Lecture 26: Sorting

CS 1110
Introduction to Computing Using Python

http://www.cs.cornell.edu/courses/cs1110/2019sp

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
Plan of Attack

- Insertion Sort
- Partition
- Quick Sort
Searching is a good motivation for Sorting

Example: 500 CS 1110 Prelims have been scanned

Grading Session: “Hey, this scan is hard to read.”

Task: go through 500 Exams, find the bad scan

Do you want this job?

Are the exams in any order? No….

Fine, go through them all.

10 minutes later “Hey, this scan is hard to read…”

Now you really wish they were in order…
Sorting: Arranging in Ascending Order

#1
3 2 K 9 7

#2
3 2 K 9 7

#3
2 3 K 9 7

#4
2 3 K 9 7

#5
2 3 K 9 7

#6
2 3 7 9 K
**Insertion Sort**

PRE: \[ b \] *(unknown values)*

\[ k = 0 \]

INV: \[ b \text{ sorted} \] *(unknown)*

\[ \text{while } k < n: \]

\# Push \( b[k] \) down into its
\# sorted position in \( b[0..k] \)

\[ k = k + 1 \]

POST: \[ b \text{ sorted} \]
def push_down(b, k):
    while k > 0:
        if b[k-1] > b[k]:
            swap(b, k-1, k)
        k = k - 1
    k = 0
while k < n:
    push_down(b, k)
    k = k + 1

How many swaps will there be?

A: 1
B: 2
C: 3
D: 4
E: 5
def push_down(b, k):
    while k > 0:
        if b[k-1] > b[k]:
            swap(b, k-1, k)
        k = k - 1
    k = 0
    while k < n:
        push_down(b, k)
        k = k + 1

3 swaps!
The Importance of Helper Functions

```python
def push_down(b, k):
    while k > 0:
        if b[k-1] > b[k]:
            swap(b, k-1, k)
        k = k - 1

k = 0
while k < n:
    push_down(b, k)
    k = k + 1
```

**VS**

```python
k = 0
while k < n:
    j = k
    while j > 0:
        if b[j-1] > b[j]:
            temp = b[j]
            b[j] = b[j-1]
            b[j-1] = temp
        j = j - 1
    k = k + 1
```

Can you understand all this code below?

Also: Is this how you want to sort 500 exams?
def push_down(b, k):
    while k > 0:
        if b[k-1] > b[k]:
            swap(b, k-1, k)
            k = k - 1
    k = k - 1

while k < n:
    push_down(b, k)
    k = k + 1

Nested loops multiply the number of operations required. We need to compare b[k] to all elements. \( \sim n \) operations

Iterating through a sequence of length \( n \) requires \( n \) operations:
push_down called \( n \) times

Approximately how many operations/swaps does this take?

A: \( \sim 1 \)  B: \( \sim n \)
C: \( \sim n^2 \)  D: \( \sim n^3 \)
E: I don’t know
Clicker Answer:
Algorithm Complexity

def push_down(b, k):
    while k > 0:
        if b[k-1] > b[k]:
            swap(b,k-1,k)
        k = k-1
    k = 0
while k < n:
    push_down(b,k)
    k = k+1

Approximately how many operations/swaps does this take?

A: $\sim 1$ operation
B: $\sim n$ operations
C: $\sim n^2$ operations \textbf{CORRECT}
D: $\sim n^3$ operations
E: I don’t know
def push_down(b, k):
    while k > 0:
        if b[k-1] > b[k]:
            swap(b, k-1, k)
        k = k - 1

k = 0
while k < n:
    push_down(b, k)
    k = k + 1

Total Swaps:
0 + 1 + 2 + ... (n-1) = n * (n-1)/2

Insertion sort requires n*n operations
https://www.youtube.com/watch?v=xxcpvCGrCBc
Plan of Attack

- Insertion Sort
- **Partition**
  - Overview in class
  - Details (optional!) are at the end of this lecture
- Quick Sort
What if we had an algorithm that could partition a list segment based on some value?

**def partition(b, h, k):**

Like separating positives from negatives but instead separating by the first value in the segment. *We can use this to make a faster sort!*
Plan of Attack

- Insertion Sort
- Partition
- Quick Sort

Let's partition the exams into 2 piles: A-M & N-Z.
   Now let's partition the A-M pile into A-F & G-M
   Now let's partition the A-F pile into A-C & D-F
   eventually the exams will all be sorted!
Sorting with Partitions

**Idea:** Pick a *pivot* element \( x \)

**Partition sequence into \( \leq x \) and \( \geq x \)**

Now Partition this and this, too

*Keep recursing…*

Sorted!

https://www.youtube.com/watch?v=m1PS8IR6Td0
def quick_sort(b, h, k):
    """Sort the array fragment b[h..k]"""
    if k<=h:
        return
    i = partition(b, h, k)
    # INV: b[h..i-1] <= b[i] <= b[i+1..k]
    # Sort b[h..i-1] and b[i+1..k]
    quick_sort(b, h, i-1)
    quick_sort(b, i+1, k)
How Fast is QuickSort?

