CMPS 12B (Winter 2002) Final Exam

March 16, 2002

This exam is 12 pages long and has 12 questions with 2 bonus questions.

This exam is being given in two halves. You are holding the first half, and you may request the second half at any time. However, **you must turn in any parts of the exam you have been given if you want to leave the room for *any* reason**. If you think you might want to leave halfway through for a bathroom break or a smoke, don't pick up the second half of the exam until you return.

You must show all of your work—partial credit may be given for partially correct answers, while answers with no justification may not receive full points.

In life, it's often best to know when you don't know the answer. Thus, answers left blank will get 20% of the points for the question (applies only to whole questions, not parts of a question). However, answers that are totally wrong will receive no credit. If you think you know some of the answer, write it down, and you'll get partial credit. If you don't have a clue, don't guess.

You may use the back of the exam sheets if you need extra space. **This exam is closed-book and closed-note.** You may use a calculator without any alphanumeric memory.

There are 205 points on this exam; 200 is considered a perfect score. In addition, there are two 10 point bonus questions (you don't get points for leaving these blank, though).

Please do the following things before starting (NOW!):

- Write your name and CATS account legibly on the top of each page. **Omitting your name & CATS account from a page will result in the loss of 5 points for each page without this information.**
- Read over the exam and do the easy questions first.

You may keep this page for reference.

Name: _____ CATS account: _____@cats

1. *(15 points)* **Short Answer in C**

a. (*5 pts***)** Write the declaration for `p` so that its type is correct for the following C statements:

```
int *arr[50];
p = &(arr[3]);
```

```
int **p;
```
The array is of type "pointer to integer," and the statement takes a pointer to an element of the array, making it "pointer to pointer to integer."

b. (*10 pts***)** Write a function `makeArray(n)` in C that will create an array of *n* `double`s and return it in such a way that the calling function can safely use it.

```
double *makeArray(int n) {
  double *d = (double *)malloc (sizeof (double) * n);
  return (d);
}
```

2. *(15 points)* **Mergesort**

Using the algorithm from Assignment #5, show the contents of the two lists below after you've performed 10 moves. Onto which list was the last item moved?

Current → 3 32 28 20 41 12 57 38

48 23 35 19 22 59 8 37 67 40

3 32 28 20 41 12 57 38	last → 12 57 38 3 32 48 19 20 22 41
48 23 35 19 22 59 8 37 67 40	59 8 37 67 40 23 23 28 35
22 32 20 3 38 12 67 41	last → 12 67 41 22 32 59 3 19 28 38
59 23 35 19 28 48 8 40 57 37	48 8 40 57 37 20 23 35

Name: _____ CATS account: _____@cats

3. *(20 points)* **C Programming**

Write a C function that takes two pointers to strings (in character arrays), and swaps the two strings. The swap ends when either string "runs out" of characters.

For example, the following code:

```
char str1[] = "abcdefg";
char str2[] = "12345";
stringSwap (str1, str2);
printf ("str1='%s', str2='%s'\n");
```

will print `str1='12345fg', str2='abcde'`. Your job is to write the function `stringSwap()`. Your function may assume that the both strings are valid and terminated by NULL characters ('\0'). In other words, don't worry about error checking.

```
void stringSwap (char *s1, char *s2)
{
  char  temp;

  while (*s1 != '\0' && *s2 != '\0') {
    temp = *s1;
    *s1 = *s2;
    *s2 = temp;
    s1++;
    s2++;
  }
}
```

Alternatively, you could do:

```
void stringSwap (char s1[], char s2[])
{
  char  temp;
  int   i;

  for (i = 0; s1[i] != '\0' && s2[i] != '\0'; i++) {
    temp = s1[i];
    s1[i] = s2[i];
    s2[i] = temp;
  }
}
```

