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.

Answer

(Lowest, smallest growth rate)

Constant time
e.g. O(1) random access of a vector

Logarithmic time
O(**log(N)**) = O(**$\log_5$(N)**)
Because $\log_5(N) = \log(N) / \log(5)$
e.g. O(log(N)) binary search of a sorted array

(In between logarithmic and linear time)
$O(N^{2/3})$ = O(**$100N^{2/3}$**)

Linear time
O(N) = O(**20N**)
e.g. O(N) sequential access of a list

Linearithmic time

O(**N log(N)**) = O(**log(N!)**) [Extra: Why?[1]]

e.g. O(N log(N)) mergesort / quicksort

Quadratic time

O(N^2) = O(**4N^2**)

e.g. O(N^2) "simple" sorting algorithms, examining pairs of elements in a list

(Other) Polynomial (N^k) time

O(**N$^{2.5}$**)

Exponential time

O(**2^N**) = O(**2^{N+1}**)

Because $2^{N+1} = 2 * 2^N$

O(**3^N**)

O(4^N) = O(**2^{2N}**)

e.g. O(2^N) / O(3^N) / O(4^N) recursive function f(x) which calls f(x-1) two / three / four times

Factorial time

O(**(N-1)!**)

O(**N!**)

e.g. O(N!) permuting (N-1) distinct objects

(Greater than factorial time)

O(**N^N**)

e.g. O(N^N) or worse - permuting N distinct objects, with replacement after each draw

This is the largest among the 15 terms, i.e. having highest growth rate.

You wouldn't want to have an O(N^N) algorithm for a medium-sized problem onwards…

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;
     }
}
```

[1] We can add coefficients to either function, so that one function bounds the other from above, i.e. both have the same big-O time complexity: N/2 log(N/2) < log(N!) < N log(N)) < 2 log(N!) = O(N log(N)) = O(log(N!))

Proof sketch: See accepted answer at http://stackoverflow.com/questions/2095395/is-logn-%CE%98n-logn

Answer

Time complexity is **O(N^2)**.

(1) Intuitively, **area** of right-angled NxN triangle is around N^2/2, right?

(2) Another way of looking at the problem:

intRow	# inner iterations
0	N
1	N-1
…	
N-2	2
N-1	1

Total # iterations is an **arithmetic series**

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = \frac{N^2}{2} + \frac{N}{2} = \boldsymbol{O(N^2)}$$

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;
          }
}
```

Answer

Trace through the code carefully.

Although there are 3 loops, loop 2 just performs 1 iteration for each value of `intIndex`.

Hence, as compared to part (a), one less '*' is printed each row.

Time complexity is still **O(N^2)**.

(1) Intuitively, **area** of rt-angled Nx(N-1) triangle is still in the order of N^2/2, right?

(2) Examining iterations and arithmetic series:

$$\sum_{i=0}^{N-1} i = \sum_{i=1}^{N-1} i = \frac{(N-1)(N-1+1)}{2} = \frac{N^2}{2} - \frac{N}{2} = \boldsymbol{O(N^2)}$$

This example highlights:

(a) **Additional nested loop** does not mean time complexity increases

(b) Changed **output**, or **execution time**, does not mean time complexity changes

(c) When implementing, debugging or analysing code, **look at counters carefully**

```
void clear(vector<int>& items) { // (c)
     int intN = items.size();

     for (int intIndex = 0; intIndex < intN; intIndex++) // loop 1
          items.erase(items.begin());
}
```

Answer

Lesson: Do not neglect the time complexity of operations on containers!

intIndex	# elem. accessed
0	N
1	N-1
…	
N-2	2
N-1	1

The number of array elements accessed each iteration is identical to the number of '*' printed in (a).

Time complexity is **O(N^2)**.

(1) Intuitively, **area** of right-angled NxN triangle is around $N^2/2$

(2) Total # iterations is an **arithmetic series**

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = \frac{N^2}{2} + \frac{N}{2} = \boldsymbol{O(N^2)}$$

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

Answer

Time complexity is **O(N)**.

N calls are made to an O(1) operation.

```
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;
     }
}
```

Answer

Time complexity is **O(N)**.

LinkedList does not have the random access property. Each call to `get(i)` is O(*min*(i, n - i)).

intN	# nodes accessed	
N/2	N/2 + 1	$2^{\log(N)-1} + 1$
N/4	N/4 + 1	$2^{\log(N)-2} + 1$
…		
2	2 + 1	$2^1 + 1$
1	1 + 1	$2^0 + 1$

Total nodes accessed form a geometric series:

$$\sum_{i=0}^{\log(N)-1} (2^i + 1) = \frac{2^{\log(N)} - 1}{2 - 1} + \log(N) = 2^{\log N} + \log(N) - 1 = O(2^{\log N}) = \boldsymbol{O(N)} \neq O(N \log(N))$$

Do NOT assume the time complexity is around O(log(N) * N/2) = O(N log(N))

just because there are log(N) iterations and (N/2 + 1) nodes accessed at first.

