DSC 40B

Lecture 5 : Best, Worst Cases

Correction (mic)

https://docs.google.com/presentation/d/1JsgA03S0rMCrWD6oORdvjVB1cdt0HQHVHEx6zszylkA/edit?slide=id.g388f030f327_0_12#slide=id.g388f030f327_0_12

Agenda

Plan for the 5-6 lectures

- Best, Worst and Average cases.

The Movie problem

The Movie Problem

The Movie Problem

- **Given**: an array movies of movie durations, and the flight duration t
- **Find**: two movies whose durations add to t
  - If no two movies sum to t, return **None**.

Exercise

- Design a brute force solution to the problem. What is its time complexity?

Brute force

```
def find_movies(movies, t):
    n = len(movies)
    for i in range(n):
        for j in range(i + 1, n):
            if movies[i] + movies[j] == t:
                return (i, j)
    return None
```

Time Complexity

- It looks like there is a **best** case and **worst** case.
- How do we formalize this?

For the future...

- Can you come up with a better algorithm?
- What is the *best possible* time complexity?

Best and Worst Cases

Definition

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

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

Example 1: mean. Best case?

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

Example 1: mean

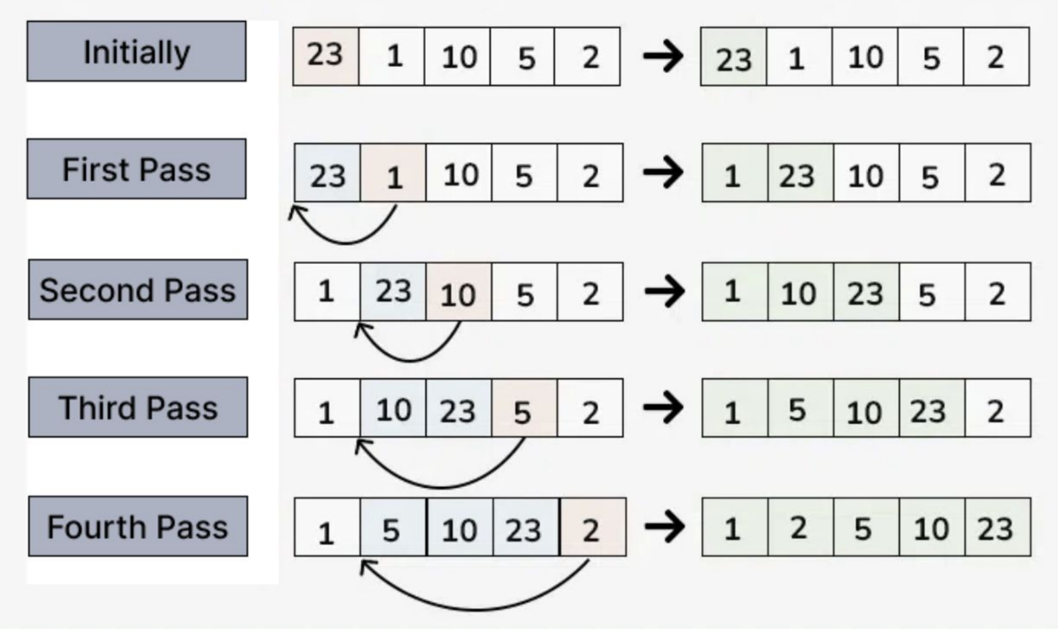
```python
def mean(arr):
    total = 0
    for x in arr:
        total += x
    return total / len(arr)
```

$T_{best}(n) = n$

Caution!

- The best case is **never**: "the input is of size one or empty".
- The best case is about the **structure** of the input, not its **size**.
- Not always constant time!
    - **Example**: sorting.

```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```



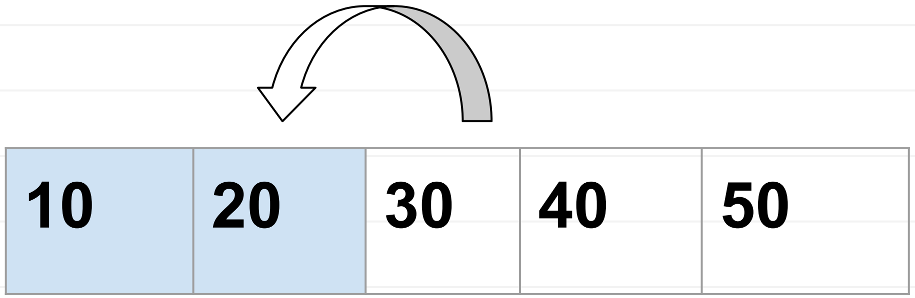
```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

10	20	30	40	50

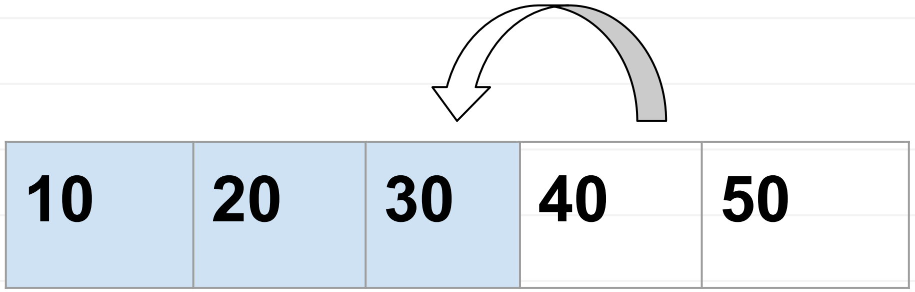
```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

10	20	30	40	50

```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```



```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```



```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

10	20	30	40	50

```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

$T_{best}(n) = n$

10	20	30	40	50

Time Complexity of mean

- Linear time **for all inputs**, $\Theta(n)$.
- Depends **only** on the array's **size**, n, not on its actual elements.

Example 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.
- **Example**: arr = [-3, -6, 7, 3, 0, 15, 4]
  - t is 7
  - 2 is returned (index of 7)

Cool function!

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

Exercise: Time complexity? Best?

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

A: Constant

B: n

C: Something else

Exercise: Time complexity? Worst?

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

Observation

- It looks like there are two extreme cases...

The Best Case

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

The Worst Case

- When the target, 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.
- It has no single, overall time complexity.
- 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?

Best Case

- In linear_search's **best case**, $T_{\text{best}}(n) = \boldsymbol{c}$, 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 $\Theta(n)$.

Exercise: times: Best and Worst

```python
def func(arr):
   n = len(arr)
   for x in arr:
      for y in arr:
         if x + y == 10
            return sum(arr)
```

A: $\Theta(1)$

B: $\Theta(n)$

C: $\Theta(n^2)$

D: $\Theta(n^3)$

Best Case

- Best case occurs when the first element is 5: 5 + 5 = 10
- `sum(arr)` takes $\Theta(n)$ time
- Exists, taking $\Theta(n)$ time in total

Worst Case

- Worst case occurs when no two numbers add to 10.
- Has to loop over all $\Theta(n^2)$ pairs.
- Worst case time complexity: $\Theta(n^2)$.
- **Note**: Not $\Theta(n^3)$ since the `sum(arr)` only runs once.

Note

- An algorithm like `linear_search` **doesn't** have one single time complexity.
- An algorithm like `mean` **does**, since the best and worst case time complexities coincide.

Main Idea

Reporting **best** and **worst** case time complexities gives us a richer of the performance of the algorithm.

Thank you!

Do you have any questions?

CampusWire!