1. Breaking Bread Together. A program to create collections of dinner parties where the participants are randomly assembled so that guests meet with the most people possible.

⟨ Includes 2 ⟩
⟨ Globals 10 ⟩
⟨ Functions 9 ⟩
⟨ Main 3 ⟩

2. Just some basic includes. I also like to define values for `TRUE` and `FALSE` using # **define**s. (The `string.h` file needs `_GNU_SOURCE` to be defined to produce a prototype for the *strndup*( ) function.)

```
#define TRUE (1 ≡ 1)
#define FALSE (¬TRUE)
```

⟨ Includes 2 ⟩ ≡

```
#define _GNU_SOURCE 1
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
```

This code is used in section 1.

3. The main program. The program command line should contain the file which has the list of guests/hosts.

⟨ Main 3 ⟩ ≡

```
int main(int argc, char **argv)
{
  int i, nEvents ← 3, ecount;
  struct event *e, events;
  argc−−;
  argv++;
  if (¬argc) {
    fprintf(stderr, "No␣guest␣file␣was␣specified\n");
    exit(1);
  }
  ⟨ Read the guest file and build tables 4 ⟩
  ⟨ Generate events 5 ⟩
  ⟨ Display events 6 ⟩
  write_statistics("stats");
}
```

This code is used in section 1.

4. ⟨ Read the guest file and build tables 4 ⟩ ≡

```
readfile(*argv);
make_relationships( );
make_hostlist( );
fprintf(stderr, "guest␣entities:␣%d␣(%d␣m,␣%d␣f)\n\n", nGuests, nMales, nFemales);
for (i ← 0; i < nGuests; i++) {
  fprintf(stderr, "%s␣:␣m%d␣f%d␣g%d\n", guestArr[i].name, guestArr[i].male, guestArr[i].female,
      guestArr[i].guests);
}
```

This code is used in section 3.

5. An event consists of the several dinner parties to be hosted. The goal is to use as many unique relationships as possible while minimizing the number of times a particular relationship is used. First, space for the *event* structure is allocated, and then the hosts are selected for each dinner party. Then the other participants are seated at one of the parties.

⟨ Generate events 5 ⟩ ≡

```
e ← &events;
for (i ← 0; i < nEvents; i++) {
   e→next ← (struct event *) malloc(sizeof(struct event));
   e ← e→next;
   e→pdesc ← determine_hosts(nMales + nFemales);
   seat_guests(e→pdesc);
}
e→next ← Λ;
```

This code is used in section 3.

6. Not a whole lot of magic here:

⟨ Display events 6 ⟩ ≡

```
e ← &events;
ecount ← 1;
while (e→next) {
   e ← e→next;
   write_event(e→pdesc, ecount++);
}
```

This code is used in section 3.

7. Write out the guest lists into individual files.

```
#define MAX_FNSTR 128

⟨ Calculation Functions 7 ⟩ ≡
  void write_event(struct party *p, int ecount)
  {
    FILE *outf;
    struct guestlist *gl;
    char fn[MAX_FNSTR];

    sprintf(fn, "Event%03d.tex", ecount);
    if ((outf ← fopen(fn, "w")) ≡ Λ) {
      fprintf(stderr, "Couldn't␣open␣\"%s\"␣for␣writing\n", fn);
      return;
    }
    fprintf(outf, "\\input␣bbtmac.tex\n\\title{%d}\n", ecount);
    while (p) {
      fprintf(outf, "\\startparty\n\\host{%s}{%d}\\hcontact{}\n", guestArr[p→hostent].name,
          p→maxguests);
      fprintf(outf, "\\allguests{%d}{%d}{%d}\n", p→males + p→females, p→males, p→females);
      gl ← p→guests;
      while (gl) {
        fprintf(outf, "\\guest{%s}{%d}{%d}\\gcontact{}\n", guestArr[gl→guestent].name,
            guestArr[gl→guestent].male, guestArr[gl→guestent].female);
        gl ← gl→next;
      }
      fprintf(outf, "\\endparty");
      p ← p→next;
    }
    fprintf(outf, "\\end\n");
    fclose(outf);
  }
```

See also sections 8, 23, and 30.

