CSE 373: Asymptotic Analysis, BSTs

Michael Lee
Friday, Jan 12, 2018

Warmup questions

Warmup: True or false:

- $5n + 3 \in \mathcal{O}(n)$
- $n \in \mathcal{O}(5n + 3)$
- $5n + 3 = \mathcal{O}(n)$
- $\mathcal{O}(5n + 3) = \mathcal{O}(n)$
- $\mathcal{O}(n^2) = \mathcal{O}(n)$
- $n^2 \in \mathcal{O}(1)$
- $n^2 \in \mathcal{O}(n)$
- $n^2 \in \mathcal{O}(n^2)$
- $n^2 \in \mathcal{O}(n^3)$
- $n^2 \in \mathcal{O}(n^{100})$

Definition: Dominated by

Definition: Dominated by
A function $f(n)$ is **dominated by** $g(n)$ when...

- There exists two constants $c > 0$ and $n_0 > 0$...
- Such that for all values of $n \geq n_0$...
- $f(n) \leq c \cdot g(n)$ is true

Definition: Big-$\mathcal{O}$
$\mathcal{O}(f(n))$ is the "family" or "set" of **all** functions that are **dominated by** $f(n)$

Definitions: Dominates

$f(n) \in \mathcal{O}(g(n))$ is like saying "$f(n)$ is less then or equal to $g(n)$".

Is there a way to say "greater then or equal to"? Yes!

Definition: Dominates
We say $f(n)$ **dominates** $g(n)$ when:

- There exists two constants $c > 0$ and $n_0 > 0$...
- Such that for all values of $n \geq n_0$...
- $f(n) \geq c \cdot g(n)$ is true

Definition: Big-Ω
$\Omega(f(n))$ is the family of all functions that **dominates** $f(n)$.

A few more questions...

True or false?

- $4n^2 \in \Omega(1)$
- $4n^2 \in \Omega(n)$
- $4n^2 \in \Omega(n^2)$
- $4n^2 \in \Omega(n^3)$
- $4n^2 \in \Omega(n^4)$

- $4n^2 \in \mathcal{O}(1)$
- $4n^2 \in \mathcal{O}(n)$
- $4n^2 \in \mathcal{O}(n^2)$
- $4n^2 \in \mathcal{O}(n^3)$
- $4n^2 \in \mathcal{O}(n^4)$

Definition: Big-Θ

Definition: Big-Θ
We say $f(n) \in \Theta(g(n))$ when both:

- $f(n) \in \mathcal{O}(g(n))$ and...
- $f(n) \in \Omega(g(n))$

...are true.

Note: in industry, it's common for many people to ask for the big-$\mathcal{O}$ when they really want the big-Θ!

Modeling complex loops

Exercise: construct a mathematical function modeling the worst-case runtime in terms of n.

Assume the println takes c time.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Foo!");
    }
}
```

A handwavy answer: $T(n) = 0c + 1c + 2c + 3c + \ldots + (n-1)c$

A not-handwavy answer: $T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} c$

7

Simplifying summations

Strategies:

- Wolfram Alpha
- Apply summation identities

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} c = \sum_{i=0}^{n-1} ci \quad \text{Summation of a constant}$$

$$= c\sum_{i=0}^{n-1} i \quad \text{Factoring out a constant}$$

$$= c\frac{n(n-1)}{2} \quad \text{Gauss's identity}$$

So, $T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} c = \underbrace{\frac{c}{2}n^2 - \frac{c}{2}n}_{\text{closed form}}$

8

Simplifying summations

Exercise: model the worst-case runtime using summations, find a closed form, find the big-Theta bound.

```
public void mystery2(int[] arr) {
    for (int i = 5; i < arr.length; i++) {
        int c = 0;
        for (int j = i; j < arr.length; j++) {
            c += arr[j];
        }
        System.out.println(c);
    }
}
```

Model: Let n be the array length. Then, $T(n) = \sum_{i=5}^{n-1}\sum_{j=i}^{n-1} 1$

9

Simplifying summations continued

$$\sum_{i=5}^{n-1}\sum_{j=i}^{n-1} 1 = \sum_{i=5}^{n-1}\left(\sum_{j=0}^{n-1} 1 - \sum_{j=0}^{i-1} 1\right) \quad \text{Normalize lower bound}$$

$$= \sum_{i=5}^{n-1}(n-i) \quad \text{Apply identity}$$

$$= \sum_{i=0}^{n-1}(n-i) - \sum_{i=0}^{5-1}(n-i) \quad \text{Normalize lower bound}$$

$$= \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{i=0}^{5-1} n + \sum_{i=0}^{5-1} i \quad \text{Split summations}$$

$$= n^2 - \frac{n(n-1)}{2} - 5n + 10 \quad \text{Apply identities}$$

10

Handling recursive functions

Exercise: model the worst-case runtime of this method.

