PDSA Notes

by

Gagneet Kaur

Date
January 16, 2022

PAGE NO.
DATE:

WEEK 3

Quick Sort

SHORTCOMINGS OF MERGE SORT

- Merge needs to create a new list to hold the merged elements
  - No obvious way to efficiently merge two lists in place
  - Extra storage can be costly
- Inherently recursive
  - Recursive calls and returns are expensive
- Merging happens because elements in the left half need to move to the right half and vice-versa
  - Consider an ~~output~~ input of the form [0,2,4,6,1,3,5,9]
- Can we divide the list so that everything on the left is smaller than everything on the right?
  - No need to merge!

DIVIDE & CONQUER WITHOUT MERGING

- Suppose the median of L is m

- Move all values $\leq m$ to left half of L
  - Right half has values $> m$
- Recursively sort left and right halves
  - L is now sorted, no merge!
- Recurrence : $T(n) = 2T(n/2) + n$
  - Rearrange in a single pass, time $O(n)$
- So, $T(n)$ is $O(n \log n)$
- How do we find the median?
  - Sort and pick up the middle element
  - But our aim is to sort the list!
- Instead pick some value in L - pivot
  - Split L with respect to the pivot element

QUICKSORT [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively, sort the two partitions

PAGE NO.
DATE:

High Level view of Quick Sort

→ Input List

[43, 32, 22, 78, 63, 57, 91, 13]

- Identify pivot
- Mark lower elements & upper elements

→ Rearrange the elements as lower - pivot - upper

[32, 22, 13, 43, 78, 63, 57, 91]

→ Recursively Sort the Lower & upper partitions.

PARTITIONING

- Scan the list from left to right
- Four segments : Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - → If it is larger than the pivot, extend Upper to include this element
  - → If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position

13	32	22	43	78	63	57	91
			↑				↑

- Pivot is always the first element.

- Maintain two indices to mark the end of the lower and upper segments.
- After partitioning, exchange the pivot with the last element of the lower segment.

CODE:

```
def quicksort (L, l, r) :                    # Sort L[l:r]
    if (r-l <= 1) :
        return (L)
    (pivot, lower, upper) = (L[l], l+1, l+1)
    for i in range (l+1, r) :
        if L[i] > pivot :                    # Extend upper segment
            upper = upper + 1
        else :
            (L[i], L[lower]) = (L[lower], L[i])
            (lower, upper) = (lower+1, upper+1)
    (L[l], L[lower-1]) = (L[lower-1], L[l])
    lower = lower - 1
    quicksort (L, l, lower)
    quicksort (L, lower+1, upper)
    return (L)
```

ANALYSIS OF QUICKSORT

- Partitioning with respect to the pivot takes time $O(n)$
- If the pivot is the median
  → $T(n) = 2T(n/2) + n$

→ $T(n)$ is $O(n \log n)$

- Worst case? Pivot is maximum or minimum
  - → Partitions are of size $0, n-1$
  - → $T(n) = T(n-1) + n$
  - → $T(n) = n + (n-1) + \ldots + 1$
  - → $T(n)$ is $O(n^2)$
- Already sorted array: worst case!
- However, average case is $O(n \log n)$
- Sorting is a rare situation where we can compute this
  - → Values don't matter, only relative order is important
  - → Analyze behaviour over permutations of $\{1, 2, \ldots, n\}$
  - → Each input permutation equally likely
- Expected running time is $O(n \log n)$.

RANDOMIZATION

- Any fixed choice of pivot allows us to construct worst case input
- Instead, choose pivot position 'randomly' at each step
- Expected running time is again $O(n \log n)$.

ITERATIVE QUICKSORT

- Recursive calls work on disjoint segments
  → No recombination of results is required
- Can explicitly keep track of left & right endpoints of each segment to be sorted

QUICKSORT IN PRACTICE

- In practice, quicksort is very fast
- Very often the default algorithm used for in-built sort functions
  → sorting a column in a spreadsheet
  → library sort function in a programming language

SUMMARY

- ~~The worst case~~ Quicksort uses divide & conquer, like merge sort.
- By partitioning the list carefully, we avoid a merge step
  → This allows in place sort
- We can also provide an iterative implementation to avoid the cost of recursive calls.

