1. (40 points) Using the multistage switch simulation program, evaluate the performance of the resequencing buffer. Specifically, simulate a network with topology $B_{256,4}$ an input buffer with 64 cells, SEs with 64 cells, a resequencer with 64 cells and an output buffer with 256 cells. Let the speedup be 1.1. Make a chart that shows the fraction of cells that are older than the age threshold when they reach the resequencer or that are discarded because there is no room for them in the resequencer, as a function of the input load. Include separate curves for age thresholds of 32, 48, 56 and 60.

Re-run the simulation using the modified traffic source you wrote for homework 5 (you may borrow from the posted solution, if you did not complete it yourself). Use a period of 20 cell times. Produce a chart showing the late percentages for this case.

The results for the fixed threshold resequencer are shown below.

Modify the resequencer to implement the adaptive resequencing algorithm described in the lecture notes on pages 4-68 and 4-69. Run the simulation for this case, using both traffic models and produce a chart showing the fraction of received cells that are late or that are discarded because there is no room for them in the resequencer (in this case, a received cell is considered late if its timestamp is smaller than that of a cell which has already been sent).
Plot separate curves for $\gamma=12$, 16, 20 and 24. Set the measurement window size equal to $\gamma$. Compare them to the results obtained earlier and comment.

The chart for the adaptive resequencer is shown below. The results for the adaptive resequencer are generally better for bursty traffic. For non-bursty traffic the results are more mixed, although the best result for the adaptive resequencer is still better than the best result for the non-adaptive resequencer. It's interesting to note that the best values of $\gamma$ differ between bursty and non-bursty traffic, with small $\gamma$ being best for bursty traffic and larger for non-bursty. There is an interesting interaction between the fraction of cells that are late and the fraction that must be discarded because there is no room for them in the resequencer. When $\gamma$ is small, most of the late-or-lost cells are late. When $\gamma$ is larger, most are lost. For the bursty case, most of the late+lost cells are lost. A larger resequencer depth is needed to improve the performance significantly.

It makes sense that the improvement for the adaptive resequencer is greatest for bursty traffic. One would expect larger differences as the burstiness increases further, although the overall performance can be expected to deteriorate.

The program that implements the adaptive resequencer is shown below.

```c
// Header file for adaptive resequencing buffer class
class ReseqA {
    int n;              // number of buffers in vector
    int depth;          // number of slots in buffer
    int win;            // size of measurement window
    int maxVar;         // max short term delay variation
    int agethresh[Maxn]; // age threshold
    int maxDnow[Maxn];  // max delay seen in current window
    int maxDlast[Maxn]; // max delay seen in previous window
    int lastts[Maxn];   // timestamp of last cell sent
}
```
int     gd[Maxn];       // downstream grants
Cqueue *q;             // actual queue of cells
int     nout;           // number output
int     nlost;          // number discarded
int     nqd;            // number queued
int     nlate;          // number older than threshold upon arrival
int     tdelay;         // time spent in resequencing buffer
int     stattime;       // time stats last cleared

public:         ReseqA(int,int,int,int);
void    grant(Flagvec,Flagvec);
void    flow(Cvec,Cvec);
void    stats(int);
void    clearstats();

#include "netsim.h"
#include "Cqueue.h"
#include "ReseqA.h"

ReseqA::ReseqA(int n1, int d, int w, int mv) {
    n = n1;
    depth = d;
    win = w;
    maxVar = mv;
    q = new Cqueue[n];
    for (int j = 0; j < n; j++) {
        agethresh[j] = maxVar;
        maxDnow[j] = maxDlast[j] = 0;
        lastts[j] = 0;
    }
}

void ReseqA::grant(Flagvec outg, Flagvec ing) {
    // Propagate grants.
    for (int j = 0; j < n; j++) {
        ing[j] = 1; gd[j] = outg[j];
    }
}

