CSCE 463/612: Networks and Distributed Processing

Homework 3 Part 3 (50 pts)

Due date: 11/19/24

1. Description

The final protocol includes additional features of TCP – cumulative ACKs with pipelining, fast retransmit, and flow control (note that a single timer for the base of the window remains the same as in Part 2). To deal with high-loss scenarios below, *set the maximum number of retx attempts for all packets to 50.*

1.1. Code (25 pts)

The input and output is the same as in Part 2, except now the window size can be above 1:

```
Main:   sender W = 5000, RTT 0.200 sec, loss 0.001 / 0.0001, link 1000 Mbps
Main:   initializing DWORD array with 2^25 elements... done in 15 ms
Main:   connected to s3.irl.cs.tamu.edu in 0.219 sec, pkt size 1472 bytes
[  2] B    184 (  0.3 MB) N    368 T 0 F 1 W 184 S 1.077 Mbps RTT 0.219
[  4] B   2821 (  4.2 MB) N   5642 T 1 F 5 W 2821 S 15.408 Mbps RTT 0.218
[  6] B  11492 ( 16.9 MB) N  16492 T 2 F 12 W 5000 S 50.741 Mbps RTT 0.218
[  8] B  18075 ( 26.6 MB) N  23075 T 5 F 18 W 5000 S 38.522 Mbps RTT 0.218
[ 10] B  24680 ( 36.3 MB) N  29680 T 5 F 23 W 5000 S 38.651 Mbps RTT 0.218
[ 12] B  30291 ( 44.6 MB) N  35291 T 7 F 28 W 5000 S 32.835 Mbps RTT 0.218
[ 14] B  33904 ( 49.9 MB) N  38904 T 7 F 32 W 5000 S 21.142 Mbps RTT 0.218
[ 16] B  37389 ( 55.0 MB) N  42389 T 8 F 37 W 5000 S 20.393 Mbps RTT 0.218
[ 18] B  48062 ( 70.7 MB) N  51849 T 9 F 42 W 5000 S 62.459 Mbps RTT 0.248
[ 20] B  52999 ( 78.0 MB) N  57999 T 11 F 50 W 5000 S 28.890 Mbps RTT 0.248
[ 22] B  63145 ( 92.9 MB) N  68145 T 13 F 58 W 5000 S 59.371 Mbps RTT 0.216
[ 24] B  75527 (111.2 MB) N  80206 T 14 F 67 W 5000 S 72.458 Mbps RTT 0.226
[ 26] B  81802 (120.4 MB) N  86802 T 14 F 76 W 5000 S 36.683 Mbps RTT 0.226
[ 28] B  90697 (133.5 MB) N  91679 T 16 F 82 W 5000 S 52.051 Mbps RTT 0.226
[29.065] <-- FIN-ACK 91679 window FC6FB7CB
Main:   transfer finished in 28.626 sec, 37509.93 Kbps, checksum FC6FB7CB
Main:   estRTT 0.226, ideal rate 258854.99 Kbps
```

1.2. Report (25 pts)

All transfers that determine speed must have sufficient user buffer size to reach steady-state dynamics (i.e., the rate becomes stable). Once the transfer speed has stabilized, record this value for the analysis below. Several questions to address:

1. Set packet loss p to zero in both directions, the RTT to 0.5 seconds, and bottleneck link speed to S = 1 Gbps. Examine how your goodput scales with window size W. This should be done by plotting the steady-state rate $r(W)$ for $W = 1, 2, 4, 8, \ldots, 2^{10}$ and keeping the x-axis on a log-scale. Your peak rate will be around 24 Mbps and, depending on your home bandwidth, usage of an on-campus server might be necessary. Using curve-fitting, generate a model for $r(W)$. Discuss whether it matches the theory discussed in class.
2. Expanding on the previous question, fix the window size at W = 30 packets and vary the RTT = 10, 20, 40, …, 5120 ms. Plot stable rate $r(\text{RTT})$, again placing the x-axis on a log-scale. Perform curve-fitting to determine a model that describes this relationship. Due to queuing/transmission delays emulated by the server and various OS kernel overhead, the

actual RTT may deviate from the requested RTT. Thus, use the *measured* average in your plots and comment on whether the resulting curve matches theory.

