

(1)What are the different approaches to prove the correctness of an algorithm? Explain Loop Invariants associated with an algorithm? Use binary search algorithm to discuss the process of proving correctness?

Solution: Approaches for algorithms correctness verification

Experimental analysis (testing).

- We test the algorithm for **different instances** of the problem (for different input data).
- The main advantage of this approach is its **simplicity** while the main disadvantage is the fact that **testing cannot cover always all possible instances of input data** (it is difficult to know how much testing is enough).
- However, the experimental analysis allows sometimes to identify situations when the algorithm doesn't work.

Formal analysis (proving).

- The aim of the formal analysis is to prove that the algorithm works for **any instance of data input**.
- The main advantage of this approach is that if it is rigourously applied it guarantee the correctness of the algorithm.
- The main disadvantage is the difficulty of finding a proof, mainly for complex algorithms.

- In this case the algorithm is decomposed in subalgorithms and the analysis is focused on these (simpler) subalgorithms.
- On the other hand, the formal approach could lead to a better understanding of the algorithms.

Loop Invariant

Logical expression with the following properties.

- Holds true before the first iteration of the loop – **Initialization.**
- If it is true before an iteration of the loop, it remains true before the next iteration – **Maintenance.**
- When the loop terminates, the **invariant – along with the fact that the loop terminated** – gives a useful property that helps show that the loop is correct – **Termination.**

Binary Search

```
def binary_search(A, target):
    lo = 0
    hi = len(A) - 1
    while lo <= hi:
        mid = (lo + hi) / 2
        if A[mid] == target:
            return mid
        elif A[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
```

You've all seen this a billion times.

But how do we prove that it's correct?

Given that A is sorted and A contains target, prove that `binary_search(A, target)` always returns target's index within A

Use Loop Invariants!!

Step 1: Construct a Loop Invariant

Say we're searching for 14 in the following array A

0	1	2	3	4	5	6
-5	10	14	33	42	42	42

2nd step: lo = 0, hi = 2, mid = 1

THIS IS OUR LOOP INVARIANT.

0	1	2	3	4	5	6
-5	10	14	33	42	42	42

↑
 lo
 hi

$$A[lo] \leq target \leq A[hi]$$


Step 2: Prove that loop invariant is inductive

- Base Case: when the algorithm begins, $lo = 0$ and $hi = \text{len}(A) - 1$. lo and hi enclose ALL values, so **target** must be between lo and hi .
- Inductive Hypothesis: suppose at any iteration of the loop, lo and hi still enclose the **target** value.
- Inductive Step:
 - Case 1: If $A[mid] > \text{target}$, then the target must be between lo and mid
 - We update $hi = mid - 1$
 - Case 2: If $A[mid] < \text{target}$, then the target must be between mid and hi
 - we update $lo = mid + 1$
 - In either cases, we preserve the inductive hypothesis for the next loop

Step 3: Prove correctness property using loop invariant

- Notice for each iteration, lo always increases and hi always decreases. These value will converge at a single location where $lo = hi$.

0	1	2	3	4	5	6
-5	10	14	33	42	42	42



- By the induction hypothesis, $A[lo] \leq \text{target} \leq A[hi]$.

Food for thought: How will the proof change if **target** isn't in the array?

(2) Correctness proof for - Linear Search, insertion Sort, array reversal

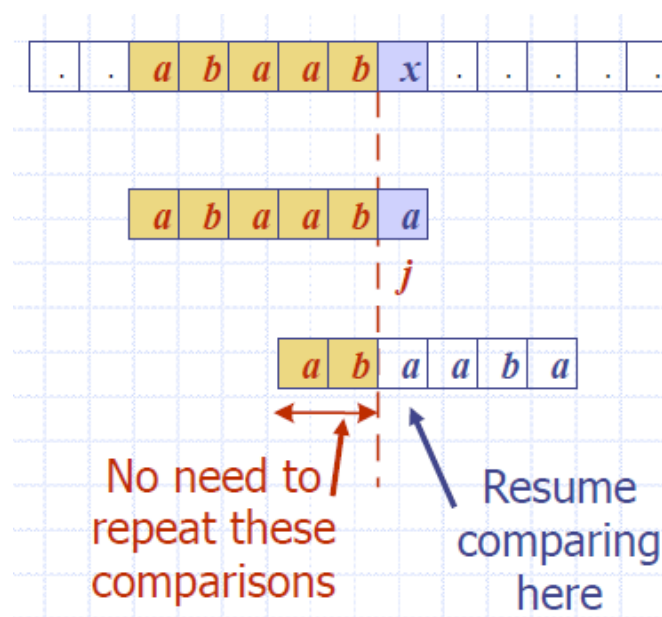
(3) What is the main idea of Knuth-Morris-Pratt (KMP) algorithm for pattern matching? Write the KMP algorithm to find the match for a given a test string T of length n and a pattern string P of length m?

