CS1020E: DATA STRUCTURES AND ALGORITHMS I

Tutorial 8 – Complexity Analysis

(Week 10, starting 17 October 2016)

1. *Big-O Analysis*

Big-O time complexity gives us an idea of the **growth rate** of a function. In other words, "for a **large input size** N, **as N increases**, in what **order of magnitude** is the volume of executed code expected to increase?" So, two functions with the same time complexity may have very different running times for all N.

Rearrange the 15 terms in ascending order of their Big-O time complexity:

$4N^2$, $\log_5(N)$, $20N$, $N^{2.5}$, $\log(N!)$, N^N, 3^N, $N \log(N)$, $100N^{2/3}$, $\log(N)$, 2^N, 2^{N+1}, $N!$, $(N-1)!$, 2^{2N}

Also, **classify** the terms as "_____ time" where possible, and try to think of examples of each.

Do note that some of the above terms have equal Big-O complexity. Remember that in Big-O notation, we only care about the dominating term of the function, without its coefficient. As N gets very large, the effect of other terms on the volume of code executed becomes insignificant.

2. *Analysis of Iterative Algorithms*

How much code is executed, relative to input size N? Often, but **NOT always**, we can get an idea from the number of times a **loop iterates**. Analyze the Big-O (worst-case) time complexity of each code snippet:

```
void printTriangle(int pintN) { // (a)
     for (int intRow = 0; intRow < pintN; intRow++) { // loop 1
          for (int intCol = intRow; intCol < pintN; intCol++) // loop 2
               cout << "*";
          cout << endl;
     }
}
```

Some **useful facts or techniques**, to analyse the complexity of simple algorithms:

(a) Arithmetic series

(b) Geometric series

(c) How to draw the number of times a block of code is executed, in the form of a tree

In this case, there is only one inner loop, so the tree becomes a list.

```
void printTriangleV2(int pintN) { // (b)
     for (int intIndex = 0; intIndex < pintN; intIndex++) // loop 1
          for (int intRow = intIndex + 1; intRow > intIndex; intRow--) {
               for (int intCol = pintN; intCol > intRow; intCol--) // 3
                    cout << "*";
               cout << endl;
          }
}
```

```
void clear(vector<int>& items) { // (c)
     int intN = items.size();
     for (int intIndex = 0; intIndex < intN; intIndex++) // loop 1
          items.erase(items.begin());
}
```

```
void clear(vector<int>& items) { // (d)
     int intN = items.size();
     for (; --intN >= 0;) items.erase(items.begin() + intN); // loop 1
}
```

```
void mystery(list<int>& items) { // (e)
     int intN = items.size() / 2;
     while (intN > 0) { // loop 1
          list<int>::iterator itr = items.begin();
          advance(itr, intN); // move forward N elements
          cout << *itr << " ";
          intN /= 2;
     }
}
```

```
void guessWhatThisIs[1](vector<int>& items) { // (f)
     int intN = items.size();
     for (int intEnd = intN - 1; intEnd > 0; intEnd--) // loop 1
          for (int intLeft = 0; intLeft < intEnd; intLeft++) // loop 2
               if (items.at(intLeft + 1) < items.at(intLeft)) {
                    int intTemp = items.at(intLeft);
                    items.at(intLeft) = items.at(intLeft + 1);
                    items.at(intLeft + 1) = intTemp;
               }
}
```

[1] What's this?

3. *Analysis of Recursive Algorithms*

For recursive problems, draw out the recursive tree. For each node in the tree:

- Identify how many calls are made directly from that node
- Identify how the **problem size** decreases each call
- Indicate how much **work is done per call**, aside from other recursive calls

Calls on the same level usually have similar problem sizes. Therefore, for each level:

- Calculate the work done per level
- Calculate the height of the tree if it helps you

These may help you to evaluate the Big-O time complexity quickly.

```
long long power(long long x, long long k, long long M) {
    if (k == 0) return 1;
    long long y = k / 2;
    if (2 * y == k) { // even power k
        long long half = power(x, y, M); // (x^y) % M
        return half * half % M; // [(x^y % M)(x^y % M)] % M
    } else { // k == 2y + 1
        long long next = power(x, 2 * y, M); // (x^2y) % M
        return x * next % M; // [x (x^2y % M)] % M
    }
}
```

Worked Example

Notice, x and M do not change, and are not useful to us. To find the next problem size, y may help.