3. Run the dummy receiver on your localhost and produce a trace using W = 8K (the other parameters do not matter as the dummy receiver ignores them, although they should still be within valid ranges). Discuss your CPU configuration and whether you managed to exceed 1 Gbps. How about 10 Gbps using 9-KB packets (see dummy-receiver discussion in Part 1)?
4. Use buffer size 2^{23} DWORDs, RTT = 200 ms, window size W = 300 packets, link capacity S = 10 Mbps, and loss *only in the reverse direction* equal to p = 0.1. Show an entire trace of execution for this scenario and compare it to a similar case with no loss in either direction. Does your protocol keep the same rate in these two cases? Why or why not?
5. Determine the algorithm that the receiver uses to change its advertised window. What name does this technique have in TCP? *Hint*: the receiver window does not grow to infinity and you need provide its upper bound as part of the answer.

1.3. Extra Credit (20 pts)

Achieve the speed in section 1.12 between ts.cse.tamu.edu (sender) and s3.irl.cs.tamu.edu (receiver). *Hint: parallelize the worker thread*.

1.4. Program Structure

It is recommended to structure your program around the three threads in Figure 1. When `ss.Open()` succeeds with the connection, it spawns the worker and stats threads to run in the background and provide support for the remainder of the transfer. These threads can be signaled to quit in `ss.Close()`; however, if the user deletes the socket class without calling `ss.Close()`, you may end up leaking memory or crashing. As a result, the destructor `~SenderSocket()` must check if these threads are still running and terminate them before deleting shared data objects inside the class (such as the queue of pending packets). The most common way of signaling termination is to set some shared event and then wait on both thread handles obtained from `CreateThread`. As a general rule, graceful termination of threads is the best method for avoiding unexpected crashes and various problems on exit.

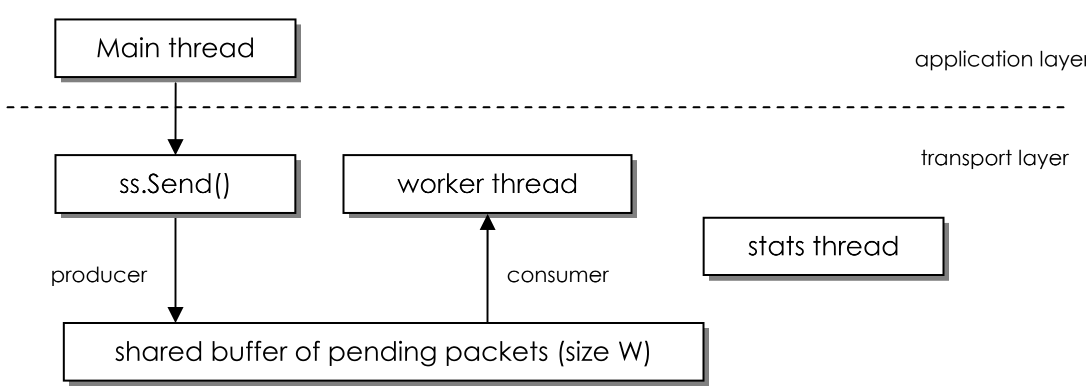

Figure 1. Organization of the program.

Function `ss.Send()` interacts with the worker thread through a shared queue of *pending* packets. This nicely maps to the bounded producer-consumer (PC) problem, where the fixed queue size is *W* packets. The easiest way to implement this is to block `ss.Send()` on a semaphore that counts the number of empty slots in the buffer. For each received ACK that moves the base forward by *k* packets, this semaphore gets released by the same *k* slots (except when doing flow control, see below). The buffer is a circular array of packets (each with `MAX_PKT_SIZE` bytes). The following pseudocode should provide a starting idea:

```
class Packet {
       int type;                           // SYN, FIN, data
       int size;                           // bytes in packet data
       clock_t txTime;                     // transmission time
       char pkt[MAX_PACKET_SIZE];          // packet with header
};

int SenderSocket::Send (char *data, int size)
{
   HANDLE arr[] = {eventQuit, empty};
   WaitForMultipleObjects (2, arr, false, INFINITE);
   // no need for mutex as no shared variables are modified
   slot = nextSeq % W;
   Packet *p = pending_pkts + slot;          // pointer to packet struct
   SenderDataHeader *sdh = p->pkt;
   sdh->seq = nextSeq;
   ...         // set up remaining fields in sdh and p
   memcpy (sdh + 1, data, size);
   nextSeq ++;
   ReleaseSemaphore (full, 1);
}
```

To reduce the amount of code duplication, it is best to process both SYN and FIN packets inside `ss.Open()` and `ss.Close()` through the same shared buffer. You can use additional variables and logic as needed, building upon the architecture explained above.

