SORTING

Algorithm	Worst-case running time	Best-case running time	Average-case running time
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Smallest data item bubbles up to top of sorted array

compares first two elements

Date:.....

BUBBLE SORT → simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Advantages.

↳ simple to write & easy to understand, as it's sequential

↳ requires few lines of code

↳ little memory ahead

Disadvantages.

↳ time consuming

↳ avg. time increases exponentially as number of elements increase

↳ not suitable for large sized data.

Best Case: $O(n)$

Worst Case: $O(n^2)$

Average Case: $O(n^2)$

Date:.....

Pseudocode.

```
Bubble Sort (array, num_elems):
    k = 1
    loop (k <= num_elems - 1)  # max passes required = n - 1
        i = 0
    # You don't want to include the last element
    because then index i+1 would go out of bounds.
    loop (i <= num_elems - 2):
        if (array[i] > array[i+1]):
            swap (array[i], array[i+1])
        i = i + 1
    k = k + 1
return.
```

↳ Bubble Sort is slow sorting algorithm

↳ Selection Sort is also quadratic time complexity, which is also slow.

Date:.....

POSSIBLE OPTIMIZATIONS.

→ at any point during bubble sort algorithm, you will have a sorted & unsorted part of your array.

→ no need to walk through the sorted part, because it's already sorted.

```
Optimized Bubble Sort (array, num_elems):
    k = 1
    loop (k <= num_elems - 1)
        i = 0
        is_swap = false
        loop (i <= num_elems - k - 1):
            if array[i] > array[i+1]:
                swap (array[i], array[i+1])
                is_swap = True
            i = i + 1
        if (is_swap) == false:
            break
        k = k + 1
    return
```

in every iteration of selection sort, the min element from unsorted array is picked & moved to beginning of unsorted array

algorithm divides the input list into two parts.

Date:.....

SELECTION SORT

→ sorts an array by repeatedly finding the minimum element from unsorted part & putting at the beginning.

Advantages.

↳ performs well on small sized

↳ in-place algorithm

Disadvantages.

↳ poor efficiency when dealing with huge list of items.

↳ requires n-squared no. of steps.

Best Case: $O(n^2)$

Worst Case: $O(n^2)$

Average Case: $O(n^2)$

- Walk through array
- Find minimum
- Swap this with value located at index = 0

- Repeat
- Walk through array
- Find second minimum
- Swap this with value at index = 1

① initialize minimum value to location 0.
② traverse the array to find minimum element
③ while traversing if any element smaller than minimum value is found then swap
④ increment min_indx to next element
⑤ repeat until array is sorted.

Pseudocode.

```
Selection Sort (A,n):
    i=0
    loop i<=n-2:
        min_index = i
        j = i+1
        loop j<=n-1:
            if A[j] < A[min_index]:
                min_index = j
            j = j+1
        temp = A[i]
        A[i] = A[min_index]
        A[min_index] = temp
        i = i+1
    return.
```

Bubble Sort & Insertion Sort what difference?

INSERTION SORT

→ inefficient on large data sets.

→ more efficient than others.

Advantages.

↳ straightforward implementation

↳ good performance when dealing with smaller lists.

Date:.....

Disadvantages.

↳ not best performing sorting algorithm.

↳ for every 'n' element that needs to be sorted, n^2 steps required

↳ time consuming.

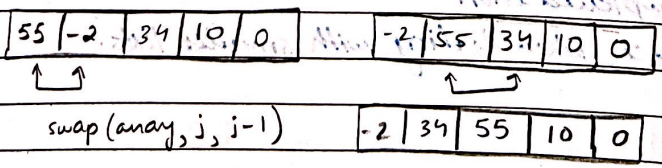
Best Case: $O(n)$

Worst Case: $O(n^2)$

Average Case: $O(n^2)$

Pseudocode.

```
insertion_sort (a_list, n):        → corresponding elements decide
    loop index : 1 to (n-1):
        current_value = a_list[index]
        free_index = index
        loop free_index > 0 and a_list[free_index-1] > currentvalue
            a_list[free_index] = a_list[free_index-1]
            free_index = free_index - 1
        a_list[free_index] = current_value
    return.
```



Date:.....

