

Calculation Of Win-Loss Distributions For Blackjack

by

JOHN HAN CHANG

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements of the Degree of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1985

© John Han Chang 1985

The author hereby grants to M.I.T permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author: _____

Department of Electrical Engineering and Computer Science

August 21, 1985

Certified by: _____

Nesmith Ankeny

Thesis Supervisor

Accepted by: _____

David Adler

Chairman, Undergraduate Thesis Committee

ARCHIVES
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

DEC 11 1985

Blackjack is the only casino game that can be beaten fairly, without the aid of any external devices. Since Professor Thorpe published *Beat The Dealer* in 1962, its popularity has increased dramatically. Although blackjack can be beaten, the casinos obviously would not continue to offer the game if they lost money on it. Winning play requires a combination of proper play and money management.

This paper discusses a procedure to calculate the distribution of win and loss for any hand in the game of blackjack. (For an explanation of the rules of blackjack, consult the Appendix.) Why is this important? Knowledge of the distribution of outcomes allows you to determine precisely how much to bet in favorable situations. Knowledge of this distribution also enables you to determine how much to bet when playing multiple hands. One interesting fact is that the dealer wins an arbitrary hand significantly more often than a player does, even when the composition of the shoe favors the player. It is just that when the player wins, he often wins after doubling or splitting. Moral: if you're out to double your money in one shot, don't play blackjack.

There are various methods that have been used to calculate the basic strategy for blackjack. Originally, a close approximation was achieved by simulating the results of thousands of hands on a computer. The precise advantage of the player and the optimal play in close decisions remained unknown. In addition, small variations in the rules would necessitate a resimulation. The major disadvantage of the Monte Carlo technique (simulation) is that it breaks down when looking at subsamples that occur infrequently. The technique requires a large number of random samples to achieve any given accuracy. One situation in which the technique fails is in determining the probability of losing four bet units from one initial hand. (This unfortunate event could arise if one were to split a pair, double down on each of the resulting hands, and lose everything.) This event occurs perhaps one in three thousand hands; a sample of 1000 would require 3 million hands to be simulated. If one were interested in an even rarer

event, say losing four when having a pair of eights, that number would be even larger.

My approach to deriving the basic strategy makes one major assumption. I assume that the cards played by the player do not change the probabilities of the dealer totals. Actually the assumption is more encompassing. I assume that the outcome of any hand does not change the probability of the outcome of any other hand. For the casino game, which involves at least one deck of cards, this assumption is valid. Only when a few cards remain does such an approximation become invalid. Consider the following situation: two cards remain unseen—a ten and an 8. The dealer has a ten showing, and the player has a total of 10 on his first two cards. This program would assign independent probabilities of .5 to both the player and the dealer getting 20 and 18. Consequently, the program would calculate a .5 probability of a tie, which results from the .25 probability of tying at 18 and the .25 probability of tying at 20. It also calculates that the player has a .25 probability of winning and a .25 probability of losing. Of course, the probabilities are dependent: if the player has 20, the dealer must have 18, and vice versa. A tie is impossible. This is, however, an extreme case.

In general, this assumption produces a slightly pessimistic expectation for the player. Epstein describes the error as 'miniscule,' amounting to an expectation error of .003 for one deck, a figure determined by simulating the results of millions of hands. The error for more than one deck is even less, since the effects of removal of any one card become smaller when the number of cards increases. The converse is also true.

Griffin describes the difference in expectation for n decks as varying as $1/n$. For example, if the expectation for one deck is x and that for an infinite deck is y , then the expectation for 2 decks is approximately halfway between at $(x + y)/2$; in general, for n decks, the expectation is $y + (x - y)/n$. An 'infinite' deck is one in which removal of any card does not affect the probability for the next card. My approximation has

no error when dealing with an 'infinite' deck, since independence between hands is a property of such a deck. For four decks, the error is approximately $.003/4 = .00075$.

In general it is more favorable to play a game with as few decks as possible. Griffin attributes the gain in playing fewer decks mostly to the increased favorability of doubling; it usually favors the player to remove the cards composing a hand that adds to 9, 10, or 11, a hand that is often doubled. These cards tend to favor the dealer when they are prevalent in the deck because they have a tendency to help his hand reach a total from 17 to 21. Tens, on the other hand, when drawn as hit cards, more often than not bust the hand to which they are drawn. For a hand totaling 9, 10, or 11, however, they are most favorable. Since it is more likely that a ten or ace will be drawn after smaller cards have been removed, it is also more likely that either the dealer will bust, or the player will have a good hand. The likelihood of these events increases as the number of decks the game is dealt from decreases, because, again, the effect of removal of any one card increases as the number of decks decreases.

Discussion of assumption

The reason for this assumption is to reduce the order of the algorithm from $O(n^m)$ to $O(nm)$, where:

n = number of card sequences that can develop from the original card

m = number of hands possible

= Dealer's hand + player's hand + number of additional hands due to splitting

= 2 + Number of times splitting is allowed.

If the probability for each hand is calculated exactly, a split pair would necessitate calculating the probability for each possible sequence of cards in a hand that could develop from the initial hand, then calculating probabilities for all possibilities for the second hand, and finally calculating those for the dealer. For each possible development

of the players hand, one must calculate the dealer probabilities, which results in n^2 calculations. If the player splits, for each possible developed hand from the first, the program must enumerate all possible hands from the second as well as those of the dealer, resulting in n^3 calculations.

An initial hand contains, on average, approximately 1000 possible card sequences. Therefore, if n is 1000, a billion (n^3) calculations ensue. If resplitting up to three times (four hands) is allowed, the analysis requires $O(n^5) \simeq 10^{15}$ operations. For a computer that does 10^7 operations per second, this would take at least three years, an unreasonable length of time for just one hand. To consider all 550 dealer-player initial hand combinations would take over a millenium.

A clever approach might avoid recalculating the probabilities for each subsequent split hand, since the hands that can result are a subset of the ones being considered. If the number of different probabilities is sufficiently small, a table can be constructed containing the probabilities for each possible hand. Nonetheless, the analysis given above still holds; the amount of time each operation takes can be reduced (because table lookup is about 1000 times faster than these calculations), but the aggregate number is still too large to deal with. The running time might be reduced to a year on a powerful minicomputer, such as a VAX with floating point hardware support.

The assumption of independence among hands allows one to calculate all probabilities for one player hand. The dealer probabilities can be calculated separately (in fact, they are calculated first). The probabilities of each total for both player and dealer are then compared to produce an expectation and a distribution of win, loss, or tie. For example, if the dealer probabilities of various totals is calculated as:

Dealer total:	17	18	19	20	21	bj	bust
P(total):	.114	.113	.114	.333	.036	.078	.213

(These are approximate probabilities when the dealer has a 10 showing.)