Next, the worker thread requires response to three events – a timeout, a packet is ready in the socket, and a new packet is in the send buffer. To manage all three, we need ability to wait on socket events in `WaitForMultipleObjects`. As mentioned in hw1.pdf, this is accomplished using `WSAEventSelect`, which associates some event `socketReceiveReady` with the socket. Make sure to create this event using `CreateEvent` (instead of `WSACreateEvent`) and specify the auto-reset option. Pseudocode:

```
void SenderSocket::WorkerRun (void)
{
  HANDLE events [] = {socketReceiveReady, full};
  while (true)
  {
    if (pending packets)
        timeout = timerExpire - cur_time;
    else
        timeout = INFINITE;

    int ret = WaitForMultipleObjects (2, events, false, timeout);
    switch (ret)
    {
        case timeout:  sendto(pending_packets[senderBase % W].pkt, ...);    // retx
                       break;
        case socket:   // move senderBase; update RTT; handle fast retx; do flow control
                       ReceiveACK ();
                       break;
        case sender:   sendto(pending_packets[nextToSend % W].pkt, ...);
                       nextToSend ++;
```

```
                    break;
        default:       handle failed wait;
    }

    if (first packet of window || just did a retx (timeout / 3-dup ACK)
        || senderBase moved forward)
        recompute timerExpire;
}
```

There are a few additional caveats. First, you need to ensure clean termination during timeouts. If the worker thread encounters the maximum number of retx on the same packet, it must unblock `ss.Send()` and somehow notify it that the connection has failed. Second, when `ss.Close()` is called, the function must block until the worker thread has collected all outstanding acknowledgments. Otherwise, the FIN packet may be rejected by the server and/or the transfer may be incomplete. Third, upon receiving an ACK that moves the base from x to $x + y$, an RTT sample is computed only based on packet $x + y – 1$ and only if there were no prior retransmissions of base x. Finally, when moving the window forward, reset the timer to current time plus the most-recent RTO (retransmission timeout).

1.5. Winsock Issues

By default, the UDP sender/receiver buffer inside the Windows kernel is configured to support only 8 KB of unprocessed data. You can achieve higher outbound performance and prevent packet loss in the inbound direction by increasing both buffers, then setting your worker thread to time-critical priority:

```
int kernelBuffer = 20e6;              // 20 meg
if (setsockopt (sock, SOL_SOCKET, SO_RCVBUF, &kernelBuffer, sizeof (int)) == SOCKET_ERROR)
       ...
kernelBuffer = 20e6;                  // 20 meg
if (setsockopt (sock, SOL_SOCKET, SO_SNDBUF, &kernelBuffer, sizeof (int)) == SOCKET_ERROR)
       ...
SetThreadPriority (GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);
```

1.6. Flow Control

The semaphore shared between the main and worker threads can be reused to easily accomplish flow control using this general architecture:

```
HANDLE empty = CreateSemaphore (NULL, 0, W, NULL)

// after the SYN-ACK, inside ss.Open()
int lastReleased = min (W, synack->window);
ReleaseSemaphore (empty, lastReleased);

// in the worker thread
while (not end of transfer)
{
    get ACK with sequence y, receiver window R
    if (y > sndBase)
    {
        sndBase = y
        effectiveWin = min (W, ack->window)

        // how much we can advance the semaphore
        newReleased = sndBase + effectiveWin - lastReleased
        ReleaseSemaphore (empty, newReleased)
        lastReleased += newReleased
    }
}
```

To test that flow control works, set the RTT to 2 seconds and observe the effective window reported by your program. It should expand once per printout.

