COL1000
Introduction to Programming

Priyanka Golia

Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

Quiz

Time complexity : $|D| = n$

```
balance, change = A, []
D.sort(reverse=True)

while balance != 0:
    for d in D:
        if d <= balance:
            k = balance // d
            change.extend([d]*k)
            balance -= d*k
            break
    else:
        return None
return change
```

Quiz

Time complexity : |D| = n

```
balance, change = A, []
D.sort(reverse=True)

while balance != 0:
    for d in D:
        if d <= balance:
            k = balance // d
            change.extend([d]*k)
            balance -= d*k
            break
    else:
        return None
return change
```

Component	Cost	Notes
Sorting denominations	n logn	Done once
Number of `while` iterations	≤n	Each iteration uses one denomination
Each `while` iteration	n	Scan for coin

Total $- nlogn + n \times n$

Algorithm

Specification —

Given (Input) — A list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

Algorithm

Specification —

Given (Input) — A list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

```
Algorithm: Find the Index of a Given Number

1. Start
2. Set an index variable i = 0
3. Repeat while i < n:
      a. If L[i] = x:
            i. Output i (the index of the number)
            ii. Stop the algorithm
      b. Else, increment i = i + 1
4. If the end of the list is reached and no match is found:
      Output -1
5. End
```

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — found if it is present, else output not found.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — found if it is present, else output not found.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Check the middle element — $8 < 10$

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — found if it is present, else output not found.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Check the middle element — $8 < 10$

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — found if it is present, else output not found.

```
[2, 4, 6, 8, 10, 12, 14]    X = 10
```

Check the middle element — $8 < 10$

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a
number x

Output — found if it is present, else output not found.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Check the middle element — $8 < 10$

`[10, 12, 14]`

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a

number x

Output — found if it is present, else output not found.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Check the middle element — $8 < 10$

`[10, 12, 14]`

Check the middle element — $12 > 10$

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — found if it is present, else output not found.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Check the middle element — $8 < 10$

`[10, 12, 14]`

Check the middle element — $12 > 10$

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a

number x

Output — found if it is present, else output not found.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Check the middle element — $8 < 10$

`[10, 12, 14]`

Check the middle element — $12 > 10$

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a

number x

Output — found if it is present, else output not found.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

↓ Check the middle element — 8 < 10

`[10, 12, 14]`

↓ Check the middle element — 12 > 10

`[10]`

↓ 10 = 10 — output found.

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

↓ Check the middle element — 8 < 10

`[10, 12, 14]`

↓ Check the middle element — 12 > 10

`[10]`

↓ 10 = 10 — output found.

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Left = 0, right = n-1

Mid = left + right // 2 **= 3rd index**

Check the middle element — $8 < 10$

`[10, 12, 14]`

Check the middle element — $12 > 10$

`[10]`

$10 = 10$ — output found.

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Left = 0, right = n-1

Check the middle element — 8 < 10

Mid = left + right // 2 = **3rd index**

`[10, 12, 14]`

Left = mid +1 **(4th index)**, right = right **(6th index)**

Mid = left + right // 2 **(5th index)**

Check the middle element — 12 > 10

`[10]`

10 = 10 — output found.

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Left = 0, right = n-1

Check the middle element — 8 < 10 → Mid = left + right // 2 = **3rd index**

`[10, 12, 14]` → Left = mid +1 **(4th index)**, right = right **(6th index)**

Mid = left + right // 2 **(5th index)**

Check the middle element — 12 > 10

`[10]` → Left = left **(4th index)**, right = mid-1 **(4th index)**

Mid = left + right // 2 = **(4th index)**

10 = 10 — output found.

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

`[2, 4, 6, 8, 10, 12, 14]` `X = 10`

Left = 0, right = n-1

Check the middle element — 8 < 10

Mid = left + right // 2 = **3rd index**

`[10, 12, 14]`

Left = mid +1 **(4th index)**, right = right **(6th index)**

Mid = left + right // 2 **(5th index)**

Check the middle element — 12 > 10

`[10]`

Left = left **(4th index)**, right = mid-1 **(4th index)**

Mid = left + right // 2 = **(4th index)**

10 = 10 — output found.

Return index = mid

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

```
def search_index(array, target):
    left = 0
    right = len(array) - 1
    while left <= right:
        mid = (left + right) // 2
        if array[mid] == target:
            return mid                     # index found
        elif array[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
   return -1                             # target not found
```

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

How much time ?

