DSC 190

Data Structures & Algorithms

Lecture 1 | Part 1

Welcome

Advanced Data Structures and Algorithms

(for data science)

- Brand new.
- Modeled (partly) after CSE 100/101.
- But with more data science flavor.

Roadmap

- Advanced Data Structures
  - Dynamic Arrays
  - AVL Trees
  - Heaps
  - Disjoint Set Forests
- Nearest Neighbor Queries
  - KD-Trees
  - Locality Sensitive Hashing

Roadmap

- Strings
  - Tries and Suffix Trees
  - Knuth-Morris-Pratt and Rabin-Karp string search
- Algorithm Design
  - Divide and Conquer
  - Greedy Algorithms
  - Dynamic Programming (Viterbi Algorithm)
  - Backtracking, Branch and Bound
  - Linear Time Sorting; Sort with Noisy Comparator

Roadmap?

- Sketching and Streaming
  - Count-min-sketch
  - Bloom filters
  - Reservoir Sampling
- Theory of Computation
  - NP-Completeness and NP-Hardness
  - Computationally-hard problems in ML/DS

Roadmap??

- Other
  - Regular Expressions
  - Linear Programming
  - ?

Prerequisite Knowledge

- Python
- Basic Data Structures and Algorithms
  - DSC 30, DSC 40B[1]

[1] outside of Winter 2019

(syllabus)

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 1 | Part 2

Review of Time Complexity Analysis

Time Complexity Analysis

- Determine efficiency of code **without** running it.
- Idea: find a formula for time taken as a function of input size.

Advantages of Time Complexity

1. Doesn't depend on the computer.
2. Reveals which inputs are slow, which are fast.
3. Tells us how algorithm scales.

Counting Operations

- Abstraction: certain basic operations take **constant time**, no matter how large the input data set is.
- Example: addition of two integers, assigning a variable, etc.
- Idea: count basic operations

Example

```
def mean(numbers):
    total = 0
    n = len(numbers)
    for x in numbers:
        total += x
    return total / n
```

$T(n) =$

$= \Theta(n)$

Theta Notation, Informally

- $\Theta(\cdot)$ forgets constant factors, lower-order terms.

$$5n^3 + 3n^2 + 42 = \Theta(n^3)$$

Theta Notation, Informally

- $f(n) = \Theta(g(n))$ if $f(n)$ “grows like” $g(n)$.

$$5n^3 + 3n^2 + 42 = \Theta(n^3)$$

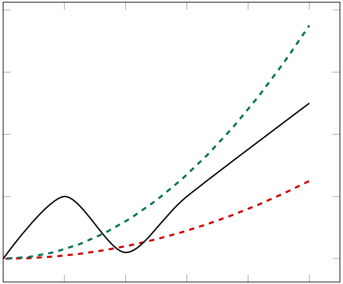
Theta Notation Examples

- $4n^2 + 3n - 20 = \Theta(n^2)$
- $3n + \sin(4\pi n) = \Theta(n)$
- $2^n + 100n = \Theta(2^n)$

Definition

We write $f(n) = \Theta(g(n))$ if there are positive constants N, c_1 and c_2 such that for all $n \geq N$:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$



Main Idea

If $f(n) = \Theta(g(n))$, then f can be "sandwiched" between copies of g when n is large.

Other Bounds

- $f = \Theta(g)$ means that f is both **upper** and **lower** bounded by factors of g.
- Sometimes we only have (or care about) upper bound or lower bound.
- We have notation for that, too.

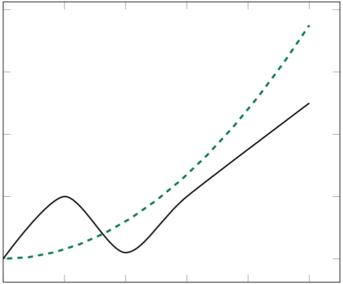
Big-O Notation, Informally

- Sometimes we only care about upper bound.
- $f(n) = O(g(n))$ if $f(n)$ “grows at most as fast” as $g(n)$.
- Examples:
  - $4n^2 = O(n^{100})$
  - $4n^2 = O(n^3)$
  - $4n^2 = O(n^2)$ and $4n^2 = \Theta(n^2)$

Definition

We write $f(n) = O(g(n))$ if there are positive constants N and c such that for all $n \geq N$:

$$f(n) \leq c \cdot g(n)$$



Big-Omega Notation

- Sometimes we only care about lower bound.
- Intuitively: $f(n) = \Omega(g(n))$ if $f(n)$ "grows at least as fast" as $g(n)$.

