Day 12
Date Tues 2016.3.1
Topic Python
Textbook for today Chapter 16
Textbook-assign Chapter 16

Quiz 6 – on Thursday will cover time complexity and searching/sorting

Project 2 will be due Wednesday at 23:30. If you don’t come to lab because you’re still working on your project, you get 0%. (If you have to miss due to illness, then you must bring an official excuse, e.g. from a doctor, nurse, or parent with telephone numbers.)

Midterm March 10: more details soon
But everything we cover up through and including March 1.

A. Postfix, Prefix and Infix

Great use of a stack.

Where do the operators appear in relation to their operands?

pre—in front of
in—between
post—after

\[ + \overbrace{5 \ 7} \]
\[ \overbrace{5 \ + \ 7} \]
\[ 5 \ 7 \ + \]

Postfix used to be called RPN (Reverse Polish Notation) and was used by HP calculators in the 60s and 70s when parsers were more primitive and paren were’t used.

Don’t need paren for pre and postfix! Also operator precedence is automatically taken care of.

Assume usual precedence: \( [\sqrt{\cdot}] \ [\pm] \ [\times \div \%] \ [^\wedge] \) [unary _minus _!]

\( (This \ is \ a \ concise \ way \ of \ showing \ the \ levels. \ Everything \ in \ square \ brackets \ has \ the \ same \ level. \ Lowest \ precedence \ is \ on \ the \ left.) \)

The _arity_ of an operator is how many operands it expects. _arity=1:_ \( \sqrt{\cdot} \) Unary _minus_ !

With postfix, push every number. If the next token is an operator, pop the appropriate number off the top and evaluate and push result back on the stack.

With prefix, it is trickier. Need to keep a stack of operators. Or you could think of rewriting the token string. Scan from left to right. If you find an operator followed the right number of values according to its _arity_, then evaluate and replace those tokens with one new number. Then start again at the left until there is just one number on the token stream.

prefix and postfix are not mirror images of each other. DO SOME EXERCISES.
B. Searching

Probabilistic searching. - what are the trade-offs?

Pros: in a very large search space spread out over many machines, you wouldn't have many searches hitting the same hardware at the same time
Cons: might take a long time

Example of probabilistic search: Testing primality of an integer:

```
def isprime(n):
    for i in range(2,n):
        if n%i == 0: return False
    return True
```

This always starts at 2 and divides on up.

But to find out if an extremely long number is prime, it would take a huge amount of time. You could use prob. searching to narrow down the possibility. If you divided by 40 random primes and didn't get 0, the chance that it is prime goes up a lot.

Do binary search (non-recursive) by tracing.

Read the binary search code in recursive format.

C. Complexity measures

There are several different complexity measures:

For linear searching:

| best case | 1 |
| average case | $\frac{N}{2}$ |
| worst case | $N$ |

There are other things that we measure:

| Time | How many seconds it will take before you know the answer. |
| Memory | How many bytes of RAM it will require when it works. |
| Complexity | How difficult the algorithm is for a human to understand. |
| Hardware | How many cores the algorithm requires (parallel computing engines.) |
D. Sorting

Sometimes called alphabetizing. Based on the **collating sequence**, which is a pairing of numbers to symbols. ASCII, Latin-1 alphabet, later Unicode

http://cs.stanford.edu/people/miles/iso8859.html

Unicode.org

MANY different sorting algorithms.

For animations and pictures of sorting:

http://www.sorting-algorithms.com/

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://www.youtube.com/watch?v=kPRA0W1kECg -- has sound!

Many sorting algorithms have a time complexity of

<table>
<thead>
<tr>
<th>T(N)</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>N^2</td>
<td>insertion sort, bubble sort, selection sort, quicksort, many others</td>
</tr>
<tr>
<td>N x log_2(N)</td>
<td>merge sort, radix sort, heapsort, and some others</td>
</tr>
</tbody>
</table>

```python
def insertSorted(target, sortedlist):
    for k in range(len(sortedlist)):
        if target < sortedlist[k]:
            sortedlist.insert(k, target)
            break
        sortedlist.append(target)
```

**Slow Sorting**

Algorithm:

Given a list with N numbers, call it the start list
Make a new empty List, call it the result list
For each number in the start list
run the insertSorted function above to insert the number into the result list

The main for loop runs N times because there are N numbers in the start list.

