COS 217 Final Exam

Princeton University, Fall 2003

January 19, 2004

	Score	Possible
1		20
2		20
3		30
4		15
5		15
Total		100

Your name:

Unix login:

Write and sign the honor pledge:

1 Assembly language

Assume the struct definition, and translate the function from C to Sparc assembly language.

```
struct tree    {
    int key;
    struct tree *left;
    struct tree *right;
};
```

```
void clearkeys (struct tree *t) {
  while (t != NULL) {
     t->key = 0;
     clearkeys(t->left);
     t = t->right;
  }
}
```

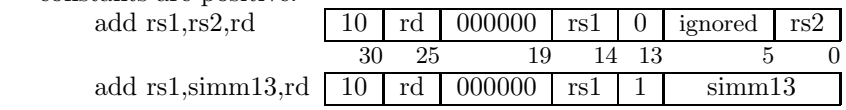
2 Machine language

Write a function to simulate a Sparc machine-language program. The array `code[]` contains a sequence of `n add` instructions (in machine language, not assembly language). Assume that all registers are initially zero. The function should return the value of register `%r1` after all the instructions are executed.

Example: If `n` is 3 and `code` contains machine code for this program,

```
add %r4,5,%r3
add %r3,2,%r6
add %r3,r6,%r1
```

then your function should return the number 12.
Hints: Use an array of 32 integers to hold the simulated registers. Do **not** execute the code by casting it to a function pointer; instead, *simulate* what the Sparc would do. You may ignore sign extension, that is, assume all 13-bit constants are positive.




```
int simulate(int n, unsigned int code[]) {

}
```

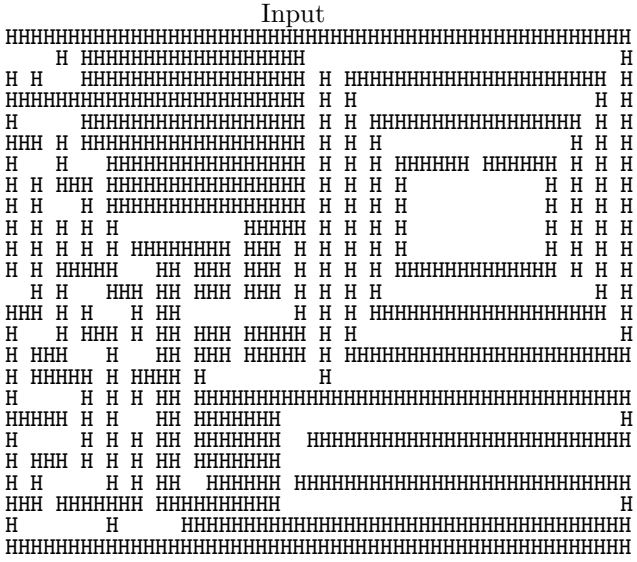
For a bit more credit (if you have time), put in assertions to make sure they're really all `add` instructions.

3 Modularity

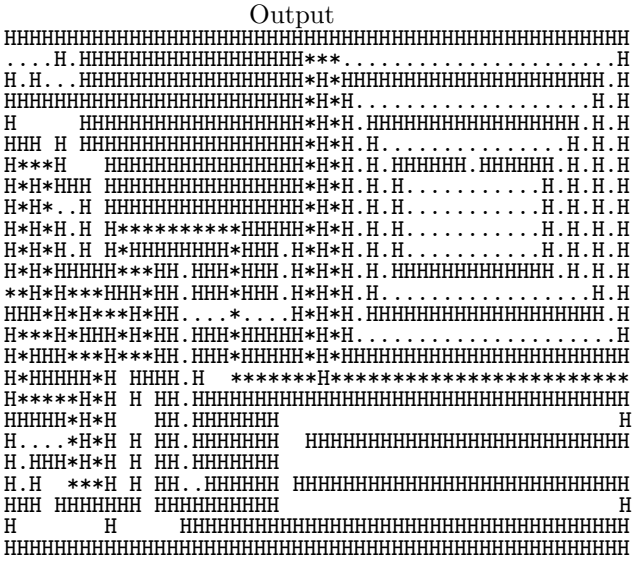
Your task: A program needs to be modularized. Show the header file that would go in between the two modules.

On the next page is a program that implements depth-first search to find a path through a maze. Given the input at left, it will produce the output at right.

Input



Output



The *'s mark the successful path; the dots mark places it visited that are not part of a successful path.

Task 1 is to design a header file that separates this program into two modules: a "search" module and a "maze" module. The search module will be the client; you write maze.h. Write data type declarations, function headers, and some comments. Limit yourself to functionality that's actually used by this particular client.