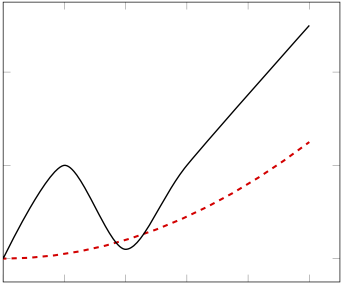
Ω $\geq$

- Examples:
  - $4n^{100} = \Omega(n^5)$
  - $4n^2 = \Omega(n)$
  - $4n^2 = \Omega(n^2)$ and $4n^2 = \Theta(n^2)$

Definition

We write $f(n) = \Omega(g(n))$ if there are positive constants N and c such that for all $n \geq N$:

$$c_1 \cdot g(n) \leq f(n)$$



Sums of Theta

- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then

$$\begin{aligned} f_1(n) + f_2(n) &= \Theta(g_1(n) + g_2(n)) \\ &= \Theta(\max(g_1(n), g_2(n))) \end{aligned}$$

- Useful for sequential code.

$\Theta(n^2)$

$\Theta(n^3)$

$\Theta(n^3)$

Products of Theta

- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then

$$f_1(n) \cdot f_2(n) = \Theta(g_1(n) \cdot g_2(n))$$

Example

```python
def foo(n):
    for i in range(3*n + 4, 5n**2 - 2*n + 5):
        for j in range(500*n, n**3):
            print(i, j)
```

$\Theta(n^2)$

$\Theta(n^3)$ per outer iter.

$\Theta(n^2) \cdot \Theta(n^3) = \boxed{\Theta(n^5)}$

[4, 12, 42, 21, 7]
0 1 2

t = 42

2

Linear Search

- **Given**: an array `arr` of numbers and a target `t`.
- **Find**: the index of `t` in `arr`, or `None` if it is missing.

```python
def linear_search(arr, t):
    for i, x in enumerate(arr):
        if x == t:
            return i
    return None
```

Exercise

What is the time complexity of `linear_search`?

The Best Case

- When `t` is the very first element.
- The loop exits after one iteration.
- $\Theta(1)$ time?

The Worst Case

- When `t` is not in the array at all.
- The loop exits after n iterations.
- $\Theta(n)$ time?

Time Complexity

- `linear_search` can take vastly different amounts of time on two inputs of the **same size**.
  - Depends on **actual elements** as well as size.
- There is no single, overall time complexity here.
- Instead we'll report **best** and **worst** case time complexities.

Best Case Time Complexity

- How does the time taken in the **best case** grow as the input gets larger?

Definition

Define $T_{\text{best}}(n)$ to be the **least** time taken by the algorithm on any input of size n.

The asymptotic growth of $T_{\text{best}}(n)$ is the algorithm's **best case time complexity**.

Best Case

- In `linear_search`'s **best case**, $T_{\text{best}}(n) = c$, no matter how large the array is.
- The **best case time complexity** is $\Theta(1)$.

Worst Case Time Complexity

- How does the time taken in the **worst case** grow as the input gets larger?

Definition

Define $T_{\text{worst}}(n)$ to be the **most** time taken by the algorithm on any input of size n.

The asymptotic growth of $T_{\text{worst}}(n)$ is the algorithm's **worst case time complexity**.

Worst Case

- In the worst case, `linear_search` iterates through the entire array.
- The **worst case time complexity** is $\Theta(n)$.

Faux Pas

- Asymptotic time complexity is not a **complete** measure of efficiency.
- $\Theta(n)$ is not always better than $\Theta(n^2)$.
- Why?

