Faculty of Applied Science and Engineering, University of Toronto
CSC181: Introduction to Computer Programming, Fall 2000

Final Examination
December 20, 2000

Examiner: R. Ortigas
Duration: 150 minutes (2.5 hours)
Aids allowed: One official double-sided 8.5 x 11 inch aid sheet and non-programmable calculator.

Do NOT turn this page until you have received the signal to start. (In the meantime, please fill out the identification section below, and read the following instructions carefully.)

This exam consists of 4 questions on 12 pages (including this cover page and 2 pages of appendices at the end). When you receive the signal to start, please make sure that your copy of the exam is complete.

Answer each question directly on the exam paper, in the space provided, and use the reverse side of the pages for rough work. (If you need more space for one of your solutions, use the reverse side of the page and indicate clearly which part of your work should be marked.)

Be aware that concise, well thought-out answers will be rewarded over long rambling ones. Also, unreadable answers will be given zero (0) so write legibly.

Unless otherwise stated, programming questions will be marked for correctness primarily, with efficiency and style as secondary considerations. Comments are not required, but they should accompany any long or tricky bits of code to aid the marker's understanding. If you need to make any assumptions in order to answer a question, be sure to state those assumptions clearly.

DON'T PANIC. KEEP COOL. IT'S ALL GOOD.

Family name: ____________________________________

Given name: ____________________________________

Student number: __ | __ | __ | __ | __ | __ | __ | __ | __

1:	____	/ 50
2:	____	/ 35
3:	____	/ 25
4:	____	/ 40
Total:	____	/ 150

Question 1. [5 + 10 + 20 + 15 = 50 marks]

A sorted set is a collection of unique elements which are sorted in some order. For the purposes of this question, consider a set of integers sorted in increasing order, on which the following operations can be performed:

- **contains(e):** Returns true if this set contains integer *e*, false otherwise.
- **add(e):** Adds integer *e* to this set.
- **addAll(s):** Adds all of the integers in set *s* to this set, effectively creating a union.
- **retrieve(k):** Returns the *k*th smallest integer in the set.

Consider the following declaration for class `SortedSet`, which represents sorted sets of integers:

```
class SortedSet {

private:
      // The number of elements in this set.
      int size;

      // The elements in this set, sorted in increasing
      // order with the lowest element at index 0 and the
      // highest element at index size-1.
      int elements[100];

public:
      // Constructs an empty set.
      SortedSet() { size = 0; }

      // SortedSet operations (described above).
      bool contains(int e) const;
      void add(int e);
      void addAll(const SortedSet& s);
      void retrieve(int k) const;
};
```

a) [5 marks]

Write the function `contains` for `SortedSet`. It doesn't have to be efficient; it just has to work.

```
bool SortedSet::contains(int e) const {
      for (int i = 0; i != size; i++)
            if (elements[i] == e)
                  return true;

      return false;
}
```

Question 1. [**continued** from previous page]

b) [10 marks]

Write the function `add` for `SortedSet`. It doesn't have to be efficient; it just has to work.

```
void SortedSet::add(int e) {
      assert(size < 100);

      if (contains(e))
            return;

      int i, j;

      for (i = 0; i != size && e > elements[i]; i++);

      for (j = size; j != i; j--)
            elements[j] = elements[j-1];

      elements[i] = e;
      size++;
}
```

c) [20 marks]

Write the function `addAll` for `SortedSet`. For this question, **efficiency is really important.** (Hint: Think about the algorithm for merging two sorted sequences.)

```
void SortedSet::addAll(const SortedSet& s) {
      // Assume number of unique integers in s and
      // this set is less than or equal to 100.

      int temp[100];
      int i, j, k;

      for (i = 0, j = 0, k = 0; j != size && k != s.size; i++) {
            if (elements[j] < s.elements[k]) {
                  temp[i] = elements[j]; j++;
            }
            else if (elements[j] > s.elements[k]) {
                  temp[i] = s.elements[k]; k++;
            }
            else {
                  temp[i] = elements[j]; j++; k++;
            }
      }

      for ( ; j != size; i++, j++)
            temp[i] = elements[j];

      for ( ; k != s.size; i++, k++)
            temp[i] = s.elements[k];

      size = i;
      memmove(elements, temp, size * sizeof(int));
}
```

Question 1. [**continued** from previous page]

d) [15 marks]

Because `SortedSet` uses a sorted array, the `retrieve` function is easy to implement. This function would be difficult to write, however, if `SortedSet` stored its numbers in a BST. We can address this problem by making a slight modification to the BST.

