one, so m goes down by exactly one.

When we now denote by R_m the (random) number of rounds till termination when there are m elements left between the prefix and suffix we know to be sorted, we can set up a recurrence inequality for its expected value $\mathbb{E}[R_m]$:

$$\begin{aligned}\mathbb{E}[R_0] &= 0 \\ \mathbb{E}[R_m] &\leq 1 + \tfrac{2}{n} \cdot \mathbb{E}[R_{m-1}] + \tfrac{n-2}{n} \cdot \mathbb{E}[R_m], \qquad (m \geq 1).\end{aligned}$$

Subtracting $\frac{n-2}{n} \cdot \mathbb{E}[R_m]$ and dividing by $1 - \frac{n-2}{n} = \frac{2}{n}$ gives

$$\mathbb{E}[R_m] \leq \frac{n}{2} + \mathbb{E}[R_{m-1}]$$

which immediately telescopes to

$$\begin{aligned}&\leq \sum_{j=1}^{m} \tfrac{n}{2} + \mathbb{E}[R_0] \\ &= m \cdot \tfrac{n}{2}\end{aligned}$$

Since we initially have $m = n$, the overall expected number of rounds in goofy-sort is $n^2/2$, as claimed.

Instead of setting up the recurrence, we can also argue as follows. The chance for choosing one of the two extremes is $2/n$. Since the draws are independent, the number of times we have to wait until one of the extremes is selected is a random variable with a *geometric distribution* with success probability $p = 2/n$ in each trial. The mean of the geometric distribution is $1/p = n/2$, so in expectation, we have to wait $n/2$ rounds for any of the extremes to be chosen. This process has to repeat n times to bring all n elements to their correct position, so at most $n \cdot n/2$ rounds are needed in expectation.

d) Here are some measured averages I found, averaging over several random permutations and runs of different sizes:

n	# rounds	$n^2/2$
32	139	512
64	364	2048
128	898	8192
256	2129	32768
512	4923	131072
1024	11260	524288

(Remark: I directly averaged here for each n over many different inputs; this does not formally fulfill the requirements of the assignment problem, which requires to show 10 averages ...)

The upper bound $n^2/2$ thus typically overestimates the actual number of rounds by far. Fitting the exponent to these six points, we obtain that they rather grow like $n^{1.26}$ rather than quadratic. (The number of points is however too small to make reliable predictions). That would mean that goofy-sort is surprisingly not much worse that binary insertion-sort (which does precisely $n-1$ rounds).

Some elements seem to be moved by chance to (or close to) their correct position even though they are not the smallest or largest. It is not obvious how to quantify this effect, though.