- The partitioning strategy we described is not only one used in the literature
  → Can build the lower & upper segments from opposite ends and meet in the middle
- The worst case complexity of quicksort is $O(n^2)$
- However, the avg. case is $O(n \log n)$
- Randomly choosing the pivot is a good strategy to beat the worst case inputs
- Quicksort works in-place & can be implemented iteratively
- Very fast in practice, and often used for built-in sorting functions
  → Good example of a situation when the worst case upper bound is pessimistic

Date
January 11, 2022

PAGE NO.
DATE:

SORTING
CONCLUDING REMARKS

STABLE SORTING

- Often list value are tuples
  - → Rows from a table, with multiple col/attributes
  - → A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll no.
- If we now sort by name, will all students with the same name remain in sorted order with respect to roll number?
- STABILITY of sorting is crucial in many applications
- Sorting on column B should not disturb sorting on column A
- The quicksort implementation we described is not stable
  - → Swapping values while partitioning can disturb existing sorted order
- Merge sort is stable if we merge carefully

→ Do not allow elements from the right to overtake elements on the left
→ While merging, prefer the left list while breaking ties

OTHER CRITERIA

- Minimizing data movement
  - → Imagine each element is a heavy carton
  - → Reduce the effort of moving values around

BEST SORTING ALGORITHM

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case.
- Merge Sort is typically used for "external" sorting
  - → Database tables that are too large to store in memory all at once
  - → Retrieve in parts from the disk & write back
- Other $O(n \log n)$ algorithms exist - heapsort
- Sometimes hybrid strategies are used
  - → Use divide & conquer for large n
  - → Switch to insertion sort when n becomes small ($n < 16$)

Date January 12, 2022

PAGE NO.
DATE:

Data Structures

DIFF. B/W LISTS AND ARRAYS

SEQUENCES

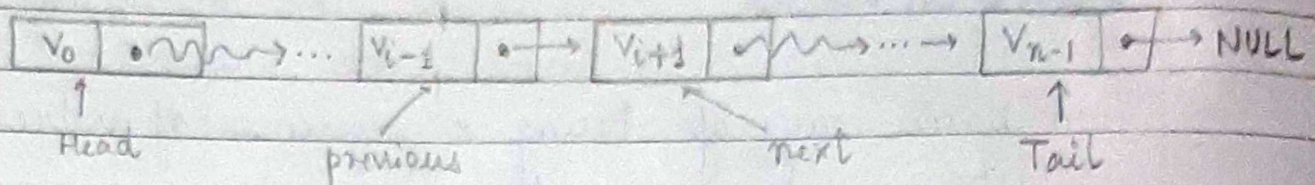
- Two basic ways of storing a sequence of values
  - → Lists
  - → Arrays
- What's the difference?

LISTS	ARRAYS
1) Flexible Length	1) Fixed size
2) Easy to modify the structure	2) Allocate a contiguous block of memory
3) Values are scattered in memory	3) Supports random access

LISTS

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
  - → 'Linked' list

- Easy to modify
  - Inserting and deletion is easy via local "plumbing"
  - Flexible size
- Need to follow links to access A[i]
  - Takes time O(i)



ARRAYS

- Fixed size, declared in advance
- Allocate a contiguous block of memory
  - n times the storage for a single value
- 'Random' access
  - Compute offset to A[i] from A[0]
  - Accessing A[i] takes constant time, independent of i
- Inserting and deleting elements is expensive
  - Expanding and contracting requires moving O(n) elements in the worst case

Index	A[0]	A[1]	...	A[i]	...	A[n-1]
Value	V_0	V_1		V_i		V_{n-1}

OPERATIONS

- Exchange A[i] and A[j]
  - Constant time for arrays
  - O(n) for lists

- Deleting A[i], insert v after A[i]
  - constant time for lists if we are already at A[i]
  - O(n) for arrays

- Need to keep implementation in mind when analyzing data structures
  - For instance, can we use binary search to insert in a sorted sequence?
  - Either search is slow, or insertion is slow, still O(n)

SUMMARY

- Sequences can be stored as lists or arrays
- Lists are flexible but accessing an element is O(n)
- Arrays support random access but are difficult to expand, contract
- Algorithm analysis needs to take into account the underlying implementation
- How does it work in Python?
  - Is the built-in list type in Python really a 'linked' list?
  - Numpy libraries provides arrays - are these faster than lists?

Designing a Flexible List

IMPLEMENTING LISTS IN PYTHON

→ Python class Node

→ A list is a sequence of nodes
  ⇒ self.value is the stored value
  ⇒ self.next points to next node

→ Empty List?
  ⇒ self.value is None

→ Creating Lists
  ⇒ l1 = Node() - empty list
  ⇒ l2 = Node(5) - singleton list
  ⇒ l1.isempty() == True
  ⇒ l2.isempty() == False

CODE:

```
class Node :
    def __init__(self, v= None)
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None :
            return (True)
```

```
    else :
        return (False)
```

APPENDING TO A LIST

→ Add v to the end of list l

→ If l is empty, update l.value from None to v

→ If at last value, l.next is None
  ⇒ Point next at new node with value v

→ Otherwise, recursively append to rest of list

→ Iterative implementation
  ⇒ If empty, replace l.value by v
  ⇒ Loop through l.next to end of list
  ⇒ Add v at the end of the list

CODE :

```
def appendi(self, v):
    if self.isempty():
        self.value = v
        return

    temp = self
    while temp.next != None:
        temp = temp.next

    temp.next = Node(v)
    return
```

INSERT AT THE START OF THE LIST

- Want to insert v at head
- Create a new node with v
- Cannot change where the head points!
- Exchange the values v_0, v
- Make new node point to head.next
- Make head.next point to new node

