Final Solutions

15-122 Principles of Imperative Computation

Friday 26th June, 2015

Name: Harry Bovik

Andrew ID: bovik

Recitation Section: S

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 180 minutes to complete the exam.
- There are 7 problems on 20 pages (including 0 blank pages at the end).
- Use a **dark** pen or pencil to write your answers.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.

	Max	Score
C0 and C	30	
Requiring Safety	30	
High-Speed Data Structures	50	
Generic Exam Question	40	
Graph Expansion	40	
Spanning Trees	45	
Minimum Spanning Trees	10	
Total:	245	

1 C0 and C (30 points)

For the C code in this question, you can assume standard implementation-defined behavior. In every case, assume we are calling `gcc` with the command

```
gcc -Wall -Wextra -Werror -Wshadow -std=c99 -pedantic example.c
```

Notice that `-fwrapv` was *not* one of the arguments. Assume the following declarations, where "`...`" refers to an arbitrary unknown value.

In C0

```
int INT_MIN = 0x80000000;

int x = ...;
int y = ...;
int z = ...;

int[] A = alloc_array(int, 5);
int* p = alloc(int);
```

In C

```
#include <limits.h>

unsigned int x = ...;
int y = ...;
int z = ...;

int *A = xcalloc(5, sizeof(int));
int *p = xmalloc(sizeof(int));
```

For the following expressions, indicate all possible outcomes among `true`, `false`, *error* (for program-stopping runtime errors), and *undefined*. If the behavior may be *undefined* then all other outcomes are automatically possible, so you don't need to list them explicitly. Assume that all initializations succeed without errors. To get you started we have already filled in first row for you.

Expression	**In C0**	**In C**
`x * 2 > x`	`false`, `true`	`false`, `true`
`x * 4 == x << 4`	true, false	true, false
`x / 8 == x >> 3`	true, false	true
`y == INT_MIN || y > y - 1`	true	true
`z == 0 || (y/z)*z + y%z == y`	true, error	undefined
`z == 0 || (122/z)*z + 122%z == 122`	true	true
`y + z >= 0 || y + z < 0`	true	undefined
`y <= 0 || z <= 0 || (y+z)/y <= z+y`	true, false	undefined
`*p == A[0]`	true	undefined
`A[0] == A[A[A[0]]]`	true	true
`A[5] == A[A[A[5]]]`	error	undefined

2 Requiring Safety (30 points)

For the functions in this question, write (additional) preconditions that are sufficient to ensure that there will be no undefined behavior and no memory leaks. Your preconditions allow the function to run *whenever* undefined behavior and memory leaks would not occur. Make sure it's not possible to cause undefined behavior in the precondition itself!

If it's not possible to ensure that a function is free of undefined behavior and memory leaks, then you can write the precondition `false`, which indicates that the function cannot be run safely. If no preconditions are needed, you can write the precondition `true`.

Do not assume *any* implementation-defined behaviors except that a byte is 8 bits. Your preconditions should make your functions safe for any implementation-defined behaviors.

5pts **Task 1**

```
unsigned long task1(unsigned long x, unsigned long y) {

    REQUIRES(             y < sizeof(unsigned long)*8              );

    return x << y;
}
```

5pts **Task 2**

```
int task2(int x, int y, int z) {
    REQUIRES(x >= 0 && y >= 0);

    REQUIRES(         (y == 0 || x <= INT_MAX/y) && z != 0          );
    return (x * y) / z;
}
```

5pts **Task 3**

```
int task3(size_t n) {

    REQUIRES(                        n == 0                          );

    int x = 0;
    int A[15];
    for (size_t i = 0; i < n; i++) {
        x += A[i];
    }
    return x;
}
```

5pts **Task 4**

```
char *task4(size_t n, size_t m) {

    REQUIRES(            m <= n / sizeof(unsigned int)            );

    REQUIRES(             m <= sizeof(unsigned int)*8             );

    unsigned int *A = xmalloc(n);
    for (unsigned int i = 0; i < m; i++) {
        A[i] = i << i;
    }
    return A;
}
```