BUBBLE SORT

```
def bubble_sort(lst):
    if len(lst)==1
        print(lst)
        return
    for j in range(len(lst)-1):
        for i in range(len(lst)-j-1):
            if lst[i]>lst[i+1]:
                lst[i], lst[i+1]= lst[i+1], lst[i]
    print(lst)
pass
bubble_sort(lst)
```

SELECTION SORT

```
def selection_sort(lst):
    for i in range(len(lst)):
        minimum=i
        for j in range(i+1, len(lst)):
            if lst[j]<lst[minimum]:
                minimum=j
        lst[i], lst[minimum]= lst[minimum], lst[i]
        print(lst)
```

Date:.....

INSERTION SORT

```
def insertion_sort(lst):
    for i in range(1,len(lst)):
        element = lst[i]
        while i > 0 and lst[i-1] > element:
            lst[i] = lst[i-1]
            i = i-1
            lst[i] = element
    print(lst)
```

* is selection better than bubble in worst case?

Date:.....

QUICK SORT

0	1	2	3	4	5
9	8	10	17	15	14

Note: that the left & right partions do not need to be sorted for 10 to be sorted.

10 is in its sorted location because all elements smaller to it are on left and all elements bigger than it are on the right.

a							b
85	24	63	45	17	31	96	50
l						r	
a						b-1	

→ our pivot is 50 for now

→ as soon as l and r switch places, we are done with finding a place for pivot.

→ is the value at l and r appropriate as compared to pivot (50 in this case)?

↳ since r is bigger than pivot so it's fine, so we update it (go left)

85	24	63	45	17	31	96	50
l					r		

→ "r" sees 31, which is smaller than the pivot (50). 31 shouldn't be on "r" side of array

31	24	63	45	17	85	96	50
	l			r			

→ swap the values of l and r.

★ r < l

→ here r < l so why not swapped.

Date:.....

Advantages.

→ are of the fastest algorithm on average.

→ does not need additional memory

Disadvantages.

→ worst-case complexity is $O(n^2)$.

Average Case: $O(n \log n)$

Worst Case: $O(n^2)$

Best Case: $O(n \log n)$

QUICK SORT

Date:.....

→ Quick sat is an example of a divide and conquer strategy.

→ What is "divide & conquer"?

→ Strategy in which you break the given problem down into subproblems recursively

↳ these subproblems are similar to original problem

→ These subproblems are solved recursively.

→ The solutions to these subproblems are combined to form solution for original problem.

- Divide: the problem into number of subproblems.
- Conquer the subproblems by solving them recursively.
- Combine the solutions to the subproblems into the solution for the original problem.

* ~~When is Quick Sort better than Merge Sort?~~

Date:.....

MERGE SORT

↳ let's say you have unsorted array.
↳ let's split it up into two equal parts.
↳ it has n=8 elements, so two subarrays of length 4 each.

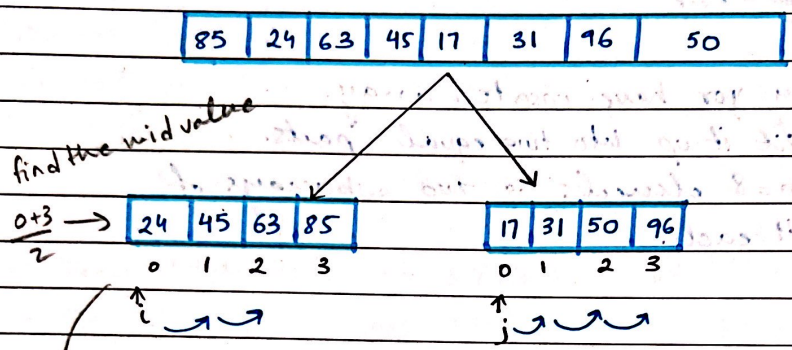
Advantages:
* Merge sort is stable sort
* it is easy to understand
* it gives better performance

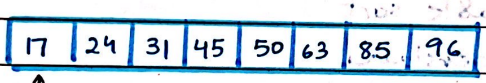
Disadvantages:
* it requires extra memory space.
* copy of elements to temporary array
* requires additional array.
* slow process.