Faux Pas

- **Why?** Asymptotic notation "hides the constants".
- $T_1(n) = 1{,}000{,}000n = \Theta(n)$
- $T_2(n) = 0.00001n^2 = \Theta(n^2)$
- But $T_1(n)$ is **worse** for all but really large n.

Main Idea

Asymptotic time complexity is not the **only** way to measure efficiency, and it can be misleading.

Sometimes even a $\Theta(2^n)$ algorithm is better than a $\Theta(n)$ algorithm, if the data size is small.

DSC 190

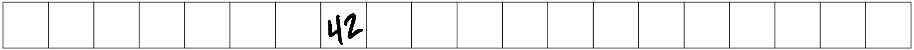
DATA STRUCTURES & ALGORITHMS

Lecture 1 | Part 3

Arrays and Linked Lists

Memory

- To access a value, we must know its **address**.



Sequences

- How do we store an **ordered sequence**?
  - e.g.: 55, 22, 12, 66, 60
- Array? Linked list?

Arrays

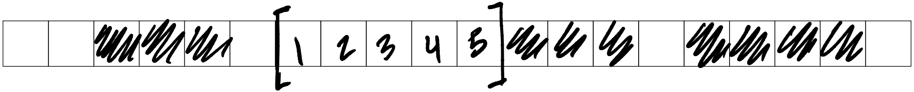
- Store elements **contiguously**.
  - e.g.: 55, 22, 12, 66, 60

			55	22	12	66	60												

- NumPy arrays are... arrays.

Allocation

- Memory is shared resource.
- A chunk of memory of fixed size has to be reserved (**allocated**) for the array.
- The size has to be known beforehand.

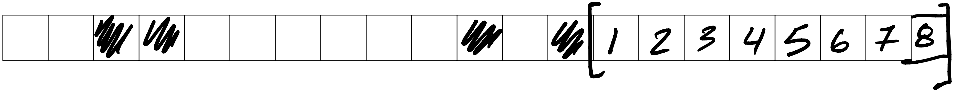


Arrays

- To access an element, we need its address.
- **Key:** Addresses are easily calculated.
  - For kth element: address of first + ($k \times 64$ bits)
- Therefore, arrays support $\Theta(1)$-time access.

Downsides of Arrays

- Homogeneous; every element must be same size.
- To **resize** the array, a totally new chunk of memory has to be found; old values copied over.



Array Time Complexities

- Retrieve kth element: $\Theta(1)$ (**good**).
- Append/pop element at end: $\Theta(n)$ (**bad**).
- Insert/remove in middle: $\Theta(n)$ (**bad**).
- Allocation: $\Theta(n)$ if initialized,[2] else $\Theta(1)$

[2]On Linux this is done lazily, as can be seen by timing `np.zeros`

Aside: `np.append`

```
>>> arr = np.array([1, 2, 3])
>>> np.append(arr, 4) # takes Theta(n) time!
array([1, 2, 3, 4])
```

Aside: np.append

```
results = np.array([])
for i in np.arange(100):
    result = run_simulation()
    results = np.append(results, result)
```

Aside: `np.append`

$$1+2+3+\ldots+n = \frac{n(n+1)}{2}$$

- This was **bad** code!
- We allocate/copy a **quadratic** number of elements:

$$\underbrace{1}_{\text{1st iter}} + \underbrace{2}_{\text{2nd iter}} + \underbrace{3}_{\text{3rd iter}} + \ldots + \underbrace{100}_{\text{last iter}} = \frac{100 \times 101}{2} = 5050$$

Aside: np.append

- Better: pre-allocate.