This code is used in section 9.

8. Now that the events have been output, show the guest array statistics.

```
⟨ Calculation Functions 7 ⟩ +≡
  void write_statistics(char *fn)
  {
    FILE *outf;
    int i;

    if ((outf ← fopen(fn, "w")) ≡ Λ) {
      fprintf(stderr, "Couldn't␣open␣\"%s\"␣for␣writing\n", fn);
      return;
    }
    for (i ← 0; i < nGuests; i++) fprintf(outf, "%25s␣h=%3d\n", guestArr[i].name, guestArr[i].hosted);
    fprintf(outf, "\ng1␣␣g2␣␣cnt\n");
    for (i ← 0; i < nRels; i++) fprintf(outf, "%3d␣%3d␣%3d\n", relArr[i].g1, relArr[i].g2, relArr[i].count);
    fclose(outf);
  }
```

9. ⟨ Functions 9 ⟩ ≡
⟨ Utility Functions 12 ⟩
⟨ Setup Functions 11 ⟩
⟨ Calculation Functions 7 ⟩

This code is used in section 1.

10. We'll start by defining storage for each guest entity. A guest entity may be more than one person, e.g., a couple who will attend each dinner party together. Each guest entity is counted as a single endpoint in a relationship, but we must count the number of seats. In case we want to track the number of males and females at each dinner party, we keep track of this info here. In addition, we keep track of the number of times the guest entity has been a host, and the number of guests that may be entertained if hosting (including the host(s)).

⟨ Globals 10 ⟩ ≡
```
struct guest {
  char *name;
  int male, female;    /* Number of males and females this guest entity represents. */
  int hosted, guests;    /* The number of times this guest entity has been a host, and the number of
      guests that may be entertained. */
  int hostingnow, seated;
} *guestArr;
int nGuests ← 0, nMales ← 0, nFemales ← 0;
```

See also sections 18, 20, and 22.

This code is used in section 1.

11. Read a file of guests.

⟨ Setup Functions 11 ⟩ ≡
```
void readfile(char *filename)
{
  FILE *inf;
  char oneline[256];
  int i;

  if ((inf ← fopen(filename, "r")) ≡ Λ) {
    fprintf(stderr, "Couldn't␣open␣%s\n", filename);
    exit(1);
  }
  fgets(oneline, sizeof (oneline), inf);
  nGuests ← atoi(oneline);
  guestArr ← malloc(sizeof(struct guest) * nGuests);
  i ← 0;
  fgets(oneline, sizeof (oneline), inf);
  while (¬feof(inf) ∧ i < nGuests) {
    getguest(&(guestArr[i]), oneline);
    i++;
    fgets(oneline, sizeof (oneline), inf);
  }
  fclose(inf);
}
```

See also sections 19, 21, 25, 26, 27, and 34.

This code is used in section 9.

12. Ingest a line of data from the guest file. Fields are separated by colons. The first field is the name (or names). The next field contains the number of males and/or females in this entity, prefixed by an `'m'` or `'f'` character, and possibly the number of guests if this entity will be a host, prefixed by a `'g'` character.

⟨ Utility Functions 12 ⟩ ≡

```
void getguest(struct guest *theguest, char *theline)
{
  char *f, *l, *e;
  theguest→male ← theguest→female ← theguest→hosted ← theguest→guests ← 0;
  f ← l ← theline;
  while (*l ∧ *l ≠ ':') l++;
  e ← l − 1;
  while (e > f ∧ isspace(*e)) e−−;
  e++;
  theguest→name ← strndup(f, (size_t)(e − f));
  f ← l + 1;
  while (f ∧ *f) {
    while (*f ∧ isspace(*f)) f++;
    switch (*f) {
    case 'm': case 'M': theguest→male ← strtol(f + 1, &l, 10);
      nMales += theguest→male;
      f ← l;
      break;
    case 'f': case 'F': theguest→female ← strtol(f + 1, &l, 10);
      nFemales += theguest→female;
      f ← l;
      break;
    case 'g': case 'G': theguest→guests ← strtol(f + 1, &l, 10);
      hosts.nHosts++;
      f ← l;
      break;
    }
  }
}
```

See also sections 13, 14, 15, and 16.