Complexity of Merge Sort. The merge sort algorithm passes over the entire list & requires at most log n passes and merges n elements in each pass. The total number of comparisons required by merge sort is given by O(nlogn).

Date:.....

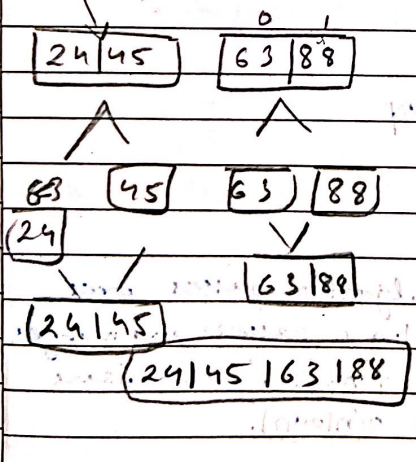
Merge Sort-Concept.





k

* i and j are compared
* j<i, j+1
* i<j, i+1
* j<i, j+1
* i<j, i+1



Date:.....

MERGE SORT

```
def merge(L,R,A)
    lenL = len(L)
    lenR = len(R)
    i=j=k=0
    while i<lenL and j<lenR:
        if L[i] <= R[j]
            A[k] = L[i]
            i=i+1
            k=k+1
        else:
            A[k]= R[j]
            j = j+1
            k=k+1
    while i is less than lenL:
        A[k] = L[i]
        i=i+1
        k=k+1
    while j is less than lenR:
        A[k] = R[i]
        j=j+1
        k= k+1
```

??

Date:.....

(2)

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:

    left_half = merge_sort
    right_half = merge_sor

    sorted_arr = []
    i = j = 0

while i < len(left_half) and j < len(right_half):
    if left_half[i] <= right_half[i]:
        sorted_arr.append(left_half[i])
        i = i+1
    else:
        sorted_arr.append(right_half)
        j = j+1
sorted_arr += left_half[i:]
sorted_arr += right_half[j:]
return.
```

→ Why is it important to create two seperate copies of subarray?

→ Can we do it without making any copies, ie in-place?

RECURSION

→ recursion is process in calls itself

There are two parts

1. Base Case → responsible for
2. Recursive Case → responsible

→ What would have happened if we did not have a base case defined?
↳ if there is no base case use of recursive definition becomes infinitely long and any program will never terminate & produce result.

→ When you call a function, that call is saved in a stack, called a stack frame.

→ All of variables and other associated information with that particular function call are saved.

→ This also means that the stack frame provides a scope for the variables for each function call.

→ So for an array of length = n,
- total number of calls = n
- recursive calls = n - 1

Date:.....

Recursion in Math

↳ one of the most obvious math definitions that can be stated in a recursive manner is definition of integer factorial.

↳ 1! = 1 (base case)

↳ N! = N * (N-1)! (recursive call)

Recursion vs Iteration.

↳ we can calculate 5! using a loop.

```
int fiveFactorial = 1;
for (int i = 1; i <= 5; i++)
    fiveFactorial *= i;
```

↳ or we can calculate using recursion.