5pts **Task 5** For this task, you should assume that casting between signed and unsigned types of the same size works the same way it does in `gcc`.

```
int task5(signed char n, int m) {

    REQUIRES(                  n >= 0 || m >= 0                   );

    unsigned int x = (unsigned int)(unsigned char)n;
    unsigned int y = (unsigned int)(signed int)n;

    if (x != y) return INT_MIN + m;
    return m;
}
```

5pts **Task 6**

```
void task6(int n) {

    REQUIRES(                        n == 1                        );

    int *A = xcalloc(5, sizeof(int));
    for (int i = 0; i < n; i++) {
        free(&A[i]);
    }
}
```

3 High-Speed Data Structures (50 points)

40pts **Task 1** Give best and worst-case running time bounds for the following operations. Some of the descriptions state that multiple operations are happening: give the cost of doing the *entire sequence* of operations, not the cost of doing a single operation within the sequence.

Always give the simplest, tightest big-O bound.

	Best-case	**Worst-case**
Adding n elements to an initially empty queue (implemented as a linked list).	$O(n)$	$O(n)$
Adding n elements to an resizing unbounded array that initially has size 0 and limit 4.	$O(n)$	$O(n)$
Adding n elements to a resizing unbounded array that initially has size 0 and limit $> n$, using **merge-sort** to re-sort the array after **every single addition**.	$O(n^2 \log n)$	$O(n^2 \log n)$
Adding n elements to a resizing unbounded array that initially has size 0 and limit $> n$, using **quicksort** to re-sort the array **only once, after all** n **additions**.	$O(n \log n)$	$O(n^2)$
Adding a *single* element to a heap data structure that has n elements in it already.	$O(1)$	$O(\log n)$
Adding n elements, which are all or almost all distinct, to an initially empty binary search tree that does not do any re-balancing.	$O(n \log n)$	$O(n^2)$
Adding n elements, which are all or almost all distinct, to an initially empty AVL tree.	$O(n \log n)$	$O(n \log n)$
Inserting n elements, which are all or almost all distinct, to an initially empty non-resizing, separate-chaining hash table with table size m.	$O(n^2/m)$	$O(n^2)$
Looking up a single element in a non-resizing, separate-chaining hash table that already contains n distinct elements and has a table size m.	$O(1)$	$O(n)$
Adding every possible edge to an initially-empty undirected graph with n vertices. Assume an adjacency matrix representation.	$O(n^2)$	$O(n^2)$

10pts **Task 2** Grace is implementing a program for the popular word game Scrabble, and she is considering the use of a hash table (using separate chaining) or a balanced binary search tree to store the dictionary of legal words with their definitions. In this case, Grace is not using a separate interface for the data structure. Instead the data structure is being integrated into the full program. (Maybe not such a good idea, but Grace has been programming for many years and wrote very careful data structure invariants.) For the hash table, she would use a hash function that adds up the ASCII values of all of the letters in the word. The binary search tree is ordered by the usual ASCIIbetical ordering.

Which data structure, hash table or binary search tree, would allow her to more easily find all words that start with the letter sequence UNI? Explain your choice in one sentence.

All the words that start with `UNI` are between `UNI` and `UNJ` in the dictionary, so a binary search tree is going to be the better way of grouping and finding all those words.

Which data structure, hash table or binary search tree, would allow her to more easily find all words that can be formed using the each of letters AERST once? Explain your choice in one sentence.

For the given hash function, a hash table would put all the words that contain `AERST` once in the same chain, making then easier to find.