void ReseqA::flow(Cvec inp, Cvec outp) {
    // Flow cells through resequencing buffers.
    int qs;
    for (int j = 0; j < n; j++) {
        if (gtime % win == 0) {
            maxDlast[j] = maxDnow[j];
            maxDnow[j] = 0;
        }
        qs = q[j].qsiz();
        if (qs>0 && gtime-q[j].qfirst()->ts >= agethresh[j] && gd[j]) {
            outp[j] = q[j].deq();
            tdelay += (gtime - outp[j]->scr0);
            lastts[j] = max(lastts[j],outp[j]->ts);
            nout++;
        } else outp[j] = NULL;
        if (inp[j] != NULL) {
            maxDnow[j] = max(maxDnow[j],gtime - inp[j]->ts);
            agethresh[j] = max(maxDnow[j],maxDlast[j]) + maxVar;
            if (inp[j]->ts < lastts[j]) {
                nlate++;
            } else {
                nlost++;
                delete inp[j];
            }
        } else {
            nqd++;
            inp[j]->scr0 = gtime;
            q[j].tenq(inp[j]);
        }
    }
}
inp[j] = NULL;
}
}
}

void ReseqA::stats(int verbose) {
// Print statistics.
    switch (verbose) {
    default: break;
    case -1: printf(" %5.2f %9.3g %9.3g ",
            double(tdelay)/double(nout),
            double(nlate)/double(nlost+nqd),
            double(nlost)/double(nlost+nqd));
            break;
    case 0: // just the facts ma'm
            printf(" %4.2f %10.8f %10.8f ",
            double(nout)/(double(gtime-stattime)*n),
            double(nlate)/double(nlost+nqd),
            double(nlost)/double(nlost+nout));
            break;
    case 2: // the works
    case 1: // terse version
            printf("#	cload frac_late  frac_lost\n");
            printf("\t%4.3f %10.8f %10.8f\n",
            double(nout)/(double(gtime-stattime)*n),
            double(nlate)/double(nout),
            double(nlost)/double(nout));
    }
}

void ReseqA::clearstats() {
// Clear statistics.
    stattime = gtime;
    for (int j = 0; j < n; j++) {
            tdelay = nout = nlost = nlate = nqd = 0;
    }
}
The chart below is a time history, showing the adaptive resequencer behavior for a test scenario in which the load at a single measured output starts out at 100%, is increased to 400% (causing cells to back up within the switching network), then decreased to 50%. Note that the number of cells in the resequencer remains small, even when the age threshold grows very large.
2. Consider a 32 port version of the system described on page 5-14 of the lecture notes. Suppose that the interconnection network uses dynamic routing, that the topology is $D_{32,2,2}$ and that the links operate at 600 M b/s. If at most 10% of the outgoing traffic is multicast, what is the maximum amount of traffic that can be carried on any of the “recycling paths”?

Since the multicast traffic is evenly distributed over all the network outputs as it is copied in the first pass, and since the total amount of outgoing multicast traffic is 60 M b/s on each outgoing link, each recycling path will carry at most 60 M b/s of traffic.

What is the maximum amount of traffic that can be present on a link in stage 1?

A stage 1 link can carry 600 M b/s of traffic from the external link, plus another 60 M b/s from the recycling data path.

What is the maximum amount of traffic on a link in stage 2? stage 3? stage 4? stage 5? stage 6? stage 7?

660 M b/s in stages 2, 5, 6 and 7. 1320 M b/s in stages 3 and 4.

3. In the WUGS architecture, many-to-many multicast can be implemented in one of two ways. Either you can use separate one-to-many multicast trees for each participant, or you can use a single shared tree. Consider a many-to-many multicast with 20 participants (each can both send and receive) and an output bandwidth of 100 M b/s (that is, each output link that is participating in the multicast sees a total of 100 M b/s, so each participating input link contributes an average of 5 M b/s).

Using separate multicast trees, how many multicast routing table entries are needed? How many are needed with a shared tree?

With separate multicast trees, you need $20 \times 18 = 360$ table entries. With a shared tree, you need $20 + 19 = 39$ entries.

What is the total recycling bandwidth used with separate multicast trees? What is the recycling bandwidth with a single shared tree?

With separate trees, you need $20 \times 17 \times 5 = 1700$ M b/s of recycling bandwidth. With a shared tree, you need $19 \times 100 = 1900$ M b/s.

How many control cells are needed to modify routing table entries when adding another participant, if we use separate multicast trees? If we use a single shared tree?

With separate trees, you need to modify $20 + 2 \times 20 = 60$ entries. With a shared tree, you need to change 3 table entries.