```
factorial (int n)
    if (n == 1)
        return 1
    else
        return n * factorial (n-1)
```

* typically, a recursive solution uses both more memory & more processing time than an iterative solution

Fibonacci Series.

↳ calculation of sequence of Fibonnaci numbers

F_n can be stated as;

$F_0 = 0$ } base cases.

$F_1 = 1$ }

$F_n = F_{(n-1)} + F_{(n-2)}$ } recursion.

Rules of Recursion.

↳ Base case: Always have atleast one case that can be solved without recursion.

↳ Make Progress: Any recursive call should make progress toward a base case.

↳ Never Duplicate work by solving same instance of a problem in seperate recursive calls.

Direct vs Indirect Recursion Call.

↳ direct recursion is when a method calls itself

↳ indirect recursion is when method calls a second method that can call first method.

Date:.....

HASHING

Unsorted Map – ADT:

↳ these are called dictionaries

↳ key-value pairs

↳ keys are unique; values not necessarily unique.

↳ order doesn't matter

#query (key)

M[key] returns the value v associated with key k in map M, if one exists; otherwise raise a KeyError.

add_modify (M, key, val) M[key] = val.

M[key] = val. Associate value v with key k in Map M replacing the existing value if map already contains an item with key equal to k.

delete (M, key) del M[key]

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then Raise a KeyError.

* Time complexity (average & worst) is O(n).
* Binary search not possible map isn't sorted.

Date:

len(M): return the number of items in map M

iter(M): the default iteration for a map generates a sequence of keys in the map.

HASH TABLES

↳ faster search approach
↳ for unsorted data, we relied on Linear search - O(n).
↳ for data arranged in sorted way, Binary search would be better - O(log n)

0	1	2	3	4	5	6	7	8	9	10
	(1:B)	(2:Z)	(3:B)		(5:H)				(9:G)	

searching is O(1)

↳ use the keys as indices in the array.

h(key) = key

input output

Date:.....

→ Hashing is the process of generating an address/index from a key

→ This is done through hash function.

→ The generated address/index is referred to as the home address

→ The area in the memory where all the home addresses are located is called the prime area.

→ A hash table is a collection of items which are stored in such a way as to make it easy to find them later.

→ Each position of hash table, is often called slot.

- Consider, h(key) = key % table_size
- Assume table_size to be 7 (slightly greater than total entries)
- Conflict or collision may occur
- To prevent these collisions, we can use good hash function.

Date:

Hashing Methods

- Direct
- Modulo Division
- Mid Square
- Rotation
- Subtraction
- Digit Extraction
- Folding
- Pseudorandom generation.

Direct Method

A direct hash function uses the item's key as bucket index. E.g if the key is 937, the index is 937.

$$h(key) = key$$

Limitations;

For example, we cannot have social security number as the key using this method because social security numbers are nine digits, and we cannot have an array as large as nine digits.

Date:.....

Subtraction Method

Modulo-Division Method

→ if key < size then return key as it is.

→ Divides the key by hash table size & uses the remainder as index.

→ ~~Use~~ Prime numbers as length of hash tables would work best.

```
h(key) = key % table_size.
```

* 1a

* Significance of having a hash table with length that is a Prime Number

  ↳ When keys are randomly distributed, it probably won't matter what the size of your table is.

  ↳ However, in the case of non-random keys, a hash table of prime number length will produce the most wide spread distribution of integers to indices.

Date:.....

Digit Extraction Method

↳ selected digits are extracted from the key and used as the address.

↳ what our hash function/mapping does is that it,

- picks the first, third and fourth digits (starting from the left)
- the resultant three-digit number is our index for that key-value pair.

* all of these weird operations are there to somehow reduce the number of collisions and get unique home address.