The main idea of the KMP algorithm is to precompute self-overlaps between portions of the pattern so that when a mismatch occurs at one location, we immediately know the maximum amount to shift the pattern before continuing the search.

Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.

When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?

Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$



KMP algorithm:

```

Algorithm KMPMatch(T, P)
  F ← failureFunction(P)
  i ← 0
  j ← 0
  while i < n
    if T[i] = P[j]
      if j = m - 1
        return i - j { match }
      else
        i ← i + 1
        j ← j + 1
    else
      if j > 0
        j ← F[j - 1]
      else
        i ← i + 1
  return -1 { no match }

```

where the **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$

(4) Discuss the concept of pattern-matching. What are the advantages of Boyer-Moore method over Brute-Force method to match a pattern P in a given string T . Explain the concept of looking-glass heuristic and character- jump heuristic used in Boyer-Moore algorithm?

In the classic pattern-matching problem, we are given a text string T of length n and a pattern string P of length m , and want to find whether P is a substring of T . If so, we may want to find the lowest index j within T at which P begins, such that $T[j:j+m]$ equals P , or perhaps to find all indices of T at which pattern P begins.

Advantages: Boyer-Moore's pattern matching algorithm takes less number of comparisons than Brute Force approach. The basic idea is to do as less comparisons

as possible and eliminate the unnecessary character comparisons.

Heuristics:

The Boyer-Moore's pattern matching algorithm is based on two heuristics

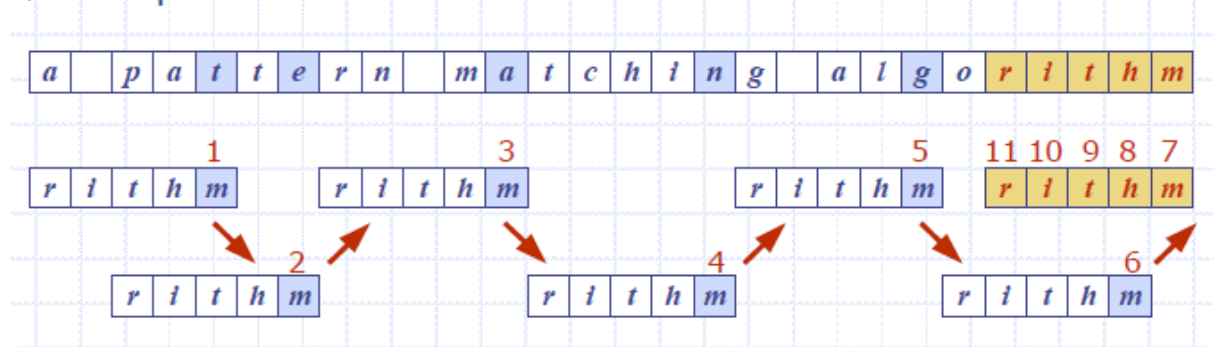
Looking-glass heuristic: Compare P with a subsequence of T moving backwards

Character-jump heuristic: When a mismatch occurs at $T[i] = c$

- If P contains c , shift P to align the last occurrence of c in P with $T[i]$
- Else, shift P to align $P[0]$ with $T[i + 1]$

The looking-glass heuristic sets up the other heuristic to allow us to avoid comparisons between P and whole groups of characters in T . In this case at least, we can get to the destination faster by going backwards, for if we encounter a mismatch during the consideration of P at a certain location in T , then we are likely to avoid lots of needless comparisons by significantly shifting P relative to T using the character-jump heuristic. The character-jump heuristic pays off big if it can be applied early in the testing of a potential placement of P against T .

