Lecture 26

Advanced Sorting
# Announcements for This Lecture

## Exam & Assignments

- **Prelim, TONIGHT at 7:30**
  - See webpage for rooms
  - Make-ups all resolved!
- Graded by **next** Thursday
- A6 is now graded
  - **Mean**: 92.3  **Median**: 95
  - **Time**: 16.4hrs  **Std Dev**: 9hr
  - **A**: 89 (80%), **B**: 70 (17%)
- A7 focus of last week of class

## Optional Videos

- **ALL** all are now posted
  - **Lesson 30** for **today**
  - **Lesson 28** is next week
Recall Our Problem

- Both insertion, selection sort are nested loops
  - **Outer loop** over each element to sort
  - **Inner loop** to put next element in place
  - Each loop is n steps. \( n \times n = n^2 \)
- To do better we must eliminate a loop
  - But how do we do that?
  - What is like a loop? **Recursion!**
  - First need an *intermediate* algorithm
The Partition Algorithm

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

  \[\begin{array}{c}
h \\
x \\
? \\
k\end{array}\]

  **Start:**

  \[\begin{array}{c}
b \\
x \\
\end{array}\]

- Swap elements of b[h..k] to get this answer

  \[\begin{array}{c}
h \\
i \\
i+1 \\
k\end{array}\]

  **Goal:**

  \[\begin{array}{c}
b \\
\leq x \\
x \\
\geq x \\
k\end{array}\]

  \[\begin{array}{c}
3 & 5 & 4 & 1 & 6 & 2 & 3 & 8 & 1
\end{array}\]

  \[\begin{array}{c}
h \\
i \\
k\end{array}\]

  **change:**

  \[\begin{array}{c}
b \\
3 & 5 & 4 & 1 & 6 & 2 & 3 & 8 & 1
\end{array}\]

  \[\begin{array}{c}
h \\
i \\
k\end{array}\]

  \[\begin{array}{c}
i+1 \\
3 & 5 & 4 & 6 & 3 & 8
\end{array}\]

  **into**

  \[\begin{array}{c}
b \\
1 & 2 & 1 & 3 & 5 & 4 & 6 & 3 & 8
\end{array}\]

  \[\begin{array}{c}
h \\
i \\
k\end{array}\]

  \[\begin{array}{c}
1 & 2 & 3 & 1 & 3 & 4 & 5 & 6 & 8
\end{array}\]

- **x** is called the pivot value
  - x is not a program variable
  - denotes value initially in b[h]
Designing the Partition Algorithm

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

  \[
  \begin{array}{c|c}
  h & k \\
  \hline
  \text{Start: } b & x \? \end{array}
  \]

- Swap elements of $b[h..k]$ to get this answer

  \[
  \begin{array}{c|c|c|c}
  h & i & i+1 & k \\
  \hline
  \text{Goal: } b & \leq x & x & \geq x \\
  \end{array}
  \]

- In Progress:

  \[
  \begin{array}{c|c|c|c|c}
  h & i & j & k \\
  \hline
  \text{In-Progress: } b & \leq x & x & \? & \geq x \\
  \end{array}
  \]

Indices $b, h$ important!
Might partition only part
Implementing the Partition Algorithm

```python
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    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
```

**partition(b,h,k), not partition(b[h:k+1])**

Remember, slicing always copies the list!

We want to partition the **original** list
def partition(b, h, k):
    
    """Partition list b[h..k] around a pivot x = b[h]""
    
    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
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    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
def partition(b, h, k):
    '''Partition list b[h..k] around a pivot x = b[h]'''
    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
**Partition Algorithm Implementation**

```python
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]

    while i < j:
        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
```

---

---

11/21/23  Advanced Sorting  10
Why is this Useful?

• Will use this algorithm to replace inner loop
  ▪ The inner loop cost us n swaps every time

• Can this reduce the number of swaps?
  ▪ Worst case is k-h swaps
  ▪ This is n if partitioning the whole list
  ▪ But less if only partitioning part

• **Idea:** Break up list and partition only part?
  ▪ This is *Divide-and-Conquer!*
Sorting with Partitions

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

  \[
  \begin{array}{cccc}
  h & x & ? & k \\
  \end{array}
  \]

  \textbf{Start:}

  \textbf{Goal:}

  \[
  \begin{array}{cccc}
  h & \leq x & x & \geq x & k \\
  \end{array}
  \]

• Swap elements of $b[h..k]$ to get this answer

  \[
  \begin{array}{cccc}
  h & i & i+1 & k \\
  \end{array}
  \]

  \textbf{Partition Recursively}

Recursive partitions = sorting

- Called \textbf{QuickSort} (why???)
- Popular, fast sorting technique
Sorting with Partitions

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

  \[
  \begin{array}{c|c|c}
  h & x & ? \\
  \end{array}
  \]

  \[
  \begin{array}{c|c|c|c|c}
  h & i & i+1 & k \\
  \end{array}
  \]

- Swap elements of $b[h..k]$ to get this answer:

  \[
  \begin{array}{c|c|c|c|c}
  h & i & i+1 & k \\
  \end{array}
  \]

  \[
  \begin{array}{c|c|c|c|c}
  y & ? & x & >= x \\
  \end{array}
  \]

Partition Recursively

Recursive partitions = sorting
- Called **QuickSort** (why???)
- Popular, fast sorting technique
Sorting with Partitions