4 Generic Exam Question (40 points)

This question involves a slight variant of the C implementation of generic queues: we've removed `queue_size`, `queue_reverse`, and `queue_peek`.

```
typedef bool check_property_fn(void* x);
typedef void* iterate_fn(void* accum, void* x);
typedef void elem_free_fn(void *x);

queue_t queue_new();
  /*@ensures \result != NULL; @*/

bool queue_empty(queue_t Q);
  /*@requires Q != NULL; @*/

void enq(queue_t Q, void *x);
  /*@requires Q != NULL; @*/

void *deq(queue_t Q);
  /*@requires Q != NULL && !queue_empty(Q) > 0; @*/

bool queue_all(queue_t Q, check_property_fn *P);
  /*@requires Q != NULL && P != NULL; @*/

void* queue_iterate(queue_t Q, void *base, iterate_fn *F);
  /*@requires Q != NULL && F != NULL; @*/

void queue_free(queue_t Q, elem_free_fn *F)
  /*@requires Q != NULL; @*/ ;
```

10pts **Task 1** Write a C function named `nestring` that matches the type `check_property_fn`. Your function should assume the **void** pointers it is given are either `NULL` or are valid C style strings (type `char*`).

Your function should be written so that `queue_all(Q,&nestring)` will return `false` if any element of the queue is `NULL` or if any element of the queue is a zero-length C-style string. (In other words, `queue_all(Q,&nestring)` should check that everything in the queue is a non-`NULL`, non-empty string.)

Make all casts to and from `void*` explicit.

```
bool nestring(void *x) {
   return x != NULL && *(char*)x != '\0';
}
```

15pts **Task 2** Recall that if the queue `Q` contains the four elements `e1`, `e2`, `e3`, and `e4`, then calling `queue_iterate(Q,base,&f)` will compute

`f(f(f(f(base,e1),e2),e3),e4)`

whereas if `Q` is empty `queue_iterate(Q,base,&f)` will just return `base`.

Respecting the interface of queues above, write `queue_size` that takes a queue and returns an integer representing the number of elements in the queue. Your solution must use `queue_iterate` (not `deq` or `enq`), and you'll need to define a helper function.

For full credit, don't heap-allocate any memory with (x)`malloc` or (x)`calloc`.

Make all casts to and from **`void`**`*` explicit.

```
void *counter(void *accum, void *x) {
  REQUIRES(accum != NULL);
  *(int*)accum += 1;
  return accum;
}

int queue_size(queue_t Q) {
  REQUIRES(Q != NULL);
  int i = 0;
  queue_iterate(Q, (void*)&i, &counter);
  return i;
}
```

(Also, don't cast between integer types like **`int`** and pointer types like **`void`**`*`. This isn't something we even mentioned the possibility of during class, so you don't know what this means you probably are not going to do it.)

15pts **Task 3** Here are two different *implementations* of `queue_all`:

```
/* Implementation A */
bool queue_all(queue *Q; check_property_fn *P) {
  REQUIRES(is_queue(Q) && P != NULL);
  for (list *L = Q->front; L != NULL; L = L->next)
    if (!(*P)(L->data)) return false;
  return true;
}
/* Implementation B */
bool queue_all(queue *Q; check_property_fn *P) {
  REQUIRES(is_queue(Q) && P != NULL);
  bool res = true;
  for (list *L = Q->front; L != NULL; L = L->next)
    res = (*P)(L->data) && res;
  return res;
}
```

Write a `main` function (and any necessary helper functions) that respects the queue interface but that returns 0 if the queue is implemented with implementation A and that returns 1 if the queue is implemented with implementation B. You can allocate memory freely and ignore memory leaks.

High level explanation: the **`bool`** returned by both impelementations is *always* the same, but `A` stops as soon as we get a `false` and `B` runs the function pointer on every data element.

So the way to distinguish the two implementations is to have the property checker modify the queue, and see if the whole queue gets modified, or just the whole queue up until the first element that causes the property checker to return `false`.

```
bool tester(void *x) {
  REQUIRES(x != NULL);
  *(int*)x += 10;
  return false;
}

int main() {
  queue_t Q = queue_new();
  enq(Q, xcalloc(1, sizeof(int)));
  enq(Q, xcalloc(1, sizeof(int)));
  queue_all(Q, &tester);
  deq(Q);
  if (*(int*)deq(Q) == 10) return 1;
  return 0;
}
```

Hint: it may be necessary for your the function you pass `queue_all` to modify memory.

If you're stumped: for partial (half) credit, you can explain the difference between the two functions without writing a test case that differentiates them.