◆ Example



Algorithm *BoyerMooreMatch*(T, P, Σ) $L \leftarrow \text{lastOccurrenceFunction}(P, \Sigma)$ $i \leftarrow m - 1$ $j \leftarrow m - 1$ **repeat****if** $T[i] = P[j]$ **if** $j = 0$ **return** i { match at i }**else** $i \leftarrow i - 1$ $j \leftarrow j - 1$ **else**

{ character-jump }

 $l \leftarrow L[T[i]]$ $i \leftarrow i + m - \min(j, 1 + l)$ $j \leftarrow m - 1$ **until** $i > n - 1$ **return** -1 { no match }**(5) Compute a table representing the KMP failure function for the pattern “Amalgamation”**

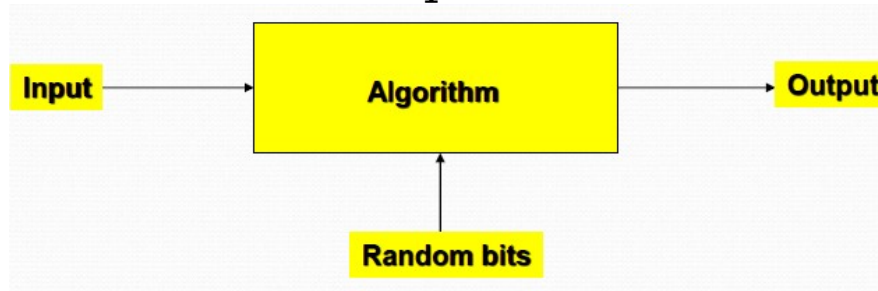
To implement the KMP algorithm, we will precompute a failure function, f , that indicates the proper shift of P upon a failed comparison. Specifically, the failure function $f(k)$ is defined as the length of the longest prefix of P that is a suffix of $P[1:k+1]$ (note that we did not include $P[0]$ here, since we will shift at least one unit). Intuitively, if we find a mismatch upon character $P[k+1]$, the function $f(k)$ tells us how many of the immediately preceding characters can be reused to restart the pattern.

The Knuth-Morris-Pratt (KMP) failure function, $f(k)$, for the string P is as shown in the following table:

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

(6) Explain what are randomized algorithms? Why we need such algorithms? What is the complexity class Randomized Polynomial time (RP)?

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation



Why Randomization ?

- Randomness often helps in significantly reducing the work involved in determining a correct choice when there are several but finding one is very time consuming.
- Reduction of work (and time) can be significant on the average or in the worst case.
- Randomness often leads to very simple and elegant approaches to solve a problem or it can improve the performance of the same algorithm.

Advantages of Randomized Algorithm

- Randomized algorithms runs faster than the known best deterministic algorithm.
- The randomized algorithms are simple to describe and implement than the deterministic algorithms for comparable performance
- The randomized algorithms also yields better complexity bounds

Randomized Polynomial time (*RP*)

Definition: The class *RP* consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^* :

$$\begin{aligned}x \in L &\Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{1}{2} \\x \notin L &\Rightarrow \Pr[A(x) \text{ accepts}] = 0\end{aligned}$$

- Independent repetitions of the algorithms can be used to reduce the probability of error to exponentially small.
- Notice that the success probability can be changed to an inverse polynomial function of the input size without affecting the definition of *RP*. Why?

(7)What is a randomized algorithm? What is the classification of randomized algorithms? Write a Monte Carlo randomized algorithm for solving 0-1 knapsack problem? Comment on class to which you algorithm belongs?

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation

Classification of Randomized Algorithm

1. Numerical Probabilistic Algorithm
2. Monte-carlo Algorithm
3. Las-Vegas Algorithm
4. Sherwood Algorithm



Numerical Probabilistic Algorithm

- Applied to the problems, when it is not possible to arrive at closed form solutions because of the uncertainties involved in the data or because of the limitations of a digital computer in representing irrationals.
- Approximation of such numerical values are to be computed (estimated) through simulation using randomness
- Quality of the solution produced by these algorithms can be improved by running the algorithms for a longer time.

