CS 341 — Lecture 3

Bartosz Antczak *Instructor: Bin Ma* *September 14, 2017*

3.1 Reduction

This section outlines how an algorithm can be simplified, thus *reducing* its run-time and complexity.

3.1.1 Example 1: two sum

- **Input:** an integer array `A` of size `n` and a value `m`
- **Output:** indices `i` and `j` such that `A[i] + A[j] = m`

Algorithm I

```
for every pair i and j
    if (A[i] + A[j] == m)
        output(i, j) and return

output(fail)
```

This algorithm has a run-time of $O(n^2)$.

Algorithm II

```
Store A in hastable
Check if m-A[i] is in hashtable
```

This algorithm has an *average* run-time of $O(n)$.

Algorithm III

```
Sort A
for i = 0 to n
    Perform binary search on A with key m-A[i]
```

This algorithm has a run-time of $O(n \log n)$ (binary search is $O(\log n)$ done n times).

Algorithm IV

```
Sort A
i = 1, j = n
while (i <= j)
    if (A[i] + A[j] == m) output and return
    else if (A[i] + A[j] < m) i++
    else j--

output(fail)
```

Let's focus on this algorithm. This algorithm doesn't intuitively look like it is correct, so let's prove it!

Proof of Correctness for Alg. IV

Let's analyze two cases:

1. If no such pair exists such that `A[i] + A[j] = m`, then the algorithm will return `fail`
2. If there does exist a pair, call them `i', j'`. Without loss of generality, assume `i` (our counter variable) becomes `i'` before `j` becomes `j'`. That is, `i = i'` and `j > j'`, thus:

   $$\texttt{A[i] + A[j] > A[i'] + A[j'] = m}$$

   According to the algorithm, `j` will keep reducing until `j = j'`. The other direction is proven by symmetry. □

Now let's analyze its time complexity. The only part of the agorithm that interests us is the `while`-loop. Noticing that `j-i` has an initial value of `n-1` and reduces by 1 every time, the `while`-loop is repeated $O(n)$ times. Thus, the entire run-time of this algorithm is $O(n \log n)$

3.1.2 Example 2: three sum

- **Input:** integer array of size `n`
- **Output:** indices `i`, `j`, `k` such that `A[i] + A[j] + A[k] = 0`

Algorithm I

```
Try all i, j, k
```

This is the trivial algorithm that has a run-time of $O(n^3)$.

Algorithm II

```
For every k
    use two-sum to solve A[i] + A[j] = -A[k]
```

If we use algorithm IV from the two-sum example, this algorithm has a run-time of $O(n^2 \log n)$

Algorithm III

We can remove the redundant sorting call in our previous algorithm:

```
Sort A
for every k
    i = 1, j = n
    while (i <= j)
    if (A[i] + A[j] == -A[k]) output and return
    else if (A[i] + A[j] < -A[k]) i++
    else j--
```

This algorithm has a run-time of $O(n^2)$.

Problem

What if we wanted to find such indices where `A[i] + A[j] + A[k] = m`? Simply rewrite the code such that you solve for `A[i] + A[j] = m - A[k]`!

3.2 Divide and Conquer

Let's analyze the merge-sort algorithm (that we are already familiar with):

```
Given an array A of size n:

1. If n <= 3, sort A with trivial algorithm      O(1)
2. Merge-sort(A[1 ... n/2])                      T(n/2)
3. Merge-sort(A[n/2 + 1 ... n])                  T(n/2)
4. Merge(A[1 .. n/2], A[n/2 + 1 ... n])          T(n)
```

Looking at its time complexity, let's define this algorithm's **recurrence relation**:

- $T(1) = O(1)$ (we must define a base case)
- $T(n) = 2 \cdot T(\frac{n}{2}) + n$

Let's solve this recurrence relation.

Method 1: unrolling

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \tag{3.1}$$

$$= 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \tag{3.2}$$

$$\vdots \quad (\log n \text{ times}) \tag{3.3}$$

$$= n \cdot T(1) + n \cdot \log n \tag{3.4}$$

$$= O(n \log n) \tag{3.5}$$

Method 2: induction

Prove that $T(n) \leq c \cdot n \log n$:

- Proving it for $n = 1$ is trivial
- Assume this statement is true for $n < m$. Next, prove it's true for $n = m$. By induction:

$$T(m) = 2 \cdot T\left(\frac{m}{2}\right) + m \tag{3.6}$$

$$\leq 2 \cdot \left(c \cdot \frac{m}{2} \log\left(\frac{m}{2}\right)\right) + m \tag{3.7}$$

$$= cm \cdot (\log m - 1) + m \tag{3.8}$$

$$\leq c \cdot m \log m \quad \square \tag{3.9}$$