1.7. Small Window, No Loss

```
Main:   sender W = 10, RTT 0.100 sec, loss 0 / 0, link 1000 Mbps
Main:   initializing DWORD array with 2^20 elements... done in 1 ms
Main:   connected to s3.irl.cs.tamu.edu in 0.102 sec, pkt size 1472 bytes
[  2] B    146 (  0.2 MB) N    156 T 0 F 0 W 10 S 0.860 Mbps RTT 0.101
[  4] B    326 (  0.5 MB) N    336 T 0 F 0 W 10 S 1.060 Mbps RTT 0.101
[  6] B    506 (  0.7 MB) N    516 T 0 F 0 W 10 S 1.060 Mbps RTT 0.102
[  8] B    696 (  1.0 MB) N    706 T 0 F 0 W 10 S 1.118 Mbps RTT 0.102
[ 10] B    876 (  1.3 MB) N    886 T 0 F 0 W 10 S 1.059 Mbps RTT 0.102
[ 12] B   1056 (  1.6 MB) N   1066 T 0 F 0 W 10 S 1.060 Mbps RTT 0.101
[ 14] B   1246 (  1.8 MB) N   1246 T 0 F 0 W 10 S 1.118 Mbps RTT 0.102
[ 16] B   1426 (  2.1 MB) N   1436 T 0 F 0 W 10 S 1.059 Mbps RTT 0.102
[ 18] B   1606 (  2.4 MB) N   1616 T 0 F 0 W 10 S 1.060 Mbps RTT 0.101
[ 20] B   1786 (  2.6 MB) N   1796 T 0 F 0 W 10 S 1.060 Mbps RTT 0.102
[ 22] B   1976 (  2.9 MB) N   1986 T 0 F 0 W 10 S 1.118 Mbps RTT 0.101
[ 24] B   2156 (  3.2 MB) N   2166 T 0 F 0 W 10 S 1.060 Mbps RTT 0.101
[ 26] B   2336 (  3.4 MB) N   2346 T 0 F 0 W 10 S 1.060 Mbps RTT 0.101
[ 28] B   2516 (  3.7 MB) N   2526 T 0 F 0 W 10 S 1.060 Mbps RTT 0.101
[ 30] B   2706 (  4.0 MB) N   2716 T 0 F 0 W 10 S 1.118 Mbps RTT 0.102
[31.938] <-- FIN-ACK 2865 window 5B0360D
Main:   transfer finished in 31.731 sec, 1057.47 Kbps, checksum 5B0360D
Main:   estRTT 0.102, ideal rate 1153.65 Kbps
```

1.8. Large Window, Low Loss

```
Main:   sender W = 12000, RTT 0.100 sec, loss 0.0001 / 0, link 1000 Mbps
Main:   initializing DWORD array with 2^28 elements... done in 882 ms
Main:   connected to s3.irl.cs.tamu.edu in 0.101 sec, pkt size 1472 bytes
[  2] B  13010 ( 19.2 MB) N  23854 T 0 F 5 W 12000 S 76.534 Mbps RTT 0.101
[  4] B  95384 (140.4 MB) N  97775 T 5 F 15 W 12000 S 484.992 Mbps RTT 0.166
[  6] B 164403 (242.0 MB) N 176403 T 6 F 22 W 12000 S 406.360 Mbps RTT 0.193
[  8] B 241856 (356.0 MB) N 248030 T 8 F 32 W 12000 S 456.028 Mbps RTT 0.103
[ 10] B 329306 (484.7 MB) N 341306 T 8 F 37 W 12000 S 514.867 Mbps RTT 0.201
[ 12] B 409685 (603.1 MB) N 420424 T 10 F 47 W 12000 S 473.247 Mbps RTT 0.152
[ 14] B 481884 (709.3 MB) N 493884 T 11 F 56 W 12000 S 425.018 Mbps RTT 0.183
[ 16] B 569132 (837.8 MB) N 581132 T 14 F 64 W 12000 S 513.688 Mbps RTT 0.190
[ 18] B 645244 (949.8 MB) N 657244 T 18 F 71 W 12000 S 448.123 Mbps RTT 0.159
[20.067] <-- FIN-ACK 733431 window E8F5B708
Main:   transfer finished in 19.854 sec, 432660.10 Kbps, checksum E8F5B708
Main:   estRTT 0.104, ideal rate 1347514.94 Kbps
```

