1. The “simple” Master method

T(n) = aT(n/b) + Θ(nd)

If d < logb a, then T(n) = Θ(nlogb a)

If d = logb a, then T(n) = Θ(ndlg n)

If d > logb a, then T(n) = Θ(nd)

* 1. 1. T(n) = 8T(n/2) + n3

log2 8 = 3 = d, so T(n) = Θ(n3lg n)

Given T(n/2) = Θ((n/2)3lg (n/2)) = c(n/2)3lg (n/2) we show T(n) = Θ(n3lg n) = cn3lg n.

T(n) = 8T(n/2) + n3

= 8c(n/2)3 lg n/2 + n3

…. (*typing maths in Google Docs sucks*)

= cn3 lg n

Need to prove T(n) <= c(n^3 lgn)

Assume this holds for all positive m < n (strong induction)

In particular for m <= n/2 yielding T(n/2) <= c((n/2)^3 lg(n/2))

By substitution:

Base case:

Say T(1) = 1

1 </= c(1^3\*lg(1)) = 0

T(2) = 16 <= c(8 \* lg(2)) = 8c holds for c >= 2

* + 1. T(n) = 9T(n/3) + n

log3 9 = 2 > 1 = d, so T(n) = Θ(n2)

To show Θ bound, separately show O and Ω bounds. For this question, we have to use the “subtract a lower order term” trick.

To prove T(n) <= cn2 - dn

Assume T(n/3) <= (n/3)2 - d(n/3)

T(n) <= 9c(n/3)2 - 9d(n/3) + n

…

= cn2 - 3dn + n

<= cn2 - dn

for d >= 1/2

Assume T(n/3) >= (n/3)2

T(n) >= 9(n/3)2 + n

= cn2 + n

>= cn2

* 1. 1. Not a fantastic algorithm, feel free to improve.

def closest\_number(n, arr):

closest = 0

for i in range(len(arr)):

if (abs(n - arr[i]) <= abs(n - arr[closest])):

closest = i

return arr[closest]

* + 1. O(n).

Alternative suggestion

I think this problem is solvable by a modified binary search. The input list is

Sorted so if the (mid > target) mid is our closest guess and we can discard the

Top half of the array. This would lead to a O(lg n) runtime

Alternative iterative binary search modification:

**find-closest**(A, target)

diff = inf

candidate = -1

left, right = 0, A.length - 1

**while** left <= right:

m = (left + right) / 2

**if** A[m] == target:

**return** A[m]

**else if** abs(target - A[m]) < diff:

candidate = A[m]

diff = abs(target - A[m])

**if** A[m] < target:

left = mid + 1

**else**:

right = mid - 1

**return** candidate

3rd Alternative solution:

int findClosest(int x, vector<int> arr, int start, int end) {

int m = (start + end) / 2;

if (arr[m] == x) { return x; }

if ((end - start) <= 1) { return arr[m]; }

int testClosest = INT32\_MAX;

if (x < arr[m]) {

testClosest = findClosest(x, arr, start, m);

} else {

testClosest = findClosest(x, arr, m, end);

}

if (abs(x - arr[m]) < abs(x - testClosest)) {

return arr[m];

} else {

return testClosest;

}

}

int findClosest(int x, vector<int> arr) {

return findClosest(x, arr, 0, arr.size());

}

* 1. 1. def differing\_element(a, b):

for i in range(len(b)):

if a[i] != b[i]:

return i

return len(a) - 1

* + 1. O(n).

Alternative solution: I think that the question specifies that there is an additional element (not differing element) in a. Perhaps something like this might work:

additional :: Array Int Int -> Array Int Int -> Int

additional a b = additional' a b (bounds a)

where

additional' a b (l, u)

| l == u = l

| a ! m == b ! m = additional' a b (m+1, u)

| otherwise = additional' a b (l, m)

where

m = (l + u) `div` 2

Complexity would be log(n)

* + 1. Reverse the string and do longest common subsequence with reversed string and input.

int longestPalindromicSubsequence(String s) {

String t = s.clone();

t.reverse();

int[][] arr = new int[s.length() + 1][t.length() + 1];

for (int i = 1; i < s.length(); i++) {

for (int j = 1; j < t.length(); j++) {

if (s.charAt(i - 1) == t.charAt(j - 1)) {

arr[i][j] = arr[i-1][j-1] + 1;

} else {

arr[i][j] = Math.max(arr[i-1][j], arr[i][j-1]);

}

}

}

return arr[s.length()][t.length()];

}

Or in Python:

def longest\_palindromic\_subsequence(s: str) -> int:

s\_r = s[::-1] # Reverse string

return longest\_common\_subsequence(s, s\_r)

def longest\_common\_subsequence(a, b):

L = [[0 for j in range(len(b)+1)] for i in range(len(a)+1)]

for i in range(len(a)+1):

for j in range(len(b)+1):

if i == 0 or j == 0:

L[i][j] = 0

elif a[i-1] == b[j-1]:

L[i][j] = L[i-1][j-1] + 1

else:

L[i][j] = max(L[i-1][j], L[i][j-1])

return L[len(a)][len(b)]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | N | K | O | O | N | J |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| J | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| N | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| O | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| O | 0 | 1 | 1 | 2 | 3 | 3 | 3 |
| K | 0 | 1 | 2 | 2 | 3 | 3 | 3 |
| N | 0 | 1 | 2 | 2 | 3 | 4 | 4 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | N | K | O | O | N | J |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| J | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| N | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| O | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| O | 0 | 1 | 1 | 2 | 3 | 3 | 3 |
| K | 0 | 1 | 2 | 2 | 3 | 3 | 3 |
| N | 0 | 1 | 2 | 2 | 3 | 4 | 4 |