And player probabilities of totals are:

Player total:	< 17	17	18	19	20	21	bj	bust
P(total):	.385	.078	.073	.078	.078	.307	0	0

(These are approximate probabilities when the player has a total of 11, given optimal play against the dealer 10.)

Then, the distribution is:

Win	-2	-1	0	1	2
P(win)	.388	.078	.063	0	.471

The program maximizes expectation to determine the optimal play; in this example, the optimal play is to double down the 11 against the dealer's 10. The result is either to win twice the original bet, to lose twice, or to tie. A third possibility, to lose the original bet, occurs only when the dealer has blackjack.

In any case, the independence assumption results in $O(2^n) \sim 2000$ calculations, regardless of whether the player has split. In comparison to the exact calculations, the results are instantaneous. If he splits, each hand is considered identical to the first; only calculating the distribution of win/loss becomes more involved. An example of one of the cases that must be enumerated is calculating the probability of tying after a split. This can result from the player losing one hand and winning the other; it can also result from tying both hands; it can even result from losing one double down and winning another.

Program discussion

Usage

The program, written in C, was compiled and run on both an IBM PC using Microsoft C 3.0 and a VAX 11/750 under 4.2 BSD Unix. It was written as a large group of sub-programs, listed on pages 24-38. Two calling routines were written to access these sub-programs. One, on pages 17-19, takes as input player and dealer hands. Each hand is delimited by a 0. The number of decks is specified on the command line when the program is run. If zero decks is input, the program takes the next 10 numbers as the number of cards corresponding to its position in the list: the first number is the number of aces; the second, the number of twos; and so on.

A second calling routine, on pages 20-23, is included for completeness. It generates all possible hands for a specified number of decks and adds their contributions to the total expectation, win-loss distribution, and distribution of hand totals.

The program not only derives the optimal strategy for any player and dealer hands, but also calculates the distribution of outcomes for such a strategy. Given such a distribution, the precise amount to bet for a given level of risk can be calculated exactly.

Dealer probabilities

Dealer probabilities are calculated independently of the player's probabilities. The algorithm for the routine is as follows:

Dealer probability routine (dprobe)

If the dealer's hand is less than 17,

For each card that does not bust the hand,

Take a card from the shoe and add it to the dealer's hand

Call the dealer probability routine.

Remove this card from the hand and replace it in the shoe.

Sum probabilities of dealer totals; subtract from one to get P(bust).

Otherwise, add the probability of this hand to the probability for its total.

Hitting routine

This is another recursive routine. The routine compares the expectation of drawing with the expectation of standing. The question is not resolved until the criterion to stop is achieved. This criterion is that the expectation of standing must be greater than -1, and that the hand must be at least a hard 17 or a soft nineteen. Why? From first principles, the stopping criterion should be a total of 21. For the vast majority of deck compositions, however, the expectation of hitting hard 17 or soft 19 versus any dealer upcard is less than that of standing. Also in the vast majority of cases, if it is best to stand on a total t , then it is best also to stand on any total greater than t . A refinement to the routine is that only cards that do not bust the hand are dealt; this increases the speed of the routine as well, since floating point calculations take place only when cards are dealt.

Splitting routine

This is the most complicated routine of all. The routine treats aces and non-aces separately as follows:

Aces

Splitting aces is the easier case to handle, because casinos generally allow only one card to be dealt to each ace. Resplitting is generally not allowed. The routine assumes an independence between each of the split aces. Therefore, one and only one card is dealt to one ace; the split distribution routine is then called with the appropriate player and dealer probabilities of totals.

Non – aces

After a player splits, his options remain essentially the same as when he started; of course, he may stand or hit. In many casinos, he may also resplit or double down. As a result, all the options must be compared in order to determine which is best. After this is done for one hand, the split distribution routine is invoked with the distribution of player totals and amounts bet for each total. This routine generates the probabilities of winning from -4 to +4 bet units. It simply considers all possible permutations of results for both hands. This program does not handle the case of resplitting, since the number of possibilities for three or more hands becomes unwieldy, though not ridiculously unmanageable.

Discussion of results

A prudent gambler (investor) never bets all he has unless he is absolutely guaranteed to win. If the investor wishes to increase his money as quickly as possible, the appropriate amount to wager in an advantageous situation is governed by the Kelly criterion, which says to bet in proportion to your expectation. In addition to the expectation, however, the distribution of results also governs the correct bet size. For example, if the expectation of some game were .01 and the distribution of its results included zero chance of losing, we would attempt to bet as much as possible on this game. If, on the other hand, the expectation were the same, but the distribution were .495 probability of losing and .505 probability of winning, we would be fools to bet any significant fraction of our wealth.

An application of the determination of the distribution of outcomes is to solve this problem. Once the distribution is known and a utility function of money is chosen, the problem becomes one of simple numerical analysis. The utility function is simply maximized with respect to the bet size. It is not the intention of this paper to discuss utility functions or betting systems in general. Rather, it is to introduce a tool that can be used in conjunction with the results of these subjects.

Consider the following situation: At the end of a round, the dealer accidentally shows the next card, a ten, to the player. What fraction of his bankroll should the player bet on the first hand in the next round?

In order to maximize the player's rate of capital growth, he should bet an amount which maximizes the expectation of his utility, i.e. $E(\ln M)$, where M is his bankroll, and $\ln M$ is his utility function. The program gives the probability distribution for a player receiving a ten as his first card in a four-deck game as approximately:

win:	-1	0	+1	+1.5
P(win):	.316	.119	.491	.074

We want to maximize $.316 \ln (M-B) + .491 \ln (M+B) + .074 \ln (M+1.5B)$. B is the amount that should be bet under the criterion of maximal growth rate. The result is $B = .0927 M$.

The program can be run for all possible dealer and player hands. For a single deck game played under current Atlantic City rules (described in the Appendix) the net expectation is .00136. The win-loss distribution is as follows:

Win	Probability
-4	.0002242
-3	.0015758
-2	.0445612
-1	.4326174
0	.0843654
1	.3242601
2	.0634407
3	.0020058
4	.0004570
bj	.0464924

The same optimization of the bet gives $B = .00109 M$. If late surrender is allowed, the probabilities for winning one, losing one, and tying one change to:

-1	.4088225
0	.0822621
1	.3171211
-.5	.0330372

As noted above, the probability of losing one-half compensates for the reduced probabilities of losing one, winning one, and tying. The expectation increases to .00150. The optimal bet is .00114 M. The value of late surrender in a single deck game is small, only +.014%.

