Priya Thapa

11/5/2025

MSCS-532

Dr. Brandon Bass

Assignment 3: Understanding Algorithm Efficiency and Scalability

Part 1: Randomized Quicksort Analysis

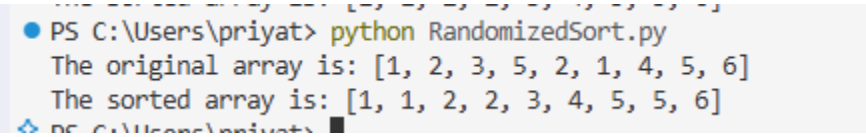
1. Implementation

Randomized Quicksort is a sorting algorithm that selects a random pivot from the array. The pivot is used to partition the remaining elements into two subarrays, one containing element less than or equal to the pivot, and the other containing elements greater than the pivot. The algorithm then recursively applies the same process to each subarray. This continues until all subarrays are reduced to single elements, at which point the concatenation of the subarrays produces a fully sorted list.

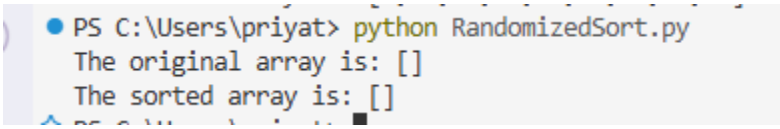

```python
import random

def quickSort(arr):
  if len(arr)<=1: return arr

 # Choose a random pivot
  pivotIndex = random.randint(0, len(arr) - 1)
  pivot = arr[pivotIndex]
  left = []
  right = []
  middle = []

  for i in range(len(arr)):
    if arr[i] < pivot:
        left.append(arr[i])
    elif arr[i] == pivot:
        middle.append(arr[i])
    else:
        right.append(arr[i])

 #recursively sort left and right, then combine
  return quickSort(left) + middle + quickSort(right)

arr = [9,8,7,6,5,4,3,2,1]
print ("The original array is:", arr)
print ("The sorted array is:", quickSort(arr))
```

Output:

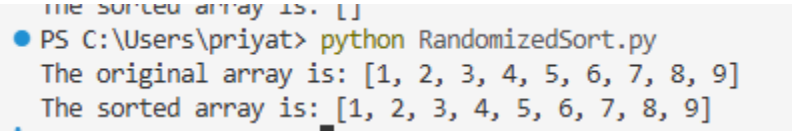
Repeated Numbers:



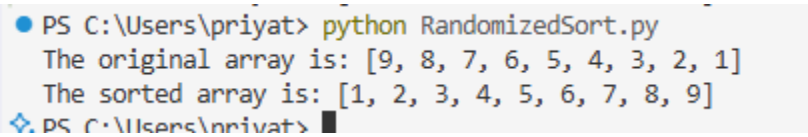
Empty Array:



Sorted Array:



Reversed Array:



2. Analysis

Average-case time complexity of Randomized Quicksort.

The following steps are the algorithm for Randomized Quicksort:

1. Choosing a pivot element uniformly at random from the array.
2. Partitioning the arrays into three parts:
   - Elements less than the pivot element (left sub array).
   - Elements more than the pivot element (right sub array).
   - Elements equal to the pivot element (middle array).
3. Recursively sort the left and right subarrays until we get our sorted array.

Randomized Quicksort chooses the pivot uniformly at random, partitions the array around it, and recursively sorts the subarrays. Let $T(n)$be the expected time for an array of size n:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

where $k$is the number of elements less than the pivot. Because the pivot is random, each $k$is equally likely, so:

$$\mathbb{E}[T(n)] = O(n) + \frac{2}{n}\sum_{k=0}^{n-1} \mathbb{E}[T(k)]$$

Solving this recurrence gives:

$$\mathbb{E}[T(n)] = O(n\log\, n)$$

On average, the pivot splits the array roughly in half, giving recursion depth $O(\log\, n)$, and each level costs $O(n)$for partitioning.

Randomized Quicksort runs in expected O(n log n) time, and random pivot selection prevents the worst-case O(n²) behavior typical of deterministic Quicksort.