If you're lucky, each partition will split the list in half. **Runtime: n * log(n)**

If you're **not** lucky, each partition removes only 1 element from the list. **Runtime = n * n**

In practice, you get lucky.

and on and on and on…
Quicksort in the real world

```
DEFINITION QUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
        NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
        THE BIGGER ONES GO IN A NEW LIST
        THE EQUAL ONES GO INTO, Uh
        THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
        THIS IS LIST A
        THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
        CALL IT LIST, Uh, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
        RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
        AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```
Appendix: Partition Details

You are **not** responsible for knowing the details of the following slides but they are a good (but difficult*) case study of how to develop an algorithm using loop invariants

* certainly more difficult than anything we would ask you on the Final Exam
Partition Algorithm

- Given a list segment $b[h..k]$ with some pivot value $x$ in $b[h]$:

  
  ![Diagram](image)

  - Swap elements of $b[h..k]$ and store in $i$ to satisfy postcondition:

    
    ![Diagram](image)

  

- **Example:**

  
  ![Diagram](image)

  - Called the **pivot value**
  - not a variable
  - $x$ = whatever value is in $b[h]$
Partition: What’s the Invariant?

• Given a list segment \( b[h..k] \) with some pivot value \( x \) in \( b[h] \):

\[
\text{pre: } \quad b[h..k] \quad \begin{array}{c}
h \quad x \quad ? \quad k \end{array}
\]

\[
\text{INV: } \quad b[h..k] \quad \begin{array}{cccc}
h \quad i \quad i+1 \quad j \quad k \end{array}
\]

\[
\begin{array}{c}
<= x \quad x \quad ? \quad >= x
\end{array}
\]

• Swap elements of \( b[h..k] \) and store in \( i \) to satisfy postcondition:

\[
\text{post: } \quad b[h..k] \quad \begin{array}{cc}
h \quad i \quad i+1 \quad k \end{array}
\]

\[
\begin{array}{c}
<= x \quad x \quad >= x
\end{array}
\]
Partition: What’s the Invariant?

• Given a list segment $b[h..k]$ with some pivot value $x$ in $b[h]$:  

  ```plaintext
  pre: b
  h x ? k
  Initially i = h, j = k+1
  
  INV: b
  h i i+1 j k
  <= x x ? >= x
  ```

• Swap elements of $b[h..k]$ and store in $i$ to satisfy postcondition:

  ```plaintext
  post: b
  h i i+1 k
  <= x x >= x
  ```
Partition: What’s the Invariant?

- Given a list segment \( b[h..k] \) with some pivot value \( x \) in \( b[h] \):

  - Swap elements of \( b[h..k] \) and store in \( i \) to satisfy postcondition:

    - **Pre:** \( b[h..k] \)
      - \( b[h..k] \)
      - \( b[h..k] \)
      - \( b[h..k] \)
    - **Post:** \( b[h..k] \)
      - \( b[h..k] \)
      - \( b[h..k] \)
      - \( b[h..k] \)

    - **Inv:** \( b[h..k] \)
      - \( b[h..k] \)
      - \( b[h..k] \)
      - \( b[h..k] \)

    - Eventually \( j = i+1 \)

- Swap elements of \( b[h..k] \) and store in \( i \) to satisfy postcondition:
Partition Algorithm Implementation

**def** partition(b, h, k):

```python
def partition(b, h, k):
    i = h
    j = k + 1
    x = b[h]
    while i < j - 1:
        if b[i + 1] >= x:
            # Move b[i+1] to end of block.
            swap(b, i + 1, j - 1)
            j = j - 1
        else:
            # b[i+1] < x
            swap(b, i, i + 1)
            i = i + 1
    return i
```

**pre:**

<table>
<thead>
<tr>
<th>b</th>
<th>h</th>
<th>?</th>
<th>k</th>
</tr>
</thead>
</table>

**INV:**

| b | <= x | x | >= x |

**while** i < j - 1:

*if* b[i + 1] >= x:

- # Move b[i+1] to end of block.
  - swap(b, i + 1, j - 1)
  - j = j - 1

*else:* # b[i+1] < x

- swap(b, i, i + 1)
- i = i + 1

**post:**

| b | <= x | x | >= x |

**return** i
```python
def partition(b, h, k):
i = h
j = k + 1
x = b[h]

while i < j - 1:
    if b[i + 1] >= x:
        # Move to end of block.
        swap(b, i + 1, j - 1)
        j = j - 1
    else:
        # b[i+1] < x
        swap(b, i, i + 1)
        i = i + 1

return i
```