For four decks, the expectation becomes -0.00403. The win-loss distribution is:

Win	Probability
-4	.0001997
-3	.0016831
-2	.0423956
-1	.4346690
0	.0870881
1	.3268699
2	.0591181
3	.0021410
4	.0003983
bj	.0454372

With late surrender, the expectation for four decks increases to -.00332, an increase of .071%. These results demonstrate that single deck blackjack is in fact more favorable than the four deck game. Surrender is only taken when the player has a bad hand. Therefore, surrender has a greater effect in the four deck game because it is a worse game; consequently, the opportunities for surrender are more frequent.

These results are only a sampling of those possible. The program contains various flags that can be changed in order to calculate the effect of various rule changes such as no double down after split, or dealer hits soft seventeen.

Other games that can be analyzed with this approach include double exposure,

and possibly some aspects of tournament blackjack. Double exposure is a game similar to blackjack, except that the dealer's second card is shown to the player before his decisions are made. To compensate for this edge, the house wins all ties and pays blackjack at 1:1. To analyze double exposure simply requires changing the expectation evaluation to make ties a loss and blackjacks pay only even money. The program takes as input any cards as dealer or player initial hands, so that no further changes are necessary.

The analysis of tournament blackjack is a much more difficult problem. Tournament blackjack differs from ordinary blackjack in that the opponent is not the dealer, but the other players. Although the rules are the same, the object of the game is not necessarily to win the most amount of money; instead it is to have more than anyone else at the table at the end of a session that typically lasts 30 rounds. In order to negate the advantage of betting last, a marker that indicates where the first player must bet rotates around the table. The program presented here can help to analyze the last rounds of this game by calculating the probabilities of win, loss, or tie. Of course, the criterion for the optimal play must be changed; the expectation of the hand is not the only important value, since winning the session involves a bonus significantly larger than the money on that hand. Consequently, a more useful criterion is maximizing the probability of winning the round, rather than maximizing the expectation of the hand. Unfortunately it seems difficult to prescribe a simple strategy; the actions of the other players, their bankrolls, and their bets all influence the what the proper play is.

This paper has introduced an approach for the analysis of blackjack-like games, an approach that allows consideration of the proper betting level as well as the proper strategy. The program presented analyzes the game as currently offered in Atlantic City. The approach has several possible applications. I leave it to the perspicacious reader to exploit them.

Appendix : The rules of blackjack

Before each round of play, the player places his bet. Each player and the dealer initially receive two cards; only one of the dealer's cards is shown to the player. A card's value is equal to its rank, but picture cards are worth 10; an ace is worth either 1 or 11. For simplicity, I will refer to all 10-valued cards as 10. The player has several options, depending on his cards. He may:

- **Hit**

Draw another card. A player is allowed to hit any hand not exceeding 21. If the player's card total exceeds 21, he has 'busted,' and automatically loses his bet.

- **Stand**

Draw no more cards.

- **Double down**

Double the original bet and receive one card only. This option is allowed on a two-card hand only.

- **Split**

If the player has two cards of equal value, he may place another bet equal to his original one, and play each card as a separate hand. Exceptions: if the player splits two aces, he receives only one card on each; an (A 10) counts as 21, not blackjack; and even if resplitting is allowed otherwise, the player may not resplit aces.

One further option available to the player is insurance, offered when the dealer has an ace showing. When a player takes insurance he is making a side bet that the dealer has blackjack. The payoff on the bet is 2 to 1. (In general, the bet is a bad one, although if the player keeps track of the tens that have been played, he can take advantage of the bet when the probability of a ten is high enough.) After the players have made their decisions, the dealer draws. As soon as his total is 17 or more, he

must stop. If the dealer busts and the player does not, the player wins and is paid 1:1. Otherwise, the one with the highest card total wins. If the card totals are the same, a situation called a 'push,' no money exchanges hands. A blackjack occurs when the initial two cards of a hand are an ace and a 10. A blackjack beats any other hand, including non-blackjack 21's; it ties against the dealer's blackjack. If the player receives blackjack and the dealer does not, he is paid 3:2. If the dealer gets blackjack and the player does not, the player loses his original bet only, regardless of whether he has split or doubled down.

A soft hand is one both containing an ace and whose total is less than 21 regardless of whether the ace counts as 1 or 11. A hard hand is one whose card total is single-valued. For example, (A 6) is soft 17. (A A 10) is 'hard' 12.

Rule variations

Variations on these rules abound. The game as described above is the one currently found in Atlantic City casinos. Common variations on these rules include:

- **Resplitting**

In Atlantic City, a player may split only once; in casinos where a player is allowed to resplit, he typically may split up to three times.

- **No doubling after splitting**

The player may not double down on the two cards on a hand developed after a split.

- **Surrender**

A few casinos allow the player to 'surrender,' to give up half his bet before playing the hand. With 'early' surrender the player loses half his bet regardless of the dealer's hand. With "late" surrender the player loses his entire bet if the dealer has blackjack.

- **Dealer hits soft 17**

The dealer must hit his soft 17.

- **No soft doubling**

The player may not double down on any soft hand.

- **No hole card**

Many European casinos have their dealers deal themselves only one initial card instead of two. There is no face down, or 'hole card.' If the dealer gets blackjack, the players lose all the money they have bet, including the extra money bet for double downs and resplits. In American casinos, a player loses only his original bet if the dealer has blackjack, regardless of his doubles or splits.

- **Doubling only on totals of 9, 10, and 11**

The player may double only on 9, 10, and 11.

Bibliography

Griffin, Peter A., *Theory of Blackjack*, 2nd edition, GBC Press. (Out of print)

Epstein, Richard A., *The Theory of Gambling and Statistical Logic*, Academic Press, 1977.

Thorpe, Edward, *The Mathematics of Gambling*, 1984.

Thorpe, Edward, *Beat The Dealer*, Random House, 1966 (revised edition).