Example: Average no of packets in router

Monte-carlo Algorithms

- Always gives an answer but the answer is not necessarily correct
- Probability of correctness can be improved by running the algorithms for a longer time.
- A Monte-carlo Algorithm is said to have an **one-sided error** if the probability that its output is wrong for at least one of the possible outputs (yes/no) that it produce.
- A Monte-carlo Algorithm is said to have an **two-sided error** if there is non-zero probability that it errors when it outputs yes or no.

Las-Vegas Algorithms

- If any answer is to be found out using Las-Vegas Algorithm, then the answer is correct.
- These are randomized algorithms which never produce incorrect results, but whose execution time may vary from one run to another .
- Random choices made within the algorithm are used to establish an expected running time for the algorithm that is, essentially, independent of the input.
- The probability of finding answer increases if the algorithm is repeated good number of times.

Sherwood Algorithms

- This algorithm always gives the answer and answer is always correct
- This algorithms are used, when a deterministic algorithm behave inconsistently
- Randomness tends to make best case's behaviour look like that of the average case behaviour

0/1 Knapsack Solution using Randomized Algorithm.

The 0-1 knapsack problem

- Even if we don't use a randomized algorithm to find a solution, we might use one to improve a solution
- The 0-1 knapsack problem can be expressed as follows:
 - A thief breaks into a house, carrying a knapsack...
 - He can carry up to 25 kg of loot
 - He has to choose which of N items to steal
 - Each item has some weight and some value
 - "0-1" because each item is stolen (1) or not stolen (0)
 - He has to select the items to steal in order to maximize the value of his loot, but cannot exceed 25 pounds
- A greedy algorithm does not find an optimal solution...but...
- **We could use a greedy algorithm to find an initial solution, then use a randomized algorithm to try to improve that solution**

Improving the knapsack solution

- We can employ a greedy algorithm to fill the knapsack
- Then--
 - Remove a randomly chosen item from the knapsack
 - Replace it with other items (possibly using the same greedy algorithm to choose those items)
 - If the result is a better solution, keep it, else go back to the previous solution
 - Repeat this procedure as many times as desired
 - You might, for example, repeat it until there have been no improvements in the last k trials, for some value of k
- You probably won't get an optimal solution this way, but you might get a better one than you started with

(8) What is the main difference between Las Vegas and Monte Carlo algorithms?

- o A Las Vegas algorithm fails with some probability, but we can tell when it fails. In particular, we can run it again until it succeeds, which means that we can eventually succeed with probability 1 (but with a potentially unbounded running time). Alternatively, we can think of a Las Vegas algorithm as an algorithm that runs for an unpredictable amount of time but always succeeds (we can convert such an algorithm back into one that runs in bounded time by declaring that it fails if it runs too long—a condition we can detect). QuickSort is an example of a Las Vegas algorithm.
- o A Monte Carlo algorithm fails with some probability, but we can't tell when it fails. If the algorithm produces a yes/no answer and the failure probability is significantly less than $1/2$, we can reduce the probability of failure by running it many times and taking a majority of the answers. The polynomial equality-testing algorithm is an example of a Monte Carlo algorithm.

- **Las Vegas algorithms**
always correct; expected running time ("probably fast")

Examples: randomized Quick sort,
randomized algorithm for closest pair
- **Monte Carlo algorithms** (mostly correct):
probably correct; guaranteed running time

Example: randomized primality test

(9) The Majority-Element Problem: Given a sequence of n elements where each element is an integer in $[1, k]$, Write a randomized algorithm to return the majority element (an element that appears more than $n/2$ times) or zero if no majority element is found.