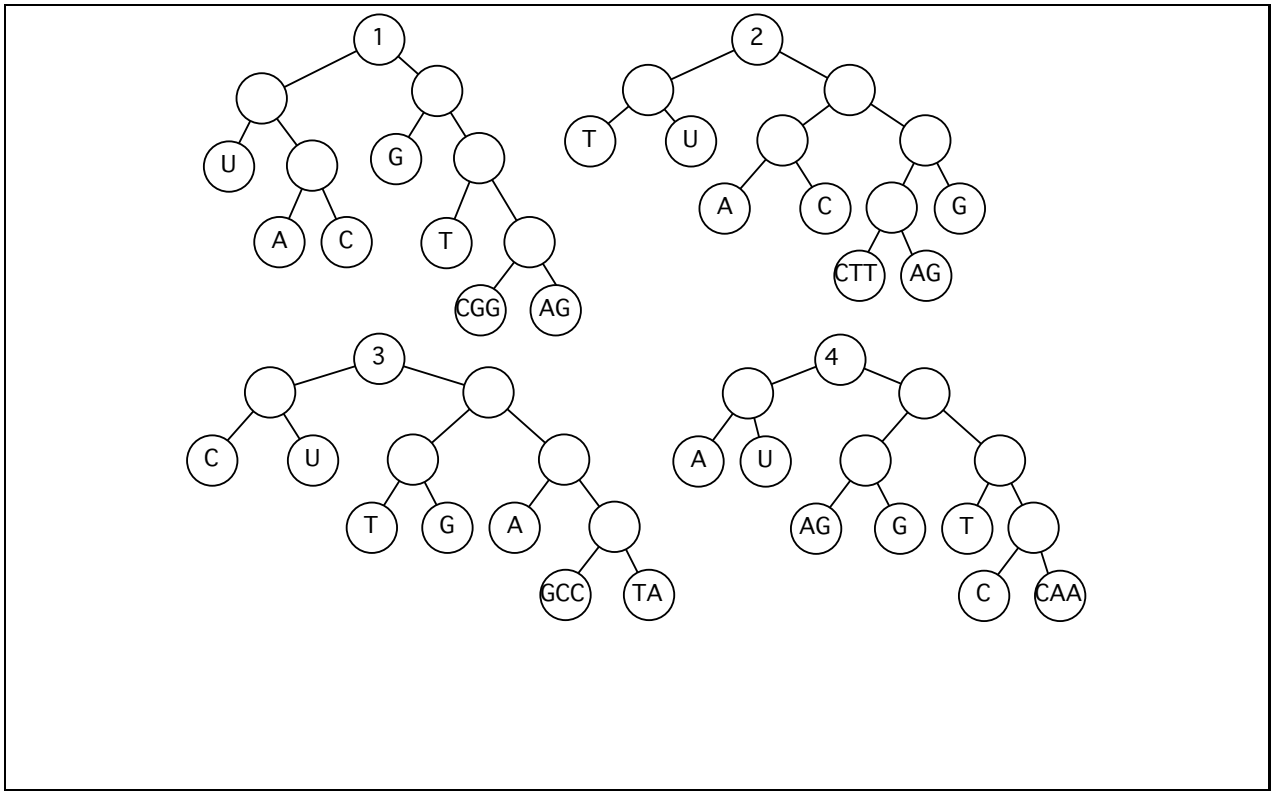
Name: _____ CATS account: _____@cats

4. *(20 points)* **Huffman decompression**

For this problem, use the Huffman decompression algorithm from Assignment #4 (where you read in a binary tree and then traverse it). The input for the algorithm is:

```
001A1U002AG1G01T01C3CAA
10111001011111011000111011010111001100011111
```

a. (*10 pts***)** Draw the binary tree that will be used to decompress the file.



b. (*10 pts***)** What are the contents of the decompressed file? In other words, what happens when you traverse the tree according to the input data?

Exam version	Answer
1	CTUUCGGUAGUTAGTUAGUCGG
2	GCUCTTUAGUAGUTGAUGUCTT
3	ATUUGCCUTAUCAGGUTAUGCC
4	GTUUCAAUAGUTTGTUAGUCAA

Name: _____ CATS account: _____@cats

5. *(20 points)* **Linked Lists in C**

Find and fix the bugs (syntax errors and/or logic errors) in the following C code to build a singly linked list of integers from an array of integers. The list should have the values in the same order as the array, starting from the head (the value returned by `buildList`). **HINT:** there are *at least* 5 different bugs in this code.

```
struct sllnode {
  int value;
  struct sllnode *next;
};

struct sllnode buildList (int v[], int n)
{
  int i;
  struct sllnode *prev, *cur;
  struct sllnode *head;
  for (i = 0; i <= n; i++) {
    prev = cur;
    cur = (struct sllnode *)malloc (sizeof (struct sllnode *));
    cur.value = v[i];
    if (i == 0) {
      first = cur;
    }
    prev.next = cur;
  }
  return (head);
}
```

The problems with this code are:

- The return type of buildList should be `struct sllnode *`.
- The loop test should be `i < n`, not `i <= n`.
- The `malloc()` call will allocate the wrong amount of space; it should be `(struct sllnode *)malloc (sizeof (struct sllnode))`.
- The fragments `cur.value` and `prev.next` should be `cur->value` and `prev->next`, respectively.
- The statement `prev->next = cur;` should be in an `else` clause of the `if` statement. In the first iteration, `prev` doesn't have a valid value.
- We need to set `cur->next = NULL;`, at least for the last value in the list.
- The variable "first" should be "head".

Name: _____ CATS account: _____@cats

6. *(10 points)* **Exceptions**

a. (*5 pts***)** Why are exceptions useful? Why not simply return an "illegal" value?