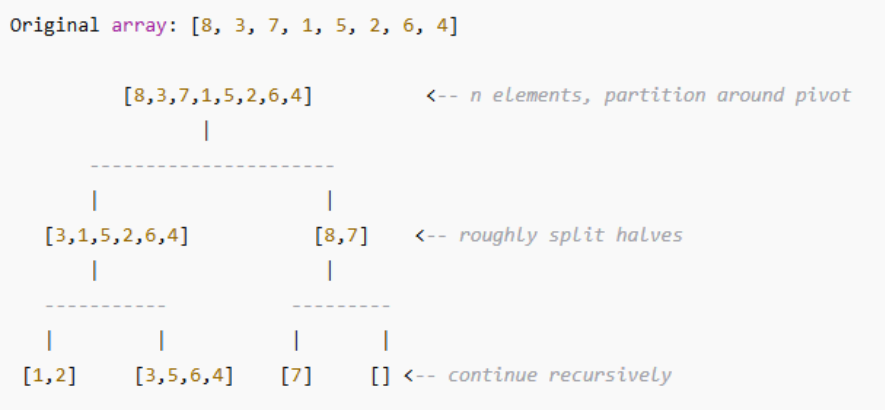
Explaining why the average-case time complexity is \(O(n \log n)\).

In Randomized Quicksort, the pivot is chosen uniformly at random. On average, this random pivot splits the array into two roughly equal halves. Each recursive call then processes a smaller subarray, and the total depth of recursion is proportional to the number of times you can halve the array until each subarray has one element, which is $O(\log\, n)$.

At each level of recursion, all $n$elements are compared once during the partitioning step. So, the total work is the number of levels multiplied by the work per level:

$$O(\text{levels}) \times O(\text{work per level}) = O(\log\, n) \times O(n) = O(n\log\, n)$$

Randomizing the pivot ensures that extremely unbalanced splits (which could lead to $O(n^2)$in deterministic Quicksort) happen with very low probability, giving an expected average-case time complexity of $O(n\log\, n)$.



Key observations:

1. **Depth of recursion:** On average, each pivot splits the array roughly in half. So the recursion depth is $O(\log\ n)$.
2. **Work per level:** At each level, every element is compared to its pivot once, so each level does $O(n)$work.
3. **Total work:** Multiply depth × work per level → $O(n\log\ n)$.

Randomization ensures that even if some pivots are poor (unbalanced splits), the expected overall behavior stays $O(n\log\ n)$.

Indicator random variables or recurrence relations

Let $X_{i,j}$be an indicator variable that is 1 if elements $i$and $j$are compared during the execution of Randomized Quicksort, and 0 otherwise.

- The probability that two elements are compared is Pr $[X_{i,j} = 1] = \frac{2}{|j-i|+1}$.
- Total comparisons: $C = \sum_{i<j} \quad X_{i,j}$.
- Expected comparisons:

$$\mathbb{E}[C] = \sum_{i<j} \quad \mathbb{E}[X_{i,j}] = \sum_{i<j} \quad \frac{2}{|\ j-i\ |+1} = O(n\log\ n)$$

In Randomized Quicksort, each pair of elements is compared at most once, and because the pivot is chosen randomly, the chance that any two elements get compared decreases as the array is divided. On average, the total number of comparisons grows like $n\log\ n$, which is why the average-case time complexity is $O(n\log\ n)$.

3. Comparison

Deterministic

```
def quickSort(arr):
  if len(arr)<=1: return arr

 # Choose the first number as pivot
  pivot = arr[0]
  left = []
  right = []
  middle = []
```

Randomized

```
def quickSort(arr):
  if len(arr)<=1: return arr

 # Choose a random pivot
  pivotIndex = random.randint(0, len(arr) - 1)
  pivot = arr[pivotIndex]
  left = []
  right = []
  middle = []
```

```
# Measure execution time
def measure_time(sort_fn, arr):
    start = time.perf_counter()
    sort_fn(list(arr))
    end = time.perf_counter()
    return round((end - start) * 1000, 3)

# Generate test arrays
def generate_arrays(n):
    random_arr = [random.randint(0, n) for _ in range(n)]
    sorted_arr = sorted(random_arr)
    reverse_arr = sorted_arr[::-1]
    repeated_arr = [random.randint(0, 10) for _ in range(n)]
    return random_arr, sorted_arr, reverse_arr, repeated_arr

# Run comparison
sizes = [100, 500, 1000]

for n in sizes:
    random_arr, sorted_arr, reverse_arr, repeated_arr = generate_arrays(n)
    print(f"\nArray size: {n}")

    print(f"Random array     - Deterministic: {measure_time(deterministic_quicksort, random_arr)}
    print(f"Sorted array     - Deterministic: {measure_time(deterministic_quicksort, sorted_arr)}
    print(f"Reverse array    - Deterministic: {measure_time(deterministic_quicksort, reverse_arr)}
    print(f"Repeated elements- Deterministic: {measure_time(deterministic_quicksort, repeated_arr)
```