Kelly, John L. Jr., *A New Interpretation of Information Rate*, *Bell System Technical Journal*, **35, No. 4, July 1956, pp. 917-926.**

```
#include "stdio.h"

#define EPS .0001
#define NOUTCOMES 8
#define HL 20
#define EARLYSURRENDER 0
#define LATESURRENDER 0
#define DOAS 1
#define RESPLIT 0
#define SOFT17 0
#define NRESPLITHANDS 4

typedef double OUTCOME[10];
typedef double EXPECTATION;
typedef double DISTRIBUTION[8];
typedef struct {
    DISTRIBUTION d1sn;
    EXPECTATION ev;
} RESULT;
typedef struct {
    EXPECTATION ev;
    DISTRIBUTION d1sn;
    OUTCOME net;
} ANSWER;
```

```

#include "hitdian.h"

#define DECKDEFAULT 8
#define newline printf("\n");
#define max(x,y) ((x) > (y) ? (x) : (y))

DISTRIBUTION dp;
unsigned dhand[HL], canresplit;
RESULT standresult, hitresult, doubleresult;

static char *mess[] = {
    "Usage: main -n < filename\n(n = decks; filename = hands)\nn = 0 allows specification of shoe composition\n",
    "Error: number of decks is non-numeric."
};

void perror(erno) unsigned erno;
{
    fprintf(stderr, mess[erno]);
    exit(0);
}

main(argc, argv) int argc; char *argv[];
{
    unsigned *noc, noc, phand[HL], i, dstore[HL], pstore[HL];
    unsigned rc(), ac();
    unsigned tot;
    DISTRIBUTION pp;
    RESULT *splitresult;
    OUTCOME pnet;
    ANSWER answers[3];

    splitresult = (RESULT *) malloc(sizeof(RESULT));
    noc = (unsigned *) malloc(11 * sizeof(unsigned));
    if (argv[1][0] == '?')
        perror(0);
    nod = ((argv[1][0] == '-') ? (unsigned) (argv[1][1] - '0') : DECKDEFAULT);
    if ((nod < 0) || (nod > 9))
        perror(1);
    if (lnod) {
        for (i = 1; i < 11; i++)
            printf("%d ", noc[i]);
        newline;
        nod = DECKDEFAULT;
    }
    md(nod, noc);

    while ((geth(dhand) != EOF) && (geth(phand) != EOF)) {
        newline;
        pstore[0] = phand[0];
        for (i = 1; i <= phand[0]; i++) {
            pstore[i] = phand[i];
            printf("%d ", phand[i]);
        }
        printf("vs ");
        dstore[0] = dhand[0];
        for (i = 1; i <= dhand[0]; i++) {
            dstore[i] = dhand[i];
            printf("%d ", dhand[i]);
        }
        printf(":\n");
        canresplit = 1;

        noc[0] = (rc(noc, dhand) + rc(noc, phand));
        for (i = 1; i < 8; i++) dp[i] = 0.0;
        dprobs(noc, dhand);

        testghit(noc, phand, &(answers[0]));
        testdbl(noc, phand, &(answers[1]));
    }
}

```

```
testp(noc,phand,splitresult,pnet,&(answers[2]));  
result(answers);  
noc[0] += (ac(noc,dstore) + ac(noc,ptore));  
}  
}
```

```

#include "hitdisk.h"
#define max(x,y) ((x) > (y) ? (x) : (y))

void show(best)
ANSWER *best;
{
    unsigned i;

    printf(" %9.7lf\n\n",best->ev);
    for (i = 0; i < 9; i++)
        if ((best->net)[i] != 0.0)
            printf("P(%d) = %9.7lf\n", (int)i - 4, (best->net)[i]);
    if ((best->net)[9] != 0.0)
        printf("P(1.5) = %9.7lf\n", (best->net)[9]);
    printf("\n");
    if ((best->disn)[0] != 0.0)
        printf("P(<17) = %9.7lf\n", (best->disn)[0]);
    for (i = 1; i < 8; i++)
        if ((best->disn)[i] != 0.0)
            printf("P(%d) = %9.7lf\n", i + 10, (best->disn)[i]);
    if ((best->disn)[6] != 0.0)
        printf("P(bj) = %9.7lf\n", (best->disn)[6]);
    if ((best->disn)[7] != 0.0)
        printf("P(>21) = %9.7lf\n", (best->disn)[7]);
    printf("\n");
}

void result(answers)
ANSWER (*answers)[3];
{
    double maxev;
    ANSWER *best;
    unsigned i, stand = 0;

    maxev = max((*answers)[0].ev, max((*answers)[1].ev, (*answers)[2].ev));

    if (maxev == (*answers)[0].ev) {
        best = &((*answers)[0]);
        for (i = 0; i < 10; i++)
            if ((best->net)[i] == 1.0) {
                printf("evs:");
                stand = i;
                break;
            }
        if (stand == 0)
            printf("evh:");
        show(best);
    }
    else if (maxev == (*answers)[1].ev) {
        best = &((*answers)[1]);
        printf("evd:");
        show(best);
    }
    else if (maxev == (*answers)[2].ev) {
        best = &((*answers)[2]);
        printf("evp:");
        show(best);
    }
}

```

```

#include "hitdian.h"

#define HL 20
#define DDAS 1
#define newline printf("\n")
#define NHANDS (sizeof(hand)/sizeof(struct hand))
#define max(a,b) (((a)>(b)) ? (a) : (b))

struct hand {
    int a, b;
} hand[] = {
    {2,3}, {2,4}, {3,4}, {2,5}, {5,3}, {6,2},
    {6,3}, {5,4}, {7,2}, {6,4}, {3,7}, {2,8}, {6,6},
    {7,4}, {6,5}, {8,3}, {9,2},
    {10,2}, {9,3}, {8,4}, {7,5}, {10,3}, {9,4}, {8,6},
    {7,6}, {10,4}, {9,5}, {8,6}, {10,6}, {9,6},
    {8,7}, {10,6}, {9,7},
    {10,7}, {9,8}, {10,8}, {10,9},
    {1,2}, {1,3}, {1,4}, {1,6}, {1,6}, {1,7},
    {1,8}, {1,9}, {1,10},
    {1,1}, {2,2}, {3,3}, {4,4}, {6,6}, {7,7},
    {8,8}, {9,9}, {10,10}
};

DISTRIBUTION dp;
unsigned dhand[HL];
RESULT standresult, hitresult, doubleresult;
double p = 0.0;
ANSWER netans = { 0 };
unsigned canresplit = 3;

double prob(a,b,c,noc) int a,b,c, noc[];
{
    double pi = 1.0;

    pi *= (noc[c]+ 1) * noc[b]/(double)((noc[0] + 1)*noc[0]);
    noc[b]--; noc[0]--;
    pi *= noc[a]/(double)noc[0];

    noc[b]++; noc[0]++;
    if (a != b) pi *= 2.0;
    return (pi);
}

main() {

    int noc[11], nod, phand[HL], i, j;
    unsigned k;
    unsigned rc(), ac();
    RESULT *splitresult;
    OUTCOME pnet;
    ANSWER answers[4];
    double prob();

    splitresult = (RESULT *) malloc(sizeof(RESULT));

    scanf("%d", &nod);
    printf("%d decks\n", nod);
    md(nod, noc);

    /* Zero out netans */
    netans.ev = 0.0;
    for (i = 0; i < 8; i++)
        (netans.dian)[i] = 0.0;
    for (i = 0; i < 10; i++)
        (netans.net)[i] = 0.0;

    for (j = 10; j > 0; j--) {

