Design and Analysis of Algorithms

PSet-1

31 August 2025

This is a compilation of solutions for PSet-1 of Design and Analysis of Algorithms class instructed by Prof. Siddharth Pritam. The solutions are of original design.

Table of contents

§0. Problems

1. Show that for any positive integer n: (a) Prove or Disprove: $n! \overset{?}{\in} \Theta(n^n)$

   (b) Prove: $\log(n!) = \Theta(n \log n)$

   (c) Prove: $H_n = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$

2. (a) Suppose we are given two sorted arrays $A[1...n]$ and $B[1...n]$. Describe an algorithm to find the median of the union of A and B in $O(\log n)$ time. Assume the arrays contain no duplicate elements.

   (b) Now suppose we are given three sorted arrays $A[1...n]$, $B[1...n]$ and $C[1...n]$. Describe an algorithm to find the median element of $A \cup B \cup C$ in $O(\log n)$ time.

   (c) You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the kth smallest element in the union of the two lists.

3. You are given an array of N integers in which the first n of them are positive and the rest are negative, but you do not know n. Give an

   (a) O(log N) algorithm to find n.

   (b) O(log n) algorithm to find n.

4. Given a list of n elements, give an efficient algorithm to determine whether they are all distinct.

5. Solve the following recurrence relations and give a $\Theta(\cdot)$ bound for each of them.

   Assume that the functions evaluate to a constant for $n = 1$.

   (a) $T(n) = 2T\left(\frac{n}{3}\right) + 1$

   (b) $T(n) = 5T\left(\frac{n}{4}\right) + n$

   (c) $T(n) = T(n-1) + 2$

   (d) $T(n) = 2T(n-1) + 1$

   (e) $T(n) = T(\sqrt{n}) + 1$

6. Suppose you are choosing between the following three algorithms:

   (a) Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

   (b) Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.

   (c) Algorithm C solves problems of size n by dividing them into 9 subproblems of size $\frac{n}{3}$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

   What are the running times of each of these algorithms (in Big-O notation), and which would you choose?

7. Given two sets S_1 and S_2, and a real number x, find whether there exists an element from S_1 and an element from S_2 whose sum is exactly x. The algorithm should run in time $O(n \log n)$, where n is the total number of elements in both sets.

§1. Problem 1

Exercise

Show that for any positive integer n: (a) Prove or Disprove: $n! \overset{?}{\in} \Theta(n^n)$

(b) Prove: $\log(n!) = \Theta(n \log n)$

(c) Prove: $H_n = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$

Proof (a). FTSOC let there exist $0 < k_1 < k_2; k_1, k_2 \in \mathbb{R}$ such that $\forall n \geq n_0 \in \mathbb{Z}$,

$$k_1 n^n < n! < k_2 n^n$$

$$k_1 < \frac{n!}{n^n} < k_2$$

$$\Rightarrow k_1 < \lim_{n\to\infty} \frac{n!}{n^n} < k_2$$

But $0 \leq \frac{n!}{n^n} < \frac{n}{n}\frac{n-1}{n}\frac{n-2}{n}...\frac{1}{n} < \frac{1}{n}$ which forces $\lim_{n\to\infty} \frac{n!}{n^n} = 0$ by squeeze theorem. Which is a contradiction as $k_1 < 0 < k_2$ is inconsistent with $k_1 > 0$. Thus, $n! \notin \Theta(n^n)$. □

Proof (b). We claim that $\frac{1}{2} n \log(n) < \log(n!) < n \log(n)$ for all $n > 3$. This is equivalent to

$$\sqrt{n^n} < n! < n^n$$

The right hand side of the inequality is trivial as $n! = n * (n-1) * ...1 < n * n * ... * n = n^n$.