1.9. Small Window, Moderate Loss

```
Main:   sender W = 10, RTT 0.010 sec, loss 0.1 / 0, link 1000 Mbps
Main:   initializing DWORD array with 2^20 elements... done in 1 ms
Main:   connected to s3.irl.cs.tamu.edu in 0.012 sec, pkt size 1472 bytes
[  2] B    512 (  0.8 MB) N    522 T 9 F 40 W 10 S 3.015 Mbps RTT 0.011
[  4] B    925 (  1.4 MB) N    935 T 25 F 74 W 10 S 2.431 Mbps RTT 0.011
[  6] B   1371 (  2.0 MB) N   1371 T 39 F 109 W 10 S 2.625 Mbps RTT 0.011
[  8] B   1808 (  2.7 MB) N   1815 T 51 F 146 W 10 S 2.572 Mbps RTT 0.011
[ 10] B   2245 (  3.3 MB) N   2255 T 63 F 189 W 10 S 2.561 Mbps RTT 0.011
[ 12] B   2696 (  4.0 MB) N   2706 T 76 F 229 W 10 S 2.654 Mbps RTT 0.011
[12.741] <-- FIN-ACK 2865 window 5B0360D
Main:   transfer finished in 12.662 sec, 2649.94 Kbps, checksum 5B0360D
Main:   estRTT 0.011, ideal rate 10889.41 Kbps
```

1.10. Bottlenecked by Win/RTT

```
Main:   sender W = 300, RTT 0.100 sec, loss 0.001 / 0, link 1000 Mbps
Main:   initializing DWORD array with 2^24 elements... done in 50 ms
Main:   connected to s3.irl.cs.tamu.edu in 0.102 sec, pkt size 1472 bytes
[  2] B   2634 (  3.9 MB) N   2934 T 0 F 2 W 300 S 15.495 Mbps RTT 0.101
[  4] B   7788 ( 11.5 MB) N   8088 T 0 F 7 W 300 S 30.345 Mbps RTT 0.101
```

```
[  6] B  12675 ( 18.7 MB) N  12975 T 0 F 11 W 300 S 28.773 Mbps RTT 0.101
[  8] B  16983 ( 25.0 MB) N  17283 T 0 F 19 W 300 S 25.364 Mbps RTT 0.101
[ 10] B  22393 ( 33.0 MB) N  22693 T 0 F 21 W 300 S 31.852 Mbps RTT 0.101
[ 12] B  27962 ( 41.2 MB) N  28262 T 0 F 25 W 300 S 32.789 Mbps RTT 0.101
[ 14] B  32918 ( 48.5 MB) N  33218 T 0 F 29 W 300 S 29.179 Mbps RTT 0.100
[ 16] B  38064 ( 56.0 MB) N  38364 T 0 F 32 W 300 S 30.298 Mbps RTT 0.101
[ 18] B  43815 ( 64.5 MB) N  43996 T 0 F 34 W 300 S 33.860 Mbps RTT 0.101
[19.051] <-- FIN-ACK 45840 window 85A854D4
Main:   transfer finished in 18.838 sec, 28499.30 Kbps, checksum 85A854D4
Main:   estRTT 0.101, ideal rate 34935.31 Kbps
```

1.11. Surviving Heavy Loss

```
Main:   sender W = 10, RTT 0.010 sec, loss 0.5 / 0, link 14 Mbps
Main:   initializing DWORD array with 2^15 elements... done in 1 ms
Main:   connected to s3.irl.cs.tamu.edu in 0.011 sec, pkt size 1472 bytes
[  2] B     23 (  0.0 MB) N     33 T 35 F 0 W 10 S 0.135 Mbps RTT 0.011
[  4] B     42 (  0.1 MB) N     52 T 71 F 0 W 10 S 0.112 Mbps RTT 0.011
[  6] B     75 (  0.1 MB) N     85 T 105 F 1 W 10 S 0.194 Mbps RTT 0.011
[ 6.829] <-- FIN-ACK 90 window FC694CF3
Main:   transfer finished in 6.698 sec, 156.54 Kbps, checksum FC694CF3
Main:   estRTT 0.011, ideal rate 10703.39 Kbps
```