4. Suppose you are asked to design a switching system that uses static routing, has 2048 inputs and outputs, uses the three stage Clos network topology and supports external links of 1 Gb/s with virtual circuit rates from 0 up to 100 M b/s. Assuming that the first stage switches have 64 inputs and the last stage switches have 64 outputs, how fast should the internal links be to make the system nonblocking for unicast traffic if there are 32 middle stage switches? How fast should they be to make it nonblocking for multicast traffic? How fast to be reroutably nonblocking for multicast traffic? In this last case what is the fanout
restriction in the first stage? Repeat the above, assuming 1 and 8 middle stage switches. Organize your answers in the form of a table with three rows and four columns.

In the table below, the first column gives the internal link bandwidth needed to make the network nonblocking for unicast traffic. The second column gives the internal link bandwidth needed to make it strictly nonblocking for multicast. The third column gives the bandwidth needed to make it reroutably nonblocking for multicast, assuming that the first stage fanout is restricted to the value given in the last column. Other values for the fanout restriction result in larger internal link bandwidths.

<table>
<thead>
<tr>
<th></th>
<th>unicast</th>
<th>multicast s.n.b.</th>
<th>multicast r.n.b.</th>
<th>fanout restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r=3$</td>
<td>3.86 Gb/s</td>
<td>64 Gb/s</td>
<td>11.38 Gb/s</td>
<td>3</td>
</tr>
<tr>
<td>$r=8$</td>
<td>12.88 Gb/s</td>
<td>64 Gb/s</td>
<td>38.44 Gb/s</td>
<td>3</td>
</tr>
<tr>
<td>$r=1$</td>
<td>64 Gb/s</td>
<td>64 Gb/s</td>
<td>64 Gb/s</td>
<td>1</td>
</tr>
</tbody>
</table>

5. For each of the following systems give the speed advantage needed to make them nonblocking for unicast traffic. Assume there is no minimum virtual circuit rate and that the highest virtual circuit rate is one fourth the link rate.

Static routing network with topology $C^{3}_{100,10,15}$.

The appropriate inequality is $r > 2\left[\frac{d - (B / \beta)}{(1 / \beta) - (B / \beta)} - 1\right]$. In this case, we have

$15 > 2\left[\frac{9.75}{(1 / \beta) - .25} - 1\right]$ or equivalently, $8.5 > \left[\frac{9.75}{(1 / \beta) - .25}\right]$. The smallest value of the speed advantage that makes this inequality hold, satisfies $8 = \frac{9.75}{(1 / \beta) - .25}$ so $1 / \beta = \frac{9.75}{8} + .25 = 1.47$.

Static routing network with topology $B_{125,5}$.

The appropriate inequality is $1 / \beta > (1 - 2 / d)(B / \beta) + (2 / d)(1 + (d - 1)(k - 1))$. In this case, we have $1 / \beta > .4(1 / 4) + (.4)(1 + (4)(2)) = 3.7$.

6. For each of the following systems give the speed advantage needed to make them strictly nonblocking for multicast traffic. Assume there is no minimum virtual circuit rate and that the highest virtual circuit rate equals half the link rate.

Static routing network with topology $B_{100,10} ; B_{100,10}$, assuming that routes are setup with a maximum fanout of 10 in each of the two subnetworks.
The appropriate inequality is $1 / \beta > (B / \beta) + ((f + 1) / d)(1 + (d - 1)(k - 1))$. In this case, we have $1 / \beta > (1 / 4) + (.9)(1 + (9)(1)) = 9.25$.

Dynamic routing network with topology $D_{4096,8,1}^*$.

The appropriate inequality is $1 / \beta > d^{\lceil (k-h)/2 \rceil} = 8^{\lceil 3/2 \rceil} = 8$.

7. Explain why the network $X_{10,5} \otimes X_{10,10} \otimes X_{5,10}$ is rearrangeably nonblocking for unicast reservations when $B=b=\beta=1/2$. Describe a procedure for selecting a set of compatible routes for a given set of compatible unicast reservations.

Because $B=b=\beta=1/2$, each external link can carry a single virtual circuit and each internal virtual circuit can carry 2. Consequently, we can route the virtual circuits using a coloring procedure similar to the one used for the Benes network. In particular, we construct a connection graph with one node for each switch in the first stage and the third stage, and we place edges between every pair of nodes where we want to have a connection. We then color the edges of this graph using 10 colors (we can always do this, since each vertex in the connection graph can have vertex degree no more than 10). We then convert this to a routing by associating pairs of colors with each center stage switch (so for example, connections whose edges are colored 0 or 1, go through the first center stage switch, connections whose edges are colored 2 or 3, go through the second center stage switch, and so forth).