This code is used in section 9.

13. If $n > 1$ return a pseudo-random value between 0 and $n - 1$, or 0 otherwise.

⟨ Utility Functions 12 ⟩ +≡

```
unsigned int cointoss(int n)
{
  if (n ≤ 1) return 0;
  return random() % n;
}
```

14. Since we are dealing with arithmetic sums a lot, it's convenient to define a quick function which calculates $\sum_{i=1}^{n} i$. This is the formula attributed to Gauss:

⟨ Utility Functions 12 ⟩ +≡

```
inline int asum(int n)
{
  return (n * (n + 1))/2;
}
```

15. Likewise, it will be convenient to quickly calculate the index into the relationship array given two entities that we wish to address. If we assume $0 \le a < b < N$ (where N is the number of entities, and a and b are respective indices of entities between which we'd like to find the index into the relationship table, the formula for the index i is

$$i = a(N-1) - \left(\sum_{j=1}^{a} j\right) + b - 1.$$

⟨ Utility Functions 12 ⟩ +≡

```
inline int reli(int a, int b)
{
  if (a < b) return a * (nGuests − 1) − asum(a) + b − 1;
  else return b * (nGuests − 1) − asum(b) + a − 1;
}
```

16. This is a little fun: create a random permutation of n numbers returned as a pointer to an array of **int**s. If *iarr* is Λ, then storage sufficient for n integers is allocated and initialized with values from 0 to $n-1$ before shuffling. Otherwise, it is assumed that *iarr* points to an array of n integers. If *no_init* is `TRUE`, then it is further assumed that *iarr* contains an array of values which should merely be shuffled. Otherwise, the array is initialized with values from 0 to $n-1$.

⟨ Utility Functions 12 ⟩ +≡

```
int *generate_permutation(int *iarr, int n, int no_init)
{
  static int initialized ← FALSE;
  int k, temp_i;
  if (¬initialized) {    /* Initialize RNG if we haven't yet */
    srandom((unsigned int) time(Λ));
    initialized ← TRUE;
  }
  if (iarr ≡ Λ) {
    iarr ← (int *) malloc(sizeof(int) * n);
    no_init ← FALSE;
  }
  if (¬no_init)
    for (k ← 0; k < n; k++) iarr[k] ← k;
  ⟨ Permute the array of numbers 17 ⟩
  return iarr;
}
```

17. The algorithm is a modern variation of the Fisher-Yates shuffle popularized by Richard Durstenfeld and Knuth. The algorithm works by swapping the tail element of the array with another element randomly selected from the array, then reducing the tail pointer.

⟨ Permute the array of numbers 17 ⟩ ≡

```
while (n > 1) {
  k ← random( ) % n;    /* integer :: 0 ≤ k < n */
  n−−;
  temp_i ← iarr[n];
  iarr[n] ← iarr[k];
  iarr[k] ← temp_i;
}
```

This code is used in section 16.

18. The principal data structure is a relationship (**struct** *rel*). For N people, the number of relationships among them is $\sum_{i=1}^{N-1} i$, i.e., the fully-connected graph of N nodes. For each event, there will be some number of hosts who may each feed some number of guests. Relationships are counted between each of the guests attending a particular host's dinner. Variables *g1* and *g2* represent the entity nodes in the relationship, and *count* contains the number of times these two entities have had dinner together.

