

0.1 Binary search

```
def binary_search(data, key):
    if len(data) == 1:
        return data[0] == key
    mid = len(data)//2
    if key < data[mid]:
        return binary_search(data[:mid], key)
    else:
        return binary_search(data[mid:], key)
```

Linked-list: is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next.

Tree: Trees are recursive data structures, like the linked lists.

```
class BinaryTree:
    def __init__(self, item):
        self.item = item
        self.left = None # type = BinaryTree
        self.right = None # type = BinaryTree
    def insert(self, item):
        # pick a random side to insert on
        left = np.random.rand() < 0.5
        if item == left:
            return
        if left:
            if self.left is None:
                self.left = BinaryTree(item)
            else:
                self.left.insert(item)
        else:
            if self.right is None:
                self.right = BinaryTree(item)
            else:
                self.right.insert(item)
    def contains(self, item):
        if self.item == item:
            return True
        if self.left is not None:
            if self.left.contains(item):
                return True
        if self.right is not None:
            if self.right.contains(item):
                return True
        return False
    def print_tree(self, level=0, prefix="Root: "):
        """Recursively prints the tree structure."""
        print(" " * level + prefix + str(self.item))
        if self.left:
            self.left.print_tree(level + 1, prefix="L--- ")
        if self.right:
            self.right.print_tree(level + 1, prefix="R--- ")
```

nearest neighbour: time complexity for n points in k dimensions
 $O(nk)$

0.2 K-D Trees

One of the classic ways to speed up nearest neighbours is a data structure call the k-d tree. Basic idea:

In each recursive step, there is a certain number of datapoints. If there's only one, we're done.

Otherwise, for one of the two dimensions (we alternate back and forth), find the median value along the dimension.

Split the data into two subsets based on being above or below that median, and build a (sub)tree for each of those subsets.

Starting from the full dataset, you will create a tree where each leaf is a datapoint.

You can find an approximate nearest neighbour by traversing the down the tree using the same decision points as were used to original split the data; the final leaf is the desired neighbour.

Other nearest neighbour approaches

Note: there are other nearest neighbour approaches besides k-d trees, including some very fast approximate algorithms.

In general, you can often do something faster if the result can be slightly wrong.

There are approaches based on hashing instead of trees.

```
def nearest_neighbour(data, query):
    if query.ndim == 1:
        query = query[None]
    return np.argmin(np.sum((data - query)**2, axis=1))
```

0.3 Amortization of hash table growth

We say the up-front effort is amortized (or spread out) over the many queries. Hash table operations are amortized constant time, meaning that although occasional slower operations occur, their cost is spread out so the average cost per operation remains $O(1)$.

0.4 Dynamic Programming

Powerful technique for solving problems that can be broken down into simpler subproblems whose solutions can be reused. DP provides a way to cache intermediate results so that each subproblem is solved only once.

Optimal substructure — the optimal solution to the problem can be composed of optimal solutions to its subproblems.

Overlapping subproblems — the space of subproblems is small and the same subproblems are solved repeatedly.

Global sequence alignment in genetics, knapsack problem, matrix chain multiplication, shortest-path algorithms such as Floyd-Warshall.

Caching: is a general term that refers to the practice of storing data for later use. To solve the problem of redundant computation.

Memoization: is a specific form of caching that involves storing the results of function calls so that when the same inputs are encountered again, the previously computed result can be reused. This avoids the need to recompute values that have already been calculated. **Top-down**: starting from (n) and recursively computing smaller subproblems while caching results.

bottom-up/tabulation: Instead of recursion, we start from the simplest subproblems, such as ways(0) and ways(1), and iteratively build up to ways(n). Eliminates recursion overhead and can sometimes reduce memory usage.

0.5 Climbing Stairs (Counting Paths)

```
#Brute Force solution O(2^n)
def climb_stairs_recurse(n):
    if n == 0 or n == 1:
        return 1
    return climb_stairs_recurse(n-1) + climb_stairs_recurse(n-2)
# same subproblems (like climb(2)) are recomputed in many branches.
#DP Solution Time O(n) Space O(n)
def climb_stairs_memoized(n: int) -> int:
    if n < 0:
        raise ValueError("n must be >= 0")
    memo = [-1] * (n + 1) # -1 marks "not computed yet"
    def _rec(k: int) -> int:
        if k <= 1: # W(0)=1, W(1)=1
            return 1
        if memo[k] != -1:
            return memo[k]
        memo[k] = _rec(k - 1) + _rec(k - 2)
        return memo[k]
```

return _rec(n)

#DP Solution Bottom-up Time: O(n) Space: O(n)

```
def climb_stairs_tab(n: int) -> int:
    if n < 0:
        raise ValueError("n must be >= 0")
    if n <= 1:
        return 1
```

dp: List[int] = [0] * (n + 1)
dp[0] = 1
dp[1] = 1
for i in range(2, n + 1):
 dp[i] = dp[i - 1] + dp[i - 2]
return dp[n]

#DP Solution Bottom-up Time: O(n) Space: O(1)

```
def climb_stairs_tab(n: int) -> int:
    if n < 0:
        raise ValueError("n must be >= 0")
    if n <= 1:
        return 1
```

a, b = 1, 1 # dp[0], dp[1]
for _ in range(2, n + 1):
 a, b = b, a + b
return b

0.6 Fibonacci

```
#Recursive solution Time: O(2^n) Space : O(n)
def f(n):
    if n == 1 or n == 2:
        return 1
    else:
        return f(n-1) + f(n-2)
#iterative Solution
def fib(n):
    f = np.zeros(n+1, dtype=int)
    f[1] = 1
    for i in range(2,n+1):
        f[i] = f[i-1] + f[i-2]
    return f[-1]
#Memoization method functools.lru_cache stores the cached results
# in memory like in temp file (function results of fib (5))
memory = joblib.Memory("/tmp", verbose=0)
@memory.cache
def fib_recursive_cache2(n):
    if n == 0 or n == 1:
        return n
    return fib_recursive_cache2(n-1) + fib_recursive_cache2(n-2)
# Memoized Fibonacci DP top down Time: O(n) Space: O(n)
def fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n - 1, memo) + fib(n - 2, memo)
    return memo[n]
# Bottom-Up Fibonacci Time: O(n) Space: O(n)
def fib(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

Edit Distance Problem

```
def edit_distance_dp(s1: str, s2: str) -> int:
    n, m = len(s1), len(s2) # If s1 = "cat" and s2 = "cut", then n = 3 and m = 3.
    # The extra row and column handle cases where one substring is empty.
    dp = np.zeros((n + 1, m + 1), dtype=int)
    for i in range(1, n + 1):
        dp[i, 0] = i
    for j in range(1, m + 1):
        dp[0, j] = j
    #Note: For s1 = "cat" and s2 = "cut", the first column becomes [0, 1, 2, 3]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i, j] = dp[i - 1, j - 1]
            else:
                dp[i, j] = 1 + min(
                    dp[i - 1, j],
                    dp[i, j - 1],
                    dp[i - 1, j - 1])
    return int(dp[n, m])
```

Dynamic Programming (DP) Problem-Solving Template

Identify subproblems: Break the problem into smaller instances (e.g., prefixes, suffixes, or states like the last move).

Set base cases: Define simplest inputs and outputs to initialise the DP table.

Form recurrence: Express each subproblem in terms of smaller ones, ensuring no cyclic dependencies.

Choose strategy: Use *top-down* (recursion + memoization) or *bottom-up* (iterative tabulation).

Implement & test: Code the recurrence, verify on small examples, and confirm expected complexity.

Optimise: Reduce space/time if possible (e.g., keep only necessary previous states).