Majority element

```
1. Algorithm Main(A, N)
2. {
3.   Counter := 0;
4.   For i:= 1 to k do
5.   {
6.     Call Majority(A, N, R, E);
7.     If R = 'True' then Counter = Counter + 1;
8.     If Counter > 0 then goto 10;
9.   }
10.  If Counter > 0 then
11.  {
12.    Print('Majority Element = ' E);
13.  Else
14.    Print('No Majority Element ');
15.  }
16. }
```

Algorithm Majority(A, N, R, E)

```
1. Algorithm Majority(A, N, R, E)
2. // A is an array of N element
3. {
4.   R := 'False';
5.   choice := Uniform(1,N);
6.   count := 0;
7.   for i:= 1 to N do
8.   {
9.     If A(i) = A(choice) then count := count +1;
10.  }
11.  If (count > N/2) then
12.  {
13.    R := 'True';
14.    E := A(choice);
15.  }
16. }
```


(10) What are the four complexity classes involving randomized algorithms? Suggest a randomized algorithm to identifying the Repeated Element from an array of n elements. Prove the run time of your algorithm?

Complexity Classes

There are some interesting complexity classes involving randomized algorithms:

- Randomized Polynomial time (RP)
- Zero-error Probabilistic Polynomial time (ZPP)
- Probabilistic Polynomial time (PP)
- Bounded-error Probabilistic Polynomial time (BPP)

Randomized Polynomial time (RP)

Definition: The class RP consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^* :

$$\begin{aligned}x \in L &\Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{1}{2} \\x \notin L &\Rightarrow \Pr[A(x) \text{ accepts}] = 0\end{aligned}$$

Independent repetitions of the algorithms can be used to reduce the probability of error to exponentially small.

Notice that the success probability can be changed to an inverse polynomial function of the input size without affecting the definition of RP . Why?

Zero-error Probabilistic Polynomial time (ZPP)

Definition: The class ZPP is the class of languages which have Las Vegas algorithms running in expected polynomial time.

$ZPP = RP \cap \text{co-}RP$. Why?

(Note that a language L is in $\text{co-}X$ where X is a complexity class if and only if its complement $\Sigma^* - L$ is in X .)

Probabilistic Polynomial time (PP)

Definition: The class PP consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^* :

$$\begin{aligned}x \in L &\Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{1}{2} \\x \notin L &\Rightarrow \Pr[A(x) \text{ accepts}] < \frac{1}{2}\end{aligned}$$

Bounded-error Probabilistic Polynomial time (BPP)

Definition: The class BPP consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^* :

$$\begin{aligned}x \in L &\Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{3}{4} \\x \notin L &\Rightarrow \Pr[A(x) \text{ accepts}] \leq \frac{1}{4}\end{aligned}$$

Identifying the repeated element:

- Consider an array $a[]$ of n numbers that has $n/2$ distinct elements and $n/2$ copies of another element. The problem is to identify the repeated element.
- Any deterministic algorithm for solving this problem will need at least $(n/2) + 2$ steps in the worst case.
- a simple and elegant randomized Las Vegas algorithm that takes only $O(\log n)$ time.
- The algorithm returns the array index of one of the copies of the repeated element

```
1  RepeatedElement( $a, n$ )
2  // Finds the repeated element from  $a[1 : n]$ .
3  {
4      while (true) do
5      {
6           $i := \text{Random}() \bmod n + 1; j := \text{Random}() \bmod n + 1;$ 
7          //  $i$  and  $j$  are random numbers in the range  $[1, n]$ .
8          if  $((i \neq j) \text{ and } (a[i] = a[j]))$  then return  $i;$ 
9      }
10 }
```

Proving the run time:

- Any iteration of the while loop will be successful in identifying the repeated number if i is any one the $n/2$ array indices corresponding to the repeated element and j is any one of the same $n/2$ indices other than i .
- The probability that the algorithm quits in any given iteration of the while loop is $P = \{(n/2(n/2-1))/n^2\}$, which is $\geq 1/5$ for all $n \geq 10$.
- Hence the probability that the algorithm does not quit in a given iteration is $< 4/5$.
- Therefore the probability that the algorithm does not quit in 10 iterations is $< (4/5)^{10} < 0.1074$.
- So, Algorithm will terminate in 10 iterations or less with probability ≥ 0.8926
- The probability that the algorithm does not terminate in 100 iterations is $< (4/5)^{100} < 2.04 * 10^{-10}$. That is, almost Certainly the algorithm will quit in 100 iterations or less.
- If n equals $2 * 10^6$, for example, any deterministic algorithm will have to spend at least one million time steps as opposed to the 100 iterations.
- In general, the probability that the algorithm does not quit in the first $c \alpha \log n$ (c is a constant to be fixed) iterations is