The insertSorted function has a loop that runs almost N times, so the total number of loops within loops would be N^2.
The inner loop (of insertSorted) isn't exactly N times.

First number it will be 0 times.
Second number it will be 1 times.
Third number it will be 2 times
...
(N-2)nd number it will be N-2 times.
(N-1)st number it will be N-1 times
Nth number it will be N times

total steps = 1 + 2 + 3 + 4 + ... + (N-2) + (N-1) + N

We can prove this is \( N \frac{N+1}{2} \) and if we multiply it out \( (N^2 - N)/2 \), which is \( N^2/2 + N/2 \)

Now if we look at \( \frac{N^2}{2} + \frac{N}{2} \) and ask what this is when \( N \to \infty \), we realize that the second term becomes insignificant and we are left with the first term.

We then don't worry about dividing by 2. So we are left with \( N^2 \)

### E. Swapping

**Given:** \( x = 2; y = 7 \)

<table>
<thead>
<tr>
<th>Classic method</th>
<th>Addition method</th>
<th>XOR method</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp = x</td>
<td>( x = x + y )</td>
<td>( x = x \oplus y )</td>
</tr>
<tr>
<td>x = y</td>
<td>( y = x - y )</td>
<td>( y = x \oplus y )</td>
</tr>
<tr>
<td>y = temp</td>
<td>( x = x - y )</td>
<td>( x = x \oplus y )</td>
</tr>
</tbody>
</table>

\( \oplus \) stands for Exclusive OR and is a bitwise operation:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a( \oplus )b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Apply this rule to every column of bits:

\[ x \quad y \]
\[ 101 \quad 110 \]

Python adds a notational shortcut:

\[ x, y = y, x \]

What is happening here is that \( = y, x \) creates a tuple, same as \( = (y, x) \)
Then you can unpack a tuple by giving as many variables as it has parts:

```
    tup = ("Mark", 59, "cats", False)
    name, age, pets, is_jesuit = tup
    name = tup[0]
    age = tup[1]
    pets = tup[2]
    is_jesuit = tup[3]
```

It unpacks each part into the corresponding variable, so `x, y = y, x` will be same as:

```
tup = (y, x)
x = tup[0]
y = tup[1]
```

**F. Look at several sorts**

Bubble sort: swap every adjacent pair that is out of order. Need to rebubble N times!

Selection sort – lower half of list is sorted, and so you need to find the upper half

Insertion sort – basically our `insertSorted()`. The first part of the list is the sorted list.

It might be easier to think about if you had 2 lists.

```
    newlist = []
    for i in range(len(start_list)):
        minimum_value = findMin(start_list)
        newlist.append(minimum_value)
        start_list.remove(minimum_value)
```

What is hidden here is that `remove()` is O(N) time complexity! Plus this uses more room.

Merge sort – requires extra memory; recursive version is easy, non-recursive version is hard!

Quicksort – great, widely used. No extra memory and it is very fast.

Quicksort's average time complexity is NlogN and requires no extra memory so it is often used.

You'll see O(N^2) when the original list is sorted, either in ascending or descending order.