Define the *weight* of a tree to be simply the number of nodes in the tree, and the *weight of a node* to be the weight of the tree rooted at that node. Now, consider a *weighted BST*, where each node stores its weight in addition to its key and pointers to its left and right children:

```
struct WeightedBSTNode {
      int key, weight;
      WeightedBSTNode *left, *right;
};
```

Given the root of a weighted BST, we can determine whether the *k*th smallest key is in the left subtree, at the root or in the right subtree, simply by examining the weight of the root's left child. Using this observation and the `WeightedBSTNode` definition given above, write a **recursive** function (which does not contain any loops) `kthSmallest` with the following prototype:

```
WeightedBSTNode* kthSmallest(int k, WeightedBSTNode* r);
```

This function should return a pointer to the node containing the `k`th smallest key in the weighted BST rooted at the node pointed to by `r`. Assume that `r` is not null, and that `k` is in the range from `1` to the weight of the node pointed to by `r`, inclusive.

Examples: (Each node is represented as **key**|`weight`.)

<table>
<tr><th>Contents of weighted BST</th><th>Arguments to kthSmallest</th><th>Result from kthSmallest</th></tr>
<tr><td rowspan="4"><pre>   ____8|8____
    /   \
   5|4   10|3
  /   \   /   \
4|1 6|2   9|1 12|1
       \
       7|1</pre></td><td>1, pointer to node w/8|8</td><td>pointer to node with 4|1</td></tr>
<tr><td>5, pointer to node w/8|8</td><td>pointer to node with 8|8</td></tr>
<tr><td>6, pointer to node w/8|8</td><td>pointer to node with 9|1</td></tr>
<tr><td>1, pointer to node w/10|3</td><td>pointer to node with 9|1</td></tr>
</table>

```
WeightedBSTNode* kthSmallest(int k, WeightedBSTNode* r) {
      int lWeight = (r->left == null) ? 0 : r->left.weight;
      if (k <= lWeight)
            return kthSmallest(k, r->left);
      else if (k == lWeight + 1)
            return r;
      else
            return kthSmallest(k - (lWeight + 1), r->right);
}
```

Question 2. [10 + 25 = 35 marks]

a) [10 marks]

Write a function `strnchr` with the following prototype:

```
char* strnchr(char* s, char c);
```

This function returns a pointer to the first character, in the string pointed to by `s`, that is *not* `c`. It returns `0` (the null pointer) if no character which is not `c` is found. Assume `s` is not null.

```
char* strnchr(char* s, char c) {
      char* result = 0;
      int n = strlen(s);
      for (int i = 0; i != n && result == 0; i++) {
            if (s[i] != c)
                  result = &s[i];
      }
      return result;
}
```

b) [25 marks]

Consider a class `StringTokenizer` with the following public interface:

```
class StringTokenizer {
public:
      StringTokenizer(const char* s);
      bool hasMoreTokens();
      char* nextToken();
};
```

This class allows a string to be broken into *tokens*, and allows these tokens to be read in one at a time. The `StringTokenizer` constructor creates a tokenizer for the string pointed to by `s`. `hasMoreTokens` tests if there are more tokens available from the tokenizer's string. If this function returns true, then a subsequent call to `nextToken` will successfully return the next token from the string. The tokenizer looks for blank spaces (single spaces or runs of spaces) to identify where each token begins and ends.

On the following page, complete the definition of `StringTokenizer` by adding private instance variables to the class and implementing its constructor and functions.

You may use `strnchr` from part (a). Assume that it works correctly, even if you didn't answer that question (correctly). You may also use any of the string processing functions in Appendix A.

Example:

Consider the following code fragment:

```
StringTokenizer st("  the  quick brown   fox ");
while (st.hasMoreTokens())
      cout << st.nextToken() << '|';
cout << endl;
```

This fragment should print: `the|quick|brown|fox|`. Note that leading and trailing spaces, as well as runs of spaces, are not included in the tokens.

