HW4 solution

1 See file hw4.cpp

2 ab) See hw4.ss

c) The variable-argument mechanism of C/C++ is type unsafe. For example, the following incorrect uses of the function max of Problem 2 cannot be detected by C/C++ compilers:

```
max("df",29,a,f);        // where a is an array, and f is a function
max("df",29,3,4.5,6);    // # of d's and f's ≠ # of arguments
max("df",29,4.5,3);      // order of d's and f's ≠ order of arguments
```

On the other hand, the variable-argument mechanism of Scheme is type safe – type errors are detected as usual at run time. For example,

```
> (mymax 2 'a)
Exception in >: a ise not a real number
```

3 a) Let e1, e2, …, e6 be the expressions specified below.

```
(k=m, f()) / ((k=n, f()) * (k=m-n, f()))
  e1   e2      e3   e4       e5     e6
```

Since

1 the operand evaluation order of / and * is unspecified

2 , requires its operands be evaluated from left to right

it follows that the only constraints on the evaluation are

a) e1 must be evaluated before e2,

b) e3 must be evaluated before e4, and

c) e5 must be evaluated before e6

Thus, the total number of ways of evaluating the six expressions is

$6!/(2!\,2!\,2!) = 90$

b) Both VC++ and GNU C++ output 0, as they don't evaluate the two operands of each comma expression consecutively. In fact, the compied code looks like:

```
k=m;
k=n;
k=m-n;
return f()/(f()*f());
```

$\Rightarrow\ 3!/(3!\,3!) = 0$

On the other hand, clang++ outputs 10, as it evlautes the two operands of each comma expression consecutively.

4 foldl is a tail-recursive function, but foldr isn't. With tail-recursive optimization, suml takes $O(n)$ time and $O(1)$ space, but sumr takes $O(n)$ time and $O(n)$ space, where $n$ is the length of the list

5 a) The worst case occurs when one of the two subarrays is empty.
In that case, the worst-case space complexity $s(n)$ satisfies
$s(n) = s(n-1) + O(1)$
Thus, $s(n) = O(n)$

b) To minimize stack space, make the call on the larger subarray tail-recursive.

```
void qsort(int l,int h)
{
    if (l<h) {
        int m=partition(l,h);
        if (m-l<h-m) { qsort(l,m-1); qsort(m+1,h); }
        else { qsort(m+1,h); qsort(l,m-1); }
    }
}
```

Then, apply tail-recursive optimization to obtain

```
void qsort(int l,int h)
{
entry:
    if (l<h) {
        int m=partition(l,h);
        if (m-l<h-m) { qsort(l,m-1); l=m+1; goto entry; }
        else { qsort(m+1,h); h=m-1; goto entry; }
    }
}
```

Finally, remove gotos to obtain

```
void qsort(int l,int h)
{
    while (l<h) {
        int m=partition(l,h);
        if (m-l<h-m) { qsort(l,m-1); l=m+1; }
        else { qsort(m+1,h); h=m-1; }
    }
}
```

c) Now, the worst-case space complexity $s(n)$ satisfies

$$s(n) \leq s\left(\frac{n}{2}\right) + O(1)$$

Thus, $s(n) = O(\log n)$

Comment

We shall learn how to solve this kind of recurrences next semester in the course *Analysis of Algorithms*.

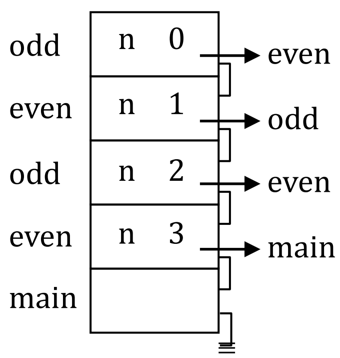
d) Regardless of the optimization, the worst case occurs when one of the two subarrays is empty. Thus, the worst-case time complexity $t(n)$ satisfies

$$t(n) = t(n-1) + O(n)$$

which means $t(n) = O(n^2)$.

However, the optimized version has a smaller coefficient of $n^2$, because it takes less time to maintain the runtime stack. More precisely, without optimization, worst-case time complexity $\Rightarrow$ worst-case space complexity, which is $O(n)$; but, with optimization, worst-case time complexity $\Rightarrow$ best-case space complexity, which is $O(1)$.

6 a)




b)