CS 162 Notes (Winter 2017)

Justin Lubin

Contents

1 Asymptotic Analysis **5**

1.1 Big-O Notation . . . 5
1.1.1 Definition . . . 5
1.1.2 Assumption: Constant Operations (Approximations) . . . 5
1.1.3 Non-Constant Operations . . . 5
1.2 Reducing Big-O Expressions . . . 6
1.2.1 What to Eliminate . . . 6
1.2.2 Examples . . . 6
1.3 Linear Search . . . 6
1.3.1 Code . . . 6
1.3.2 Analysis . . . 6
1.4 Binary Search . . . 7
1.4.1 Code . . . 7
1.4.2 Analysis . . . 7
1.5 Methods of Asymptotic Analysis . . . 7
1.5.1 Expansion Method . . . 7
1.5.2 Substitution Method . . . 8
1.6 The Towers of Hanoi . . . 9
1.6.1 Gameplay . . . 9
1.6.2 Algorithmic Solution . . . 9
1.7 Mergesort . . . 10
1.7.1 Description . . . 10
1.7.2 Example . . . 10
1.7.3 Analysis . . . 10

2 Algorithm Correctness **13**

2.1 Key Parts of an Algorithm . . . 13
2.2 Proving Correctness . . . 13
2.2.1 Basic Example . . . 13
2.2.2 Loop Invariants . . . 14
2.2.3 Strong Induction . . . 14
2.2.4 Mergesort . . . 14

Chapter 1

Asymptotic Analysis

1.1 Big-O Notation

1.1.1 Definition

We have that a function $f \in O(g(n))$ if and only if there exists constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

We have that a function $f \in \Omega(g(n))$ if and only if there exists constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \leq n_0$.

We have that a function $f \in \Theta(g(n))$ if and only if $f \in O(g(n))$ and $f \in \Omega(g(n))$.

This notation can be used to analyze the best-case, average-case, and worse-case efficiency of an algorithm, but this class typically concerns the worst-case efficiency of an algorithm.

Note that efficiency can be a measure of time, space, or even power complexity.

1.1.2 Assumption: Constant Operations (Approximations)

- Arithmetic (fixed width)
- Assignment
- Access any array element

1.1.3 Non-Constant Operations

Control Flow	Time
Consecutive statements	Sum of time of each statement
Conditional	Time of test + time of the slower branch
Loop	Number of iterations * time of body
Function call	Time of function body

Control Flow	Time
Recursion	Solve recurrence relation

1.2 Reducing Big-O Expressions

1.2.1 What to Eliminate

- Eliminate low-order terms
- Eliminate coefficients

1.2.2 Examples

- $4n + 5 \in O(n)$
- $\frac{1}{2} n \log n + 2n + 7 \in O(n \log n)$
- $n^3 + 2^n + 3n \in O(2^n)$
- $n \log(10n^2) + 2n \log n \in O(n \log n)$
  - Note that $n \log(10n^2) = 2n \log(10n)$

1.3 Linear Search

1.3.1 Code

```
int find(int[] arr, int arr_length, int k) {
    for (int i = 0; i < arr_length; ++i) {
        if (arr[i] == k) {
            return 1;
        }
    }
    return 0;
}
```

1.3.2 Analysis

- Best case: approximately six steps
  - $O(1)$
- Worst case: $6 * \text{arr\_length}$ steps
  - $n = \text{arr\_length}$, so $O(n)$

1.4 Binary Search

1.4.1 Code

```
int find(int[] arr, int k, int lo, int hi) {
    return help(arr, k, 0, arr_length);
}

int find(int[] arr, int arr_length, int k) {
    int mid = (hi + lo) / 2;
    if (lo == hi) {
        return 0;
    }
    if (arr[mid] == k) {
        return 1;
    }
    if (arr[mid] < k) {
        return help(arr, k, mid + 1, k);
    } else {
        return help(arr, k, lo, mid);
    }
}
```

1.4.2 Analysis

Let $T(n)$ be the efficiency of `find`. Then, because each split takes approximately ten operations, we have that:

$$\begin{aligned} T(n) &= 10 + T\left(\frac{n}{2}\right) \\ &= 10 + \left(10 + T\left(\frac{n}{4}\right)\right) \\ &= 10 + \left(10 + \left(10 + T\left(\frac{n}{8}\right)\right)\right) \\ &= 10k + T\left(\frac{n}{2^k}\right). \end{aligned}$$

To solve this, there are a couple methods.