5 Graph Expansion (40 points)

In this question, we use the following modified interface of undirected graphs, which incorporates ideas from unbounded arrays.

```
typedef unsigned int vertex;

unsigned int graph_size(graph_t G);
  //@requires G != NULL;

graph_t graph_new();
  //@ensures \result != NULL;
  //@ensures graph_size(\result) == 0;

void graph_grow(graph_t G);
  //@requires G != NULL;

bool graph_hasedge(graph_t G, vertex v, vertex w);
  //@requires G != NULL;
  //@requires v < graph_size(G) && w < graph_size(G);

void graph_addedge(graph_t G, vertex v, vertex w);
  //@requires G != NULL;
  //@requires v < graph_size(G) && w < graph_size(G);
  //@requires v != w && !graph_hasedge(G, v, w);

void graph_free(graph_t G);
  //@requires G != NULL;
```

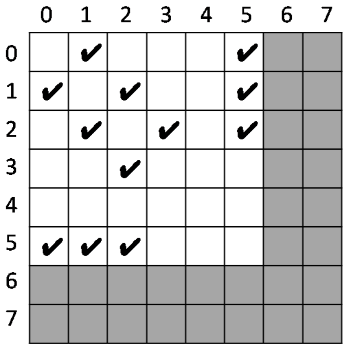
A new graph is always created with 0 vertices (meaning `graph_size` is initially 0), and `graph_grow` increments `graph_size` by one:

```
graph_t G = graph_new();
graph_grow(G); // Adds the vertex 0
graph_grow(G); // Adds the vertex 1
graph_addedge(G, 0, 1);
graph_grow(G); // Adds the vertex 2
graph_grow(G); // Adds the vertex 3
graph_addedge(G, 2, 3);
graph_addedge(G, 0, 3);
assert(graph_size(G) == 4); // 4 vertices, numbered 0, 1, 2, and 3
```

6pts **Task 1** We create a large graph, and in the process call `graph_new` once, call `graph_grow` k times, and call `graph_addedge` $3k$ times. Is the graph sparse or dense?

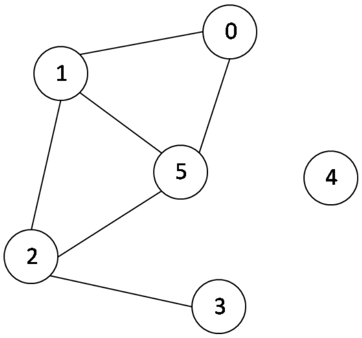
Sparse: the number of edges would need to be proportional to k^2 for it to be dense.

We can implement this interface using adjacency matrices: we'll hold the adjacency matrix in a 1-D array, using the same layout we used for the images assignment, where the element in row `i` and column `j` is stored at index `i*G->limit + j` in the array.



	0	1	2	3	4	5	6	7
0		✔				✔	▨	▨
1	✔		✔			✔	▨	▨
2		✔		✔		✔	▨	▨
3			✔				▨	▨
4							▨	▨
5	✔	✔	✔				▨	▨
6	▨	▨	▨	▨	▨	▨	▨	▨
7	▨	▨	▨	▨	▨	▨	▨	▨

size = 6, limit = 8



```
typedef struct graph_header graph;
struct graph_header {
    size_t size;
    size_t limit;
    bool *adj;
};
```

A well-formed undirected graph (the `is_graph(G)` data structure invariant) is a non-`NULL` struct with `size` strictly less than `limit`. The array of boolean values `adj` must be a non-`NULL` array `adj`, and all the unused cells in the array (the grayed-out portion in the illustration above) must be set to `false`. Finally, the length of `G->adj` must be equal to `limit*limit`, but in C, it is impossible to actually check this last condition.

Because the graphs are undirected, we also require that `G->adj[i*G->limit + j]` is equal to `G->adj[j*G->limit + i]` for every `i` and `j` in the range `[0..G->limit)`.