Output:

```
Array size: 100
Random array     - Deterministic: 0.109 ms | Randomized: 0.113 ms
Sorted array     - Deterministic: 0.288 ms | Randomized: 0.093 ms
Reverse array    - Deterministic: 0.364 ms | Randomized: 0.103 ms
Repeated elements- Deterministic: 0.045 ms | Randomized: 0.044 ms

Array size: 500
Random array     - Deterministic: 0.694 ms | Randomized: 0.676 ms
Sorted array     - Deterministic: 7.351 ms | Randomized: 0.66 ms
Reverse array    - Deterministic: 7.652 ms | Randomized: 0.652 ms
Repeated elements- Deterministic: 0.175 ms | Randomized: 0.184 ms

Array size: 1000
Random array     - Deterministic: 1.378 ms | Randomized: 1.583 ms
Sorted array     - Deterministic: 48.803 ms | Randomized: 1.457 ms
Reverse array    - Deterministic: 30.281 ms | Randomized: 1.539 ms
Repeated elements- Deterministic: 0.39 ms | Randomized: 0.344 ms
PS C:\Users\priyat>
```

Array Type	**Observation** **(deterministic & randomized)**	**Explanation**
Random Array	Both algorithms performed similarly across all input sizes.	For randomly distributed data, both deterministic and randomized pivot selections are likely to pick a reasonably balanced pivot, giving the expected $O(n\log\ n)$performance.
Sorted Array	Deterministic Quicksort was significantly slower (e.g., 48.8 ms vs 1.4 ms at $n = 1000$).	Using the first element as the pivot in sorted data always produces highly unbalanced partitions, leading to the worst-case $O(n^2)$behavior. Randomized Quicksort avoids this by choosing pivots uniformly at random, maintaining balanced partitions and $O(n\log\ n)$average complexity.

Reverse Array	Similar trend — deterministic version degraded sharply (30.3 ms vs 1.5 ms).	Reverse-sorted arrays are another pathological case for deterministic pivot selection, again triggering $O(n^2)$performance, while randomized pivot selection remains efficient.
Repeated Elements	Both algorithms performed similarly and efficiently.	When many duplicate values exist, partitions tend to be naturally balanced. Pivot choice has little impact, keeping both algorithms near $O(n\log\ n)$.

Discussion

Empirically, the randomized version consistently outperforms the deterministic one on sorted and reverse-sorted data, confirming the theoretical prediction that Randomized Quicksort's expected running time is $O(n\log\ n)$for all input distributions.

Minor timing fluctuations (e.g., randomized version slightly slower on small random arrays) are due to random pivot selection overhead and normal timing noise, not algorithmic inefficiency. Overall, the empirical results strongly align with theoretical expectations.