Question 2. [**continued** from previous page]

[Space for your solution to part (b)]

```
class StringTokenizer {
private:
      char* string;
      char* cursor;

public:
      StringTokenizer(const char* s);
      bool hasMoreTokens();
      char* nextToken();
};

StringTokenizer::StringTokenizer(const char* s) {
      string = new char[strlen(s)+1];
      strcpy(string, s);
      cursor = strnchr(string, ' ');
}

bool StringTokenizer::hasMoreTokens() {
      return cursor != 0;
}

char* StringTokenizer::nextToken() {
      assert(hasMoreTokens());

      char* result;
      char* end = strchr(cursor, ' ');
      if (end != 0) {
            result = new char[end-cursor+1];
            strncpy(result, cursor, end-cursor);
            result[end-cursor] = (char)0;
            cursor = strnchr(end+1, ' ');
      }
      else {
            result = new char[strlen(cursor)+1];
            strcpy(result, cursor);
            cursor = 0;
      }

      return result;
}
```

Question 3. [25 marks]

Consider the game of Tic-Tac-Toe. If you are not familiar with the rules of Tic-Tac-Toe, please consult Appendix B.

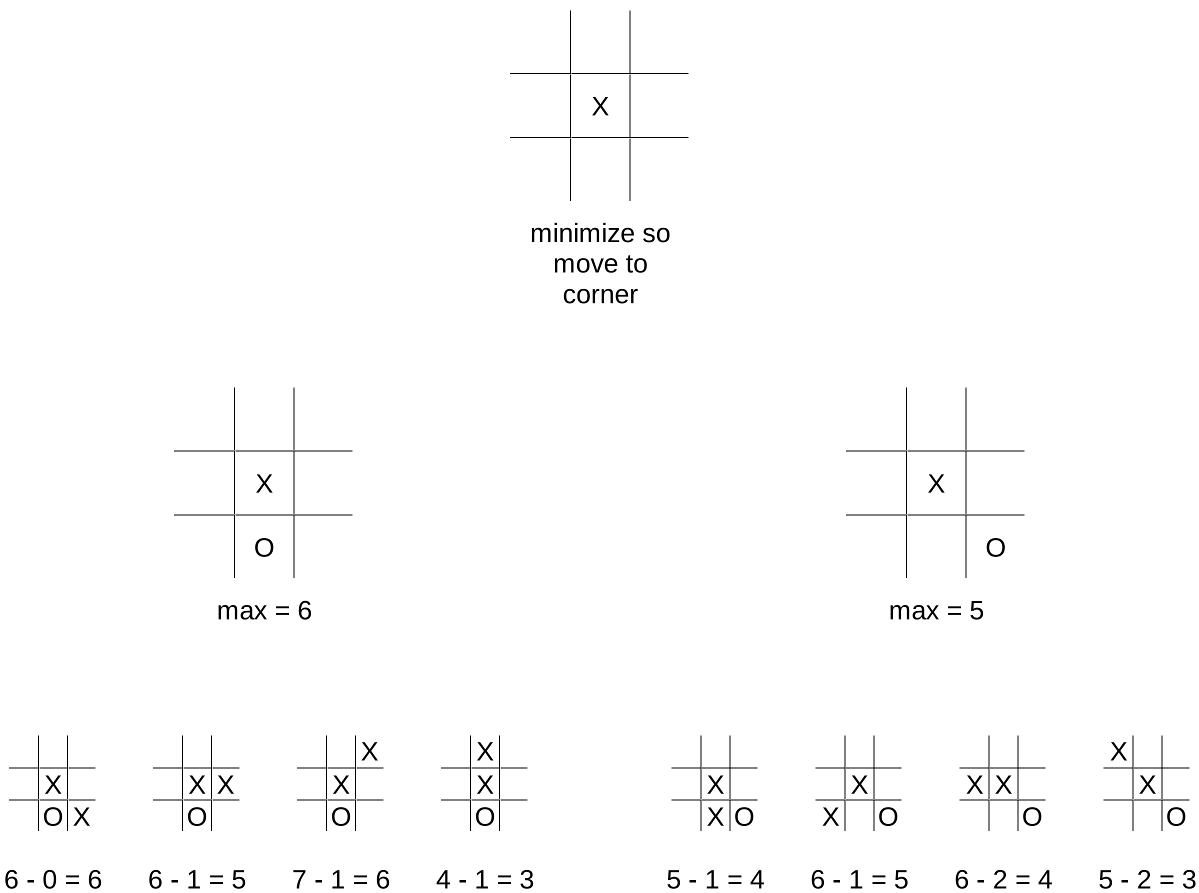
Starting from the initial board below, which has one X in the centre square, carry out the minimax algorithm to a cutoff depth of 2 (i.e. two X's and one O on the board), taking symmetry into account. If you are not familiar with the minimax algorithm, please consult Appendix C.