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

How much time ?

```
Each time you perform a comparison, you cut the
search space in half.
That means:
After the 1st comparison → n/2 elements remain
After the 2nd comparison → n/4 elements remain
After the 3rd comparison → n/8 elements remain
…and so on.
So after k steps, the number of remaining
elements is
```

$$\frac{n}{2^k}$$

Algorithm

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

How much time ?

Each time you perform a comparison, you cut the search space in half.
That means:
After the 1st comparison → n/2 elements remain
After the 2nd comparison → n/4 elements remain
After the 3rd comparison → n/8 elements remain
…and so on.
So after k steps, the number of remaining elements is
$$\frac{n}{2^k}$$

And, when do you stop,
when only one element remains
$$\frac{n}{2^k} = 1$$

$$k = log_2 n$$

Binary Search

Specification —

Given (Input) — A **sorted without duplicates** list of numbers $L = [a_1, a_2, \ldots, a_n]$ and a number x

Output — Find the index (position) of x in the list if it is present, else output -1.

```
def Binary_search(array, target):
    left = 0
    right = len(array) - 1
    while left <= right:
        mid = (left + right) // 2
        if array[mid] == target:
            return mid                    # index found
        elif array[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
   return -1                             # target not found
```

$log_2 n$ Time

N = 1,000, 000 —
only 20 steps to search

Revision: Algorithms

Algorithm design:
It's a way of thinking — a process of **analyzing problems**, **breaking them down**, **refining solutions**, and **reasoning about correctness and efficiency**.

1. Abstraction — to ignore irrelevant details and focusing on the core problem structure. we don't begin with coins, rupees, or actual currency. We abstract them to values and quantities.

Instead of thinking "sort student names alphabetically," we abstract it to:
"Given a list of comparable items, rearrange them into non-decreasing order."
This abstraction allows the same algorithm (say, merge sort) to work for integers, names, or even custom objects.

Revision: Algorithms

2. Decomposition — dividing a big problem into smaller parts that are easier to handle. Good decomposition reveals structure — often showing how a solution for a small case can help solve the full problem.

3. Top-Down and Step-Wise Refinement — Algorithm design is best done top-down — start with a high-level idea, and refine it step by step until every part is implementable.

Revision: Algorithm

4. Completion and Correctness

A working program is not automatically a correct algorithm. We must reason carefully about termination (it finishes) and correctness (it produces the right result).

```
We show that the algorithm always finishes.

1. In the greedy coin change, balance decreases by at least one each
iteration (since we subtract some coin value c>0). So the loop must end
after at most n/min(D) steps.

2. In merge sort, the list size halves each time. Recursion depth = log2 n.
Eventually, lists of length 1 are reached, so recursion stops.
```

Revision: Algorithm

4. Completion and Correctness

A working program is not automatically a correct algorithm. We must reason carefully about termination (it finishes) and correctness (it produces the right result).

Correctness: We prove that, if the algorithm stops, its output satisfies the specification.

```
Coin Change Invariant
At every step:
            sum(change)+balance=n
This ensures that we never lose or gain money. When balance becomes 0:
sum(change)=n. Thus, the list change is a valid solution.

Merge Sort:
At every stage of merging:
The merged list M is sorted. It contains all elements processed so far.
When both halves are exhausted, the merged list is completely sorted.
```

Revision: Algorithm

5. Correctness vs. Optimality — Correctness means “gives a valid result.” Optimality means “gives the best possible result.”

Greedy + Backtracking: correct for canonical denominations, but not always optimal.

Brute-force: both correct and optimal (since all combinations are considered).

Greedy: For activity selection, both correct and optimal.
Brute-force: both correct and optimal (since all combinations are considered).

Revision: Algorithm

6. Performance Analysis — Estimating how long an algorithm takes, or how much memory it uses.

7. Different approaches — greedy, dynamic programming, divide and conquer, brute force, etc.

Revision: Algorithm

6. Performance Analysis — Estimating how long an algorithm takes, or how much memory it uses.

7. Different approaches — greedy, dynamic programming, divide and conquer, brute force, etc.

Technique	Basic Idea	Speed	Always Correct?	Example
Brute Force	Try every possible combination	Very slow	Yes	Small puzzles, exhaustive search
Greedy	Pick best local choice	Very fast	Sometimes fails	coin system
Divide & Conquer	Split, solve, combine	Moderate	Yes	Sorting, searching
Reorder Input	Simplify by sorting	Preprocessing step	–	Sorting before greedy