```

```

dhand[1] = j;
phand[0] = 2;
dhand[0] = 1;

printf("\n\nversus dealer %d :\n\n",dhand[1]);

noc[0] -= rc(noc,dhand);

for (k = 0; k < MHANDS; k++) {
  phand[1] = hand[k].a;
  phand[2] = hand[k].b;
  p = prob(hand[k].a, hand[k].b, j, noc);
  noc[0] -= rc(noc,phand);

  printf("(%d %d) v %d:\n",phand[1],phand[2],dhand[1]);
  for (i = 1; i < 8; i++) dp[i] = 0.0;
  dproba(noc,dhand);

  if ((phand[1] == 1) && (phand[1] == phand[2])) {
    testp(noc,phand,splitresult,pnet,&(answers[2]));
    answers[0].ev = answers[1].ev = -2.0;
  }
  else {

    testghit(noc,phand,&(answers[0]));
    testdbl(noc,phand,&(answers[1]));
    testp(noc,phand,splitresult,pnet,&(answers[2]));
  }
  result(answers);
  newline;
  noc[0] += ac(noc,phand);
}
noc[0] += ac(noc,dhand);

newline;
newline;
}
printf("Net results:\n");
show(&netans);
}

```

```

#include "hitdisn.h"
#define max(x,y) ((x) > (y) ? (x) : (y))

extern double p;
extern ANSWER netans;

void show(best)
ANSWER *best;
{
    register unsigned i;

    if (&netans != best) {
        printf(" %9.7lf\n\n",best->ev);
        netans.ev += p * best->ev;
        for (i = 0; i < 9; i++)
            if ((best->net)[i] != 0.0) {
                printf("P(%d) = %9.7lf\n", (int)i - 4, (best->net)[i]);
                (netans.net)[i] += (best->net)[i] * p;
            }
        if ((best->net)[9] != 0.0) {
            printf("P(1.6) = %9.7lf\n", (best->net)[9]);
            (netans.net)[9] += (best->net)[9] * p;
        }
        printf("\n");
        if ((best->dian)[0] != 0.0) {
            printf("P(<17) = %9.7lf\n", (best->dian)[0]);
            (netans.dian)[0] += (best->dian)[0] * p;
        }
        for (i = 1; i < 6; i++)
            if ((best->dian)[i] != 0.0) {
                printf("P(%d) = %9.7lf\n", i + 16, (best->dian)[i]);
                (netans.dian)[i] += (best->dian)[i] * p;
            }
        if ((best->dian)[6] != 0.0) {
            printf("P( b j) = %9.7lf\n", (best->dian)[6]);
            (netans.dian)[6] += (best->dian)[6] * p;
        }
        if ((best->dian)[7] != 0.0) {
            printf("P(>21) = %9.7lf\n", (best->dian)[7]);
            (netans.dian)[7] += (best->dian)[7] * p;
        }
    }
    else {
        printf(" %9.7lf\n\n",best->ev);
        for (i = 0; i < 9; i++)
            if ((best->net)[i] != 0.0)
                printf("P(%d) = %9.7lf\n", (int)i - 4, (best->net)[i]);
        if ((best->net)[9] != 0.0)
            printf("P(1.6) = %9.7lf\n", (best->net)[9]);
        printf("\n");
        if ((best->dian)[0] != 0.0)
            printf("P(<17) = %9.7lf\n", (best->dian)[0]);
        for (i = 1; i < 6; i++)
            if ((best->dian)[i] != 0.0)
                printf("P(%d) = %9.7lf\n", i + 16, (best->dian)[i]);
        if ((best->dian)[6] != 0.0)
            printf("P( b j) = %9.7lf\n", (best->dian)[6]);
        if ((best->dian)[7] != 0.0)
            printf("P(>21) = %9.7lf\n", (best->dian)[7]);
    }
    printf("\n");
}

void result(answers)
ANSWER (*answers)[];
{
    double maxev;
    ANSWER *best;
    unsigned i, stand = 0;

```

```

maxev = max((*answers)[0].ev,max((*answers)[1].ev,(*answers)[2].ev));

if (maxev == (*answers)[0].ev) {
    best = &((*answers)[0]);
    for (i = 0; i < 10; i++)
        if ((best->net)[i] == 1.0) {
            printf("evs:");
            stand = 1;
            break;
        }
    if (stand == 0)
        printf("evh:");
    show(best);
}
else if (maxev == (*answers)[1].ev) {
    best = &((*answers)[1]);
    printf("evd:");
    show(best);
}
else if (maxev == (*answers)[2].ev) {
    best = &((*answers)[2]);
    printf("evp:");
    show(best);
}
}
}

```

```

#include "stdio.h"

/***** Routines in this file *****/
/*
/* md Make shoe
/* ctot return sum of cards appropriate for dealer total
/* rc return number of cards in hand, and remove these cards from shoe
/* ac return number of cards in hand, and add these cards to shoe
/* geth get hand; return number of cards in hand
/*
/*****

void md(unsigned nod, unsigned noc[]):
{
    unsigned i;

    if (nod) {
        for (i = 1; i < 10; i++) noc[i] = nod * 4;
        noc[10] = nod * 16;
        noc[0] = nod * 52;
    }
    else {
        printf("Enter noc vector: ");
        noc[0] = 0;
        for (i = 1; i < 11; i++) {
            scanf("%d",&noc[i]);
            noc[0] += noc[i];
        }
    }
}

unsigned ctot(unsigned h[]):
{
    register unsigned short sum = 0;
    register unsigned short i;
    register unsigned short a = 0;

    /* Microsoft 3.0 doesn't support more than 2 register
    variables; it also doesn't support short variables, but who knows,
    I may use this routine elsewhere. */

    for (i = 1; i <= h[0]; i++) {
        sum += h[i];
        if (h[i] == 1) a++;
    }
    return(((a > 0) && (sum < 12) && (sum > 6)) ? (sum + 10) : sum);
}

unsigned rc(unsigned noc[], unsigned h[]):
{
    unsigned i;

    for (i = 1; i <= h[0]; i++) noc[h[i]]--;
    return (h[0]);
}

unsigned ac(unsigned noc[], unsigned h[]):
{
    unsigned i;

    for (i = 1; i <= h[0]; i++) noc[h[i]]++;
    return (h[0]);
}

geth(unsigned h[]):
{
    unsigned dindex = 1, dc = 1;

    while (dc) {