In carrying out the minimax algorithm, use the following evaluation function:

$$\text{eval(board)} = 3X_2 + X_1 - (3O_2 + O_1)$$

X_n is the number of rows, columns or diagonals with exactly *n* X's and no O's, and O_n is similarly defined as the number of rows, columns or diagonals with just *n* O's.

Show all your work. That is, show the whole game tree starting from the initial board below down to depth 2 (i.e. two X's and one O on the board); show the evaluations of the boards at the cutoff depth; show the values calculated by the minimax algorithm for the boards higher up the tree; and at the end, indicate what is the best move for O from the initial board below, as suggested by minimax.



Question 4. [2 x 20 = 40 marks]

Consider the selection sort algorithm. If you are not familiar with the selection sort algorithm, please consult Appendix D.

a) [20 marks]

Consider the following declaration of `ListNode`, which represents a node in a singly-linked list of integers. Each node contains an integer and a link to the next node:

```
struct ListNode {
      int data;
      ListNode* next;
};
```

Write a function `linkedSort` with the following prototype:

```
ListNode* linkedSort(ListNode* x);
```

This function should sort the integers in the linked list whose first node is pointed to by `x`, in nondecreasing order, using the selection sort algorithm. It should also return a pointer to the first node of the sorted linked list (which may or may not be `x` depending on how you write the function). Assume `x` is not null.

You may not use any auxiliary data structures. You may write helper functions for `linkedSort`, but these also cannot use any auxiliary data structures.

```
ListNode* linkedBringMinToFront(ListNode* x) {
      ListNode* min = x;
      ListNode* curr = x->next;

      while (curr != 0) {
            if (curr->data < min->data) {
                  min = curr;
            }
            curr = curr->next;
      }

      if (min != curr) {
            int temp = min->data;
            min->data = x->data;
            x->data = temp;
      }

      return min;
}

ListNode* linkedSort(ListNode* x) {
      if (x->next != 0) {
            linkedBringMinToFront(x);
            linkedSort(x->next);
      }
      return x;
}
```

Question 4. [**continued** from previous page]

b) [20 marks]

Write a function `arraySort` with the following prototype:

```
void arraySort(int a[], int n);
```

This function should sort the first n integers of array a in nondecreasing order using the selection sort algorithm. Assume n > 0, and that a has at least n integers.

You must use **recursion**. You may not use any loops or auxiliary data structures. You may write helper functions for `arraySort`, but these also cannot use any loops or auxiliary data structures.

```
int arrayMinIndex(int a[], int n) {
      if (n == 1) {
            return 0;
      }
      else {
            int minIndexOfRest = 1 + arrayMinIndex(&a[1], n-1);
            return (a[minIndexOfRest] < a[0]) ? minIndexOfRest : 0;
      }
}

void arrayBringMinToFront(int a[], int n) {
      int minIndex = arrayMinIndex(a, n);
      if (minIndex != 0) {
            int temp = a[0];
            a[0] = a[minIndex];
            a[minIndex] = temp;
      }
}

void arraySort(int a[], int n) {
      if (n > 1) {
            arrayBringMinToFront(a, n);
            arraySort(&a[1], n-1);
      }
}
```

Question 0. [5 x 2 = 10 bonus marks]

a) [2 bonus marks]

Write your student number LEGIBLY on the top of every page of this exam. (On the first page of this exam, write your student number where you are asked to write it.)

b) [2 bonus marks]

What's my symbol for Bad? **A cross. (×)**

c) [2 bonus marks]

What's my symbol for Good? **A checkmark. (✓)**

d) [2 bonus marks]

You've been a great group of students, and I've really enjoyed teaching you. Congratulations on your first semester of Engineering Science at U of T, and best of luck in the future.

e) [2 bonus marks]

Thanks for coming, and happy holidays.