The left hand side follows from

$$\begin{aligned}(2n)! &= 2n * (2n-1) * ... * 1 \\ &= (2n * 1) * ((2n-1) * 2) * ... * (n * n) \\ &> 2n * 2n * ...2n \\ &= (2n)^n\end{aligned}$$

and

$$\begin{aligned}(2n-1)! &= (2n-1) * (2n-2) * ... * 1 \\ &= ((2n-1) * 1) * ((2n-2) * 2) * ... * ((n+1) * (n-1)) * n \\ &> (2n-1) * (2n-1) * ... * (2n-1) * \sqrt{2n-1} \\ &= (2n-1)^{\frac{2n-1}{2}}\end{aligned}$$

for $n > 3$ as $ab \geq a + b$ for $a, b > 1$ and $x > \sqrt{x}$ for $x > 1$ and $3 > 1$. □

Proof (c). As $\frac{1}{x}$ is decreasing,

$$\lfloor x \rfloor \leq x \leq \lceil x \rceil$$
$$\Rightarrow \frac{1}{\lfloor x \rfloor} \geq \frac{1}{x} \geq \frac{1}{\lceil x \rceil}$$
$$\Rightarrow \int_1^n \frac{1}{\lfloor x \rfloor}\,\mathrm{d}x \geq \int_1^n \frac{1}{x}\,\mathrm{d}x \geq \int_1^n \frac{1}{\lceil x \rceil}\,\mathrm{d}x$$
$$\Rightarrow 1 + \frac{1}{2} + ... + \frac{1}{n-1} \geq \ln(n) \geq \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$
$$\Rightarrow H_{n-1} \geq \ln(n) \geq H_n - 1$$

This implies

$$H_n \geq \ln(n+1)$$
$$\Rightarrow H_n = \Omega(\log n)$$

and

$$H_n \leq \ln(n) + 1$$
$$\Rightarrow H_n = O(n)$$

which implies $H_n = \Theta(\log n)$. □

§2. Problem 2

Exercise

(a) Suppose we are given two sorted arrays $A[1...n]$ and $B[1...n]$. Describe an algorithm to find the median of the union of A and B in $O(\log n)$ time. Assume the arrays contain no duplicate elements.

(b) Now suppose we are given three sorted arrays $A[1...n]$, $B[1...n]$ and $C[1...n]$. Describe an algorithm to find the median element of $A \cup B \cup C$ in $O(\log n)$ time.

(c) You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the kth smallest element in the union of the two lists.

Solution. (a)

MEDIAN OF TWO SORTED LISTS

```
1  i_a := 1
2  j_a := n
3  i_b := 1
4  j_b := n
5  for i_a ≠ j_a and i_b ≠ j_b:
6      a = A[⌊(i_a + j_a)/2⌋]
7      b = B[⌊(i_b + j_b)/2⌋]
8      if a > b
9          j_a = ⌊(i_a + j_a)/2⌋
10         i_b = ⌊(i_b + j_b)/2⌋
11     if a < b
12         i_a = ⌊(i_a + j_a)/2⌋
13         j_b = ⌊(i_b + j_b)/2⌋
14 return min(A[i_a], B[i_b])
```

Proof of Correctness. We prove the correctness of given algorithm by induction on n.

(B) For $n = 1$, we are done as we take the lower of the two.

(S) If our algorithm works for all $k \leq n$, we will show that it works for $n + 1$. In the first iteration of the for loop, we reject the elements smaller than the small median and greater than big median as they can't be the median as there are more than n elements greater and smaller than them respectively.

This reduces the problem to finding the median of union of two sorted lists of length $\frac{n}{2}$. Our algorithm can do so by the induction hypothesis. Thus, the algorithm works. □

We do a consent amount of work every for loop. This leads to the reccurence:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$$\Rightarrow T(n) = 1T\left(\frac{n}{2}\right) + O(n^0)$$

$$\text{As } \frac{1}{2^0} = \frac{1}{1} = 1$$

$$\therefore T(n) = O(n^0 \log(n)) = O(\log n)$$

(b)