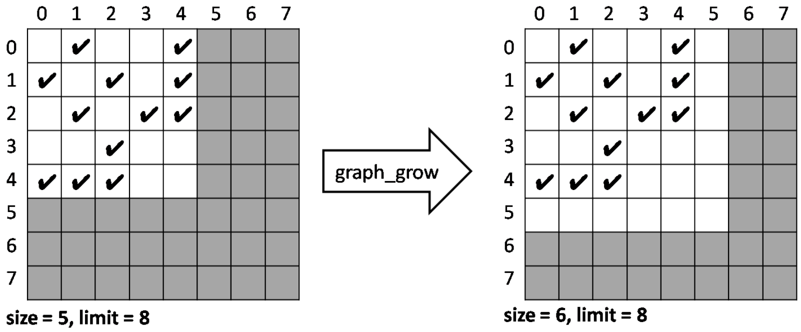
6pts **Task 2** Implement `graph_addege` for the data structure described above.

```
void graph_addedge(graph *G, vertex v, vertex w) {
    REQUIRES(is_graph(G) && v < G->size && w < G->size);
    REQUIRES(v != w && !graph_hasedge(G, v, w));

    G->adj[v*G->limit + w] = true;
    G->adj[w*G->limit + v] = true;

    ENSURES(is_graph(G));
}
```

When we call `graph_grow`, we increase the `size`: if it's still less than `limit`, we're done.



size = 5, limit = 8

size = 6, limit = 8

If `size` is equal to `limit`, we double `limit`, which quadruples the length of the array.

19pts **Task 3** Implement `graph_grow` according to the description above. Avoid memory leaks.

```
void graph_grow(graph *G) {
    REQUIRES(is_graph(G));

    G->size += 1;
    if (G->size == G->limit) {
        assert(G->limit*G->limit < SIZE_T_MAX / 4); // Not required
        bool *OLD = G->adj;
        bool *NEW = xcalloc(G->limit*G->limit*4, sizeof(bool));
        for (size_t row = 0; row < G->limit; row++) {
           for (size_t col = 0; col < G->limit; col++) {
              NEW[row*G->limit*2 + col] = OLD[row*G->limit + col];
           }
        }
        G->limit *= 2;
        G->adj = NEW;
        free(OLD);
    }

    ENSURES(is_graph(G));
}
```

9pts **Task 4** In most cases, `graph_grow` requires 0 array writes. In the worst case, we will have `size` $=$ `limit` $= n$, and the running time of `graph_grow` in terms of n will be in

$O(\underline{\quad n^2 \quad})$

Assuming the array has resized before, that expensive operation was necessarily preceded by exactly…

$\underline{\quad n/2 - 1 \quad}$

…cheap calls to `graph_grow`. The amortized cost of `graph_grow` is therefore in

$O(\underline{\quad n \quad})$

6 Spanning Trees (45 points)

In this problem we consider a version of Prim's algorithm for computing a minimum weight spanning tree for connected graphs, the way we presented it in class. First, a summary of the algorithm.

We start with a partial spanning tree T consisting of a single arbitrary vertex v_0 and empty set of edges A. We also initialize a candidate set H to contain all the edges incident to v_0.

We now repeatedly remove a minimum weight edge e from H. If it connects two vertices already in T we just drop it. If it connects one vertex in T with one not in T (call it u), we add u to T and e to A. We then add all the edges from u to vertices not already in T to the candidate set H.

We stop when we have constructed a spanning tree.

10pts **Task 1** State the *invariants* of this algorithms in terms of the sets V (vertices), E (edges), T (vertices of partial spanning tree), A (edges added to make up a partial spanning tree), and H (candidate edges). Your invariants should be strong enough to prove that we have a spanning tree when the algorithm terminates. You may state more than two additional invariants.

1. T is a subset of V, and A is a subset of E.
2. Vertices T with edges A form a minimum spanning tree for the subgraph with vertices from T and edges from E.
3. Every edge from any vertex in T to a vertex not in T is contained in H.