Substituting k/2 for y and disregarding O(1) statements, the code can be **reduced** to:

```
void powerReduced(long long k) {
    if (k == 0) return; // base case, to identify leaf nodes
    if (k % 2 == 0) powerReduced(k/2); // even k, recursive call
    else powerReduced(k-1); // odd k, recursive call
} // O(1) work done per call (aside from rec. calls)
```

Next, draw out the recursive tree (list, in this case, as only one call is made within another)

We here assume k is a power of 2 (e.g. 64, 1024, …)

Level	Problem size		# mtd calls in level	Work done per call	Work done in level
	Intuitive	Based on level			
1	k	$2^{\log(k)}$	1	O(1)	O(1)
2	k/2	$2^{\log(k)-1}$	1	O(1)	O(1)
…					
h-1	2	2^{1}	1	O(1)	O(1)
Height h	1	2^{0}	1	O(1)	O(1)

How much work is done in total?

We can sum the work done in each level to get the answer.

However, since every level does O(1) work, we first need to find the height h, which is log(k)+1.

Therefore, time complexity is **O(log(k))**.

You may ask, what happens if k is not a power of 2?

The worst case occurs when k is a ((power of 2) -1).

- Every call to powerReduced(x) results in a call to powerReduced(x-1) and powerReduced((x-1)/2)
- Each call does O(1) work aside from other recursive calls
- The height (length) of the list is 2log(k) + 1
- Therefore, the time complexity is still O(log(k)).

Your Turn!

Analyze the Big-O (worst-case) time complexity of each code snippet:

```
long long powerSum(long long x, long long k, long long M) { // (a)
    if (k == 1) return x % M;
    long long y = k / 2;
    if (k % 2 == 0) { // even power k
        long long half = powerSum(x, y, M); // S(y) % M
        long long pw = power(x, y, M); // (x^y) % M
        long long ans = half * (pw + 1); // (S(y) % M) [1 + (x^y % M)]
        return ans % M;
    } else { // k == 2y + 1
        long long next = powerSum(x, y + y, M); // S(2y) % M
        long long pw = power(x, k, M); // x^(2y + 1) % M
        long long ans = next + pw; // (S(2y) % M) + (x^(2y + 1) % M)
        return ans % M;
    }
}
```

```
bool lookHere(vector<int>& items, int value, int low, int hi); // (b)
bool lookHere(vector<int>& items, int value) {
     int intN = items.size() – 1;
     return lookHere(items, value, 0, intN);
}
bool lookHere(vector<int>& items, int value, int low, int hi) {
     if (low > hi) return false;
     int mid = (low + hi) / 2;
     // do some O(1) stuff
     if (items.at(mid) > value)
          return lookHere(items, value, low, mid - 1);
     else
          return lookHere(items, value, mid + 1, hi);
}
```

```
void lookHere(vector<int>& items, int value, int low, int hi); // (c)
void lookHere(vector<int>& items, int value) {
     int intN = items.size() - 1;
     lookHere(items, value, 0, intN);
}
void lookHere(vector<int>& items, int value, int low, int hi) {
     if (low >= hi) return;
     int mid = (low + hi) / 2;
     // do some O(1) stuff
     lookHere(items, value, low, mid);
     lookHere(items, value, mid + 1, hi);
}
```

```
void lookHere(vector<int>& items, int value, int low, int hi); // (d)
void lookHere(vector<int>& items, int value) {
     int intN = items.size() - 1;
     lookHere(items, value, 0, intN);
}
void lookHere(vector<int>& items, int value, int low, int hi) {
     if (low >= hi) return;
     int mid = (low + hi) / 2;
     // do some O(N) stuff
     lookHere(items, value, low, mid);
     lookHere(items, value, mid + 1, hi);
}
```

```
void lookHere(vector<int>& items, int value, int low, int hi); // (e)
void lookHere(vector<int>& items, int value) {
     int intN = items.size() - 1;
     lookHere(items, value, 0, intN);
}
void lookHere(vector<int>& items, int value, int low, int hi) {
     if (low >= hi) return;
     int mid = (low + hi) / 2;
     // do some O(N^2) stuff
     lookHere(items, value, low, mid);
     lookHere(items, value, mid + 1, hi);
}
```

Get enough practice
Draw out the recursive tree
Revise your mathematics
Have fun ^_^