We need to remember that the shape is **neither** a **square nor a triangle**.

```
void guessWhatThisIs(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;
               }
}
```

Answer

Guess what this is[2]?

Even if you don't know what this does, examine each statement quickly.

The condition and 3 statements within loop 2 all execute in O(1) time.

intEnd	# inner iterations
N-1	N-1
N-2	N-2
…	
2	2
1	1

$$\sum_{i=1}^{N-1} i = \frac{(N-1)(N-1+1)}{2} = \frac{N^2}{2} - \frac{N}{2} = \boldsymbol{O(N^2)}$$

We do not always need to completely understand an algorithm in order to analyze its complexity.

[2] Ans: Bubble sort!

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;
    }
}
```

Answer
Again, x and M 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 powerSumRed(long long k) {
    if (k == 1) return; // base case, to identify leaf nodes
    if (k%2==0) { powerSumRed(k/2); doOhLogK(k/2);} // even k, rec. call
    else { powerSumRed(k-1); doOhLogK(k); } // odd k, rec. call
}
```

Next, draw out the recursive tree / list, assuming 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	log(k/2)	log(k/2)
2	k/2	$2^{\log(k)-1}$	1	log(k/4)	log(k/2) – 1
…					
h-2	8	2^3	1	log(4)	2
h-1	4	2^2	1	log(2)	1
Height h	2	2^1	1	1	1

As the work done per level is an arithmetic sequence, we do not need to find the height of the recursive list.

$$1 + \sum_{i=1}^{\log(k/2)} i = 1 + \frac{(\log(k) - 1)(\log(k/2) + 1)}{2} = \boldsymbol{O((\log(k))^2)} \neq O(\log(\boldsymbol{k^2}))$$

Therefore, the time complexity is $((\log(\boldsymbol{k}))(\log(\boldsymbol{k})))$.

Again, the time complexity in Big-O notation does not change if k is not a power of 2.

```
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);
}
```

Answer

What's this[3]?

We are not concerned with items and value. Of course, we should first check that the code runs repeatedly and terminates, based on the values of low and hi. Our problem size is now (hi - low).

```
void lookHereRec(int intN) {
     if (intN < 0) return;
     doOhOne();
     lookHereRec(intN / 2); // condition removed, both branches combined
}
```

Next, draw out the recursive tree / list, assuming N 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	N	$2^{\log(N)}$	1	1	1
2	N/2	$2^{\log(N)-1}$	1	1	1
…					
h-2	2	2^1	1	1	1
h-1	1	2^0	1	1	1
Height h	0	0	1	1	1

Height of the recursive list is log(N)+2.

[3] Binary search with some parts hidden.

Time complexity is **O(log(N))**. Even if N is not a power of 2, we have height 2log(N) = O(log(N)).

```
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);
}
```

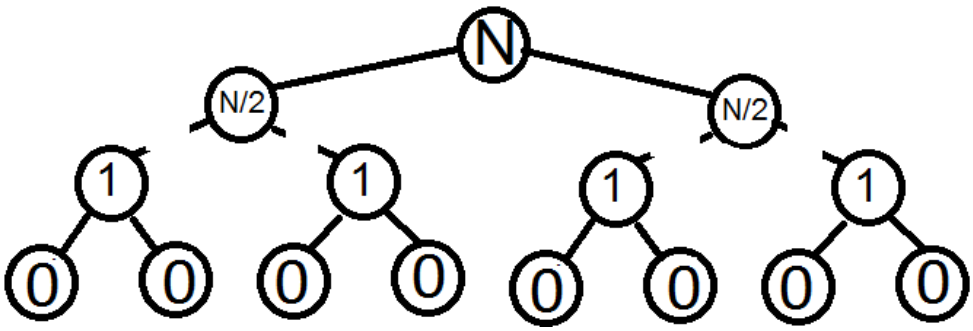
Answer

Simplify the problem as we have been doing.

```
void lookHereRec2(int intN) {
     if (intN <= 1) return;
     doOhOne();
     lookHereRec2(intN / 2); lookHereRec2(intN / 2);
}
```

Oh look! A tree at last! The problem size is written in each node.

Remember, our objective is to find the sum of the **work done by all recursive function calls**.



Level	Problem size		# mtd calls in level	Work done per call	Work done in level
	Intuitive	Based on level			
1	N	$2^{\log(N)}$	1	1	1
2	N/2	$2^{\log(N)-1}$	2	1	2
3	N/4	$2^{\log(N)-2}$	4	1	4
…					
h-2	2	2^{1}	2^{h-3}	1	2^{h-3}
h-1	1	2^{0}	2^{h-2}	1	2^{h-2}
Height h	0	0	2^{h-1}	1	2^{h-1}

h = log(N) + 2

Total work done forms a geometric series:

$$\sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1} = 4\left(2^{\log N}\right) - 1 = 4N - 1 = \boldsymbol{O(N)} \neq O(N \log(N))$$

```
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);
}
```

