DSC 40B

Lecture 12 : Partition and QuickSort

Agenda

Partition

Partitioning

- Given an array of n numbers and the index of a **pivot** p.
- Rearrange elements so that:
  - Everything < p is first.
  - Everything = p is next.
  - Everything > p is last.
- Return index of first element ≥ p.

Approach #1

- [77, 42, 11, 99, 0, 101], **pivot** is 11.
- [, , , , ,]
- [, , , , , **77**]
- [, , , , **42**, 77]
- skip for now
- [, , , **99**, 42, 77]
- [**0**, , , 99, 42, 77]
- [0, , **101**, 99, 42, 77]
- [0, **11**, 101, 99, 42, 77]

Approach #1

- [77, 42, 11, 99, 0, 101], **pivot** is 11.
- [, , , , ,]
- [, , , , , **77**]
- [, , , , **42**, 77]
- skip for now
- [, , , **99**, 42, 77]
- [**0**, , , 99, 42, 77]
- [0, , **101**, 99, 42, 77]
- [0, **11**, 101, 99, 42, 77]

How long does it take?

A: Constant

B: Log

C: Linear

D: n log n

E: n^2

Approach #1

- [77, 42, 11, 99, 0, 101], **pivot** is 11.
- [, , , , ,]
- [, , , , , **77**]
- [, , , , **42**, 77]
- skip for now
- [, , , **99**, 42, 77]
- [**0**, , , 99, 42, 77]
- [0, , **101**, 99, 42, 77]
- [0, **11**, 101, 99, 42, 77]

Issue: Not in-place.

Partition

- `partition` takes $\Theta(n)$ time.
- This is **optimal**.
- But we can use memory **more efficiently.**

Approach #2. Motivation

- Similar to selection sort, we'll use two barriers:
- **"Middle"** barrier:
  - Separates things $<$ pivot from things $\geq$
  - Index of first thing in "right"
- **"End"** barrier:
  - Separates **processed** from *processed*.
  - Index of first "unprocessed" thing.

[$<$ | middle $\geq$ | end ?]

[< | middle ≥ | end ?]

Example.

arr = [77, 42, 11, 99, 0, 101]

0 1 2 3 4 5

pivot_ix = 1

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| | 77, 42, 11, 99, 0, 101]

0 1 2 3 4 5

pivot_ix = 1

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| | 77, **42**, 11, 99, 0, 101]

0 1 2 3 4 5

pivot_ix = 1

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [77, 101, 11, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| |77, 101, 11, 99, 0, 42]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| 77, | **101**, 11, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| 77, 101, | **11**, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| 77, 101, | **11**, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Swap!

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| 11, 101, | 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, **0**, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, | **0**, 77, 99, | 101, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 0, 77, 99, 101, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 0, 77, 99, 101, 42]

0 1 2 3 4 5

pivot_ix = 5

What to do with the pivot?

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 0, **42**, 99, 101, 77]

0 1 2 3 4 5

pivot_ix = 2

Swap!

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 0, 42, 99, 101, 77]

0 1 2 3 4 5

pivot_ix = 2

Simplification: start by moving pivot to the end of the list

Loop Invariants

- After **each** iteration:
  - everything in `arr[start:middle_barrier]` is < pivot.
  - everything in `arr[middle_barrier:end_barrier]` is ≥ pivot.
  - everything in `arr[end_barrier:stop]` is “unprocessed”

```python
def in_place_partition(arr, start, stop, pivot_ix):
    def swap(ix_1, ix_2):
        arr[ix_1], arr[ix_2] = arr[ix_2], arr[ix_1]

    pivot = arr[pivot_ix]
    swap(pivot_ix, stop-1)
    middle_barrier = start
    for end_barrier in range(start, stop - 1):
        if arr[end_barrier] < pivot:
            swap(middle_barrier, end_barrier)
            middle_barrier += 1
        # else:
            # do nothing
    swap(middle_barrier, stop-1)
    return middle_barrier
```

Efficiency

- Also takes $\Theta(n)$ time.
- **No auxiliary memory required.**

Time Complexity Analysis

```python
import random
def quickselect(arr, k, start, stop):
    """Finds kth order statistic in numbers[start:stop])"""
    pivot_ix = random.randrange(start, stop)
    pivot_ix = partition(arr, start, stop, pivot_ix)
    pivot_order = pivot_ix + 1
    if pivot_order == k:
        return arr[pivot_ix]
    elif pivot_order < k:
        return quickselect(arr, k, pivot_ix + 1, stop)
    else:
        return quickselect(arr, k, start, pivot_ix)
```

Problem

- We **don't know** the size of the subproblem.
  - Is **random**, can be anywhere from 1 to $n - 1$.
- Difficult to write recurrence relation.

Good and Bad Pivots

- Some pivots are better than others.
- **Good**: splits array into roughly balanced halves.
- **Bad**: splits array into **wildly** unbalanced pieces.