MEDIAN OF THREE SORTED ARRAYS

```
1  i_a := 1, j_a := n
2  i_b := 1, j_b := n
3  i_c := 1, j_c := n
4  while at least two arrays have more than one element:
5      m_a := A[⌊(i_a + j_a)⌋ / 2]
6      m_b := B[⌊(i_b + j_b)⌋ / 2]
7      m_c := C[⌊(i_c + j_c)⌋ / 2]
8      min_med := min(m_a, m_b, m_c)
9      max_med := max(m_a, m_b, m_c)
10     if m_a = min_med:
11         remove_count := min((⌊(i_a + j_a)⌋ / 2) − i_a + 1, j_c − (⌊(i_c + j_c)⌋ / 2))
12         i_a := i_a + remove_count
13     else if m_b = min_med:
14         remove_count := min((⌊(i_b + j_b)⌋ / 2) − i_b + 1, j_c − (⌊(i_c + j_c)⌋ / 2))
15         i_b := i_b + remove_count
16     else:
17         remove_count := min((⌊(i_c + j_c)⌋ / 2) − i_c + 1, j_c − (⌊(i_c + j_c)⌋ / 2))
18         i_c := i_c + remove_count
19     if m_a = max_med:
20         j_a := j_a − remove_count
21     else if m_b = max_med:
22         j_b := j_b − remove_count
23     else:
24         j_c := j_c − remove_count
25 if only one array has elements remaining:
26     return median of that array segment
27 else if two arrays have elements remaining:
28     return median of two sorted arrays (using two-array algorithm)
```

Proof of Correctness. We prove the correctness of given algorithm by induction on total number of elements in all the lists.

(B) For $n = 1$, we are done as we take the last element as

(S) If our algorithm works for all $k \leq n$, we will show that it works for $n+1$. In the first iteration of the for loop, we reject the elements smaller than the smallest median and greater than biggest median as they can't be the median as there are more than $3\frac{n}{2}$ elements greater and smaller than them respectively. The fact that we are deleting the same number of elements on both the sides, keeps the median invariant.

This allows us the induction hypothesis to finish the problem. □

Notice that starting with $k \leq l \leq m$ sized lists, we delete atleast $\frac{k}{2} + \frac{k}{2}$ elements. This implies

$$T(k+l+m) = T(l+m) + O(1)$$
$$T(k+l+m) < T(2m) + O(1)$$
$$T(3k) < T(2m) + O(1)$$

Where k is the length of the shortest list and m the longest. As we start with $k = l = m = n$,

$$T(3n) < T(2n) + O(1)$$
$$\Rightarrow T(3n) = O(\log(3n)) = O(\log n)$$

(c)

K-TH ELEMENT IN UNION OF TWO SORTED LISTS OF DIFFERENT SIZE

```
1  def kthSmallest(A[1...m], B[1...n], k)
2      if k == 1:
3          return min(A[1], B[1])
4      i = min(m, k/2)
5      j = min(n, k/2)
6      if A[i] > B[j]:
7          return kthSmallest(A, B[j+1...n], k-j)
8      ELSE:
9          return kthSmallest(A[i+1...m], B, k-i)
```

Proof of Correctness. The correctness follows by strong induction on k and m, n.

For $k = 1$, we take the minimum of the first indecies of the two lists.

If we assume the algorithm is correct for $a \leq k$ and lists of size $b \leq \max(m, n)$; We can show the algorithm is correct for $k+1$ and list of size (m, n); $(m+1, n)$; $(m, n+1)$; $(m+1, n+1)$.

This follows from the recursion scheme being correct which is obvious to notice. □

We only need to consider the first k elements in each of the lists. That implies that we half one of the lists every loop. That implies

$$T(m, n, k) = O(\log(\min(m, k)) + \log(\min(n, k))) = O(\log(m) + \log(n))$$

for most values of m, n, k. □

§3. Problem 3

Exercise

You are given an array of N integers in which the first n of them are positive and the rest are negative, but you do not know n. Give an

(a) O(log N) algorithm to find n.

(b) O(log n) algorithm to find n.

Solution.

(a)

BINARY SEARCH

```
low = 1
high = N
while low ≠ high:
    mid = ⌊(low+high)/2⌋
    if A[mid] > 0:
        low = mid
    else:
        high = mid
return low
```

Proof of Correctness. We shall show the proof of correctness by induction.

(B) For $N = 1$, we are trivially done.

(S) Assuming our algorithm works for all $k \leq N$, we will show the correctness for $N + 1$. In the first iteration of the for loop, we reject half the list as the turning point would be below or above it respectively; and by then by the induction hypothesis we are done. □

The time complexity is

$$T(N) = T\left(\frac{N}{2}\right) + O(1)$$
$$\Rightarrow T(N) = O(\log N)$$

using the same calculation as above.