Appending to a List

- Create a new node with v
- Exchange the values v_0, v
- Make new node point to head.next
- Make head.next point to new node

Delete a value v

- Remove the first occurence of v
- Scan list for first v - look ahead at next node
- If next node value is v, bypass it

→ Can't bypass the first node in the list
  ↳ Instead, copy the second node value to head
  ↳ Bypass second node

→ Recursive implementation

- ~~Exercise~~ CODE :

```
def insert (self, v):
    if self.isempty():
        self.value = v
        return

    newnode = Node(v)
    (self.value, newnode.value) = (newnode.value, self.value)
    (self.next, newnode.next) = (newnode.next, self.next)

    return

def delete (self, v):
    if self.isempty():
        return

    if self.value == v:
        self.value = None
        if self.next != None:
            self.value = self.next.value
            self.next = self.next.next
        return
    else:
        if self.next != None:
```

```
self.next.delete(v)
if self.next.value == None:
    self.next = None

return
```

SUMMARY

- Use a linked list of nodes to implement a flexible list
- Append is easy
- Insert requires some care, cannot change where the head points to
- When deleting, look one step ahead to bypass the node to be deleted

Lists & Arrays in Python

LISTS

→ Python lists are <u>not</u> implemented as flexible linked lists

→ Underlying interpretation maps the list to an array
  ⇒ Assign a fixed block when you create a list
  ⇒ Double the size if the list overflows the array

→ Keep track of the last position of the list in the array
  ⇒ L.append() & L.pop() are constant time, amortised – O(1)
  ⇒ Insertion / deletion require time O(n)

→ Effectively, Python lists behave more like arrays than lists.

ARRAYS vs LISTS IN PYTHON

→ Arrays are useful for representing matrices

→ In list notations, these are nested lists

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

[[1,0], [0,1]]

PAGE NO.
DATE:

→ Need to be careful when initializing a multi dimensional list

CODE:

```
zerolist = [0,0,0]
zeromatrix = [zerolist, zerolist, zerolist]

zeromatrix[1][1] = 1
print(zeromatrix)
```

Output: [[0,1,0], [0,1,0], [0,1,0]]

→ Mutability Aliases different values

→ Instead, use list comprehension

```
zeromatrix = [[0 for i in range(3)] for j in range(3)]
```

NUMPY ARRAYS

→ The Numpy library provides arrays as a basic type

```
import numpy as np
zeromatrix = np.zeros(shape = (3,3))
```

→ Can create an array from any sequence type

```
identity matrix = np.array([[1,0],[0,1]])
```

→ arange is equivalent of range for lists

row2 = np.arrange(5)

→ Can operate on a matrix as a whole

- C = 3*A + B
- C = np.matmul(A,B)
- very useful for data science

SUMMARY

- Python lists are not implemented as flexible linked structures
- Instead, allocate an array, and double space as needed
- Append is cheap, insert is expensive
- Arrays can be represented as multidimensional lists, but need to be careful about mutability, aliasing
- Numpy arrays are easier to use

Date
January 13, 2022

PAGE NO.
DATE:

Implementing Dictionaries

DICTIONARY

→ An array/list allows access through positional indices

→ A dictionary allows access through arbitrary keys
  - A collection of key-value pairs
  - Random access - access time is the same for all keys

→ How is a dictionary implemented?

IMPLEMENTING A DICTIONARY

→ The underlying storage is an array
  - Given an offset i, find $A[i]$ in constant time

→ Keys have to be mapped to $\{0, 1, \ldots, n-1\}$
  - Given a key k, convert it to an offset i

→ Hash function
  - $h: S \to X$ maps a set of values S to a small range of integers $X = \{0, 1, \ldots, n-1\}$

- Typically $|X| \ll |S|$, so there will be collisions, $h(s) = h(s')$, $s \neq s'$
- A good hash function will minimize collisions
- SHA-256 is an industry standard hashing function whose range is 256 bits
  - Use to hash large files - avoid uploading duplicates to cloud storage

HASH TABLE

- An array A of size n combined with a hash function h
- h maps keys to $\{0, 1, \ldots, n-1\}$
- Ideally, when we create an entry for key k, $A[h(k)]$ will be unused
  - What if there is already a value at that location?
- Dealing with collisions
  - Open adressing (closed hashing)
    - Probe a sequence of alternate slots in the same array
  - Open hashing
    - Each slot in the array points to a list of values
    - Insert into the list for the given slot

Shriansh
PAGE NO.
DATE:

→ SUMMARY ←

- A dictionary is implemented as a hash table
  - → An array plus a hash function
- Creating a good hash function is important (and hard!)
- Need a strategy to deal with collisions
  - → open addressing / closed hashing - probe for free space in the array
  - → open hashing - each slot in the hash table points to a list of key-value pairs
  - → Many heuristics / optimizations possible for dea