⟨ Globals 10 ⟩ +≡

```
struct rel {
    int g1, g2, count, unavailable, assigned;
} *relArr;
int nRels;
```

19. Once we've read all the guests, we build a table for keeping track of the relationship interactions. We build the table so that $g_1 < g_2$, starting with all the relationships the lowest-numbered entity may have with the other entities. We build the table in this order so that we more easily locate a relationship in the table between any two entities by calculating the index. The table is laid out like this:

index	g_1	g_2
0	0	1
1	0	2
2	0	3
$\vdots$	$\vdots$	$\vdots$
$n-2$	0	$n-1$
$n-1$	1	2
n	1	3
$\vdots$	$\vdots$	$\vdots$
$\left(\sum_{i=1}^{n-1} i\right) - 1$	$n-2$	$n-1$

⟨ Setup Functions 11 ⟩ +≡

```
void make_relationships()
{
    int i, j, c;
    nRels ← asum(nGuests − 1);
    relArr ← (struct rel *) malloc(sizeof(struct rel) * nRels);
    c ← 0;
    for (i ← 0; i < nGuests − 1; i++)
        for (j ← i + 1; j < nGuests; j++) {
            relArr[c].g1 ← i;
            relArr[c].g2 ← j;
            relArr[c].count ← relArr[c].unavailable ← relArr[c].assigned ← 0;
            c++;
        }
}
```

20. After initialization, *hostList* will contain a list index values into *guestArr* for those entities willing to host. The rest of the variables are used to manage permutations of this list.

⟨ Globals 10 ⟩ +≡

```
struct {
    int *hostList, nHosts;
    int current_host;
} hosts;
```

21. Create a list of entities willing to be hosts from data read from the guest configuration file. This list will be permuted so that we get random collections of hosts for each event.

```
⟨ Setup Functions 11 ⟩ +≡
  void make_hostlist( )
  {
    int i, h;
    hosts.hostList ← (int *) malloc(sizeof(int) * hosts.nHosts);
    for (i ← h ← 0; i < nGuests; i++)
      if (guestArr[i].guests) hosts.hostList[h++] ← i;
    hosts.current_host ← hosts.nHosts;
  }
```

22. Each event consists of a number of dinner parties. Each party has a host entity and guests.

```
⟨ Globals 10 ⟩ +≡
  struct guestlist {
    int guestent;
    struct guestlist *next;
  };
  struct party {
    int hostent, maxguests;
    int males, females;
    struct guestlist *guests;
    struct party *next;
  };
  struct event {
    struct party *pdesc;
    struct event *next;
  };
```

23. Generate a list of hosts for an event able to accommodate *seats* guests.

```
#define ERR_OUTOFHOSTS (1)
⟨ Calculation Functions 7 ⟩ +≡
  struct party *determine_hosts(int seats)
  {
    struct party *p ← Λ;
    int h;
    ⟨ Clear hosting status 24 ⟩
    while (seats > 0) {
      h ← getnexthost( );
      if (h < 0) {
        fprintf(stderr, "Ran␣out␣of␣hosts\n");
        exit(ERR_OUTOFHOSTS);
      }
      p ← add_host(h, p);
      seats −= guestArr[h].guests;
    }
    return p;
  }
```

24. ⟨ Clear hosting status 24 ⟩ ≡

for $(h \leftarrow 0;\ h < hosts.nHosts;\ h{+}{+})$ $guestArr[hosts.hostList[h]].hostingnow \leftarrow$ `FALSE`;

This code is used in section 23.

25. The key to generating a list of hosts for an event is generating a permutation. The permutation is exhausted when $hosts.current\_host \geq hosts.nHosts$, in which case a new permutation is created. If a new permutation is created, then it's possible that a host may be selected a second time for the same event. If we find that the host we're considering is already selected, we swap positions in the permutation with the first available host. If there are no more hosts available, then we do not have enough seats provided by enough hosts, and so we return an error value (a host index $h < 0$).