```
public static int sum(int[] arr) {
    return sumHelper(0, int[] arr);
}

private static int sumHelper(int curr, int[] arr) {
    if (curr == arr.length) {
        return 0;
    } else {
        return arr[curr] + sumHelper(curr + 1);
    }
}
```

Answer: create a *recurrence*.

$$T(n) = \begin{cases} c_1 & \text{when n} = 0 \\ c_2 + T(n-1) & \text{otherwise} \end{cases}$$

Note: here, n is the number of items we need to visit, and c_1 and c_2 are some constants.

11

Simplifying recurrences

How do we find a *closed form* for:

$$T(n) = \begin{cases} c_1 & \text{when n} = 0 \\ c_2 + T(n-1) & \text{otherwise} \end{cases}$$

One method: the "unfolding" method.

Observation: when $n = 4$, $T(n) = c_2 + (c_2 + (c_2 + (c_2 + c_1)))$

We repeat c_2 four times, so $T(4) = 4c_2 + c_1$.

After generalizing: $T(n) = c_1 + \sum_{i=0}^{n-1} c_2 = c_1 + c_2 n$.

12

The Dictionary ADT

A dictionary contains a bunch of key-value pairs. Every key is unique (no duplicate keys allowed); the values can be arbitrary. A client can provide a key to look up the corresponding value.

Supported operations:

- **get:** Retrieves the value corresponding to the given key
- **put:** Updates the value corresponding to the given key
- **remove:** Removes the given key (and corresponding value)
- **containsKey:** Returns whether dictionary contains given key
- **size:** Returns the number of key-value pairs

Alternative names: map, lookup table

The Set ADT

A set is a collection of items. A set cannot contain any duplicate items: each item must be unique.

Supported operations:

- **add:** Adds the given item to the set
- **remove:** Removes the given item to the set
- **contains:** Returns 'true' if the set contains this item
- **size:** Returns the number of items in the set

Two questions:

1. Do sets (and dictionaries) need to 'order' items in some way?
2. We can implement a set on top of some dictionary: how?

Algorithm design practice: ArrayDictionary

Ex: consider your ArrayDictionary implementation; fill in table:

Operation	Description of algorithm	Big-Θ bound
get	Scan through the internal array, see if the key exists. Return value if it does.	$\Theta(n)$
put	Scan through the internal array, replace the value if we find the key-value pair. Otherwise, add the new pair at the end.	$\Theta(n)$
remove	Scan through the internal array and find the key-value pair. Remove it, and shift over the remaining elements.	$\Theta(n)$
containsKey	Scan through the array...	$\Theta(n)$

Idea: exploit additional property of keys

Observation: sometimes, keys are *comparable* and *sortable*.

Idea: Can we exploit the "sortability" of these keys?

Design practice: implementing get

Suppose we add the following **invariant** to ArrayDictionary:

SortedArrayDictionary invariant

The internal array, at all times, **must** remain sorted.

How do you implement get? What's the big-Θ bound?

The binary search algorithm

Core algorithm (in *pseudocode*):

```
public V get(K key):
    return search(key, 0, this.size)

private K search(K key, int lowIndex, int highIndex):
    if lowIndex > highIndex:
        key not found, throw an exception
    else:
        middleIndex = average of lowIndex and highIndex
        pair = this.array[middleIndex]

        if pair.key == key:
            return pair.value
        else if pair.key < key:
            return search(key, lowIndex, middleIndex)
        else if pair.key > key:
            return search(key, middleIndex + 1, highIndex)
```

Ex: model the worst-case runtime. Assume the time needed to compare two keys takes c time. Let $n = \text{highIndex} - \text{lowIndex}$.

Finding a closed form

Our answer: $T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T\left(\frac{n}{2}\right) & \text{Otherwise} \end{cases}$

Question: how do we find a closed form? Try unfolding?

$$T(n) = c + (c + (c + \ldots + (c + 1))) = \underbrace{c + c + \ldots + c}_{t = \text{Num times}} + 1$$

n	0	2	4	6	8	10	12	16	...	32	...	64
t	0	2	3	3	4	4	4	5	...	6	...	7

What's the relationship? $n \approx 2^{t+1}$

Solve for t: $t \approx \log(n) - 1$

Final model: $T(n) \approx c(\log(n) - 1) + 1$

So, we conclude: $T(n) \in \Theta(\log(n))$

SortedArrayDictionary

Fill in the remainder of this table for SortedArrayDictionary:

Operation	Description of algorithm	Big-Θ bound
get	Use binary search.	$\Theta(\log(n))$
put	Use binary search to find key. If it doesn't exist, insert into array.	$\Theta(n)$
remove	Use binary search to find key. Once found, remove it and shift over remaining elements.	$\Theta(n)$
containsKey	Use binary search.	$\Theta(\log(n))$