1 Exercise 1

```
function fib(n)
  if n <= 2
    1.0
  else
    fib(n-1)+fib(n-2);
  end
end

[@time fib(i) for i=35:45]
elapsed time: 0.04645490646362305 seconds
elapsed time: 0.07510113716125488 seconds
elapsed time: 0.12166213989257812 seconds
elapsed time: 0.1966111660003662 seconds
elapsed time: 0.3180971145629883 seconds
elapsed time: 0.512732982635498 seconds
elapsed time: 0.8321151733398438 seconds
elapsed time: 1.3448238372802734 seconds
elapsed time: 2.177851915359497 seconds
elapsed time: 3.525499105453491 seconds
elapsed time: 5.701272010803223 seconds
```

We observe that the running for `fib(n)` is essentially the double of that for `fib(n-1)`. When we look at the formula $F_n = F_{n-1} + F_{n-2}$ this makes sense.

2 exercise 2

```
 function mmult(A,B)
          (M,N) = size(A);
          C = zeros(M,M);
          for i=1:M
            for j=1:M
              for k=1:M
                C[i,j] += A[i,k]*B[k,j];
              end
            end
          end
          C;
        end

for d in [500,1000,1500,2000]
        a=rand(d,d)
        b=rand(d,d)
        @time mmult(a,b)
        end
```

```
elapsed time: 0.3470189571380615 seconds
elapsed time: 3.170802116394043 seconds
elapsed time: 23.14421010017395 seconds
elapsed time: 62.64995193481445 seconds
```

Theoretically, the number of arithmetic operations required to multiply two square matrices of order n is proportional to n^3. Hence, one could expect that the running time for $n = 1000$ should be $8 = (1000/500)^3$ times that for $n = 500$. But the above experimental results show a ratio close to 10. This is due to the high rate of cache misses in the naive algorithm for matrix multiplication. We saw in class better algorithms for this operation, in particular one based on a blocking strategy.

3 exercise 3

```
function qsort!(a,lo,hi)
           i, j = lo, hi
           while i < hi
               pivot = a[(lo+hi)>>>1]
               while i <= j
                   while a[i] < pivot; i = i+1; end
                   while a[j] > pivot; j = j-1; end
                   if i <= j
                       a[i], a[j] = a[j], a[i]
                       i, j = i+1, j-1
                   end
               end
               if lo < j; qsort!(a,lo,j); end
               lo, j = i, hi
           end
           return a
       end

function sortperf(n)
           qsort!(rand(n), 1, n)
       end

issorted(sortperf(5000)) // to test whether your alog is correct
true

[@time sortperf(2^e*1000000) for e=[0 1 2 3 4 5 6 7]]


elapsed time: 0.2307720184326172 seconds
elapsed time: 0.21168804168701172 seconds
```

```
elapsed time: 0.4441850185394287 seconds
elapsed time: 0.912261962890625 seconds
elapsed time: 1.9140989780426025 seconds
elapsed time: 3.977203130722046 seconds
elapsed time: 8.138957023620605 seconds
elapsed time: 16.795485973358154 seconds
```

Theoretically, the number of integer comparisons required to sort a list of n integers (using quick-sort) is proportional to $O(n \log(n))$. Hence, one could expect that the running time for $n = 2^7\,1000000$ should be 7/3 times that for $n = 2^6\,1000000$. (To verify this claim approximate 1000 with 2^{10}.) And, indeed, the above experimental results show a ratio close to 2. This is due to the relatively low rate of cache misses in quick sort algorithms.

4 exercise 4

```
function mergesort(data, istart, iend)
                if(istart < iend)
                        mid = (istart + iend) >>>1
                        mergesort(data, istart, mid)
                        mergesort(data, mid+1, iend)
                        merge(data, istart, mid, iend)
                end
        end

function merge( data, istart, mid, iend)
                n = iend - istart + 1
                temp = zeros(n)
                s = istart
                m = mid+1
                for tem = 1:n
                        if s <= mid && (m > iend || data[s] <= data[m])
                                temp[tem] = data[s]
                                s=s+1
                        else
                                temp[tem] = data[m]
                                m=m+1
                        end
                end
                data[istart:iend] = temp[1:n]
        end

issorted(mergesort(rand(100),1,100))   // to test whether your algorithm is correct
true
```

```
[@time mergesort(rand(2^e*1000000),1,2^e*1000000) for e=[0 1 2 3 4 5 6]]
elapsed time: 0.47666501998901367 seconds
elapsed time: 0.9581248760223389 seconds
elapsed time: 1.9626061916351318 seconds
elapsed time: 3.9991350173950195 seconds
elapsed time: 8.226263046264648 seconds
elapsed time: 16.87691307067871 seconds
elapsed time: 34.648082971572876 seconds
```

Theoretically, the number of integer comparisons required to sort a list of n integers (using merge-sort) is proportional to $O(n \log(n))$, as for quick-sort. Nevertheless, we can see that, for our input data sets, merge-sort is essentially twice slower than quick-sort. However, merge-sort has nicer properties with respect to parallelism, as we shall see later.