Final Exam

15-122 Principles of Imperative Computation
Frank Pfenning, Tom Cortina, William Lovas

December 10, 2010

Name: **Sample Solution** Andrew ID: **fp** Section:

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 3 hours to complete the exam.
- There are 6 problems.
- Read each problem carefully before attempting to solve it.
- Consider writing out programs on scratch paper first.

	Integers	Generic Sort	Graphs	Tries	Generic Traversal	Dynamic Programming	
	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Total
Score	**40**	**30**	**60**	**40**	**40**	**40**	**250**
Max	40	30	60	40	40	40	250
Grader							

1 Integers (40 pts)

Task 1 (15 pts). For each of the following expressions, indicate whether it is **always true**, **always false**, **true or false**, **abort** (must abort), **sometimes undefined** (there are some values for x and y where the result is not defined). For both languages we assume declarations

```
int x = ...;
int y = ...;
```

so x and y are both properly initialized to arbitrary values.

Expression	Value in C0	Value in C
`x + 0 == x`	**aw.true**	**aw.true**
`x + y == y + x`	**aw.true**	**s.undef.**
`0 - x == -x`	**aw.true**	**s.undef.**
`x > 0 || x + 1 > x`	**aw.true**	**aw.true**
`1/0 == 1/0`	**abort**	**s.undef.**

Task 2 (10 pts). Write a C0 function `bit` which returns bit number i in the two's complement representation of n. Also supply a postcondition capturing the value range of the output.

```
int bit(int n, int i)
//@requires 0 <= i && i < 32;
//@ensures i == 0 || i == 1;
{
  return (n >> i) & 1;
}
```

Task 3 (15 pts). Write a C0 function `uleq` (*unsigned less or equal*) which compares two integers x and y interpreted as *unsigned* numbers. So, for example, `uleq(x,-1)` is `true` for all x. You may use the `bit` function from Task 2.

```
bool uleq(int x, int y) {
  int i;
  for (i = 31; i >= 0; i--)
    if (bit(x,i) < bit(y,i)) return true;
    else if (bit(x,i) > bit(y,i)) return false;
  return true;
}
```

2 Generic Sort (30 pts)

The sorting functions we studied in C0 just sorted arrays of integers into ascending order. In practice, we would want a sorting function that can sort arrays of arbitrary data according to an arbitrary ordering. In this problem we will sketch portions of such a generic sorting function and how to use it.

Assume we have, in C, an in-place sorting function `sort` with declaration

```
void sort(int[] A, int lower, int upper);
```

where in the body of the function we compare array elements only using the inequality `A[i] <= A[j]`.

Task 1 (10 pts). Give a declaration of a generic sorting function `sort` in C which permits sorting an array A of pointers to arbitrary data, taking as argument a generic comparison function *leq* on arbitrary data elements. Only show the function prototype, the way it would appear in an interface, do *not* give its definition.

```
void sort(void** A, bool (*leq)(void* x, void* y), int lower, int upper);
```

Task 2 (5 pts). In order to obtain a generic `sort` function, we have to replace comparisons

```
A[i] <= A[j]
```

in the original sorting function. Show the new form of the comparison (without writing the rest of the function).

```
(*leq)(A[i], A[j])
```

Task 3 (10 pts). Assume we have data structure of weighted edges in an undirected graph, where vertices are represented as integers.

```
struct edge {
  int u1;
  int u2;
  int weight;
};
typedef struct edge* edge;
```

Here `u1` is one end point of the edge, `u2` the other, and `weight` its weight. Write a function `edge_leq` to compare edges, taking into account that you want to pass a pointer to this function to the function `sort`.

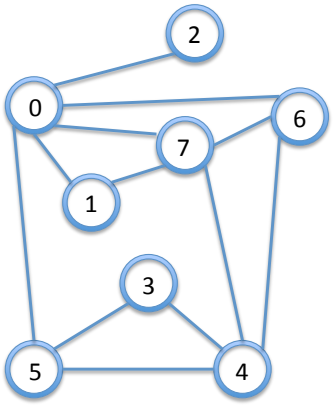
```
bool edge_leq(void* e1, void* e2) {
  REQUIRES(e1 != NULL && e2 != NULL);
  return ((edge)e1)->weight <= ((edge)e2)->weight;
}
```