8. Consider an ATM switch using the topology $C_{4096,16,20}^5$. Assume that the external links have a bandwidth of 1 Gb/s and the internal links have a bandwidth of 1.5 Gb/s. Use Lee’s method to derive an expression for the blocking probability for this network when all virtual circuits have a bandwidth of 1 Gb/s. Express the blocking probability in terms of $p$, the expected fraction of each external link that is in use. What if every virtual circuit has a bandwidth of 500 Mb/s? What if every virtual circuit has a bandwidth of 300 Mb/s? Plot the blocking probability as a function of $p$ for values of $p$ between .5 and 1 for all three cases. Use a logarithmic scale for the y axis.

The diagram below shows the set of all such channels between an input $x$ and output $y$. The labels indicate the probability that channels at each stage are busy.
So the expression for blocking probability is

\[ p = \left(1 - (1 - 0.8)^3 \right) \left\{ 1 - \left[ \left(1 - (1 - 0.64)\right)^2 \right]^{20} \right\} \]

If all virtual circuits have a bandwidth of 500 M b/s, then each internal link can carry exactly three virtual circuits. This is equivalent to replacing each internal link with three parallel links, each capable of carrying one virtual circuit. So, the blocking probability expression becomes

\[ p = \left(1 - (1 - 0.8)^3 \right) \left\{ 1 - \left[ \left(1 - (1 - 0.64)\right)^2 \right]^{20} \right\} \]

If all virtual circuits have a bandwidth of 300 M b/s, each internal link can carry five virtual circuits. So, the blocking probability expression becomes

\[ p = \left(1 - (1 - 0.8)^5 \right) \left\{ 1 - \left[ \left(1 - (1 - 0.64)\right)^2 \right]^{20} \right\} \]

The chart is shown below.

9. Give an expression the blocking probability for the following networks using Lee's method. Assume each external link and internal link can carry just one virtual circuit at a time. Let \( p \) denote the probability that each network input is busy.

\[ C_{30,10,15}^3 \quad [1 - (1 - (2p/3))^2]^{15} \]
\[(X_{2,2} \times X_{2,2}) \otimes X_{2,2} \otimes X_{4,4} \quad \{1 - (1 - p)[1 - (1 - p)^2] \}^2\]

\[D_{64,2,2}^* \quad \{1 - (1 - p)^2 [1 - (1 - p)^5] \}^2\]

10. (40 points) Write a simulator to determine the blocking probability for a static routing network in which all flows have the same bandwidth. In particular, simulate the network B_{256,4} and evaluate the blocking probability for flows with bandwidth equal to the internal link rate, or \(\frac{1}{2}\) the internal link rate or \(\frac{1}{4}\), or \(\frac{1}{16}\) or \(\frac{1}{64}\). Produce a chart of the blocking probability as a function of the input load, showing a separate curve for each value of the flow bandwidth. Use a log scale for the blocking probability. Also on your chart, plot comparable results obtained using Lee’s method.

To perform the simulation, first create a background load, by reserving random paths through the network in order to create a specified input load. Then randomly select an input and an output with spare bandwidth available and determine if there is an available path connecting the selected input and output. Repeat this last step for 1000 randomly selected input and output pairs, then create a new random background load and select 1000 more input/ output pairs. Continue until you have checked a total of \(10^6\) pairs. The fraction of pairs for which no path could be found is the simulated blocking probability.

The program implementing the simulation is shown below.

```c
#include "stdinc.h"
#define MAXSTAGES 31

int d;   // SE dimension
int k;   // 2k-1 stages
int n;   // n inputs and outputs
int ld;   // lg(d)
int m;   // number of channels per link
int *busy[MAXSTAGES+1]; // busy[i][j] = number of channels

int rho(int a, int b) {
    // Rotate the b rightmost d-ary digits of a one position to the right.
    int t1 = a & (-1 << b*ld);
    int t2 = (a & ((1 << ld) - 1)) << ld*(b-1);
    a = a & ((1 << b*ld) - 1);
    return (a >> ld) | t1 | t2;
}
```