Exceptions are useful because there is no illegal value for many functions. For example, a function that returns a character may be able to return *any* character, making it impossible to select on for an "error" return.

b. (*5 pts***)** Write the Java code fragment to generate a `ListException`. Assume the class `ListException` is already defined.

```
ListException e = new ListException;
throw e;
```

Name: _____ CATS account: _____@cats

7. *(10 points)* **Sorting speed**

Are there any arrays of integers that bubble sort would sort in *less* time than the recursive version of mergesort? If so, describe the kinds of arrays that might cause this behavior. If this would never happen, why not? Assume that memory usage is not an issue, and that both sorts are run on the same system under the same conditions.

There are two reasons why bubble sort might be faster than recursive mergesort.

- The sort is being done on relatively few items. In this case, the constant multiplier for sort time increases the $O(n \log n)$ sort time for mergesort above the $O(n^2)$ running time of bubble sort .
- The data being sorted are nearly sorted. In this case, bubble sort will spend relatively little time moving items because they're in approximately the right place. In comparison, mergesort still has to do all of the merges.

8. *(10 points)* **Program development**

a. (*5 pts***)** List two benefits for using a Makefile when developing C programs.

- You can do less typing.
- You make fewer mistakes when typing (particuarly those that might cause damage, like accidental file removal).
- You can automatically recompile only those files that need recompiling. Also, you can set up rules to build the executable using only the necessary files.
- You can get something working once and do it many times.
- You can set up rules to (for example) build object files from source files.

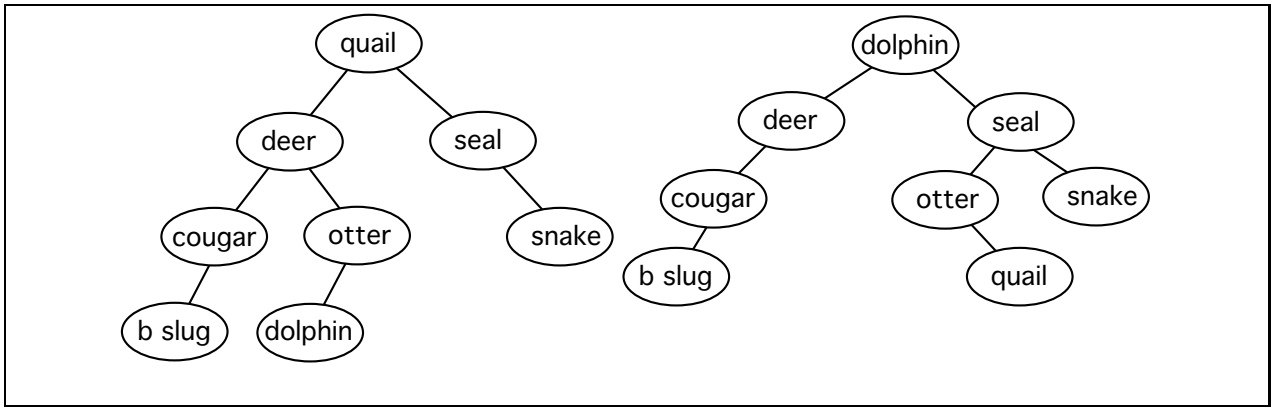
b. (*5 pts***)** Write the Unix command(s) necessary to compile the two C code files `main.c` and `utility.c` and header file `utility.h` and link them together into the executable file `myprogram`.

```
gcc -o myprogram main.c utility.c
```

Name: _____ CATS account: _____@cats

9. *(20 points)* **Binary Search Trees**

a. (*5 pts***)** Insert the following strings into an (initially empty) binary search tree, and show the resulting tree. "dolphin", "seal", "otter", "snake", "deer", "cougar", "banana slug", "quail"



b. (*3 pts***)** What is the depth of the resulting tree? How do you know?

The depth is 4 because the longest path from top to bottom has 4 nodes on it.

c. (*2 pts***)** Is this tree full? Why or why not?

The tree is not full. A full binary tree must have $2^n - 1$ elements in it; this tree has 8 elements, and thus is not full.

d. (*5 pts***)** Is this tree balanced? If not, how could you change it so it was balanced? Explain.

The tree is not balanced. In order for it to be balanced, the maximum depth must be no more than 1 greater than the minimum depth. If the tree is full, all leaves must be at the same depth for the tree to be balanced. You could relocate some of the low-hanging leaves to make the tree balanced (depending on which tree you had).

e. (*5 pts***)** Write out the strings produced using a post-order traversal of the tree.