Subsequence is NOON

Alternative solution

int longestSubstring(string s) {

int arr[s.size()][s.size()];

for (int i = 0; i < s.size(); i++) {

for (int j = 0; j < s.size(); j++) {

arr[i][j] = 0;

}

}

for (int j = 0; j < s.size(); j++) {

for (int i = j; i >= 0; i--) {

if (i == j) {

arr[i][j] = 1;

} else if (s[i] == s[j]) {

arr[i][j] = arr[i + 1][j - 1] + 2;

} else {

arr[i][j] = max(arr[i + 1][j], arr[i][j - 1]);

}

}

}

return arr[0][s.size() - 1];

}

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | N | K | O | O | N | J |
| N | 1 | 1 | 1 | 2 | 4 | 4 |
| K | 0 | 1 | 1 | 2 | 2 | 2 |
| O | 0 | 0 | 1 | 2 | 2 | 2 |
| O | 0 | 0 | 0 | 1 | 1 | 1 |
| N | 0 | 0 | 0 | 0 | 1 | 1 |
| J | 0 | 0 | 0 | 0 | 0 | 1 |

We only care about diagonal movements, so the only characters that count are N and O ( the ones in darker blue) and since it’s a palindrome, we flip them too, so: NOON

1. ai) can get O(nlogk) complexity: use min heap to store first k elements, compare elements to minheap root, pop root and push new element if its bigger. At the end of list pop heap for result.
   1. 1. This can be done using a priority queue or just sort it.

Something like:

Convert array to a hash set (O(n ))

Merge sort it (O(nlog(n)))

Return kth element or some error if less than k unique elements exist

def largestK(k, arr):

li = list(dict.fromkeys(arr).keys())

li.sort(reverse=True)

print(li)

return li[k-1]

* + 1. O(n log n)

Alternative to 2) a) is quickselect (identical to quicksort except we don’t bother expanding both sides, only the side in which the kth largest element will be in, using the fact that the index of an element selected to be a pivot will never change after it is processed). Something along the lines of:

def quickselect(arr, k, lo, hi):

p = random\_partition(arr, lo, hi)

If p == k:

Return arr[p]

Elif p < k:

Return quickselect(arr, k, p+1, hi)

Else:

Return quickselect(arr, k, lo, p-1)

Def random\_partition(arr, lo, hi):

Pivot = random(lo, hi)

swap arr[pivot] and arr[hi]

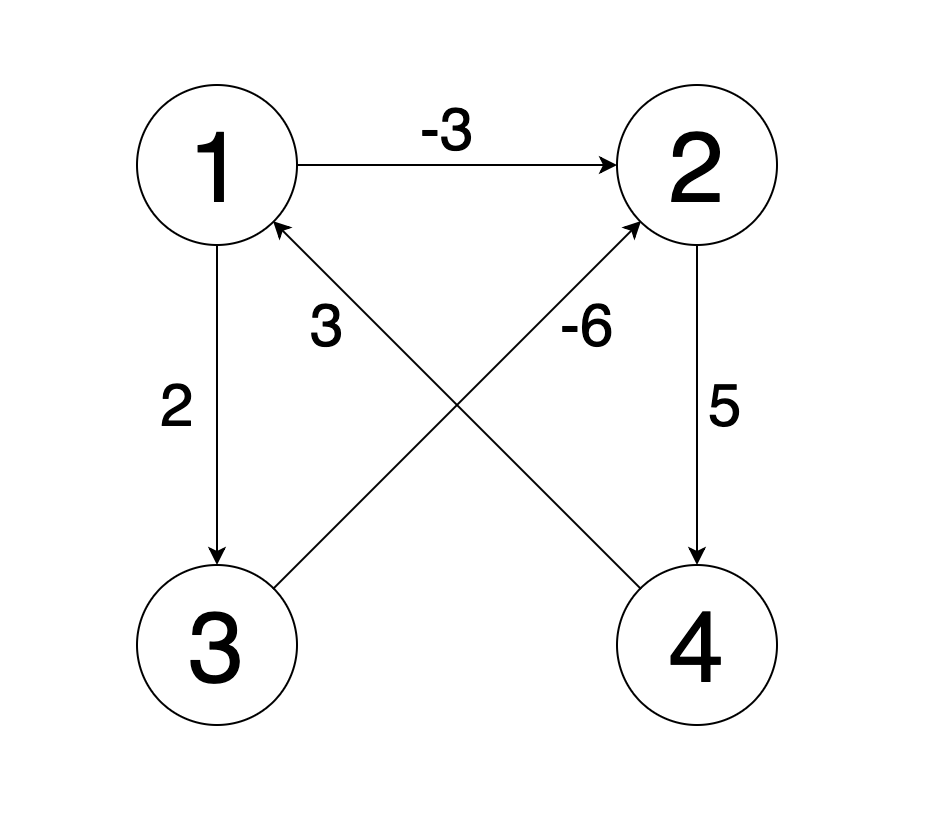
Return partition(arr, lo, hi)

Where partition is the standard quicksort partition function

This is O(n) average complexity (but O(n^2) worst case, so the other answer might be a safer choice).

* 1. 1. π[q] is the length of the longest prefix of P that is a suffix of Pq.
     2. Not examinable.
     3. Not examinable.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| P[i] | m | a | m | a | m | i | a |
| π[i] | 0 | 0 | 1 | 2 | 3 | 0 | 0 |

* 1. 1. 

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | -4 | 2 | 1 |
| 8 | 0 | 10 | 5 |
| 2 | -6 | 0 | -1 |
| 3 | -1 | 5 | 0 |



def is\_a\_subsequence(a, b):

i = 0

j = 0

while i < len(a) and j < len(b):

if a[i] == b[j]:

i += 1

j += 1

return i == len(a)

* + 1. O(n). Design paradigm is greedy.