- 10 -
int irho(int a, int b) {
// Rotate the b rightmost d-ary digits of a one position to the left.
    int t1 = a & (-1 << b*ld);
    int t2 = (a >> ld*(b-1)) & ((1 << ld) - 1);
    a = a & ((1 << ld*(b-1)) - 1);
    return (a << ld) | t1 | t2;
}

int suc(int i, int j) {
// Return successor of link (i,j) in Benes network
    if (i == 0) return j;
    return i< k? rho(j,k+1-i): irho(j,i+2-k);
}

int pred(int i, int j) {
// Return predecessor of link (i,j) in Benes network
    if (i == 1) return j;
    return i<=k? irho(j,k+2-i): rho(j,i+1-k);
}

void bgLoad(double p) {
// Create background load.
    int h,i,j,s,x,y;
    static int *vec = NULL;

    // create input load
    h = 0;
    for (i = 0; i < m*n; i++) {
        if (randfrac() < p) {
            if (h < m*n-1) {
                busy[0][i/m]++; h++;
            }
        }
    }

    // Create a vector listing all integers from [0,md-1].
    // Used below to sample-without-replacement
    if (vec == NULL) {
        vec = new int[d*m];
        for (i = 0; i < d*m; i++) vec[i] = i;
    }

    // propagate input load through following stages
    for (i = 1; i <= 2*k-1; i++) {
        for (j = 0; j < n/d; j++) {
            s = 0;
            for (h = 0; h < d; h++) {
                s += busy[i-1][pred(i,j*d+h)];
            }
            for (h = 0; h < s; h++) {
                x = randint(h,d*m-1);
                busy[i][j*d+vec[x]/m]++;
                y = vec[h]; vec[h] = vec[x]; vec[x] = y;
            }
        }
    }
}

int connect(int i, int j, int i2) {
// Return 1 if there is a path available from link (i,j1) to link
// (2k-1-i,j2), else return 0.
    int r;
    if (busy[i][j1] == m || busy[(2*k-1-i)][j2] == m)
        return 0;
    if (i == k-1) return 1;
for (r = 0; r < d; r++) {
    if (connect(i+1, d*(suc(i, j1)/d)+r,
          pred((2*k-1)-i, d*(j2/d)+r)
        ) == 1)
        return 1;
}
}

int attempt() {
    // Select an input and output with available capacity at random
    // and attempt to find a path between them. Return 1 if path found,
    // else zero.
    
    int lnk1, lnk2;
    do {
        lnk1 = randint(0, n-1);
    } while (busy[0][lnk1] == m);
    do {
        lnk2 = randint(0, n-1);
    } while (busy[2*k-1][lnk2] == m);
    return connect(0, lnk1, lnk2);
}

void dump() {
    // For debugging.
    int i, j;
    for (j = 0; j < n; j++) {
        for (i = 0; i <= 2*k-1; i++)
            printf(" %2d", busy[i][j]);
        printf("\n");
    }
    printf("\n");
}

main(int argc, char* argv[]) {
    int i, j, r1, r2, lnk, blocked;
    double p;
    argc--; *argv++;
    if (argc != 6 ||
        sscanf(*argv++, "%d", &d) != 1 ||
        sscanf(*argv++, "%d", &k) != 1 ||
        sscanf(*argv++, "%d", &m) != 1 ||
        sscanf(*argv++, "%lf", &p) != 1 ||
        sscanf(*argv++, "%d", &r1) != 1 ||
        sscanf(*argv++, "%d", &r2) != 1 ) {
        fatal("usage: benesBlock d k m p r1 r2");
    }
    if (p >= 1) fatal("Background load must be <1.");
    n = int(pow(double(d), double(k)));
    ld = 1; for (i = d; i > 2; i >>= 1) ld++;
    blocked = 0;
    for (i = 0; i <= 2*k-1; i++) busy[i] = new int[n];
    for (i = 1; i <= r2; i++) {
        for (j = 0; j <= 2*k-1; j++)
            busy[i][j] = 0;
        bgLoad(p);
        for (j = 1; j <= r1; j++)
            blocked += (attempt() == 1 ? 0 : 1);
    }
    printf("%2d %2d %3d %5.3f %11.4e\n",
            d, k, m, p, double(blocked)/double(r1*r2));
}
The chart is shown below. The simulation data is shown as squares, the analytical results are shown as solid lines. The analysis predicts higher blocking probability than the simulation. This is expected, since the analysis is based on Lee's algorithm and the independence assumption made by Lee's algorithm causes it to consistently over-estimate the true blocking probability. The difference is fairly large in this case, because the switch elements are small (4 ports). For larger switch elements, the difference between the simulation and analytical results would be much smaller.