```

```
    if (scanf("%d",&dc) == EOF) return (EOF);
    h[dindex++] = dc;
}
h[0] = dindex - 2;
return(h[0]);
}
```

```

#include "hitdisn.h"

extern DISTRIBUTION dp;

/*
  This streamlined dealer probability procedure declares all the unchanged
  variables in the recursion as static variables, thereby saving stack space and
  increasing speed.
*/

static unsigned *noc1, *dealer, tot;
static DISTRIBUTION dps;
static double p;
unsigned ctot();

void dprob(cp) double cp;
{
  unsigned lim;
  register unsigned i;

  tot = ctot(dealer);

  lim = ((tot > 11) ? 21 - tot : 10);

  if (tot < 17) {
    for (i = lim; i > 0; i--)
      if (noc1[i] > 0) {
        noc1[i]--; noc1[0]--; dealer[++dealer[0]] = i;
        dprob(cp * (noc1[i] + 1)/(double)(noc1[0] + 1));

        noc1[i]++; noc1[0]++; dealer[0]--;
      }
    p = 0.0;
    for (i = 1; i < 7; i++) p += dps[i];
    dps[7] = 1.0 - p;
  }
  else if ((tot == 21) && (dealer[0] == 2))
    dps[6] += cp;
  else
    dps[tot - 16] += cp;
}

void dprobs(noc,dhand) unsigned noc[],dhand[];
{
  register unsigned i;

  noc1 = noc;
  for (i = 0; i < NOUTCOMES; i++)
    dps[i] = 0.0;
  dealer = dhand;
  dprob(1.0);
  for (i = 0; i < NOUTCOMES; i++) {
    dp[i] = dps[i];
  }
}

```

```

#include "hitdisn.h"

extern RESULT standresult,doublresult;
extern DISTRIBUTION dp;

/* To use this routine, dp must already be calculated */

void ddisn(noc,phand,net) unsigned noc[], phand[]; OUTCOME net;
{
    unsigned i, j, lim, sum = 0, bustcda = 0, tot, ctot();
    double pi;
    DISTRIBUTION pp;

    if (phand[0] != 2) {
        printf("Double allowed on two cards only\n");
        return;
    }
    for (i = 1; i <= phand[0]; i++) sum += phand[i];
    lim = ((sum > 11) ? 22 - sum : 11);
    doublresult.ev = 0.0;
    for (i = 0; i < NOUTCOMES; i++)
        (doublresult.disn)[i] = 0.0;
    for (i = lim; i < 11; i++) bustcda += noc[i];

    for (i = 1; i < lim; i++)
    if (noc[i] > 0) {
        pi = noc[i]/(double) noc[0];
        noc[i]--; noc[0]--; phand[0]++;
        /* if the dealer has bj, then gevs returns a different ev */
        /* We need a different gevs, or must modify its result for ev */
        /* Now, gevd takes this situation into consideration */

        for (j = 0; j < NOUTCOMES; j++)
            pp[j] = 0.0;
        if ( (tot = ctot(phand)) < 17)
            pp[0] = 1.0;
        else
            pp[tot - 16] = 1.0;
        gevd(pp,net);

        doublresult.ev += standresult.ev * pi;

        for (j = 0; j < NOUTCOMES; j++)
            (doublresult.disn)[j] += (standresult.disn)[j] * pi;
        noc[i]++; noc[0]++; phand[0]--;
    }
    (doublresult.disn)[7] += bustcda/(double) noc[0];
    for (i = 0; i < NOUTCOMES; i++)
        pp[i] = (doublresult.disn)[i];
    gevd(pp,net);
    doublresult.ev = standresult.ev;
    for (i = 0; i < NOUTCOMES; i++)
        (doublresult.disn)[i] = (standresult.disn)[i];
}

```

```

#include "hitdisn.h"
#define true 1

extern DISTRIBUTION dp;
extern unsigned dhand[];
extern RESULT hitresult, standresult, doubleresult;
extern unsigned canresplit;

/*
   canresplit must be declared and initialized in main.c
   dp must be calculated already before this routine is invoked
   splitresult must be cleared before this routine is called
*/

void pdisn(noc,phand,splitresult,net)
unsigned noc[], phand[];
RESULT *splitresult;
OUTCOME net;
{
    unsigned i, j, tot, ctot();
    double p1;
    DISTRIBUTION w1,w2;
    DISTRIBUTION pp;
    OUTCOME dummynet;

    for (i = 0; i < NOUTCOMES; i++)
        w1[i] = w2[i] = 0.0;
    if (phand[0] != 2) {
        printf("Allowed to split on two cards only\n");
        return;
    }
    else if (phand[1] != phand[2]) {
        printf("Allowed to split cards of the same value only\n");
        return;
    }

    else if (phand[1] == 1) {
        phand[0] = 1;
        for (i = 1; i < 11; i++)
            if (noc[i] > 0) {
                p1 = noc[i] / (double) noc[0];
                noc[i]--; noc[0]--; phand[++phand[0]] = 1;
                for (j = 0; j < NOUTCOMES; j++)
                    pp[j] = 0.0;
                tot = ctot(phand);
                pp[(tot < 17) ? 0 : tot - 16] = 1.0;
                gevs(pp,dummynet);

                for (j = 0; j < NOUTCOMES; j++)
                    w1[j] += p1 * (standresult.disn)[j];
                phand[0]--; noc[i]++; noc[0]++;
            }

        gevpb(w1,w2,splitresult,net);

        phand[0] = 2; phand[2] = phand[1];
    }
    else {
        /* What about resplits? */
        phand[0] = 1;
        for (i = 1; i < 11; i++)
            if (noc[i] > 0) {
                p1 = noc[i] / (double) noc[0];
                noc[i]--; noc[0]--; phand[++phand[0]] = 1;

                if ((RESPLIT == true) && (canresplit < NRESPLITHANDS) && (phand[1] == phand[2])) {
                    canresplit++;
                    pdisn(noc,phand,splitresult,net);
                }
            }
        else {