1.5 Methods of Asymptotic Analysis

1.5.1 Expansion Method

We know $T(1)$, so we should try to express this formula in terms of $T(1)$. To do so, let $\frac{n}{2^k} = 1$, so then $k = \log n$. Then

$$\begin{aligned} T(n) &= 10\log n + T\left(\frac{n}{2^{\log n}}\right) \\ &= 10\log n + T(1) \\ &= 10\log n + 10 \\ &\in O(\log n). \end{aligned}$$

However, this method actually gives you a big-theta approximation for T; in other words, not only is $T \in O(\log n)$, we also have that $T \in \Theta(\log n)$.

1.5.2 Substitution Method

IMPORTANT NOTE: This method is *not* recommended. Because it actually requires an inductive proof (not covered).

Guess $O(?)$, then check. For example (in this case), guess $\log n$ because we have something like $\frac{n}{2^n}$ in the formula. Then:

$$\begin{aligned} T(n) &= 10 + T(n/2) \\ &= 10 + \log(n/2) \end{aligned}$$

Because we have guessed that $T \in O(\log n)$, we have that $T(n) \leq c\log n$ for all $n \geq n_0$, for some constants c, n_0.

$$\begin{aligned} T(n) &\leq c\log n \\ 10 + \log(n/2) &\leq c\log n \\ 10 + \log(n) - \log(2) &\leq c\log n \\ \frac{10 + \log(n) - \log(2)}{\log n} &\leq c \\ \frac{10}{\log n} + 1 - \frac{\log(2)}{\log n} &\leq c \\ \frac{10}{\log n} + 1 - \frac{1}{\log n} &\leq c \end{aligned}$$

Now take $n_0 = 2$, so $n \geq 2$ and thus $\log n \geq 1$ (log is base two). We then have:

$$\begin{aligned} c &\geq \frac{10}{1} + 1 - \frac{1}{1} \\ c &\geq 10 \end{aligned}$$

Therefore, $T \in O(\log n)$ because with $c = 10$ and $n_0 = 2$, we have that $T(n) \leq c\log n$ for all $n \geq n_0$.

Note that the substituion method is more general than the **Expansion Method**, but it does *not* give you a big-theta approximation (unlike the **Expansion Method**).

1.6 The Towers of Hanoi

1.6.1 Gameplay

The goal of the Towers of Hanoi is to move all disks to goal peg, with the following rules:

- You can only move one disk at a time
- You can only move the top-most disk in a pile
- You cannot put a larger disk on top of a smaller one

1.6.2 Algorithmic Solution

```
if n = 1:                                  => T(1)
    move to goal (base case)                    = 1
else:                                      => T(n)
    move top n-1 disks to temporary peg         = T(n - 1)
    move bottom disk to goal                    + T(1)
    move the n-1 disks to goal                  + T(n - 1)
```

Thus, $T(n) = T(n-1) + T(1) + T(n-1)$, with $T(1) = 1$. So:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 4T(n-2) + 3 \\ &= 4(2T(n-3) + 1) + 3 \\ &= 8T(n-3) + 7. \end{aligned}$$

By inspection, we have:

$$T(n) = 2^{n-1}T(1) + (2^{n-1} - 1),$$

but $T(1) = 1$, so we have:

$$\begin{aligned} T(n) &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2^n - 1 \\ &\in \Theta(2^n). \end{aligned}$$

1.7 Mergesort

1.7.1 Description

To mergesort a list, split the list into two and sort the sublists. To merge them back together, interleave the elements. Interleaving / merging is $O(n)$ and there are $O(\log n)$ splits, so mergesort is $O(n \log n)$.

Mergesort is based on the trick that it is really easy to interleave two sorted lists.

1.7.2 Example

```
8 2 9 4 5 3 1 6
=> 8 2 9 4
   => 8 2
      => 8
      => 2
      merge: 2 8
   => 9 4
      => 9
      => 4
      merge: 4 9
   merge: 2 4 8 9
=>  5316
   => 5 3
      => 5
      => 3
      merge: 3 5
   => 1 6
      => 1
      => 6
      merge: 1 6
   merge: 1 3 5 6
merge: 1 2 3 4 5 6 8 9
```

1.7.3 Analysis

We have that $T(1) = 1$ and $T(n) = 2T(n/2) + n$ (split into two sublists, then mergesort them, then merge / interleave them). We then have that:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(T(n/4) + n/2) + n \\ &= 2T(n/4) + 2n \\ &= 4(2T(n/8) + n/8) + 2n \\ &= 8T(n/8) + 3n \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

Set $n/2^k = 1$ (because we are using the **Expansion Method**), so $k = \log n$. Then:

$$\begin{aligned} T(n) &= 2^{\log n} T(n/2^{\log n}) + (\log n)n \\ &= nT(1) + n \log n \\ &= n + n \log n \\ &\in \Theta(n \log n) \end{aligned}$$

Chapter 2

Algorithm Correctness

2.1 Key Parts of an Algorithm

There are a couple key things that every algorithm needs:

- Inputs
- Outputs
- Preconditions (restrictions on input)
- Postconditions (restrictions on output)
- Step-by-step process specification (either in natural language or pseudocode)

Therefore, we can define a "correct" algorithm to be one that, given any input data that satisfies the precondition, produces output data that satisfies the postcondition *and* terminates (stops) in finite time.