Root	Strings produced
quail	bananaslug cougar dolphin otter deer snake seal quail
dolphin	bananaslug cougar deer quail otter snake seal dolphin

Name: _____ CATS account: _____@cats

10. *(20 points)* **Running Time**

You want to write code to read k values (possibly reading some values more than once) from a total of n items. For both parts, your answer should be in terms of n and k. Assume that the items are randomly distributed, and not in a "bad" order that would cause worst-case running time.

a. (*10 pts***)** What will the total running time be if you insert the items into a binary search tree and then read them out as needed?

It will take n insertions of $O(\log n)$ each to build the tree, and k reads of $O(\log n)$ each to read the values. Total time is $O((n+k)\log n)$.

b. (*10 pts***)** What will the total running time be if you build an array with all of the items, sort it using the best possible algorithm (ignoring radix sort) and then look up each item as needed?

Sorting the array will take time $O(n \log n)$ using either quicksort or mergesort. You can then look up each item in $O(\log n)$ time using binary search; there are k such lookups. Total time is, again, $O((n+k)\log n)$.

Name: _____ CATS account: _____@cats

11. *(20 points)* **Quicksort**

a. (*12 pts***)** Write the pseudocode for quicksort.

```
quicksort (int theArray[], int nElem)
{
  // We're done if there's at most 1 element
  if (nElem <= 1)
    return;
  Choose a pivot item p from theArray[]
  Partition the items of theArray about p
    Items less than p precede it
    Items greater than p follow it
    p is placed at index pIndex
  // Sort the items less than p
  quicksort (theArray, pIndex);
  // Sort the items greater than p
  quicksort (theArray+pIndex+1, nElem-(pIndex+1));
}
```

b. (*4 pts***)** What is quicksort's best-case, average-case, and worst-case running time?

Best case and average case running time are $O(n \log n)$. There's no way to do this sort without $\log n$ levels of recursion and n operations per level. Worst case running time is $O(n^2)$.

c. (*4 pts***)** How can the basic algorithm be modified to make the worst case running time less likely?

We can choose pivots in a different way than just using the first element. Instead, we can choose a random element or (better still) choose the median element from a random set of 3 or 5.

Name: _____ CATS account: _____@cats

12. *(25 points)* **Recursion**

Write the C code for a function `printPerm (int arr[], int n)` that prints all of the permutations (orderings) of the array `arr[]`, one permutation per line. You should print permutations and return 1 if $0 < n \leq 20$, and return 0 without doing anything if $n > 20$ or $n \leq 0$.

HINTS:

- You may want to write a separate recursive function `printPerm2 (int arr[], int n, int m)` that prints all of the permutations of array `arr[]` of length n given that the first m values are fixed.
- If the first m values are fixed, there are $n - m$ choices for element $m + 1$.
- Don't print anything until you're at the lowest level of recursion.

```
void swap(int arr[], int a, int b)
{
  int tmp = arr[a];
  arr[a] = arr[b];
  arr[b] = tmp;
}
void printPerm2 (int arr[], int n, int m)
{
  int i;
  if (m == n) {
    for (i = 0; i < n; i++) {
      printf ("%d ", arr[i]);
    }
    printf ("\n");
  }
  for (i = m; i < n; i++) {
    swap (arr, m, i);
    printPerm2 (arr, n, m+1);
    swap (arr, m, i);
  }
}

int printPerm (int arr[], int n)
{
  if (n > 20 || n < 0) {
    return (0);
  }
  printPerm2 (arr, n, 0);
  return (1);
}
```

Name: _____ CATS account: _____@cats

13. *(10 points)* **BONUS: these points will not count above a score of 89%, but cannot lower your grade.**

How can a computer system ensure that a user has entered the correct password without actually storing the password itself anywhere? Are there any potential drawbacks to this approach?

The computer stores the hash value of the user's password rather than storing the password itself. When the password is entered, the computer hashes it and compares it to the value that was stored. If they match, the password is likely to be correct.

The weaknesses of this approach are:

- There's a small chance that you can find another password with the same hash value. For an n bit hash value, the chance should be $2^{-(n-1)}$.
- You can crack the password by trying commonly used values.

14. *(10 points)* **BONUS #2: these points *will* count above a score of 89%.**

How could you use a binary tree to build a data structure that quickly allows the removal of the smallest element (a *heap*)? Keep in mind that removing an element with children may cause problems. Also, the tree should stay balanced to ensure good running time. **HINT:** consider changing the rules the relationship of a node to its children. Also, consider inserting items at the bottom and then making sure that the relationship always holds for all nodes in the tree.

The solution is to build a binary tree where each node is less than *both* of its children. This condition can be ensured by inserting new items at the *bottom* of the tree (keeping the tree balanced) and then going up the tree, swapping parent and child if parent is less than the child. Removal consists of removing the element at the root of the tree, replacing it with the "last" element in the bottom row, and going down the tree doing the same swap procedure.