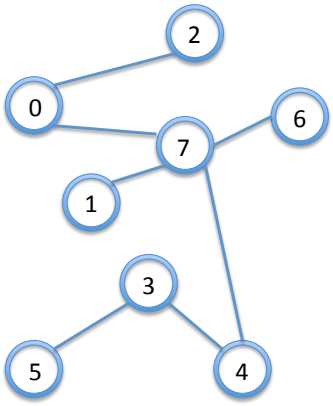
Task 4 (5 pts). Assume we have an array E containing e edges that we would like to sort by increasing weight (as we have to, for example, in Kruskal's algorithm). Based on your answers in Tasks 1–3, write out the correct function call to sort E in place, using the generic function `sort`.

```
  sort(E, &edge_leq, 0, e)
```

3 Graphs (60 pts)

Task 1 (15 pts). We can apply Kruskal's algorithm to find a minimum spanning tree for the graph on the left





with the edge weights shown below on the left. In the table below on the right, fill in the edges in the order considered by Kruskal's algorithm and indicate for each whether it would be added to the spanning tree (**Yes**) or not (**No**). Do not list edges that would not even be considered. When you are done, draw in the spanning tree above on the right.

edge	weight
$0-6$	51
$0-1$	32
$0-2$	29
$4-3$	34
$5-3$	18
$7-4$	46
$5-4$	40
$0-5$	60
$6-4$	51
$7-0$	31
$7-6$	25
$7-1$	21

edge considered	added?
$5-3$	**yes**
$7-1$	**yes**
$7-6$	**yes**
$0-2$	**yes**
$7-0$	**yes**
$0-1$	**no**
$4-3$	**yes**
$5-4$	**no**
$7-4$	**yes**

For this problem we represent vertices by integers $0, 1, \ldots$ and then edges (as in Problem 2, Task 4) and graphs by the following structs. We assume weights are integers greater or equal to 0.

```
struct edge {
  int u1;                       /* one endpoint */
  int u2;                       /* other endpoint */
  int weight;
};
typedef struct edge* edge;

struct graph {
  int num_vertices;             /* number of vertices */
  int num_edges;                /* number of edges */
  edge* edges;                  /* edge array */
};
typedef struct graph* graph;
```

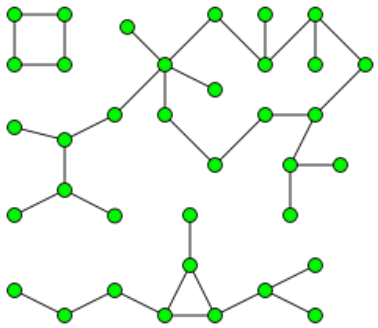
Task 2 (10 pts). Write a C function to allocate a new graph with n vertices and space for e edges, where all the edges are initialized to the null pointer. You may use the functions `xmalloc(size_t nbytes)` and `xcalloc(size_t nobj, size_t size)` which raise an exception if not enough memory is available.

```
graph graph_new(int n, int e) {
  graph G = xmalloc(sizeof(struct graph));
  edge* E = xcalloc(e, sizeof(edge)); /* will init to null */
  G->num_vertices = n;
  G->num_edges = e;
  G->edges = E;
  return G;
}
```

Task 3 (15 pts). Complete the following function to compute a minimum spanning tree for a given graph G, to be stored in H. Assume that H is initialized with `graph_new(n, n-1)` where `n = G->num_vertices`. The auxiliary data structure `eqs` is there to efficiently detect cycles. It maintains equivalence classes of vertices via union/find; for this problem it is only important that `cycle(eqs, u, v)` returns true if adding the edge from u to v would create a cycle. For this task you should assume the given graph G is connected, that is, there is a path between any two vertices. You may use functions `edge_leq` and `sort` from Problem 2 (Generic Sort).

```
graph mst (graph G, graph H) {
  assert(G != NULL);            /* graph must be defined */
  int n = G->num_vertices;
  int e = G->num_edges;
  edge* E = G->edges;
  edge* M = H->edges;
  ufs eqs = singletons(n);
  sort(E, &edge_leq, 0, e);     /* sort E[0..e) */
  int i = 0, k = 0;
  while (i < e && k < n-1)
    { if (!cycle(eqs, E[i]->u1, E[i]->u2)) {
        M[k] = E[i];            /* add to mst */
        k++;
      }
      i++;
    }
  assert(k != n-1);             /* graph was not connected */
  return;
}
```

Task 4 (10 pts). A *connected component* of a graph is a set of vertices where each node can reach every other node in the component along the given edges, and which is connected to no additional vertices. For example, the graph below has 3 connected components.



Explain, in words, how to use Kruskal's algorithm to compute the number of connected components in a graph.

