

Reading

Wednesday 13 April: Sethi, Sections 12.3, 12.5
Friday 15 April: Sethi, Section 12.6
Monday 18 April: Sethi, Sections 6.1, 6.5
Wednesday 20 April: Sethi, Sections 6.6, 6.8

Exercises

Due date: Tuesday 19 April. Please leave your paper in the Taylor Hall homework box (it's located in the breezeway between 2.132 and 2.136) **by 4pm**.

- [10] The snapshots of the stack of activation records in Sethi's Fig. 5.21 and 5.23 illustrate recursive and nonrecursive binary searches for 55. Draw corresponding snapshots to illustrate searches for 24, which is not in the table.
- [20] Write a tail-recursive C or C++ function `int power(int a, int b)` that computes the b^{th} power of a by repeated multiplication. Also write an iterative version of the function (call it `ipower`).
Run both versions and measure the difference in speed. One simple approach to measurement is to use the system function `clock`, which is defined in `time.h`. Execution of `clock()` returns a value of type `clock_t`, which is a synonym for an integral type; the difference between the values returned by two successive calls to `clock` is the number of "ticks" of the system clock that occurred between the two calls.
For a good measurement you will probably have to call the functions with fairly large arguments (but not too large—remember the danger of stack overflow!) and then iterate the calls a good many times. You may have to try several combinations of argument values and iteration counts to get good results.
- [20] Measure the effectiveness of in-line functions using your favorite C++ compiler. Make the body of the inner loop of your `ipower` function (from Exercise 2) into a function `ibody`, and measure the effect on the speed of your program of declaring `ibody` as an inline function.
- [10] Given the following program, what *offset* would a compiler generate for each of the variable occurrences in the procedures' bodies? Indicate the distinction between local and global variables by prefixing a **G** to global-variable references. Assume that the compiler allocates space for variables in the order in which they are declared, beginning with offset 0, and that there is no requirement that variables be aligned on word boundaries.

```
program
  type R = record integer d[20], double e[10] end;
  integer b; float x[5], y;
  procedure p()
    char m[5], R n[10]; double a, b; integer k;
  begin
    ... b ... m[k] ... y ... n[9].d[5] ...
  end p;
  procedure q()
    boolean f, h; R r[10], s, t;
  begin
    ... b ... x[3] ... r[8].e[5] ... t.d[b] ...
  end q;
begin
  ... x[2] ... y ...
end program.
```

Assume that offsets are given in bytes, and that the basic data types' space requirements (in bytes) are as follows:

| | | | | | | | | | |
|----------------|----------|-------------|----------|----------------|----------|--------------|----------|---------------|-----------|
| boolean | 1 | char | 1 | integer | 4 | float | 4 | double | 10 |
|----------------|----------|-------------|----------|----------------|----------|--------------|----------|---------------|-----------|

5. [15] Modify Sethi's Example 12.1 (page 480) so that instead of outputting each of the words that occur in a given text, it outputs only those words which contain at least three letters, all of which occur in alphabetical order. The words' counts should be included as in Sethi's example.

Your solution, like Example 12.1, should be a Unix shell program in the form of a pipeline. You may use any Unix tools, including the ones in Example 12.1 and any others you may find. At least one of the tools in your pipeline, however, should be a C/C++ program that you write yourself (a good candidate for such a program would be a filter that passes only words meeting the stated criterion).

Run your program on one of the department's Unix workstations, or on any other system that supports the pipe construct and offers the tools you need. For input, use the file `//www.cs.utexas.edu/users/ham/UTCS/CS345/Homework/Elites.text`.

6. [25] Solve Exercise 5 again, this time in Haskell. Simulate each stage in the Unix pipe with a Haskell function (each Haskell function should be designed specifically for its role in the pipeline—don't try to make it as general as the corresponding Unix utility). If you run into problems with heap or stack overflow, remember the function (`$!`) discussed in Slide 65.

To read a text file in Hugs98, use the function

```
readFile :: FilePath -> IO String
```

The argument of `readFile` is a `String` containing the path of the file to be read.

7. [20] Sethi, Exercise 12.10.

Clarification: The "readers-and-writers" problem to which Sethi refers is a variation on the producer-consumer buffer-reading and -writing problem described in Section 12.7 and in lectures. It allows multiple consumers ("readers"), which do not update the resource, to access it simultaneously. Producers, which do update the resource, must have exclusive access to it. Both of your solutions (12.10a and 12.10b) should be in Ada; Solution 12.10a should define a single coordinator task which provides coordination service to readers and writers, whereas Solution 12.10b should use semaphores, defined in Ada code, whose *enter* and *leave* entries are called within the readers' and writers' code.

CS 345 Richards

Assignment **9**

Authors:

1. _____

2. _____

3. _____

4. _____