```

```

ghit(noc.phand.dumynet);
if (DDAS == true) {
  ddis(noc.phand.dumynet);
  if (doublesresult.ev > hitresult.ev) {
    for (j = 0; j < NOUTCOMES; j++)
      (splitresult->disn)[j] = w2[j] + p1 * (doublesresult.disn)[j];
  }
  else {
    for (j = 0; j < NOUTCOMES; j++)
      (splitresult->disn)[j] = w1[j] + p1 * (hitresult.disn)[j];
  }
} /* End DDAS */
else {
  for (j = 0; j < NOUTCOMES; j++)
    (splitresult->disn)[j] = w1[j] + p1 * (hitresult.disn)[j];
} /* End nonresplit routine */
phand[0]--; noc[1]++; noc[0]++;
} /* End for ... if noc[1] > 0 loop */

gevpb(w1,w2,splitresult,net);
phand[0] = 2; phand[2] = phand[1];
} /* End non-ace splits */
}

```

```

#include "hitdisn.h"

extern RESULT standresult, hitresult;

/*****
/*                               Hit distribution routine                               */
*****/

/*
Calculate standing ev (and distribution)
if optimal decision is to stand, then return standing results
else
  for each card that doesn't bust the hand
    hit
    call this routine recursively with the new hand and shoe
    save the result
  Subtract P(bust) from evh and add to P(bust)?
  Compare hit and stand evs & return result of optimal decision
*/

void ghit(noc,phand,hnet) unsigned noc[],phand[]; OUTCOME hnet;
{
  register unsigned short i;
  register unsigned short j;
  double evh,evs;
  unsigned tot,lim,bustcda,sum,ctot();
  DISTRIBUTION pps,pph;
  double p1;

  evh = 0.0; bustcda = sum = 0;
  for (i = 0; i < NOUTCOMES; i++) pph[i] = pps[i] = 0.0;
  tot = ctot(phand);
  for (j = 1; j <= phand[0]; j++) sum += phand[j];
  if (tot < 17)
    pps[0] = 1.0;
  else if ((tot == 21) && (phand[0] == 2))
    pps[6] = 1.0;
  else
    pps[tot - 16] = 1.0;

  gevs(pps,hnet);
  evs = standresult.ev;
  lim = ((sum > 11) ? 22 - sum : 12);
  for (j = lim; j < 11; j++) bustcda += noc[j];

  if ((evs > -1.0) && ((sum > 16) || (tot > 18))) {
    hitresult.ev = evs;
    for (i = 0; i < NOUTCOMES; i++)
      (hitresult.dsn)[i] = pps[i];
  }
  else {
    for (j = 1; j < lim; j++)
      if (noc[j] > 0) {
        p1 = noc[j]/(double) noc[0];
        noc[j]--; noc[0]--; phand[++phand[0]] = j;
        ghit(noc,phand,hnet);
        evh += hitresult.ev * p1;
        for (i = 0; i < NOUTCOMES; i++)
          pph[i] += (hitresult.dsn)[i] * p1;
        phand[0]--; noc[0]++; noc[j]++;
      }
    evh -= bustcda/(double) noc[0];
    pph[7] += bustcda/(double)noc[0];

    if (evh > evs) {
      hitresult.ev = evh;
      for (i = 0; i < NOUTCOMES; i++)
        (hitresult.dsn)[i] = pph[i];
      gevs(pph,hnet);
    }
  }
}

```

```
}  
else {  
    hitresult.ev = evs;  
    for (i = 0; i < NOUTCOMES; i++)  
        (hitresult.dian)[i] = pps[i];  
    gevs(pps,hnet);  
}  
}
```

```
#include "hitdisn.h"
```

```
extern DISTRIBUTION dp;  
extern RESULT standresult;
```

```
/*  
/* Get expectation given dealer and player probabilities of totals */  
/*  
*/
```

```
void gevs(pp.net) double pp[]; OUTCOME net;
```

```
{  
  register unsigned short i;
```

```
  for (i = 0; i < 10; i++)  
    net[i] = 0.0;
```

```
/* These subtractions to get the probabilities of occurrence have the  
unfortunate property of sometimes producing negative numbers.  
The error is miniscule, however, and simply ignoring the -0.000000 displayed is  
the appropriate response. Code to test for this condition would slow the  
program down considerably, since this routine is among the most heavily used in  
the program, accounting for as much as 1/3 the total execution time.  
*/
```

```
net[8] = dp[7] * (1.0 - pp[6] - pp[7])  
  + dp[1] * (pp[2] + pp[3] + pp[4] + pp[8])  
  + dp[2] * (pp[3] + pp[4] + pp[8])  
  + dp[3] * (pp[4] + pp[8])  
  + dp[4] * pp[8];
```

```
net[9] = pp[6] * (1.0 - dp[6]);
```

```
for (i = 1; i < 7; i++) net[4] += dp[i] * pp[i];
```

```
net[3] = dp[6] * (1.0 - pp[6])  
  + dp[8] * (1.0 - pp[6] - pp[8])  
  + dp[4] * (1.0 - pp[6] - pp[8] - pp[4])  
  + dp[3] * (pp[7] + pp[0] + pp[1] + pp[2])  
  + dp[2] * (pp[7] + pp[0] + pp[1])  
  + dp[1] * (pp[7] + pp[0])  
  + dp[7] * pp[7];
```

```
standresult.ev = net[8] + 1.6 * net[9] - net[3];
```

```
for (i = 0; i < NOUTCOMES; i++)  
  (standresult.disan)[i] = pp[i];
```

```
}
```

```
#include "hitdian.h"
```

```
extern DISTRIBUTION dp;  
extern unsigned dhand[];  
extern RESULT standresult;
```

```
/*.....*/  
/* Get expectation given dealer and player probabilities of totals */  
/*.....*/
```

```
void gevd(pp.net) double pp[]; OUTCOME net;  
{  
    register unsigned short i;  
  
    for (i = 0; i < 10; i++)  
        net[i] = 0.0;  
    net[6] = dp[7] * (1.0 - pp[7])  
        + dp[1] * (1.0 - pp[0] - pp[1] - pp[7])  
        + dp[2] * (pp[3] + pp[4] + pp[6])  
        + dp[3] * (pp[4] + pp[6])  
        + dp[4] * pp[6];  
  
    net[3] = dp[6];  
  
    for (i = 1; i < 8; i++) net[4] += dp[i] * pp[i];  
  
    net[2] = dp[5] * (1.0 - pp[6])  
        + dp[4] * (1.0 - pp[6] - pp[4])  
        + dp[3] * (1.0 - pp[3] - pp[4] - pp[3])  
        + dp[2] * (pp[7] + pp[0] + pp[1])  
        + dp[1] * (pp[7] + pp[0])  
        + dp[7] * pp[7];  
  
    standresult.ev = 2.0 * (net[6] - net[2]) - net[3];  
    for (i = 0; i < NOUTCOMES; i++)  
        (standresult.dian)[i] = pp[i];  
}
```

```

#include "hitdisn.h"

#define true 1
#define false 0

extern DISTRIBUTION dp;
/*
  This routine gets the expectation and distribution given the player and
  dealer probabilities of totals.

  I will assume no replitting
*/

void gevpb(w1,w2,splitresult,net)
DISTRIBUTION w1,w2;
RESULT *splitresult;
OUTCOME net;
{
  unsigned i,j,k;
  int m,n;
  double bad[2][8];
/*
  bad contains 2 distributions of totals. The first is
  the disn with one unit bet; the other is with 2.
  The first element of each disn contains the probability of both
  busting and getting a stiff.
*/

  double t,nobust1 = 0.0,nobust2 = 0.0;

/* stiffs and busts are the same thing, except when the dealer busts */

  for (i = 0; i < 10; i++)
    net[i] = 0.0;
  bad[0][0] = w1[0] + w1[7];
  bad[1][0] = w2[0] + w2[7];
  for (i = 0; i < NOUTCOMES; i++)
    (splitresult->disn)[i] = w1[i] + w2[i];

/* You cannot have bj when you split */
  (splitresult->disn)[6] = (splitresult->disn)[6] + (splitresult->disn)[6];
  (splitresult->disn)[6] = 0.0;

  for (i = 1; i < NOUTCOMES; i++) {
    bad[0][i] = w1[i];
    bad[1][i] = w2[i];
  }
/*
  Since the routine that passes w1 & w2 doesn't know that 10 A cannot be bj,
  this routine assumes that any values passed in w1[6] should be added to
  w1[6].
*/
  bad[0][6] = w1[6] + w1[6];

  net[3] = dp[6];

  for (i = 1; i < 6; i++) { /* Over all dealer totals, except bust & bj */
    /* do the diagonals and nondiagonals separately */

    for (j = 0; j < 6; j++) {
      for (m = 0; m < 2; m++) {
        t = dp[i] * bad[m][j] * bad[m][j];
        if (i > j) {
          net[2-2*m] += t;
        }
        else if (i == j) {
          net[4] += t;
        }
        else if (i < j) {