In a graph with c connected components, the spanning trees for the components have a total of $n - c$ edges, where n is the total number of vertices in the graph. Therefore, if we execute Kruskal's algorithm we will always end up with exactly $k = n - c$ edges, so $c = n - k$.

Task 5 (10 pts).

Show how to modify the `mst` function from Task 3 so that it returns the number of connected components of the graph, and H (initialized as in Task 3) contains the graph consisting of minimum spanning trees for each connected component. If you need to add additional lines before or after the loop, please write them in and indicate clearly where they belong.

```
graph mst (graph G, graph H) {
  assert(G != NULL);            /* graph must be defined */
  int n = G->num_vertices;
  int e = G->num_edges;
  edge* E = G->edges;
  edge* M = H->edges;
  ufs eqs = singletons(n);
  sort(E, &edge_leq, 0, e);     /* sort E[0..e) */
  int i = 0, k = 0;
  while (i < e && k < n-1)
    { if (!cycle(eqs, E[i]->u1, E[i]->u2)) {
        M[k] = E[i];            /* add to msts */
        k++;
      }
      i++;
    }
  return n-k;
}
```

4 Tries (40 pts)

In this problem we explore how to store fixed-precision integers in a trie. We use the following representation in C0

```
struct trie {
  int data;
  trie left;
  trie right;
}
typedef struct trie* trie;
```

The function below inserts n into a trie T where i is initially highest bit in n. At each level it tests the ith bit of n, inserting n into the left subtrie the bit is 0 and the right subtrie if the bit is 1. At the leaves, with both children null, we store the number n itself. The data fields in non-leaves are ignored.

```
trie trie_insert(trie T, int n, int i) {
  if (T == NULL) {
    T = alloc(struct trie);
    T->left = NULL; T->right = NULL;
  }
  if (i < 0) {
    T->data = n;
    return T;
  }
  if (bit(n,i) == 0)  /* as defined in Problem 1, Task 2 */
    T->left = trie_insert(T->left, n, i-1);
  else
    T->right = trie_insert(T->right, n, i-1);
  return T;
}
```

Task 1 (5 pts). What is the asymptotic runtime complexity of inserting n numbers with k bits into a binary trie? Use big-O notation and explain briefly.

$O(n * k)$: we have to extract and compare k bits and perform a constant number of operations for each bit.

In the remainder of this problem we use queues holding integers with the following interface.

```
bool is_empty(queue Q);          /* O(1) */
queue queue_new();               /* O(1) */
void enq(queue Q, int n);        /* O(1) */
int deq(queue Q);                /* O(1) */
```

Task 2 (15 pts). Given a try that stores n integers (all greater or equal to 0) as described in the previous problem, write a recursive function `trie_collect` that inserts these integers into an integer queue in ascending order.

```
void trie_collect(trie T, queue Q) {
  if (T->left == NULL && T->right == NULL)
    enq(Q, T->data);
  if (T->left != NULL) trie_collect(T->left, Q);
  if (T->right != NULL) trie_collect(T->right, Q);
}
```

Task 3 (15 pts). Complete the following code to sort an array of 32-bit integers, each greater or equal to 0. You should use functions `trie_insert` and `trie_collect`. Assume that the input array A has no duplicates.

```
void sort(int[] A, int n) {
  int i;
  queue Q = queue_new();
  trie T = NULL;
  for (i = 0; i < n; i++)
    T = trie_insert(T, A[i]);
  trie_collect(A, Q);
  for (i = 0; i < n; i++)
    A[i] = deq(Q);
}
```

Task 4 (5 pts). What is the asymptotic running time of this sorting algorithm? Give your answer in big-O notation in terms of the number of bits k in the integers stored and the number of integers n in the array.

$O(k*n)$: each of the n insertion takes $O(k)$ steps, and `trie_collect` takes $O(k*n)$. Finally dequeuing the elements requires $O(n)$, which sums up to $O(k*n)$.

5 Generic Tree Traversal (40 pts)

In this problem we consider binary search trees with generic data, implemented in C.

```
typedef struct tree* tree;
struct tree {
  void* data;
  tree left;
  tree right;
};
```

Task 1 (10 pts). Assume we are given a tree T which is a binary search tree. Write a generic recursive function in C to traverse a binary search tree and free each node. In addition, it should apply a function `elem_free` to each stored data item.

```
void tree_free(tree T, void (*elem_free)(void* x)) {
  if (T == NULL) return;
  traverse(T->left);
  if (elem_free != NULL) (*elem_free)(T->data);
  traverse(T->right);
}
```