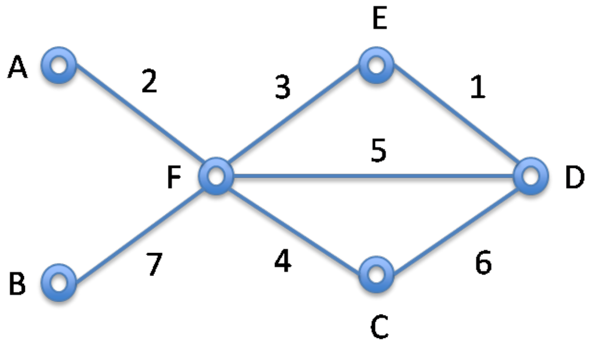
4pts **Task 2** Let D represent the set of edges that are dropped in this algorithm as the minimum spanning tree is constructed. State the *variant* of this algorithm, that is, the quantity that either strictly increases and is bounded above or strictly decreases and is bounded below. Also state the bound. For partial credit, you may explain instead why the algorithm terminates in one or two sentences.

The number of edges in A plus the number of dropped edges D increases in each step and is bounded by the number of edges in E.

Because of the operations we have to perform on H (adding an edge, removing the minimum weight edge), it is both efficient and convenient to maintain it as a min-heap.

15pts **Task 3** For the following graph fill in the table below to show how the sets T, A, and H change with each step. Draw the min-heaps as trees. The edge weights are unique so we can, for example, write 2 to denote the edge AF of weight 2. Write "Add k" to add an edge of weight k to the tree and "Drop k" if an edge of weight k is deleted from the heap but not added to the tree. When multiple edges have to be added to the heap add them in order of increasing weight. We have started with vertex A.

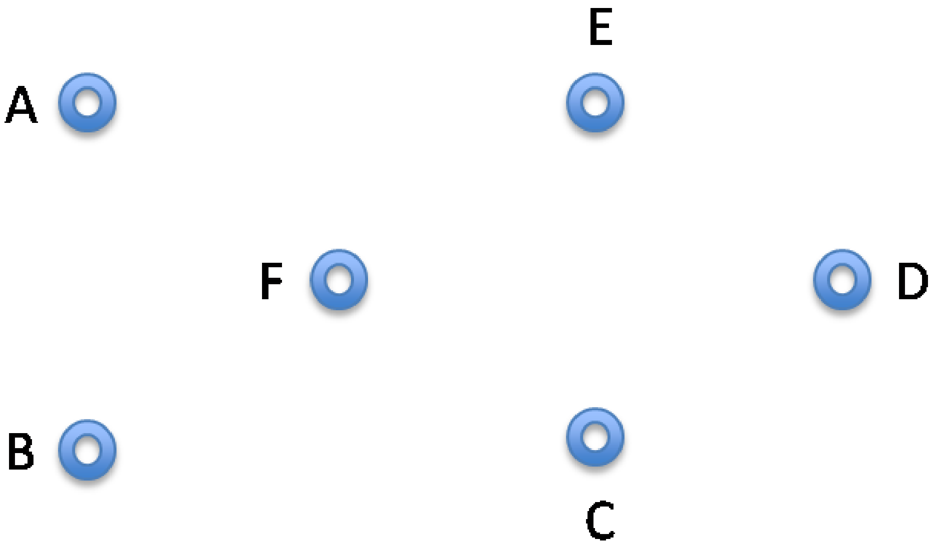
The table continues on the next page, where there is space for scratch work and an empty graph as a sketching aid. Note that we will not grade your graph drawings.



Operation	**T** vertices in tree	**A** edges in tree	**H** candidate heap
Init	A	$(none)$	2
Add 2	A, F	2	3 4 5 7
Add 3	A,F,E	2,3	1 4 5 7
Add 1	A,F,E,D	2,3,1	4 6 5 7
Add 4	A,F,E,D,C	2,3,1,4	5 6 7
Drop 5	A,F,E,D,C	2,3,1,4	6 7