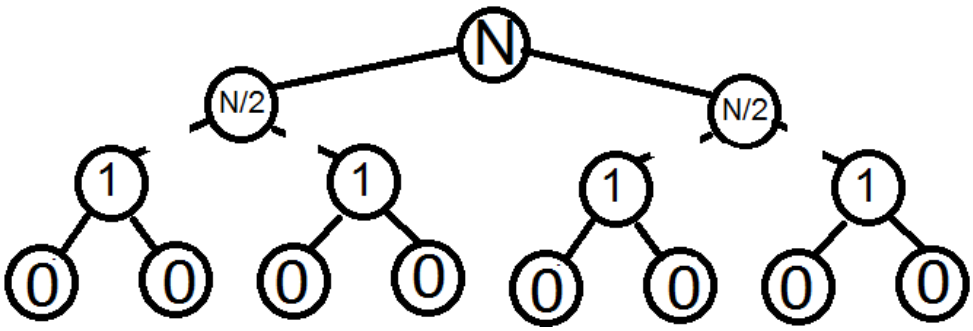
Answer

What's this[4]?

Simplify the problem as we have been doing.

```
void lookHereRec3(int intN) {
     if (intN <= 1) return;
     doOhN(intN);
     lookHereRec3(intN / 2); lookHereRec3(intN / 2);
}
```

The recursive tree is the same. However, the amount of work done per call (hence per level) is different.



Level	Problem size		# mtd calls in level	Work done per call	Work done in level
	Intuitive	Based on level			
1	N	$2^{\log(N)}$	1	N	N
2	N/2	$2^{\log(N)-1}$	2	N/2	N
3	N/4	$2^{\log(N)-2}$	4	N/4	N
…					
h-2	2	2^{1}	2^{h-3}	2	2^{h-2}
h-1	1	2^{0}	2^{h-2}	1	2^{h-2}
Height h	0	0	2^{h-1}	1	2^{h-1}

h = log(N) + 2, therefore 2^{h-2} = N

Total work done= $(\log N + 1)N + 2N = N \log N + 3N = \boldsymbol{O(N \log N)}$

[4] Merge sort with some parts hidden. Merge sort does O(N) work **after** the two recursive calls, unlike this example.

```
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);
}
```

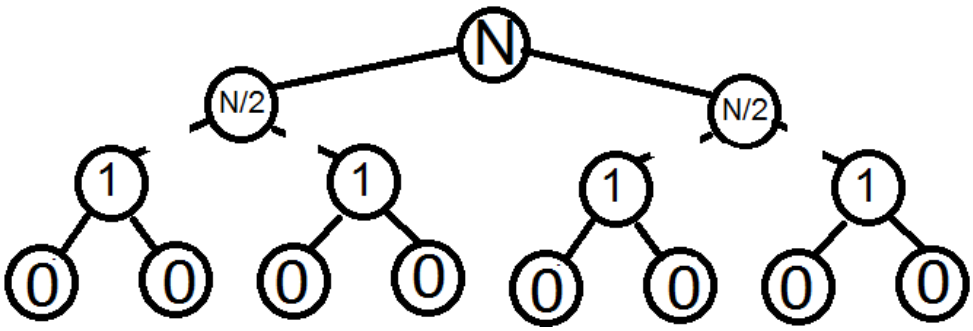
Answer

Simplify the problem as we have been doing.

```
void lookHereRec4(int intN) {
     if (intN <= 1) return;
     doOhNSquared(intN);
     lookHereRec4(intN / 2); lookHereRec4(intN / 2);
}
```

Again, the recursive tree is the same due to similar recursive calls.

However, the amount of work done per call (hence per level) is different.



Level	Problem size		# mtd calls in level	Work done per call	Work done in level
	Intuitive	Based on level			
1	N	$2^{\log(N)}$	1	N^2	N^2
2	N/2	$2^{\log(N)-1}$	2	$(N/2)^2$	$N^2/2$
3	N/4	$2^{\log(N)-2}$	4	$(N/4)^2$	$N^2/4$
…					
h-2	2	2^1	2^{h-3}	2^2	2N
h-1	1	2^0	2^{h-2}	1	N
Height h	0	0	2^{h-1}	1	2N

h = log(N) + 2, therefore $2^{h-2} = N$

Total work done can be transformed into a geometric series:

$$2N + \sum_{i=0}^{\log(N)} 2^i N = 2N + \frac{2^{\log(N)+1} - 1}{2 - 1} N = 2N + 2N\left(2^{\log N}\right) - N = N + 2N^2 = \boldsymbol{O(N^2)} \neq O(N^2 \log(N))$$

Alternatively, the total work done can be transformed into sum-to-infinity of a geometric sequence:

$$2N + \frac{N^2}{2} + \frac{N^2}{4} + \cdots + 2N + N = 2N + N^2(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{0.5N} + \frac{1}{N})$$

$$2N + \frac{N^2}{2} \leq 2N + N^2\left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{0.5N} + \frac{1}{N}\right) < 2N + N^2(\frac{1}{1 - 0.5})$$

$\boldsymbol{O(N^2)}$ function $\leq 2N + N^2\left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{0.5N} + \frac{1}{N}\right) \leq \boldsymbol{O(N^2)}$ function

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