- Given a list segment b[h..k] with some value x in b[h]:
  
  \[
  \begin{array}{c|c|c|c|c}
  h & x & \text{?} & k \\
  \end{array}
  \]

- Swap elements of b[h..k] to get this answer
  
  \[
  \begin{array}{c|c|c|c|c|c}
  h & i & i+1 & k \\
  \end{array}
  \]

  \[
  \begin{array}{c|c|c|c|c|c}
  \leq y & y & \geq y & x & \geq x \\
  \end{array}
  \]

  Partition Recursively

  Recursive partitions = sorting
  - Called \textbf{QuickSort} (why???)
  - Popular, fast sorting technique
def quick_sort(b, h, k):
    """Sort the array fragment b[h..k]"""
    if b[h..k] has fewer than 2 elements:
        return
    j = partition(b, h, k)
    # b[h..j–1] <= b[j] <= b[j+1..k]
    # Sort b[h..j–1] and b[j+1..k]
    quick_sort(b, h, j–1)
    quick_sort(b, j+1, k)

• Worst Case:
  array already sorted
  ▪ Or almost sorted
  ▪ n^2 in that case

• Average Case:
  array is scrambled
  ▪ n log n in that case
  ▪ Best sorting time!
So Does that Solve It?

- Worst case still seems bad! Still $n^2$
  - But only happens in small number of cases
  - Just happens that case is common (already sorted)
- Can greatly reduce issue with randomization
  - Swap start with random element in list
  - Now pivot is random and already sorted unlikely

Start: $b \begin{array}{ccc} x & ? & y & ? \end{array}$
So Does that Solve It?

- Worst case still seems bad! Still $n^2$
  - But only happens in small number of cases
  - Just happens that case is common (already sorted)
- Can greatly reduce issue with randomization
  - Swap start with random element in list
  - Now pivot is random and already sorted unlikely

Makes it “good enough” for most applications
Can We Do Better?

• Recursion seems to be the solution
  ▪ Partitioned the list into two halves
  ▪ Recursively sorted each half

• How about a traditional \textit{divide-and-conquer}?
  ▪ \textbf{Divide} the list into two halves
  ▪ \textbf{Recursively sort} the two halves
  ▪ \textbf{Combine} the two sort halves

• How do we do the last step?
Combining Two Sorted Lists

1 4 5 6 11 12

2 3 7 8 9 10
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Pick from list with the least
Combining Two Sorted Lists

Finish off remaining list
Combining Two Sorted Lists

Finish off remaining list

1
2
3
4
5
6
7
8
9
10
11
12
Merge Sort

```python
def merge_sort(b, h, k):
    """Sort the array fragment b[h..k]""
    if b[h..k] has fewer than 2 elements:
        return
    # Divide and recurse
    mid = (h+k)//2
    merge_sort(b, h, m)
    merge_sort(b, m+1, k)
    # Combine
    merge(b,h,mid,k) # Merge halves into b
```

- Seems simpler than **qsort**
  - Straight-forward d&c
  - Merge easy to implement
- What is the **catch**?
  - Merge requires a **copy**
  - We did not allow copies
  - Copying takes n steps
  - But so does merge/partition
- **n log n ALWAYS**
def merge_sort(b, h, k):

    """Sort the array fragment b[h..k]"""
    if b[h..k] has fewer than 2 elements:
        return
    # Divide and recurse
    mid = (h+k)//2
    merge_sort(b, h, m)
    merge_sort(b, m+1, k)
    # Combine
    merge(b,h,mid,k) # Merge halves into b

• Seems simpler than qsort
  ▪ Straight-forward d&c
  ▪ Merge easy to implement

• What is the catch?
  ▪ Merge requires a copy
  ▪ We did not allow copies
  ▪ Copying takes n time
  ▪ But so does merge/partition

• n log n ALWAYS

Proof beyond scope of course
What Does Python Use?

• The sort() method is **Timsort**
  - Invented by Tim Peters in 2002
  - Combination of insertion sort and merge sort

• Why a combination of the two?
  - Merge sort requires copies of the data
  - Copying pays off for large lists, but not small lists
  - Insertion sort is not that slow on small lists
  - Balancing two properly still gives \( n \log n \)
What Does Python Use?

- The `sort()` method is **Timsort**
  - Invented by Tim Peters in 2002
  - Combination of insertion sort and merge sort
- Why a combination of the two?
  - Merge sort requires copies of the data
  - Copying pays off for large lists, but not small lists
  - Insertion sort is not that slow on small lists
  - Balancing two properly still gives $n \log n$
What Does Python Use?

- The `sort()` method is **Timsort**
  - Invented by Tim Peters in 2002
  - Combination of insertion sort and merge sort
- Why a combination of the two?
  - Merge sort requires copies of the data
  - Copying pays off for large lists, but not small lists
  - Insertion sort is not that slow on small lists
  - Balancing two properly still gives $n \log n$
What Does Python Use?

• The `sort()` method is **Timsort**
  - Invented by Tim Peters in 2002
  - Combination of insertion sort and merge sort

• Why a combination
  - Merge sort requires copies of the data
  - Copying pays off for large lists, but not small lists
  - Insertion sort is not that slow on small lists
  - Balancing two properly still gives $n \log n$

This strategy allows AI to find even **better** sorting algorithms