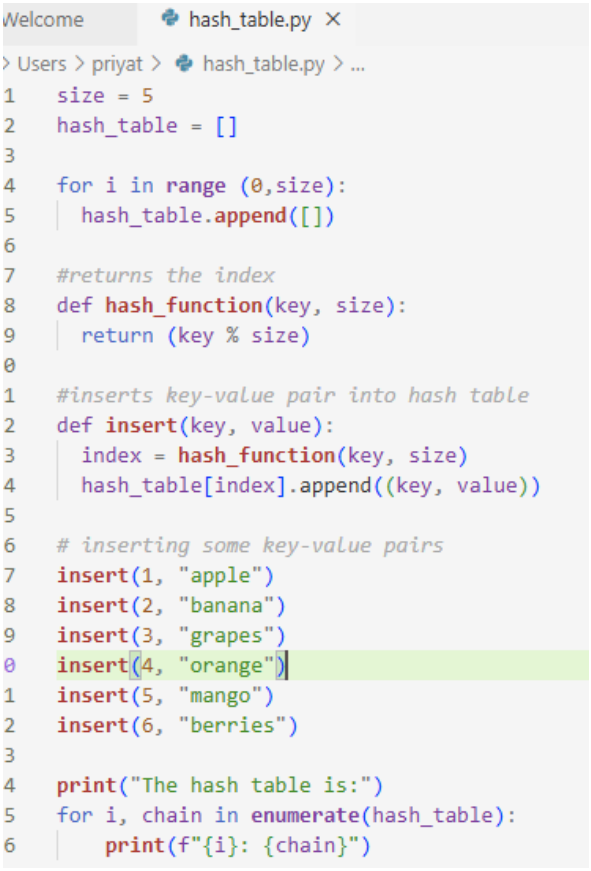
Part 2: Hashing with Chaining

1. Implementation

A Hash Table is a data structure where we store data in key value pairs at specific indices. We use Hash Function that receives a key to compute the index to for the key value pair. We can perform inserts, search, and delete operations using the hash table. We can also add more than one key value pair at a certain index which is often described as collision. To handle collision, we store the multiple key value pair as a list in the same index for better operations.

Insertion

A Hash Function that inserts key value pairs into the Hash Table, I have caused collision on index 1.

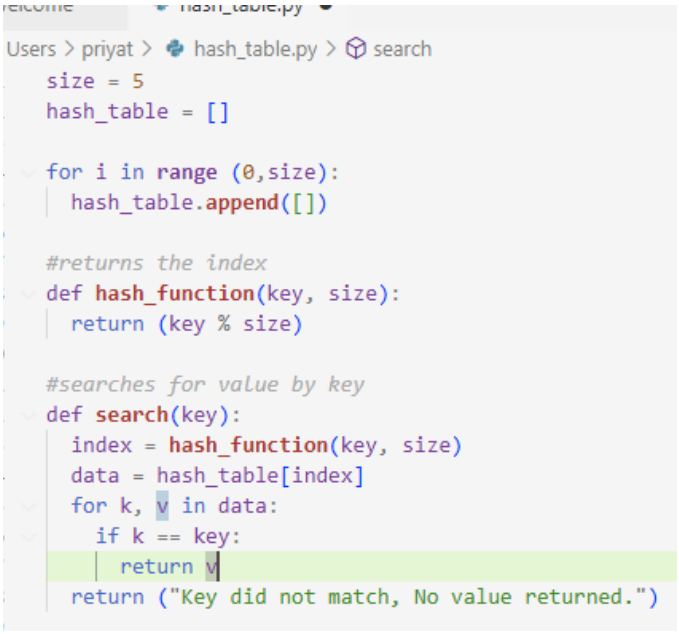


```python
size = 5
hash_table = []

for i in range (0,size):
    hash_table.append([])

#returns the index
def hash_function(key, size):
    return (key % size)

#inserts key-value pair into hash table
def insert(key, value):
    index = hash_function(key, size)
    hash_table[index].append((key, value))

# inserting some key-value pairs
insert(1, "apple")
insert(2, "banana")
insert(3, "grapes")
insert(4, "orange")
insert(5, "mango")
insert(6, "berries")

print("The hash table is:")
for i, chain in enumerate(hash_table):
    print(f"{i}: {chain}")
```

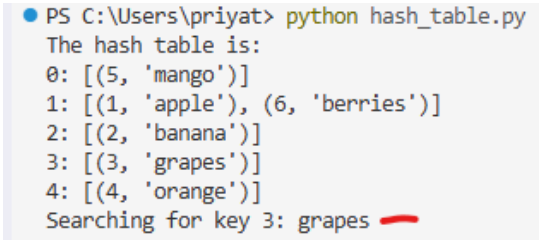
Output:

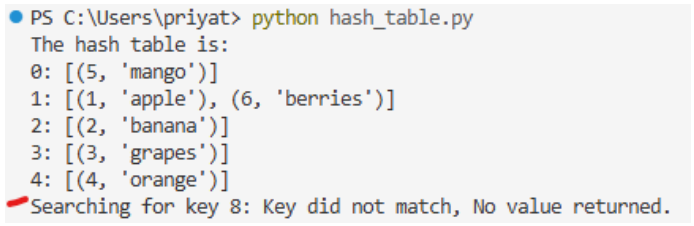
```
PS C:\Users\priyat> python hash_table.py
The hash table is:
0: [(5, 'mango')]
1: [(1, 'apple'), (6, 'berries')]
2: [(2, 'banana')]
3: [(3, 'grapes')]
4: [(4, 'orange')]
```