[END OF EXAM]

Appendix A: String Library Functions

(Source: Appendix D of *C Programming: A Modern Approach* by K.N. King.)

`char *strcat(char *s1, const char *s2);`

Appends characters from the string pointed to by `s2` to the string pointed to by `s1`. Returns `s1`.

`char *strchr(const char *s, int c);`

Returns a pointer to the first occurrence of the character `c` in the string pointed to by `s`. Returns `0` (null pointer) if `c` isn't found.

`int strcmp(const char *s1, const char *s2);`

Returns a negative, zero or positive integer, depending on whether the string pointed to by `s1` is less than, equal to, or greater than the string pointed to by `s2`.

`char *strcpy(char *s1, const char *s2);`

Copies the string pointed to by `s2` into the array pointed to by `s1`. Returns `s1`.

`size_t strlen(const char *s);`

Returns the length of the string pointed to by `s`, not including the null character.

`char *strncat(char *s1, const char *s2, size_t n);`

Appends characters from the array pointed to by `s2` to the string pointed to by `s1`. Copying stops when a null character is encountered or `n` characters have been copied. Returns `s1`.

`char *strncpy(char *s1, const char *s2, size_t n);`

Copies the first `n` characters of the array pointed to by `s2` into the array pointed to by `s1`. If it encounters a null character in the array pointed to by `s2`, `strncpy` adds null characters to the array pointed to by `s1` until a total of `n` characters have been written. Returns `s1`.

`char *strrchr(const char *s, int c);`

Returns a pointer to the last occurrence of the character `c` in the string pointed to by `s`. Returns `0` (null pointer) if `c` isn't found.

`char *strstr(const char *s1, const char *s2);`

Returns a pointer to the first occurrence in the string pointed to by `s1` of the characters in the string pointed to by `s2`. Returns `0` (null pointer) if no match is found.

`void *memcpy(void *s1, const void *s2, size_t n);`

Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. May not work properly if the objects overlap. Returns `s1`.

`void *memmove(void *s1, const void *s2, size_t n);`

Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. Will work properly if the objects overlap, although it may be slower than `memcpy`. Returns `s1`.

Appendix B: Rules of Tic-Tac-Toe

The game of Tic-Tac-Toe is played between two players, X and O, who each have a set of playing pieces they can place on a grid with 3 columns and 3 rows of squares. It is governed by the following rules:

- X and O alternate moves (with X starting) until one of them has won, or there are no more legal moves.
- To move, a player puts one of their pieces on the grid. Each square on the grid can hold only one piece.
- The winner is the first player to connect three consecutive pieces of theirs in a line, either horizontally, vertically or diagonally. It is possible for the game to end in a draw; in this case, the entire grid is filled with neither player having achieved the winning condition.

Appendix C: Minimax Algorithm

(Source: Chapter 5 of *Artificial Intelligence: A Modern Approach* by S. Russell and P. Norvig.)

function MINIMAX-DECISION(*game*) **returns** *an operator*

  for each *op* **in** OPERATORS[*game*] **do**
    VALUE[*op*] ← MINIMAX-VALUE(APPLY(*op*, *game*), *game*)
  end
  return the *op* with the highest VALUE[*op*]

---

function MINIMAX-VALUE(*state, game*) **returns** *a utility value*

  if TERMINAL-TEST[*game*](*state*) **then**
    return UTILITY[*game*](*state*)
  else if MAX is to move in *state* **then**
    return the highest MINIMAX-VALUE of SUCCESSORS(*state*)
  else
    return the lowest MINIMAX-VALUE of SUCCESSORS(*state*)

MINIMAX-CUTOFF is identical to MINIMAX-VALUE except TERMINAL-TEST is replaced by CUTOFF-TEST (which indicates whether the given game state/board is at the cutoff depth) and UTILITY is replaced by EVAL (the evaluation function).

Appendix D: Selection Sort Algorithm

Consider a sequence of *n* integers. To rearrange these integers in nondecreasing order, use the following algorithm:

- Find the smallest integer in the sequence and swap it with the first integer.
- Find the second smallest integer in the sequence and swap it with the second integer.
- Continue in this manner for the *n* integers of the sequence.