$\Theta(n)$

```
results = np.empty(100)
for i in np.arange(100):
    results[i] = run_simulation()
```

CPU

Cache

RAM

SSD

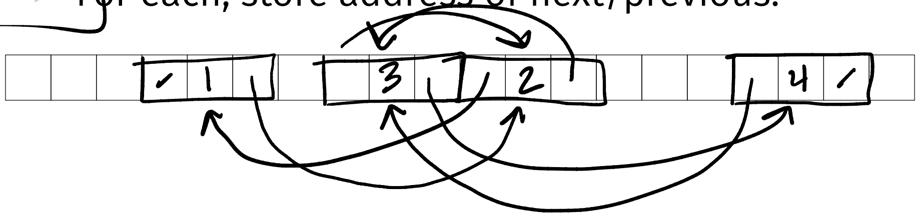
x, y

$x + z$

1, 2, 3, 4

(Doubly) Linked Lists

- Scatter elements throughout memory.
- For each, store address of next/previous.



Linked Lists

- Each element has an **address**.
- Keep track of the address of first/last elements.
- Have to **find** address of middle elements by looping.

Linked List Time Complexities

- Retrieve kth element:
  - $\Theta(k)$ if you don't know address (**bad**)[3]
  - $\Theta(1)$ if you do
- Append/pop element at start/end: $\Theta(1)$ (**good**).
- Insert/remove kth element:
  - $\Theta(k)$ if you don't know address (**bad**)
  - $\Theta(1)$ if you do
- Allocation not needed! (**good**)

[3] assumes search starts from beginning

Tradeoffs

- Arrays are better for numerical algorithms.
  - Arrays have good cache performance.
- Linked lists are better for stacks and queues.

Main Idea

Different data structures optimize for different operations.

DSC 190

Data Structures & Algorithms

Lecture 1 | Part 4

Dynamic Arrays

Motivation

- Can we have the best of both worlds?
- $\Theta(1)$ time access like an array.
- $\Theta(1)$ time append like a linked list.
- **Yes!** (sort of)

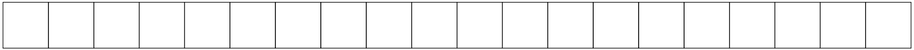
The Idea

- Allocate memory for an **underlying array**.
  - say, 512 elements
  - This is the **physical size**.
- To append element, insert into first unused slot.
  - Number of elements used is the **logical size**.
  - $\Theta(1)$ time.

The Idea

- We'll eventually run out of unused slots.
- Fix: allocate a new underlying array whose physical size is γ times as large.
  - γ is the **growth factor**.
  - Commonly, $\gamma = 2$; i.e., double its size.
  - Takes $\Theta(k)$ time, where k is current size.

Example



```
>>> arr = DynamicArray(initial_physical_size=4)
>>> arr.append(1)
>>> arr.append(2)
>>> arr.append(3)
>>> arr.append(4)
>>> arr.append(5)
```

(notebook)

DSC 190

Data Structures & Algorithms

Lecture 1 | Part 5

Amortized Analysis

Analysis

- Appending takes $\Theta(1)$ time *usually*...
- ...but takes $\Theta(k)$ time when we run out of slots.
  - Where k is current size of sequence.

The Key

- Resizing is expensive, but rare.
  - If $\gamma = 2$, each new resize is twice as expensive, but happens half as often.
- Thus, the **cost per append** is small.
- **Amortize** the cost over all previous appends.

Amortized Time Complexity

- The **amortized** time for an append is:

$$T_{\text{amort}}(n) = \frac{\text{total time for } n \text{ appends}}{n}$$

- We'll see that $T_{\text{amort}}(n) = \Theta(1)$.

Amortized Analysis

total time for *n* appends

=

total time for **non-growing** appends

+

total time for **growing** appends

Counting Growing Appends

- Want to calculate time taken by growing appends.
- First: how many appends caused a resize?
  - β: initial physical size
  - γ: growth factor

Counting Growing Appends

- Suppose initial physical size is $\beta = 512$, and $\gamma = 2$
- Resizes occur on append #:

$$512, 1024, 2048, 4096, \ldots$$

- In general, resizes occur on append #:

$$\beta\gamma^0, \beta\gamma^1, \beta\gamma^2, \beta\gamma^3, \ldots$$

Counting Growing Appends

- In a sequence of n appends, how many caused the physical size to grow?

- Simplification: Assume n is such that nth append caused a resize. Then, for some $x \in \{0, 1, 2, \ldots\}$:

$$n = \beta\gamma^x$$

- If $x = 0$ there was 1 resize; if $x = 1$ there were 2; etc.

