Chapter 2 Problem Solutions

Jörg Barkoczi

2-2 *Correctness of bubblesort*

a) What we need to prove:

- valid loop invariant
  - Initialization
  - Maintenance
  - Termination
- that it contains all original elements.

b) **Loop invariant (2-4)**:

The for-loop in lines 2-4 will maintain a loop invariant such that the element at position $A[j]$ will be the smallest in the subarray $A[n-j]$.

Initialization:: $A[n-j]$ will contain a single element, thus it'll be the smallest.

Maintenance: If $A[j-1] > A[j]$, both will be swapped, thus maintaining the loop invariant.

Termination: Element $A[j]$ will be the smallest in the subsequence $A[n-j]$.

c) **Loop invariant (1-4)**:

Each iteration sorts the sequence, such that the ith smallest element will be on the ith position. This will partition the sequence in the following subsequences after each iteration:

$A[1...i]$ which is sorted such that $A[1] \leq A[2] \leq \ldots \leq A[i]$.
$A[i+1 \ldots n]$ which is unsorted.

In the end the $n-1$ smallest elements will be positioned at positions $A[1 \ldots n-1]$ respectively, thus leaving the nth smallest element - which is the biggest - at position n.

Initialization: $i = 1$, therefore $A[1 \ldots i]$ contains only a single element, which is trivially sorted.

Maintenance: i is incremented, thus fullfilling the loop invariant yielding a sorted subarray $A[1 \dots i]$.

Termination: When $i = n - 1$, the sorted subsequence will be of size $n - 1$ and will contain the $n - 1$th smallest elements, thus leaving the last element $A[n]$ which must be the largest.

d) The worst case running time occurs everytime, because regardless of how sorted the sequence already is, the inner loop will always iterate $n - i + 1$ times, thus giving a total runtime of:

$$\Theta((n-1)(n-i+1)) \Leftrightarrow \Theta(n^2 - ni + i - 1) \equiv \Theta(n^2) \tag{1}$$

2-3 *Correctness of Horner's rule*

a) The running time of the given codefragment is $\Theta(n)$.

b)

```
1         template<typename X>
2         long double naive_polynomial_evaluator(const X x, const
              std::vector<int>& arr2)
3         {
4             std::vector<int> powers(arr2.size());
5
6             powers[0] = 1;
7             powers[1] = x;
8             for (int i = 2; i < powers.size(); ++i)
9             {
10                powers[i] = x * powers[i-1];
11            }
12            long double y = 0;
13
14            for (int i = 0; i < arr2.size(); ++i)
15            {
16                y += arr2[i] * std::pow(x, arr2.size() - 1 - i);
17            }
18
19            return y;
20        }
```

The running time of this algorithm is $\Theta(n^2)$ and thus clearly slower than Horner's rule.

c) Let y' denote the value of y after each iteration.
Given that $y' = a_i + xy$ and $y = \sum_{k=0}^{n-(i+1)} a_{i+k+1}x^k$ we can rewrite it as
$y' = a_i + x\sum_{k=0}^{n-(i+1)} a_{i+k+1}x^k$
$\Leftrightarrow a_i + \sum_{k=0}^{n-(i+1)} a_{i+k+1}x^{k+1}$
$\Leftrightarrow a_i x_0 + \sum_{k=0}^{n-(i+1)} a_{i+k+1}x^{k+1}$
$\Leftrightarrow \sum_{k=-1}^{n-(i+1)} a_{i+k+1}x^{k+1}$
$\Leftrightarrow \sum_{k=0}^{n-i} a_{i+k}x^k$

Now at termination where $i = 0$, we get

$$\sum_{k=0}^{n-0} a_{0+k}x^k \Leftrightarrow \sum_{k=0}^{n} a_k x^k \tag{2}$$

which is indeed the same as the given summation.

d) As we have just proved, we can say that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

2-4 *Inversions*

a) The five inversions of the array $2, 3, 8, 6, 1$ are as follows:

(a) (8,6)
(b) (2,1)
(c) (3,1)
(d) (8,1)
(e) (6,1)

b) An array from the set $1, 2, \ldots, n$ has the most inversions, if it has the property $A[1] > A[2] > \ldots > A[n]$. It will have $\frac{n(n-1)}{2}$ inversions.

c) Each inversion corresponds to an iteration of the inner loop in the insertion sort algorithm.
Given an array $A[1 \ldots n]$ and a function $I(x)$ which takes an element of A and outputs the number of inversion for that element, we in total have $\sum_{i=1}^{n} I(A[i])$ iterations of the inner loop.

d)
Given the hint of modifying our existing merge sort algorithm, we can add another statement of constant time on line 33, which calculates the remaining elements in array $v1$ - which is left of $v2$ - and thus reflects the number of inversions for element $v2[j]$. We also change the return value to the number of inversions, such that it adds up and propagates through the recursion. Since we only added operations of constant time, the worst case running time of the algorithm stays at $\Theta(nlgn)$ and thus satisfies the required worst case running time.

```
1         template<typename T>
2         int merge(std::vector<T>& v, int l, int r, int p)
3         {
4             int s1 = p - l + 2;
5             int s2 = r - p + 1;
6
7             std::vector<T> v1(s1);
8             for (int i = 0; i < s1-1; ++i)
9             {
10                v1[i] = v[l+i];
11            }
12            v1[s1-1] = std::numeric_limits<T>::max();
13
```

```
14              std::vector<T> v2(s2);
15              for (int i = 0; i < s2-1; ++i)
16              {
17                  v2[i] = v[p+i+1];
18              }
19              v2[s2-1] = std::numeric_limits<T>::max();
20
21              int inv = 0;
22              int i = 0;
23              int j = 0;
24              for (int a = l;a <= r; ++a)
25              {
26                  if (v1[i] <= v2[j])
27                  {
28                      v[a] = v1[i++];
29                  }
30                  else
31                  {
32                      v[a] = v2[j++];
33                      inv += s1 - i - 1;
34                  }
35              }
36              return inv;
37          }
38
39          template<typename T>
40          int merge_sort(std::vector<T>& v, int l, int r)
41          {
42              auto inv = 0;
43              if (l < r)
44              {
45                  int p = (l + r) / 2;
46                  inv += merge_sort(v, l, p);
47                  inv += merge_sort(v, p+1, r);
48                  inv += merge<T>(v, l, r, p);
49              }
50              return inv;
51          }
```