(b)

ONE DIRECTIONAL BINARY SEARCH

```
itt = 1
while True:
    if itt*2 > N:
        BinarySearch(itt, N)
```

ONE DIRECTIONAL BINARY SEARCH

```
5           exit loop
6       else if A[itt] > 0 and A[itt*2] < 0:
7           BinarySearch(itt, itt*2)
8           exit loop
9       itt = 2*itt
```

Proof of Correctness. The algorithm is correct as for all $n \in Z$ as there will always exist k such that $2^k \leq n < 2^{k+1}$ which our algorithm will sandwich it between. □

Notice that for all $n < 2^k$, we take $O(k) + O\big(\log\big(2^k - 2^{k-1}\big)\big)$ time. This implies:

$$\begin{aligned} T(n) &< O(\log n) + O\big(\log\big(2^{\log(n)} - 2^{\log(n)-1}\big)\big) \\ \Rightarrow T(n) &< O(\log n) + O(\log n - 1) \\ \Rightarrow T(n) &= O(\log n) \end{aligned}$$

□

§4. Problem 4

Exercise

Given a list of n elements, give an efficient algorithm to determine whether they are all distinct.

Proof.

Claim 5.1. Given that the elements can only be compared to be equal or not but don't have a canonical order, the worst case time complexity of the most efficient algorithm is bounded by $\Theta(n^2)$.

Proof. Without a canonical order, it is imperative that we need to check all $\binom{n}{2}$ equalities. This would make the time complexity atleast $\Theta(n^2)$. We can show that this is sharp by realizing the complexity.

Distinct Element

```
for i in A:
    for j in [i+1, n]:
        if A[i] == A[j]:
            return False
return True
```

□

We can although be faster when the elements admit an order.

Distinct Elements

```
distinct lis = check $ sort lis
check [] = True
check [x] = True
check (x:y:ys) = if x == y then False else check (y:ys)
```

This is written in Haskell psudocode as it is much easier for me in this case.

Proof of Correctness. The algorithm clearly works as intended as if two adjacent elements are not equal, we can be sure the smaller is not equal to any of the following elements by the property of being ordered. □

Notice,

$$T(n) = O(n \log n) + O(n) = O(n \log n)$$

as intended. □

§5. Problem 5

Exercise

Solve the following recurrence relations and give a $\Theta(\cdot)$ bound for each of them.

Assume that the functions evaluate to a constant for $n = 1$.

(a) $T(n) = 2T(\frac{n}{3}) + 1$

(b) $T(n) = 5T(\frac{n}{4}) + n$

(c) $T(n) = T(n-1) + 2$

(d) $T(n) = 2T(n-1) + 1$

(e) $T(n) = T(\sqrt{n}) + 1$

Solution.

(a)

Claim 6.1. $T(n) = \Theta\left(n^{\log_3(2)}\right)$

For $T(1)$, it takes constant time and hence, the hypothesis holds.

If we assume the hypothesis to be true for all $k \leq n-1$, we will show this is true for n.

$$T(n) = 2T\left(\frac{n}{3}\right) + 1$$

$$\Rightarrow T(n) = 2\Theta\left(\left(\frac{n}{3}\right)^{\log_3(2)}\right) + 1$$

$$\Rightarrow T(n) = \Theta\left(n^{\log_3(2)}\right) + 1 = \Theta\left(n^{\log_3(2)}\right)$$

(b)

Claim 6.2. $T(n) = \Theta\left(n^{\log_4(5)}\right)$

For $T(1)$, it takes constant time and hence, the hypothesis holds.

If we assume the hypothesis to be true for all $k \leq n-1$, we will show this is true for n.

$$T(n) = 5T\left(\frac{n}{4}\right) + n$$

$$\Rightarrow T(n) = 5\Theta\left(\left(\frac{n}{4}\right)^{\log_4(5)}\right) + n$$

$$\Rightarrow T(n) = \Theta\left(n^{\log_4(5)}\right) + n = \Theta\left(n^{\log_4(5)}\right)$$

(c)

$$
\begin{aligned}
T(n) &= T(n-1) + 2 \\
\Rightarrow T(n) &= T(n-k) + 2k \\
\Rightarrow T(n) &= T(1) + 2 * (n-1) \\
\Rightarrow T(n) &= \Theta(n)
\end{aligned}
$$