Counting Growing Appends

- Solving for x:

$$x = \log_\gamma \frac{n}{\beta}$$

- Check: without assumption, $x = \lfloor \log_\gamma \frac{n}{\beta} \rfloor$
- Number of resizes is $\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$

Counting Growing Appends

- Number of resizes is $\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$
- Check with $\gamma = 2$, $\beta = 512$, $n = 400$
  - Correct # of resizes: 0
- Check with $\gamma = 2$, $\beta = 512$, $n = 1100$
  - Correct # of resizes: 2

Time of Growing Appends

- How much time was taken across all appends that caused resizes?
- Assumption: resizing an array with physical size k takes time $ck = \Theta(k)$.
  - c is a constant that depends on γ.

Time of Growing Appends

- Time for first resize: $c\beta$.
- Time for second resize: $c\gamma\beta$.
- Time for third resize: $c\gamma^2\beta$.
- Time for jth resize: $c\gamma^{j-1}\beta$.
- This is a **geometric progression**.

Time of Growing Appends

- Time for jth resize: $c\gamma^{j-1}\beta$.
- Suppose there are r resizes.
- Total time:

$$c\beta \sum_{j=1}^{r} \gamma^{j-1} = c\beta \sum_{j=0}^{r} \gamma^{j}$$

Recall: Geometric Sum

- From some class you've taken:

$$\sum_{p=0}^{N} x^p = \frac{1 - x^{N+1}}{1 - x}$$

- Example:

$$1 + 2 + 4 + 8 + 16 = \sum_{p=0}^{4} 2^p = \frac{1 - 2^5}{1 - 2} = 31$$

Time of Growing Appends

- Total time:

$$c\beta \sum_{j=0}^{r} \gamma^j = c\beta \frac{1 - \gamma^{r+1}}{1 - \gamma}$$

Time of Growing Appends

- Remember: in n appends there are $r = \lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$ resizes.

- Total time:

$$c\beta \frac{1 - \gamma^{r+1}}{1 - \gamma} = c\beta \frac{1 - \gamma^{\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 2}}{1 - \gamma}$$
$$= \Theta(n)$$

Amortized Analysis

total time for n appends

=

total time for **non-growing** appends

+

$\Theta(n)$ $\leftarrow$ total time for **growing** appends

Time of Non-Growing Appends

- In a sequence of n appends, how many are **non-growing**?

$$n - \left(\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1\right) = \Theta(n)$$

- Time for one such append: $\Theta(1)$.
- Total time: $\Theta(n) \times \Theta(1) = \Theta(n)$.

Amortized Analysis

total time for n appends

$=$

$\Theta(n)$ ← total time for **non-growing** appends

$+$

$\Theta(n)$ ← total time for **growing** appends

Amortized Time Complexity

- The **amortized** time for an append is:

$$\begin{aligned} T_{\text{amort}}(n) &= \frac{\text{total time for } n \text{ appends}}{n} \\ &= \frac{\Theta(n)}{n} \\ &= \Theta(1) \end{aligned}$$

Dynamic Array Time Complexities

- Retrieve kth element: $\Theta(1)$
- Append/pop element at start/end:
  - $\Theta(1)$ best case
  - $\Theta(n)$ worst case (where n = current size)
  - $\Theta(1)$ amortized
- Insert/remove in middle: $\Theta(n)$
  - May or may not need resize, still $\Theta(n)$!

DSC 190

Data Structures & Algorithms

Lecture 1 | Part 6

Practicalities

Advantages

- Great cache performance (it’s an array).
- Fast access.
- Don’t need to know size in advance of allocation.

Downsides

- Wasted memory.
- Expensive deletion in middle.

Implementations

- Python: `list`
- C++: `std::vector`
- Java: `ArrayList`

Exercise

Why do we need `np.array`? Python's `list` is a dynamic array, isn't that better?

In defense of `np.array`

- Memory savings are one reason.
- Bigger reason: using Python's `list` to store numbers does not have good **cache** performance.