```
def digit_extraction(key):
    d3 = (key // 100) % 10
    d2 = (key // 1000) % 10
    d1 = (key // 100000) % 10
    home_address = (d1 * 100) + (d2 * 10) + (d3 * 1)
    return home_address
```

Midsquare Method

def midsquare (key):

- caiupute the square of your key
- select your address from the middle of squared number.

$9452^2 = 89340304$ address is 3403

Folding Methods

→ two methods:

① Fold Shift ② Fold Boundary

① Fold Shift

- the key is divided into parts whose size matches the size of required address
- then the left & right parts are shifted and added with middle part.
- any extra digit is dropped off or take mod here with table size.

Date:.....

② Fold Boundary

- you can think of the key is being folded along certain boundaries.
- this results in the key being divided into parts.
- but alternate sets of numbers are reversed.

Rotation Method

↳ works well with keys that are in serial/sequence e.g
- student/employee ID's, etc.
- 1001, 1002, 1003, 1004, ...

↳ a simple hashing algorithm tends to create synonyms when hashing keys like these.
- rotating the last character to front of the key minimizes this effect.

Pseudorandom Hashing

↳ the key is used as seed in a pseudorandom-number generator, and ...
- ... the resulting random number is then scaled into possible address range using modulo-division.

Perfect Hash Function

↳ a hash function that maps each item into a unique slot is referred to as a perfect hash function.

↳ you can increase the size of table to reduce collisions

- more vacant slots means that probability of collision is lower.
- not always feasible though

↳ so we also need a good hash function to ensure less collisions.

Characteristics of "Good" Hash Functions

- Fast to compute so that querying, addition and deletion are done typically in O(1).
- Minimizes collisions.
- outputs are uniformly distributed along hash table
- function is deterministic for a given input, the same output index is returned

Date:.....

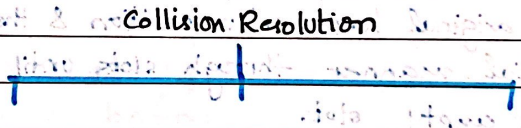
Collision Resolution.

↳ two or more unique keys hash to same slot in hash table is called collision

↳ $h(key_1) = h(key_2) = j$

↳ Load factor

- $\lambda = \frac{\text{no of items}}{\text{table size}}$
- measure of how much of the hash table is occupied - bigger load factor, the more hash table filled.



Open Addressing Linked List Buckets

↳ linear probe

↳ quadratic probe

↳ pseudorandom

↳ key offset

Date:.....

Open Addresing

↳ collisians are resolved in prime airea.

↳ in case of collisians, you look for vacant space within prime area where you can store incoming key.

↳ unlike chaining, you do not introduce an additional data structure.

① Linear Probing

↳ you perform hashing on a key k, and get and address j,

↳ That slot, A[j] is already occupied - collisian.

↳ start at an original hash value position & then move in a sequential manner through slots until we encounter first empty slot.

- ↳ h(k) = j, A[j] is occupied
- ↳ try looking in, A[(j+1) mod N]
- ↳ if collisian, then, A[(j+2) mod N]
- ↳ A[(j+3) mod N] ...
- ↳ keep looking until you found empty slot.

↳ so you search until you either,

- find the key you are looking for
- reach an empty slot
- return back to index from where you started.

Date:.....

↳ if you want to remove a key from hash table.
- you can pass your key in hash function
- get index
- and set the value at that index to None.

↳ and instead of placing None at the deleted key's index in hash table, place a special marker called "available" or "tombstone".
↳ this indicates that a certain slot wasn't always empty (or None) & there was something here before.

None broke the search path to get to key 15

Disadvantage: → clustering
→ if many collisions occur at same hash value, a number of surrounding slots will be filled by linear probing resolution.
→ impact on other items a that are being inserted

Primary clustering around the canon have address.

- can we make collision path independent of have address?

Date:

how to know jump size?

s bigger than size one.

N], where skip is an integer

er clustering.

A[(org_ind + probe-step) % N]

↳ in linear probing, $f(i) = i$, for $i = 0, 1, 2, \ldots$

↳ in quadratic probing, $f(i) = i^2$, for $i = 0, 1, 2, \ldots$

- $A[h(k)+0]$, $A[h(k)+1]$, $A[h(k)+4]$, $A[h(k)+9]$ ...

↳ this avoids primary clustering and instead creates primary clustering.

↳ linear probing leads to primary clustering

↳ quadratic solves this, but creates secondary clustering.

↳ secondary clustering is problematic because collision path is dependent

- on open addresing
- on have address.

(1) we want to come up with a collision path that is unique for each key

(2) Each key is unique, so if our path is dependent on key, then path is also unique.

* this better than primary clustering because there are gaps in the middle but hopefully you understand WHY it's a problem.

(3)

Pseudorandom Numbers

```
import random
key = int(input("Enter key = "))
random.seed(key)
for x in range(5):
    print(random.randint(1,19), end=" ")
```

random.randint(a,b) returns a random integer N such that a <= N <= b.

→ A[h(k)+f(i) %N], where f(i) is the pseudorandom number sequence produced using k as seed.

(4)

Seperate Chaining

↳ simple and efficient way of resolving collisions.

↳ each slot in hash table, instead of storing keys/items directly would possess a secondary container, which instead of would contain the keys/items.

↳ more memory required. (hash table + secondary containers)