Look at insertion sort.
\[
((3+6) \times (14-8))^3
\]

**postfix**

3 6 + 14 8 - * 3 ^

**prefix**

\[
\wedge \times - + 3 \hspace{1cm} 6 \hspace{1cm} 14 \hspace{1cm} 8 \hspace{1cm} 3
\]

**infix**

\[
(3 \times \times y \geq 4 - + 2)^
\]

invalid

\[
((3 \times x) + ((4 \times 2) -
\]

\[
((3 + 6) \times (14 - 8))^3
\]
5 \cdot 7 + 3 \cdot 6 - \\
(5 + 7) \cdot 3 \\
(5 + 7) \cdot 3 - 6 \\
(((5 + 7) \cdot 3) - 6) \\
- \cdot 5 \cdot 7 \cdot 3 \cdot 6 \\
5 \cdot 7 + 3 \cdot 6 - 0
$4 \times * 3 \ 6 - 1 \ 2$

$((4 \times x) \div (3 - 6))^2$
**Probabilistic Search**

```python
import random

numlooks = 0

def probabilistic_search(numbers, key):
    assert type(numbers) is list, "1st arg must be a list"
    global numlooks; numlooks = 0
    while True:
        numlooks += 1
        z = random.randint(0, len(numbers) - 1)
        if numbers[z] == key:
            return z
    # when do you stop?

def main():
    numbers = [2, 4, 7, 10, 11, 32, 45, 87, 15, 88, 72, 35, 48, 26, 13, 91, 70, 52, 63, 82, 80, 15, 5, 92, 37, 33, 29]

    key = int(input("Enter a value: "))
    keyIndex = probabilistic_search(numbers, key)

    if keyIndex == -1:
        print(key, "was not found")
    else:
        print("found", key, "at index", keyIndex)
    print("Numlooks = ", numlooks)

main()
```

**Linear Search**

```python
def linearSearch(numbers, key):
    assert type(numbers) is list, "1st arg must be a list"
    for i in range(len(numbers)):
        if numbers[i] == key:
            return i
    return -1  # not found

def main():
    numbers = [2, 4, 7, 10, 11, 32, 45, 87]

    key = int(input("Enter a value: "))
    keyIndex = linearSearch(numbers, key)

    if keyIndex == -1:
        print(key, "was not found")
    else:
        print("found", key, "at index", keyIndex)

main()
```
**Binary Search**

```python
def binarySearch(numbers, key):
    assert type(numbers) is list, "lst arg must be a list"
    mid = 0; low = 0; high = len(numbers) - 1
    while high >= low:
        mid = (high + low) // 2
        if numbers[mid] < key:
            low = mid + 1
        elif numbers[mid] > key:
            high = mid - 1
        else:
            return mid
    return -1  # not found

def main():
    numbers = [2, 4, 7, 9, 10, 11, 32, 45, 87, 99, 100, 105, 112, 126, 129,
               130, 131, 132, 435, 499, 501, 516, 525, 572, 583, 594, 600]
    key = int(input("Enter a value: "))
    keyIndex = binarySearch(numbers, key)
    if keyIndex == -1:
        print (key,"was not found")
    else:
        print ("found", key, "at index",keyIndex)

main()
```

**Binary Search (recursive)**

```python
def binarySearch(numbers, key):
    assert type(numbers) is list, "lst arg must be a list"
    mid = len(numbers) // 2
    if len(numbers) == 0:
        return False
    if numbers[mid] == key:
        return True
    else:
        if key < numbers[mid]:
            return binarySearch(numbers[0:mid], key)
        else:
            return binarySearch(numbers[mid+1:], key)

def main():
    numbers = [2, 4, 7, 9, 10, 11, 32, 45, 87, 99, 100, 105, 112, 126, 129,
               130, 131, 132, 435, 499, 501, 516, 525, 572, 583, 594, 600]
    print("size of list=",len(numbers))
    print("Midpoint index = ",len(numbers)//2)
    print("value at midpoint = ",numbers[len(numbers)//2])
    key = int(input("Enter a value: "))
    keyIndex = binarySearch(numbers, key)
    if keyIndex == -1:
        print (key,"was not found")
    else:
        print ("found", key, "at index",keyIndex)

main()
```
\[
\left\lfloor \log_2(100) \right\rfloor
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>

\[
2^n = 100
\]

\[
2^{10} = 1024
\]

\[
\log_2 N = \frac{\log_{10} N}{\log_{10} 2} = \frac{\log_{10} N}{0.30103}
\]