```

```

    net[6+2*m] += t;
}
for (k = 0; k <= j; k++) {
  for (n = 0; n < 2; n++) {
    if ((k == j) && (m == n))
      continue;
    else if ((k == j) && (m != n))
      t = dp[i] * bad[m][j] * bad[n][k];
    else
      t = 2.0 * dp[i] * bad[m][j] * bad[n][k];
    if ((i > j) && (i > k)) {
      net[2-m-n] += t;
    }
    else if ((i > j) && (i == k)) {
      net[3-m] += t;
    }
    else if ((i == j) && (i > k)) {
      net[3-n] += t;
    }
    else if ((i == j) && (i == k)) {
      net[4] += t;
    }
    else if ((i > j) && (i < k)) {
      net[4-m+n] += t;
    }
    else if ((i < j) && (i > k)) {
      net[4-n+m] += t;
    }
    else if ((i < j) && (i == k)) {
      net[6+m] += t;
    }
    else if ((i == j) && (i < k)) {
      net[6+n] += t;
    }
    else if ((i < j) && (i < k)) {
      net[m+n+6] += t;
    }
  } /* end n */
} /* end k */
} /* end m */
} /* end j */
} /* end i */
/* handle dealer bust here */

for (i = 0; i < 6; i++) {
  nobust1 += w1[i];
  nobust2 += w2[i];
}
/* No bj when splitting */
nobust1 += w1[6];

net[0] += dp[7] * w2[7] * w2[7];
net[8] += dp[7] * nobust2 * nobust2;
net[2] += dp[7] * w1[7] * w1[7];
net[6] += dp[7] * nobust1 * nobust1;
net[4] += 2.0 * dp[7] * (w2[7] * nobust2 + w1[7] * nobust1);
net[8] += 2.0 * dp[7] * nobust2 * w1[7];
net[1] += 2.0 * dp[7] * w2[7] * w1[7];
net[7] += 2.0 * dp[7] * nobust2 * nobust1;
net[3] += 2.0 * dp[7] * w2[7] * nobust1;

splitresult->ev = 4.0 * (net[8] - net[0]) + 3.0 * (net[7] - net[1]) + 2.0 *
(net[6] - net[2]) + net[8] - net[3];
}

```

```

#include "hitdisn.h"

extern RESULT doubleresult;
extern DISTRIBUTION dp;
extern unsigned dhand[];

void testdbl(noc,phand,ans) unsigned noc[], phand[]; ANSWER *ans;
{
    unsigned i;
    OUTCOME dnet;

    if (phand[0] != 2) {
        ans->ev = -2.0;
        return;
    }
    ddisn(noc,phand,dnet);
    ans->ev = doubleresult.ev;
    for (i = 0; i < NOUTCOMES; i++) {
        (ans->disn)[i] = (doubleresult.disn)[i];
    }
    for (i = 0; i < 10; i++) {
        (ans->net)[i] = dnet[i];
    }
}

```

```

#include "hitdis.h"

extern DISTRIBUTION dp;
extern unsigned dhand[];
extern RESULT hitresult;

void testghit(noc,phand,ans) unsigned noc[],phand[]; ANSWER *ans;
{
    unsigned i;
    OUTCOME hnet;

    ghit(noc,phand,hnet);

    ans->ev = hitresult.ev;
    for (i = 0; i < NOUTCOMES; i++) {
        (ans->disn)[i] = (hitresult.disn)[i];
    }
    for (i = 0; i < 10; i++) {
        (ans->net)[i] = hnet[i];
    }
}

```

```

#include "hitdisn.h"

extern DISTRIBUTION dp;
extern unsigned dhand[];

void testp(noc,phand,splitresult,net,ans)
unsigned noc[], phand[];
RESULT *splitresult;
OUTCOME net;
ANSWER *ans;
{
    unsigned i;

    if ((phand[0] != 2) || (phand[1] != phand[2])) {
        ans->ev = -2.0;
    }
    else {
        splitresult->ev = 0.0;
        for (i = 0; i < NOUTCOMES; i++)
            (splitresult->disn)[i] = 0.0;
        pdisn(noc,phand,splitresult,net);
        ans->ev = splitresult->ev;
        for (i = 0; i < NOUTCOMES; i++) {
            (ans->disn)[i] = (splitresult->disn)[i];
        }
        for (i = 0; i < 10; i++) {
            (ans->net)[i] = net[i];
        }
    }
}
}

```