Task 2 is to translate my sketch of the client module (on the page after next) into C code

Do not implement the "maze" module in C code, just its header file.

The search algorithm works on all kinds of directed and undirected graphs, not just two-dimensional rectilinear ASCII mazes. Your `maze.h` should be useful for any situation where there are "places" and "adjacencies".

- If the "places" are squares in an ASCII maze, then two places are "adjacent" if they differ by row-number ± 1 or by column-number ± 1.
- If the places are airports, they are adjacent if there's a flight between them.

```
#include <stdio.h>
#include <assert.h>

#define ROWS 25
#define COLS 50

char maze[ROWS][COLS];

void search(int row, int col) {
  if (col==COLS) {
    /* we're deep in the recursion, we found a way all the way to
       the right-hand wall, so print out the board with H's and *'s in it. */
    for (row=0; row<ROWS; row++) {
      for (col=0; col<COLS; col++)
        putchar(maze[row][col]);
      putchar('\n');
    }
    exit(0);
  }
  else if (row >= 0 && row < ROWS     /* we didn't step out of bounds */
           && col >= 0 && col < COLS
           && maze[row][col] == ' ') {
    maze[row][col] = '*';  /* tentatively mark this square as part of
                              the solution */
    search(row,col+1);  /* try going right */
    /* if we get here, then "going right" did not lead to a solution */
    search(row,col-1);  /* try going left  */
    search(row+1,col);  /* try going up    */
    search(row-1,col);  /* try going down  */
    /* if we get here, then this square can't be part of any solution */
    maze[row][col] = '.';  /* mark this square so we never try it again */
  }
}

int main(int argc, char **argv) {
  int row, col;
  int ch;
  /* Read in the maze */
  for (row=0; row<ROWS; row++) {
     for (col=0; col<COLS; col++)
       maze[row][col] = getchar();
     ch = getchar();
     assert (ch == '\n');
  }
  ch = getchar();
  assert (ch == EOF);

  /* Try each possible door in the left-hand wall */
  col = 0;
  for (row=0; row<ROWS; row++)
    search(row,col);  /* Find the path, if any, and print it */

  /* No path worked. */
  printf("No path through the maze.\n");
  exit(1);
}
```

Question 3, continued. On this page is a sketch of the client, which uses *depth-first search*, an algorithm for finding a path through a maze. When the root-level call to Search returns, every reachable place will have been visited and marked, or else a destination will have been found.

```
Search(p) {
    If p is not marked (i.e., first time we've been here)
        If p is your destination
            Display the maze with the path marked
            Exit the process
        Mark p as (hypothetically) "in the path"
        For each adjacent place a
            Search(a)
        Mark p as "not in the path, but we've been here" }

Main(...) {
    Read the maze
    For each potential start point p
        Search(p)
    If we get here, no path worked; print message.
}
```

Put your search.c module here

Put your maze.h module here

4 Process management

Write a Kalah player that cheats, in the following way.[1] The referee creates a process to run your player, and makes sure that your process takes no more than 2 minutes. You will fork a child process and run a (10-minute) Kalah player in the child. Your own process will spend practically no time computing.

Don't create any new pipes; use the ones the referee already created, and let the referee communicate directly with its grandchild (your child).

The file `/u/mydir/slowplayer` is an executable file for a highly skilled (but slow) Kalah-player program. As usual, it expects one command-line argument (MIN or MAX).

```
int main (int argc, char *argv[]) {




}
```

[1] This cheat wouldn't actually work, because process resource limits are inherited, but some variations of it would work. I've written the question this naive way to keep things simple for you.

5 The last question

```
#include <stdio.h>

int main(int argc, char **argv) {
  int i, a=0, n=1;
  int fd[2];
  pipe(fd);
  dup2(fd[0],0);
  dup2(fd[1],1);
  for (i=0;i<5;i++) {
    printf("%d\n",n);
    fflush(stdout);
    n = n+a;
    fprintf(stderr, "%d\n",n);
    scanf("%d", &a);
  }
}
```

```
#include <stdio.h>

int main(int argc, char **argv) {
  int i, a=0, n=1;
  int fd[2];
  pipe(fd);
  close(0);
  dup(fd[0]);
  close(1);
  dup(fd[1]);
  for (i=0;i<5;i++) {
    printf("%d\n",n);
    fflush(stdout);
    n = n+a;
    fprintf(stderr, "%d\n",n);
    scanf("%d", &a);
  }
}
```

These programs are equivalent; use whichever one you like.
Either program is compiled and run with the input 10 20 30 40 50, as follows:

```
% a.out >output
10 20 30 40 50
```

A. What is the sum of all the numbers written to the file "output"?

B. What is printed on the standard error?