1.12. Extra Credit

```
Main:   sender W = 3000, RTT 0.010 sec, loss 0 / 0, link 10000 Mbps
Main:   initializing DWORD array with 2^30 elements... done in 3563 ms
Main:   connected to s3.irl.cs.tamu.edu in 0.012 sec, pkt size 1472 bytes
[  2] B 127425 (187.6 MB) N 128424 T 0 F 0 W 3000 S 749.636 Mbps RTT 0.010
[  4] B 276493 (407.0 MB) N 278623 T 0 F 0 W 3000 S 877.665 Mbps RTT 0.022
[  6] B 439231 (646.5 MB) N 441182 T 0 F 0 W 3000 S 958.148 Mbps RTT 0.023
[  8] B 601846 (885.9 MB) N 603857 T 0 F 0 W 3000 S 957.424 Mbps RTT 0.024
[ 10] B 764429 (1125.2 MB) N 766370 T 0 F 0 W 3000 S 957.238 Mbps RTT 0.024
[ 12] B 926865 (1364.3 MB) N 928692 T 0 F 0 W 3000 S 956.314 Mbps RTT 0.024
[ 14] B 1089542 (1603.8 MB) N 1091262 T 0 F 0 W 3000 S 957.469 Mbps RTT 0.020
[ 16] B 1251824 (1842.7 MB) N 1254240 T 0 F 0 W 3000 S 955.404 Mbps RTT 0.028
[ 18] B 1414565 (2082.2 MB) N 1416538 T 0 F 0 W 3000 S 957.869 Mbps RTT 0.025
[ 20] B 1577166 (2321.6 MB) N 1579261 T 0 F 0 W 3000 S 956.920 Mbps RTT 0.027
[ 22] B 1739768 (2560.9 MB) N 1741676 T 0 F 0 W 3000 S 957.111 Mbps RTT 0.026
[ 24] B 1902007 (2799.8 MB) N 1904350 T 0 F 0 W 3000 S 955.204 Mbps RTT 0.030
[ 26] B 2064474 (3038.9 MB) N 2066779 T 0 F 0 W 3000 S 956.258 Mbps RTT 0.024
[ 28] B 2227129 (3278.3 MB) N 2229134 T 0 F 0 W 3000 S 957.666 Mbps RTT 0.021
[ 30] B 2389773 (3517.7 MB) N 2391930 T 0 F 0 W 3000 S 957.578 Mbps RTT 0.024
[ 32] B 2552442 (3757.2 MB) N 2554286 T 0 F 0 W 3000 S 957.766 Mbps RTT 0.024
[ 34] B 2715008 (3996.5 MB) N 2716917 T 0 F 0 W 3000 S 956.921 Mbps RTT 0.024
[ 36] B 2877201 (4235.2 MB) N 2879481 T 0 F 0 W 3000 S 954.741 Mbps RTT 0.027
[36.731] <-- FIN-ACK 2933721 window 39F2A0E6
Main:   transfer finished in 36.697 sec, 936307.17 Kbps, checksum 39F2A0E6
Main:   estRTT 0.024, ideal rate 1456380.85 Kbps
```

463/612 Homework 3 Grade Sheet (Part 3)

Name: ______________________________

Function	Points	Break down	Item	Deduction
Test case 1.7	5	1	Incorrect rate (not within 20%)	
		1	Incorrect T	
		1	Incorrect F	
		1	Incorrect FIN-ACK sequence	
		1	Checksum does not match receiver	
Test case 1.8	5	1	Incorrect rate (not within 20%)	
		1	Incorrect T (not within 20%)	
		1	Incorrect F (not within 20%)	
		1	Incorrect FIN-ACK sequence	
		1	Checksum does not match receiver	
Test case 1.9	5	1	Incorrect rate (not within 20%)	
		1	Incorrect T (not within 20%)	
		1	Incorrect F (not within 20%)	
		1	Incorrect FIN-ACK sequence	
		1	Checksum does not match receiver	
Test case 1.10	5	1	Incorrect rate (not within 20%)	
		1	Incorrect T (not within 20%)	
		1	Incorrect F (not within 20%)	
		1	Incorrect FIN-ACK sequence	
		1	Checksum does not match receiver	
Test case 1.11	5	1	Incorrect rate (not within 20%)	
		1	Incorrect T (not within 20%)	
		1	Incorrect F (not within 20%)	
		1	Incorrect FIN-ACK sequence	
		1	Checksum does not match receiver	
Report	25	5	Model of r(W)	
		5	Model of r(RTT)	
		5	Dummy receiver trace	
		5	Reverse packet loss trace	
		5	Receiver window-resize algorithm	

Total points: ________________