(d)

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
\Rightarrow T(n) &= 2(2 * T(n-2) + 1) + 1 \\
&\vdots \\
\Rightarrow T(n) &= 2^n T(1) + 2^{n-2} + ... + 1 \\
\Rightarrow T(n) &= 2^n T(1) + 2^{n-1} - 1 \\
\Rightarrow T(n) &= \Theta(2^n)
\end{aligned}
$$

(e) Let $R(k) = T\left(2^k\right) \iff T(n) = R(\log n)$. The recursion on R is:

$$
\begin{aligned}
R(k) &= R\left(\frac{k}{2}\right) + 1 \\
\Rightarrow R(k) &= O(\log n) \\
\Rightarrow T(n) &= O(\log \log n)
\end{aligned}
$$

□

§6. Problem 6

Exercise

Suppose you are choosing between the following three algorithms:

(a) Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

(b) Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.

(c) Algorithm C solves problems of size n by dividing them into 9 subproblems of size $\frac{n}{3}$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in Big-O notation), and which would you choose?

Solution.

(a) The statement reduces to this recurrence $T(n) = 5T(\frac{n}{2}) + O(n)$. By Masters theorem, $a = 5, b = 2$ and $d = 1$ we would get the time complexity $O(n^{\log(5)})$.

(b) The statement reduces to this recurrence $T(n) = 2T(n-1) + O(1)$. This implies

$$
\begin{aligned}
T(n) &= 2T(n-1) + O(1) \\
\Rightarrow T(n) &= 2(2 * T(n-2) + O(1)) + O(1) \\
&\vdots \\
\Rightarrow T(n) &= 2^n T(1) + 2^{n-2} O(1) + \ldots + 1O(1) \\
\Rightarrow T(n) &= 2^n T(1) + (2^{n-1} - 1)O(1) \\
\Rightarrow T(n) &= O(2^n)
\end{aligned}
$$

(c) The statement reduces to $T(n) = 9T(\frac{n}{3}) + O(n^2)$. By Masters theorem, $a = 9, b = 3$ and $d = 2$, we would get the time complexity $O(n^2 \log(n))$.

I would use algorithm C as $O(n^2 \log(n)) < O(n^{\log(5)}) < O(2^n)$ as

- $O(\log n) < O(n^{\varepsilon})$
- $5 > 4 = 2^2$
- $O(n^p) < O(2^n)$

where $O(f(n)) < O(g(n))$ if there exist an n_0 such that $\forall n > n_0$, $f(n) < g(n)$. □

§7. Problem 7

Exercise

Given two sets S_1 and S_2, and a real number x, find whether there exists an element from S_1 and an element from S_2 whose sum is exactly x.

The algorithm should run in time $O(n \log n)$, where n is the total number of elements in both sets.

Proof.

Solution to problem 7

```
answer = solve toList S1 reverse.toList S2 x
solve [] [] _ = False
solve _ [] _ = False
solve [] _ _ = False
solve (y:ys) (z:zs) target =
    if y + z == target
        then True
    else
        if y+z > target
            then solve (y:ys) zs
        else
            solve ys (z:zs) target
```

This has been written as haskell psudocode as I want to.

Proof of Correctness. We claim we keep the loop invariant as the fact that, if their exist a_i, b_j such that $a_i + b_j = x$, then it will be in our search space.

This is true initially as our original space is $A[1...r] \times B[1...s]$. Every iteration, we terminate the programme if the sum hits the target. We discard the larger element if the sum is too large and vice versa.

This makes the sample space smaller every iteration as well as maintains the invariant property. Thus, ensuring termination and correctness. □

We already spend $O(r \log r + s \log s)$ time converting the sets to lists.

Furthermore, we reduce one of the lists every iteration. Thus,

$$\begin{aligned} T(r+s) &= T(r+s-1) + O(1) \\ \Rightarrow T(n) &= T(n-1) + O(1) \quad \text{using r+s = n} \\ \Rightarrow T(n) &= O(n) \end{aligned}$$

Thus, The time complexity of the algorithm is $O(r \log r + s \log s + r + s) = O(n \log n)$.

□