Answer the questions in the spaces provided. If you run out of room for an answer, continue on the back of the page.

Question	Points	Score
1	15	
2	20	
3	34	
Total:	69	

Name: ______________________________

Section: ______________________________

1. (15 points) **Fast exponentiation**: Repeated exponentiation of numbers is frequently required in cryptography, where large numbers are used to ensure security. But when working with really large numbers, runtime can start to be a concern. The need for quick ways to compute large exponents of numbers were thus required, which emerged in the form of *fast exponentiation algorithms*.

   One such algorithm takes advantage of the fact that $a^{2^n} = a^{2^{2^{n/2}}}$ to reduce the number of times one needs to multiply a by itself to obtain a^{2^n}. The process for this algorithm is simple, for the computation of m^e:

   - Obtain a shortest binary representation $e_2 = b_k b_{k-1} \ldots b_0$ for e, where $b_i \in \{0, 1\}$ for all $i \in [0..k]$ and $b_k = 1$.
   - Compute m^{2^k} by repeated squaring.
   - Compute $m^e = \prod_{i:b_i=1} m^{2^i}$.

   Implement this algorithm in Python.

2. (20 points) **LISP, JSON, and arithmetic**: Suppose you want to parse a mathematical expression in a LISP-like syntax from a JSON string. An example JSON assignment to JavaScript variable `x` might be as follows:

```
x = { expr : [
    delimiter : "(",
    operator : "+"
    delimiter : "(",
    operator : "-",
    value : "3",
    value : "4",
    delimiter : ")",
    delimiter : "(",
    operator : "*",
    value : "7",
    value : "5",
    delimiter : ")",
    delimiter : ")",
] }
```

This would turn into the following LISP syntax, once parsed correctly:

```
(+ (- 3 4) (* 7 5))
```

This will evaluate to `(+ (-1) (35))` which is `34`.

Your task is to write code that, given some Python variable `x` which holds some string representing a JSON of the above form (containing `delimiters`, `operators`, and `values`), prints the result of computing the expression represented therein. In the above example, given `x = '{ expr : [...] }'`, your code should return `34`.

The first line of code is written for you. The `'...'` for `x` is meant to represent a general string of the form described above. Your code should end in a `print` statement.

You **may not use** Python's `JSON` module. You should parse the string manually.

(*Hint:* You may want to use the *stack* data structure, which functions exactly as it sounds. You can implement it with a Python `list`, using the `append` and `pop` operators.)

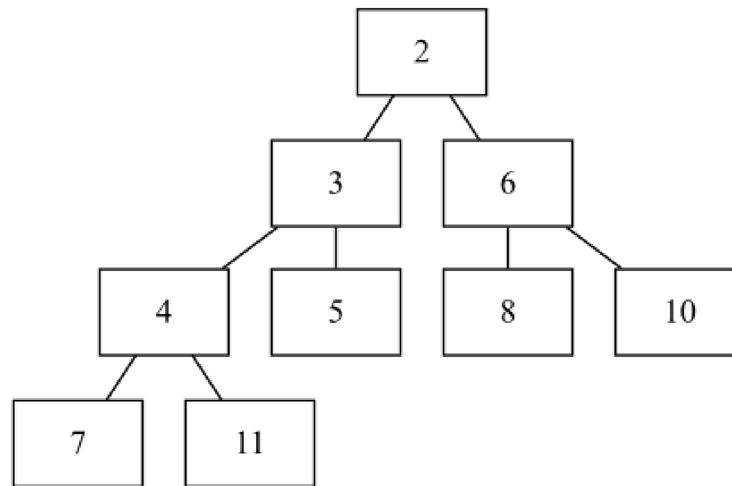
```
x = '...'
```

3. (34 points total) **Heaps**

A **heap** is a data structure frequently used in implementations of *priority queues*, as it has the convenient property of always being partially ordered[1]. In this question we will consider **binary heaps**, which are a kind of *binary tree*. Heaps obey the *heap property*, which is as follows:

> If a heap is a *min*-heap, then for all child nodes c of some parent node p, the value of c is *greater than* that of p. If a heap is a *max*-heap, then for all child nodes c of some parent node p, the value of c is *less than* that of p. This ensures that the root node is the minimum or maximum respectively of the set containing itself and both of its children.

An example binary min-heap might be as follows:



If we need to access the lowest element in the above heap, we can simply pop off the top element. Convenient! But how do we create a heap? It's not so great if we need to put everything in manually.

That's why heaps come with a set of common operations. These include:

- `peek`: Returns the root element of the heap.
- `pop`: Removes and then returns the root element of the heap.
- `create-heap`: Creates an empty heap.
- `heapify`: Turns a binary tree into a heap.
- `size`: Returns the size of the heap.

Rather than working with a set of nodes and whatnot, we can also use an array data structure, in which the children of parent node at index p are at indices $c_1 = 2p + 1$ and $c_2 = 2p + 2$ respectively. The above heap would be represented by the following array:

```
[2, 3, 6, 4, 5, 8, 10, 7, 11]
```

For each of the first five of the following parts (part (a) through part (e)) your task will be to implement one of the above operations. You should use the array data structure for the heap as defined above, and should consider an array to be a Python `list`.

[1] This is technically incorrect if we use the mathematical definition of a (weak) partial order as a relation which is reflexive, antisymmetric, and transitive; heaps are simply "ordered to some degree but not completely".

(a) (2 points) Implement `peek`.

(b) (4 points) Implement `size`.

(c) (2 points) Implement `create-heap`.

(d) (6 points) Implement `heapify`.

(e) (3 points) Implement `pop`.

(f) (4 points) List the Big-O complexities of each of the above operations.

(g) (12 points) **Heapsort**: You may have noticed that the array given for the example above is very close to completely sorted. In fact, heaps *can* be used to sort data! Implement an algorithm, *heapsort*, that uses a heap to sort an array of integers from smallest to largest. Your algorithm should have the following performance metrics:

- Worst-case performance: $O(n \log n)$
- Best-case performance[2]: $O(n \log n)$
- Average case performance (this follows from the first two): $O(n \log n)$

(h) (1 point) Why might someone choose to use heapsort over another sorting algorithm, such as quicksort or mergesort?

[2] Assume that the keys for the heap are all unique.