⟨ Setup Functions 11 ⟩ +≡

```
int getnexthost()
{
  int h, i, swapped, t;
  if (hosts.current_host ≥ hosts.nHosts) {
    generate_permutation(hosts.hostList, hosts.nHosts, TRUE);
    hosts.current_host ← 0;
  }
  h ← hosts.hostList[hosts.current_host];
  if (guestArr[h].hostingnow) {
    swapped ← FALSE;
    for (i ← hosts.current_host + 1; i < hosts.nHosts; i++)
      if (¬guestArr[hosts.hostList[i]].hostingnow) {
        t ← hosts.hostList[i];
        hosts.hostList[i] ← h;
        h ← hosts.hostList[hosts.current_host] ← t;
        swapped ← TRUE;
        break;
      }
    if (¬swapped) return −1;
  }
  hosts.current_host++;
  return h;
}
```

26. Create a new **party** structure around a host and add it to the party list. Return a pointer to the new list head. Then mark the host entity as hosting. The value *maxguests* should include the host(s).

#define ERR_MEM (2)

⟨ Setup Functions 11 ⟩ +≡
```
struct party *add_host(int h, struct party *p)
{
  struct party *newp ← (struct party *) malloc(sizeof(struct party));
  if (¬newp) {
    fprintf(stderr, "Could␣not␣allocate␣new␣party␣structure\n");
    exit(ERR_MEM);
  }
  newp→hostent ← h;
  newp→males ← guestArr[h].male;
  newp→females ← guestArr[h].female;
  newp→maxguests ← guestArr[h].guests;
  newp→guests ← Λ;
  newp→next ← p;
  guestArr[h].hostingnow ← TRUE;
  guestArr[h].hosted++;
  return newp;
}
```

27. Here we attempt to calculate the "goodness" of seating the guest entity g with the party p. Lower scores are better than higher scores. There is a MAX_BADVALUE which indicates an "infinitely bad" seating.

#define MAX_BADVALUE 10000
#define MF_WEIGHT 20
#define SEEN_WEIGHT 100
#define UNSEEN_WEIGHT −10

⟨ Setup Functions 11 ⟩ +≡
```
int calc_objective(int g, struct party *p)
{
  struct guestlist *gl ← p→guests;
  int objvalue, count, seen_count, unseen_count, mf_diff;
  ⟨ Check seating availability 28 ⟩
  ⟨ Initialize the counters 29 ⟩
  while (gl) {
    count ← relArr[reli(g, gl→guestent)].count;
    if (count) seen_count += count * count;
    else unseen_count++;
    gl ← gl→next;
  }
  mf_diff ← abs(p→males + guestArr[g].male − p→females − guestArr[g].female);
  objvalue ← SEEN_WEIGHT * seen_count + MF_WEIGHT * mf_diff + UNSEEN_WEIGHT * unseen_count;
  return objvalue;
}
```

28. If there are no seats available for a guest entity at a party, there's no point in checking further. The count of males and females at a party represents the number of guests already seated before considering the current guest entity g. If adding the current guest entity to the guest list will cause the number of guests to exceed the host's *maxguests*, then we return `MAX_BADVALUE` right away.

⟨ Check seating availability 28 ⟩ ≡
```
if (p→males + p→females + guestArr[g].male + guestArr[g].female > p→maxguests) return MAX_BADVALUE;
```
This code is used in section 27.

29. ⟨ Initialize the counters 29 ⟩ ≡
```
count ← relArr[reli(g, p→hostent)].count;
if (count) {
  seen_count ← count * count;
  unseen_count ← 0;
} else {
  unseen_count++;
  seen_count ← 0;
}
```
This code is used in section 27.

30. This is probably the most interesting process. Here's where we try to decide where each guest is seated. We try to minimize the number of times guests encounter the same people while maximizing the number of new people they meet.