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[77, 42, 11, 99, 101]

A: 77

B: 42

C: 11

D: 99

E: 101

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[77, 42, 11, 99, 101]

[77, 42, 11, 99] ~~101~~

A: 77

B: 42

C: 11

D: 99

E: 101

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

[**77, 42, 11**, **99**] if pivot is 99

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

[**77, 42, 11**, **99**] if pivot is 99

[**77**, **42, 11]** if pivot is 77 -> [**42, 11, 77]**

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

[**77, 42, 11**, **99**] if pivot is 99

[**77**, **42, 11]** if pivot is 77 -> **[42, 11, 77]**

[**42**, **11**] if pivot is 42 -> [**11**, **42**]

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[77, 42, 11, 99, **101**]

[77, 42, 11, **99**] if pivot is 99

[**77**, 42, 11] if pivot is 77 -> [42, 11, 77]

[**42**, 11] if pivot is 42 -> [11, **42**]

[**11**]

Worst Case

- Suppose we're searching for $k = 1$ (minimum).
- **Worst pivot**: the maximum.
- **Worst case**: use max as pivot every time.
- Subproblem size: $n - 1$.

Worst Case

- Every recursive call is on problem of size $n - 1$.
- $T(n) = T(n-1) + \Theta(n)$.
  - Solution: $\Theta(n^2)$.
- Intuitively, randomly choosing largest number as pivot every time is **very** unlikely!

$$\frac{1}{n} \times \frac{1}{n-1} \times \frac{1}{n-2} \times \cdots \times \frac{1}{3} \times \frac{1}{2} = \frac{1}{n!}$$

Equally Unlikely

- Pivot falls exactly in the middle, every time.
- Subproblems are of size $n/2$.
- $T(n) = T(n/2) + \Theta(n)$.
  - Solution: $\Theta(n)$.

Typically

- Pivot falls somewhere in the middle.
- Sometimes **good**, sometimes **bad**.
- But **good** pivots reduce problem size by **so much** that they make up for **bad** pivots.

Analogy

- You’re 100 miles away from home.
- You have a button that, if you press it, teleports you **1 mile** closer to home.
- How many times must you press it before you are 1 mile away from home?
- 99

Analogy

- You're 100 miles away from home.
- You have a button that, if you press it, teleports you **half the distance** closer to home.
- How many times must you press it before you are < 1 mile away from home?
- $\mathrm{Log}_2 100 \sim 6.64$

Analogy

- You’re 100 miles away from home.
- You have a button that, if you press it, teleports you **half the distance** to home with probability 1/2, does nothing with probability 1/2.
- How many times do you **expect** to press it before you are < 1 mile away from home?
- $2 * \log_2 100 \sim 13.28$

Quickselect

- The same reasoning applies to `quickselect`.
- If we always get a **good** pivot, time taken is $\Theta(n)$.
- If half the time we get a **bad** pivot, we expect:
  - To make *twice* as many recursive calls.
  - Take *twice* as much time as before.
- But $2\ \Theta(n) = \Theta(n)$.

Quickselect

- **Expected time** complexity: $\Theta(n)$.
- **Worst case**: $\Theta(n^2)$, but ***very unlikely.***

Median

- We can find the median in expected linear time with **quickselect**.

QuickSort

Last Time

- We saw `mergesort`.
- **Divide**: split list directly down the middle
- **Conquer**: sort each half
- **Combine**: merge sorted halves together

`merge` *does all the work*

- In `mergesort`, we are *lazy* when we divide.
- So we have to work to combine.

[4,1,3,2] → [4,1], [3,2]

`merge` *does all the work*

- In `mergesort`, we are *lazy* when we divide.
- So we have to work to combine.

[4,1,3,2] → [4,1], [3,2] → [4,3], [2,3]

`merge` *does all the work*

- In `mergesort`, we are *lazy* when we divide.
- So we have to work to combine.

[4,1,3,2] → [4,1], [3,2] → [4,3], [2,3] → [1,2,3,4]

What if?

- Suppose we divide so that everything in `left` is smaller than everything in `right`:
- After sorting, no need for merge.
- [5,1,3,8,6,2] → [1,3,2], [5,8,6]
- **This is what partition does!**

Quicksort

```python
def quicksort(arr, start, stop):
    """Sort arr[start:stop] in-place."""
    if stop - start > 1:
        pivot_ix = random.randrange(start, stop)
        pivot_ix = partition(arr, start, stop, pivot_ix)
        quicksort(arr, start, pivot_ix)
        quicksort(arr, pivot_ix+1, stop)
```

Time Complexity

- **Average case**: $\Theta(n \log n)$
- **Worst case**: $\Theta(n^2)$.
- Like with `quickselect`, worst case is **very rare**.

Mergesort vs Quicksort

- `Mergesort` has better worst case complexity.
- But in practice, `Quicksort` is often faster.
- Takes less memory, too.

Memory Requirements

- `merges` requires output array, **$\Theta(n)$ additional space.**
- `partition` **works in-place**, requires no additional space
  - Call stack for quicksort requires $\Theta(\log n)$ additional space.
- **Example**: sorting 3 GB of data with 4 GB of RAM.

Python sort, what does it use?

Python's default sort uses **Tim Sort**, which is a combination of both `mergesort` and `insertion` sort.

Thank you!

Do you have any questions?

CampusWire!