Search



Output:





Delete

```
Users > priyat > hash_table.py > delete
size = 5
hash_table = []

for i in range (0,size):
  hash_table.append([])

#returns the index
def hash_function(key, size):
  return (key % size)

#deletes key-value pair from hash table
def delete(key):
  index = hash_function(key, size)
  data = hash_table[index]
  for item in data:
    if item[0] == key:
      data.remove(item)
      return ("Data removed.")
  return ("Could not find the data, no data was removed")
```

Output:

As we can see the image shows output of Hash table before the deletion and after the deletion of key value (1, apple).

```
PS C:\Users\priyat> python hash_table.py
The hash table is:
0: [(5, 'mango')]
1: [(1, 'apple'), (6, 'berries')]
2: [(2, 'banana')]
3: [(3, 'grapes')]
4: [(4, 'orange')]
Deleting Key 1: Data removed.
The hash table after deleting Key 1:
0: [(5, 'mango')]
1: [(6, 'berries')]
2: [(2, 'banana')]
3: [(3, 'grapes')]
4: [(4, 'orange')]
PS C:\Users\priyat>
```

2. Analysis

Search, Insert, and Delete Time Complexity (Assuming Simple Uniform Hashing)

Under the assumption of **simple uniform hashing**, every key is equally likely to be mapped to any index in the hash table, independent of other keys.

- **Search:**
  The expected time to search for an element is O(1 + α), where α (alpha) is the load factor. On average, the key will be found in constant time because each index (chain) has approximately α elements.

- **Insert:**
  The expected time to insert a key-value pair is also O(1) on average.
  Insertion involves computing the hash function and appending the element to the list (chain) at that index.

- **Delete:**
  Deletion requires locating the element within its chain, so the expected time is O(1 + α).
  Like search, deletion is constant on average if the load factor is small.

When hashing is uniform and collisions are evenly distributed, these operations are very efficient typically near constant time.

Load factors (the ratio of the number of elements to the number of slots) affecting the performance of these operations.

The load factor (α) represents how full the hash table is and is given by:

$$\alpha = \frac{n}{m}$$

where

- **n** = number of stored elements
- **m** = number of slots (buckets)

- When $\alpha < 1$, most slots contain at most one element, so collisions are rare and operations are fast.
- As α increases, more elements are stored per slot, causing longer chains and slower searches or deletions.
- In the worst case, when all keys hash to the same index, the time complexity can degrade to O(n).

Thus, keeping the load factor small is crucial to maintain near-constant operation times.

Strategies for maintaining a low load factor and minimizing collisions, including dynamic resizing of the hash table.

To ensure high performance and avoid excessive collisions, several strategies can be used:

1. **Dynamic Resizing (Rehashing):**
    - When the load factor exceeds a threshold (commonly 0.7 or 0.75), increase the table size (e.g., double it or move to the next prime number).
    - Recalculate the index for each key (rehash) in the larger table to spread elements more evenly.
    - This lowers α and maintains efficient operations.
2. **Efficient Hash Function Design:**
    - Use a hash function that distributes keys **uniformly** across all slots.
    - Avoid simple modulo operations that may cluster keys if patterns exist in the data.
3. **Collision Handling with Chaining:**
    - Store multiple elements at the same index as a list or linked structure.
    - This allows multiple keys to coexist in one slot without overwriting.

- Some implementations (like Java's HashMap) use **balanced trees** for long chains to maintain good performance even when α grows.

Operation	**Average Case**	**Worst Case**
Search	O(1 + α)	O(n)
Insert	O(1)	O(n)
Delete	O(1 + α)	O(n)

In conclusion, when the hash table maintains a low load factor and employs a well-designed hash function, the expected performance of search, insertion, and deletion operations approaches constant time. Proper management of collisions and dynamic resizing are therefore fundamental to sustaining the efficiency of hashing systems.