2.2 Proving Correctness

2.2.1 Basic Example

Consider the following pseudocode to swap two variables:

```
swap(x, y)
    aux := x
    x := y
    y := x
```

The following is a proof of correctness of `swap`:

1. Precondition: `x = a` and `y = b`.
2. Postcondition: `x = b` and `y = a`.
3. `aux := x` implies `aux := a`.
4. `x := y` implies `x := b`.
5. `y := aux` implies `y := a`.

6. Thus, `x := b` and `y := a`, so the postcondition is satisfied.

2.2.2 Loop Invariants

A *loop invariant* is a logical predicate that, if true before any single iteration of a loop, then is also true after the iteration of the loop. It is called an "invariant" because it is always true.

When using induction to prove the correctness of an algorithm, the loop invariant is the inductive hypothesis.

Consider the following psuedocode to sum the first `N` elements of a list `a`:

```
sum_list(a, N)
    s := 0
    k := 0
    while (k < N):
        s := s + a[k]
        k := k + 1
```

To prove this algorithm is correct, we must show:

1. The loop invariant (that at step `k`, `s` = sum of first `k` numbers in `a`) holds true; and
2. The algorithm terminates.

We want to show that, at step `k`, `s` = sum of first `k` numbers in `a`. Hence, we will induct on `k`.

1. At `k = 0`, we have that `s = 0`, so the algorithm is correct for `k = 0`.
2. We will assume that the algorithm holds for some arbitrary `k`.
3. We will now prove that the algorithm holds for `k + 1`. In the `(k + 1)`-th iteration of the loop, we set `s = s + a[k + 1]`, so `s` is the sum of the first `k` numbers (because, by the induction hypothesis, before the iteration of the loop we have that `s` is the sum of the first `k` numbers) plus `a[k + 1]`; i.e., it is the sum of the first `k + 1` numbers.

Hence, the loop invariant holds for all `k`. However, we must also prove that the algorithm terminates. For each iteration, we have that `k := k + 1`, so after `N` iterations, `k = N`, and hence the loop will terminate.

2.2.3 Strong Induction

Strong induction is just like regular induction, but instead of assuming the inductive hypothesis for some `k`, we assume it for all values less than or equal to `k`. We will use strong induction in the proof of correctness of mergesort.

2.2.4 Mergesort

Consider the following pseudocode of mergesort:

```
mergesort(A, l, r):
    if l < r:
        m = floor((l + r) / 2)
```

```
mergesort(A, l, m)
mergesort(A, m + 1, r)
merge(A, l, m, r)
```

Note that the precondition of `merge` is that `A[l ... m]` and `A[m + 1 ... r]` are sorted, and the postcondition is that `A` is sorted.

The precondition for this algorithm is that `A` has at least `1` element between the indices `l` and `r` (otherwise, this code doesn't have anything to do). The postcondition is that the elements between `l` and `r` are sorted.

To prove the correctness of this algorithm, we must show that the postcondition holds. To do so, we will prove that `mergesort` can sort `n` elements (and so it will therefore be able to sort `n = r - l + 1` elements).

The base case of `n = 1` is true because `A` is simply a one-element list (which is always sorted).

We will let the inductive hypothesis be that mergesort correctly sorts `n = 1...k` elements (i.e., everything less than or equal to `k`) This is strong induction.

To show the inductive step (`n = k + 1`) true, consider the two recursive calls in the function.

1. For the first recursive call, we have that `mergesort` is sorting `m - l + 1 = (k + 1) / 2 <= k` elements, so by the inductive hypothesis, `mergesort` holds. Hence, `A[l ... m]` is sorted.
2. For the second recursive call, we have that `mergesort` is sorting `r - (m + 1) + 1 = r - m = (k + 1) / 2 <= k` elements, so by the inductive hypothesis, `mergesort` holds. Hence, `A[m + 1 ... r]` is sorted.

We therefore have that the precondition of `merge` is upheld, so the postcondition of `merge` must also be upheld. Therefore, we have that `A` is sorted, and so the postcondition of `mergesort` is upheld.

Lastly, we must show that `mergesort` terminates. Note that for each recursive call, the length of the subarray between `p` and `q` decreases, so eventually it must reach a point where there is only one element in the array, in which case we reach termination.