# Trie Data Structure

Define a class `Trie` that can be used for fast lookups of strings. Such classes are frequently used in many natural language processing applications.

(Writing such a class is also a common interview question; you need to be able to do it in real time.)

Have your class update a global variable `nops` for each node traversal during `add`, `lookup`, and `remove` operations.

In [81]:
```
class Trie:
    def __init__(self):
        self.root = dict()
        pass
    def add(self,s,value):
        """Add the string `s` to the `Trie` and
        map it to the given value."""
        global nops
        current_dict = self.root
        for letter in s:
            nops += 1
            current_dict = current_dict.setdefault(letter, {})
        current_dict = current_dict.setdefault('value', value)
        pass
    def lookup(self,s,default=None):
        """Look up the value corresponding to the
        string `s`."""
        global nops
        current_dict = self.root
        for letter in s:
            nops += 1
            if letter in current_dict:
                current_dict = current_dict[letter]
            else:
                if default != None:
                    return default
                else:
                    return False
        if 'value' in current_dict:
            return current_dict['value']
        else:
            if default != None:
                return default
            else:
                return False

    def remove(self,s):
        """Remove the string s from the Trie.
        Returns True if the string was a member."""
        current_dict = self.root
        global nops
        for letter in s[:-1]:
            nops += 1
            if letter in current_dict:
                current_dict = current_dict[letter]
            else:
                return False
        if s[-1] in current_dict:
            if len(current_dict[s[-1]]) > 1:
                del current_dict[s[-1]]['value']
            else:
                del current_dict[s[-1]]
        else:
            return False
        return True
    def prefix(self,s):
        """Check whether the string `s` is a prefix
        of some member."""
```

Typesetting math: 100%

In [82]:
```
nops = 0
trie = Trie()
trie.add("hello",1)
trie.add("world",2)
trie.add("worlds",2)
for entry in trie:
    print entry
```

```
('world', 2)
('worlds', 2)
('hello', 1)
```

# Unit Tests

Write some unit tests demonstrating that your class works as intended. The next cell gives some examples, but you need to write additional tests for other methods and common sequences of operations.

In [49]:
```
nops = 0
trie = Trie()
trie.add("hello",1)
trie.add("world",2)
trie.add("worlds",2)
assert not trie.lookup("street")
assert not trie.lookup("house")
assert trie.lookup("hello")
assert trie.lookup("world")
assert trie.prefix("wor")
assert not trie.prefix("war")
assert not trie.lookup("worlds") == 3
assert trie.lookup("worlds") == 2
assert trie.lookup("worlds",4) == 2
assert trie.lookup("war",4) == 4
assert not trie.lookup("war",4) == 3
```

In [4]:
```
trie = Trie()
nops = 0
trie.add("hello",1)
assert nops>0
```

# Performance Measurements

Next, let's measure how `Trie` performance scales on real data.

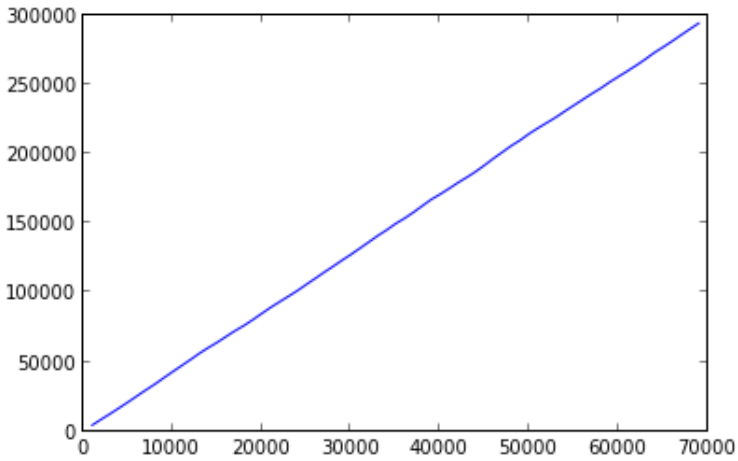
In [5]:
```
import re
words = re.findall(r'\w+',open("tomsawyer.txt").read())
words = [w.lower() for  w in words]
print len(words),words[:10]
```

```
74354 ['the', 'adventures', 'of', 'tom', 'sawyer', 'mark', 'twain',
'harper', 'and', 'brothers']
```

In [6]:
```
counts = []
for n in range(1000,70000,1000):
    trie = Trie()
    nops = 0
    for i,w in enumerate(words[:n]):
        trie.add(w,i)
    counts.append((n,nops))
```

In [7]:
```
counts = array(counts)
plot(counts[:,0],counts[:,1])
```

Out[7]: [<matplotlib.lines.Line2D at 0x3b418d0>]




In [7]:

Typesetting math: 100%