We now use a generic stack data structure with the following relevant part of the interface

```
typedef struct stack* stack;
bool stack_empty(stack S);      /* O(1) */
stack stack_new();              /* O(1) */
void push(void* x, stack S);    /* O(1) */
void* pop(stack S);             /* O(1) */
void stack_free(stack S, void (*data_free)(void* x));
```

Task 2 (20 pts). Again, assume we are given a tree T which is a binary search tree. Write a function `traverse` that applies a function `visit` to each data item in the tree, in ascending order of keys. This time, your function should use a loop and maintain an *explicit stack* instead of using recursion. A correct function using recursion will get partial credit.

```
void traverse(tree T, void (*visit)(void* x)) {
  stack S = stack_new();
  while (!(stack_empty(S) && T == NULL)) { /* (1) */
    if (T == NULL) {
      T = pop(S);                          /* (2) */
      if (visit != NULL) (*visit)(T->data); /* (3) */
      T = T->right;
    } else {
      push(T, S);                          /* (4) */
      T = T->left;                         /* (5) */
    }
  }
  stack_free(S, NULL);
}
```

Task 3 (10 pts). Explain for your program in Task 2, why we never dereference a null pointer or pop from an empty stack. Number the lines you refer to, and write down explicit pre- or post-conditions or loop invariants as needed.

1. A loop invariant at (1) is that the stack contains only trees T that are not null.
2. At line (2), S is not empty because the loop condition holds and T is null.
3. At line (3) we just checked that `visit` is not null.
4. At line (4) we preserve the loop invariant because T is not null in the else-clause of the conditional.
5. At line (5), T is not null because we are in the else-clause of the conditional testing whether T is null.

6 Dynamic Programming (40 pts)

In this problem we explore dynamic programming using the so-called *maximum segment sum* problem, implemented in C0. We are given an integer array $A[0..n)$ and we have to find the segment $A[i..j)$ such that the sum of the integers $A[i] + A[i+1] + \cdots + A[j-1]$ is maximal. We simplify this slightly by only requesting the maximal sum itself, but not the bounds of the segment. The problem is not trivial because we can negative integers in the array.

Task 1 (10 pts). The following is a specification of the maximum segment sum problem, omitting pre- and post-conditions and loop invariants. Fill these in. We consider an empty segment as having sum 0.

```
int max_seg_sum(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \result >= 0;
{
  int i; int j; int k;
  int max = 0;
  for (j = 0; j <= n; j++)
    //@loop_invariant 0 <= j && j <= n+1;
    // max == max_seq_sum(A, j)
    { for (i = 0; i < j; i++)
        //@loop_invariant 0 <= i && i <= j;
        { int sum = 0;
          for (k = i; k < j; k++)
            //@loop_invariant i <= k && k <= j;
            sum += A[k];
          if (sum > max) max = sum;
        }
   }
  return max;
}
```

Task 2 (5 pts). What is the asymptotic complexity of `max_seg_sum` as a function of the n, the length of the array A?

$O(n^3)$ because there are three nested loops of order n, where the innermost loop body takes constant time.

Alternatively, we can say that the code checks every segment. There are n left endpoints, on the average $n/2$ right endpoints for each, and, on the average, $n/2$ elements in each segment to sum. So we obtain $O(n^3)$ in total.

We use dynamic programming by introducing a new array M, where $M[j]$ holds the maximum sum for any segment $A[i..j)$ ending in $j-1$.

Task 3 (10 pts). If $M[j]$ holds the maximum sum for any segment $A[i..j)$ ending in $j-1$ how can we efficiently compute the maximum sum for any segment $A[i..j+1)$ ending in j? **[Hint:** you do not need a loop or recursive function.]

If $M[j]$ is positive, then it pays for $A[j]$ to include the maximal sum segment ending in $j-1$. If $M[j]$ is negative, then the singleton $A[j]$ is the maximal segment with sum $A[j]$ (including any preceding nodes would only make it smaller).

Task 4 (10 pts). Based on your observation in Task 3, complete the following program.

```
int dp_max_seg_sum(int[] A, int n) {
  int j;
  int max = 0;
  int[] M = alloc_array(int, n+1);
  M[0] = 0;
  for (j = 0; j < n; j++)
    { M[j+1] = A[j] + ((M[j] > 0) ? M[j] : 0);
      if (M[j+1] > max) max = M[j+1];
    }
}
```

Task 5 (5 pts). What is the asymptotic complexity of your dynamic programming solution to the maximum segment sum problem?

$O(n)$, because we traverse the array only once and the operations in the loop body take constant time.