⟨ Calculation Functions 7 ⟩ +≡
```
typedef struct party *PartyPtr;
void seat_guests(PartyPtr phead)
{
  PartyPtr p, *bestp;
  int i, *gp, g, bestvalue, testvalue, nbest, need_minguests, *bg;
  ⟨ Initialize seating chart 31 ⟩
  ⟨ Calculate host-preferred seating 32 ⟩
  bestp ← (PartyPtr *) calloc(hosts.nHosts, sizeof(PartyPtr));
  gp ← generate_permutation(Λ, nGuests, FALSE);
  for (i ← 0; i < nGuests; i++) {
    g ← gp[i];
    if (guestArr[g].seated) continue;
    p ← phead;
    bestvalue ← MAX_BADVALUE;
    nbest ← 0;
    ⟨ Find the best party score 33 ⟩
    if (nbest) {
      add_guest(g, bestp[cointoss(nbest)]);
      guestArr[g].seated ← TRUE;
    } else fprintf(stderr, "Couldn't␣add␣guest(s)␣%s\n", guestArr[g].name);
  }
  free(gp);
  free(bestp);
}
```

31. First, clear any previous seating indications. Then mark all the hosts as having been seated (at their own tables, of course!).

⟨ Initialize seating chart 31 ⟩ ≡
```
for (i ← 0; i < nGuests; i++) guestArr[i].seated ← FALSE;
p ← phead;
while (p) {
  guestArr[p→hostent].seated ← TRUE;
  p ← p→next;
}
```
This code is used in section 30.

32. As a first pass at seating, we attempt to put a minimum number guests at each party.

```
#define MIN_GUESTS 6
```

⟨ Calculate host-preferred seating 32 ⟩ ≡
```
bg ← (int *) calloc(nGuests, sizeof(int));
do {
  need_minguests ← FALSE;
  p ← phead;
  while (p) {
    if (p→males + p→females < p→maxguests ∧ p→males + p→females < MIN_GUESTS) {
      gp ← generate_permutation(Λ, nGuests, FALSE);
      bestvalue ← MAX_BADVALUE;
      nbest ← 0;
      for (i ← 0; i < nGuests; i++) {
        g ← gp[i];
        if (guestArr[g].seated) continue;
        testvalue ← calc_objective(g, p);
        if (testvalue < bestvalue) {
          bestvalue ← testvalue;
          bg[0] ← g;
          nbest ← 1;
        } else if ((bestvalue < MAX_BADVALUE) ∧ (testvalue ≡ bestvalue)) bg[nbest++] ← g;
      }
      if (nbest) {
        g ← bg[cointoss(nbest)];
        add_guest(g, p);
        guestArr[g].seated ← TRUE;
      }
      if (p→males + p→females < p→maxguests ∧ p→males + p→females < MIN_GUESTS)
        need_minguests ← TRUE;
      free(gp);
    }
    p ← p→next;
  }
} while (need_minguests);
free(bg);
```
This code is used in section 30.

33. For the guest g, find the lowest scoring party according to the function *calc_objective*( ). If more than one party has the same "best" score, save it on the best score stack, and we'll randomly choose among them later. When a new best score is found, clear the stack and save the pointer.

```
⟨ Find the best party score 33 ⟩ ≡
  while (p) {
    testvalue ← calc_objective(g, p);
    if (testvalue < bestvalue) {
      bestvalue ← testvalue;
      bestp[0] ← p;
      nbest ← 1;
    } else if (bestvalue < MAX_BADVALUE ∧ testvalue ≡ bestvalue) bestp[nbest++] ← p;
    p ← p→next;
  }
```

This code is used in section 30.

34. Add the guest entity indexed by g onto the guestlist for the party pointed to by p, updating all the counts. Then update the relationship table, incrementing the number of times every other person seated at the same party has seen guest entity g.

```
⟨ Setup Functions 11 ⟩ +≡
  void add_guest(int g, struct party *p)
  {
    struct guestlist *gl ← (struct guestlist *) malloc(sizeof(struct guestlist));
    gl→guestent ← g;
    gl→next ← p→guests;
    p→guests ← gl;
    p→males += guestArr[g].male;
    p→females += guestArr[g].female;
    relArr[reli(g, p→hostent)].count++;
    gl ← p→guests→next;
    while (gl) {
      relArr[reli(g, gl→guestent)].count++;
      gl ← gl→next;
    }
  }
```

35. Index.

BBT