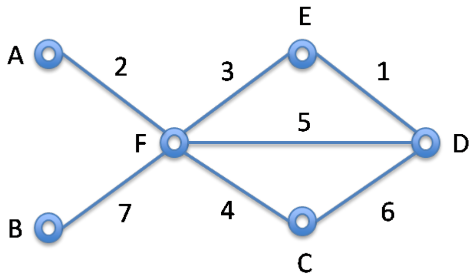
← start here

Operation	**T** vertices in tree	**A** edges in tree	**H** candidate heap
Drop 6	A,F,E,D,C	2,3,1,4	7
Add 7	A,F,E,D,C,B	2,3,1,4,7	



10pts **Task 4** Next we are interested in applying Kruskal's algorithm to the same graph. Since this was discussed in lecture in more detail, we do not review the algorithm here. Please fill in the table below with the union-find data structure where each array element stores either (a) the array index of its parent, or (b) for a root, the negated value of the maximal length of a path to this root (counting the number of vertices). When the update of the union-find data structure is not uniquely defined by the algorithm, point the vertex with the higher array index to the lower one. Do not perform path compression!

We have filled in the beginning for you and already listed the edges in the order they are considered. Please write "Add" if the edge is added to the spanning tree or "Drop" if it is not added.



Edge considered	Action	union-find data structure					
		A	*B*	*C*	*D*	*E*	*F*
	initialize	-1	-1	-1	-1	-1	-1
1	*Add*						
2							
3							
4							
5							
6							
7							

← start here

```
           A  B  C  D  E  F
           -1 -1 -1 -1 -1 -1
1 Add               -2  3
2 Add      -2              0
3 Add      -3        0
4 Add             0
5 Drop
6 Drop
7 Add      -3  0  0  0  3  0
```

6pts **Task 5** What is the worst-case asymptotic time complexity of this exam's version of Prim's algorithm in term of $v = |V|$ and $e = |E|$? Use big O notation.

Answer: $O(e \log(e))$

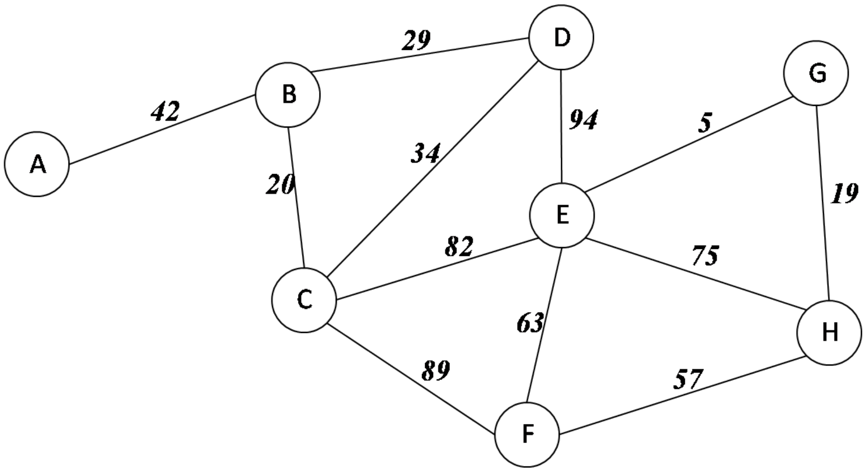
What is the worst-case asymptotic time complexity of this exam's version of **Kruskal**'s algorithm in term of $v = |V|$ and $e = |E|$? Use big O notation.

Answer: $O(e \log(e))$

7 Minimum Spanning Trees (10 points)

In this question, we look at Kruskal's algorithm for computing the minimum spanning tree (MST) of a graph. Kruskal's algorithm requires a sorted sequence of edges by weight.

We can apply Kruskal's algorithm to find a minimum spanning tree for the graph shown below:



In the table below, 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 *any edges that would not even be considered by Kruskal's algorithm.* We have filled in the first two edges for you, and listed all the weights in ascending order

Weight	Edge Considered	Added to MST?
5	(E,G)	**Yes**
19	(H,G)	**Yes**
20	(B,C)	*Yes*
29	(B,D)	*Yes*
34	(C,D)	*No*
42	(A,B)	*Yes*
57	(F,H)	*Yes*
63	(E,F)	*No*
75	(E,H)	*No*
82	(C,E)	*Yes*
89		
94		