$$< (4/5)^{c \alpha \log n} = n^{c \alpha \log (5/4)}$$
 which will be $< n^{-\alpha}$ if we pick $c \geq (1/(\log 5/4))$
- Thus the algorithm terminates in $(1/(\log 5/4)) \alpha \log n$ iterations or less with probability $\geq (1 - n^{-\alpha})$.
- Since each iteration of the while loop takes $O(1)$ time, the run time of the algorithm is $O(\log n)$

(11) The QUICK-SORT algorithm is an efficient and popular sorting technique that sorts a list of keys $S[1], S[2], \dots, S[n]$, recursively by choosing a pivot key. The best-case running

time of QUICK-SORT IS $O(n \log_2 n)$ and its worst-case running time is $O(n^2)$. Several improvements and modifications have been proposed to improve QUICK-SORT's worst-case behavior. Write a randomized algorithm for improving the running time of QUICK-SORT.

Randomized Quicksort

Algorithm RANDOMIZEDQUICKSORT

Input: An array $A[1..n]$ of n elements.

Output: The elements in A sorted in nondecreasing order.

1. $rquick\text{sort}(1, n)$

Procedure $rquick\text{sort}(low, high)$

1. **if** $low < high$ **then**
2. $v \leftarrow \text{random}(low, high)$
3. interchange $A[low]$ and $A[v]$
4. $\text{SPLIT}(A[low..high], w)$ $\{w \text{ is the new position of the pivot}\}$
5. $rquick\text{sort}(low, w - 1)$
6. $rquick\text{sort}(w + 1, high)$
7. **end if**

Informal analysis

Randomized quicksort has **expected time** (averaged over all choices of pivots) of $O(n \log n)$.

Assume we sort the set and divide into four parts, the middle two contain the best pivots.

Each is larger than at least $\frac{1}{4}$ of the elements and smaller than at least $\frac{1}{4}$ of the elements.

Choosing an element from the middle two means we split at most $2 \log 2n$ times.

A random choice will only choose from these middle parts half the time, but this is good enough to give an average call depth of $2(2 \log 2n)$.

And hence the expected time of $O(n \log n)$.

(12) Write a randomized algorithm (Quick select) to find the k th smallest element in an unsorted array? Prove that average case time complexity of Quick select is $O(n)$?

Input: Array $A[1..n]$ of the elements in the an arbitrary order, and an index k

Output: the k th smallest element in $A[1..n]$.

If $k = 1$, we are asking for the smallest element

If $k = n$, we are asking for the largest element

If $k = n/2$, we are asking for the **median**

Quick-Selection(A, p, r, k)

Quick-Selection(A, p, r, k)

if $p = r$ and $k = 1$, do return $A(p)$

if $p < r$ then

$q \leftarrow \text{RANDOM}(p, r)$

$t = \text{Partition}(A, p, r, q)$

 if $t = k$ do return $A[t]$

 else if $t > k$ do return **Quick-Selection($A, p, t-1, k$)**

 else if $t < k$ do return **Quick-Selection($A, t+1, r, k-t$)**

Partition(A,p,r,q)

Suppose there are $t-1$ elements in A that is smaller than $s = A[q]$.
Then return t and reorder A so that
 $A[p..p+t-1] < A[t] = s \leq A[t+1..r]$

- Partition(A,p,r,q)
 - [REDACTED] SWAP(A[q],A[r])
 - for $j \leftarrow p-1$ to $r-1$
 - do if
 - then $A[j] \leq x$
 - $i \leftarrow i+1; \text{SWAP}(A[i], A[j])$
 - return $i+1$

Average case time complexity:

Expected Time Complexity

$$\begin{aligned}
 E(T(n)) &\leq \frac{\sum_{i=1}^{n-1} E(T(\max(i, n-i)))}{n} + 3n \\
 &\leq \frac{2 \sum_{i=n/2}^{n-1} E(T(i))}{n} + 3n \\
 &= \frac{2 \sum_{i=n/2}^{3n/4} E(T(i))}{n} + \frac{2 \sum_{i=3n/4+1}^{n-1} E(T(i))}{n} + 3n \\
 &\leq \frac{(n/2)E(T(3/4n))}{n} + \frac{(n/2)E(T(n-1))}{n} + 3n
 \end{aligned}$$

$$E(T(n)) \leq \frac{E(T(3/4n))}{2} + \frac{E(T(n-1))}{2} + 3n$$

- Let $G(n) = E(T(n))$, we have

$$G(n) \leq \frac{G(3/4n)}{2} + \frac{G(n-1)}{2} + 3n$$

$$G(n) \leq \frac{G(3/4n)}{2} + \frac{G(n-1)}{2} + 3n$$

- Conjecture: $G(n) = O(n)$.
- Use the substitution method
 - Assuming for all $m < n$,
 - Need to show

$$G(m) \leq cm$$

$$G(n) \leq cn$$

Time Complexity
(if $c \geq 24$)

$$\begin{aligned} G(n) &\leq \frac{G(3/4n)}{2} + \frac{G(n-1)}{2} + 3n \\ &\leq \frac{c(3/4n)}{2} + \frac{c(n-1)}{2} + 3n \\ &\leq \frac{7cn}{8} - \frac{c}{2} + 3n \\ &\leq cn \end{aligned}$$

(13) Choosing a random pivot point improves quick sort by removing the worst case due to bad data. What effect would happen to Insertion Sort if we chose a random element to insert rather than the next one in the input sequence?

<http://www.cs.tulane.edu/~carola/teaching/cms2200/fall14/slides/Lecture-randomizedAlgos.pdf>

(14) Define subset paradigm and ordering paradigm in the context of greedy approach?

In subset paradigm, greedy approach creates a subset of items to find the optimal solution. In ordering Paradigm, greedy approach generates some arrangement/order to get the optimal solution.

Example: Greedy Method with SUBSET Paradigm

- There are $n = 5$ objects with integer weights $w[1..5] = \{1, 2, 5, 6, 7\}$, and values $p[1..5] = \{1, 6, 18, 22, 28\}$. Assuming a knapsack capacity of **11**.

Item/ weight	Status	Profit
1/1	1	1
2/2	1	1+6=7
3/5	1	7+18=25
4/6	0	25
5/7	0	25

GreedyKnapsack(Subset paradigm

```
1.  Algorithm GreedyKnapsack(m, n)
2.  //p[1 : n] and w[1 : n] contain the profits and weights
3.  // respectively of the n objects
4.  // m is the knapsack size and x[1 : n] is the solution vector.
5.  {
6.  for i := 1 to n do x[i] := 0.0;
7.  U:=m; P:=0;
8.  for i := 1 to n do
9.  {
10. if (w[i] > U) then break;
11. x[i] := 1.0; U:= U - w[i]; P := P + p[i]
12. }
13. Return P;
14. }
```

Example: Greedy Method with ORDERING Paradigm

- There are $n = 5$ objects with integer weights $w[1..5] = \{1,2,5,6,7\}$, and values $p[1..5] = \{1,6,18,22,28\}$. Assuming a knapsack capacity of **11**.

Item	w	v	v/w	Status	Profit
5	7	28	4	1	28
4	6	22	3.66	0	
3	5	18	3.6	0	
2	2	6	3	1	28+6=34
1	1	1	1	1	34+1=35

Optimal solution :40

GreedyKnapsack(ordering paradigm)

```
1.  Algorithm GreedyKnapsack(m, n)
2.  // p[1 : n] and w[1 : n] contain the profits and weights respectively
3.  // of the n objects ordered such that  $p[i]/w[i] \geq p[i+1]/w[i+1]$ .
4.  // m is the knapsack size and x[1 : n] is the solution vector.
5.  {
6.  for i := 1 to n do x[i] := 0.0;
7.  U:=m; P :=0;
8.  for i := 1 to n do
9.  {
10. if (w[i] > U) then break;
11. x[i] := 1.0; U.:= U - w[i]; P := P + p[i]
12. }
13. Return P;
14. }
```

(15) What are the general characteristic of dynamic programming algorithm?

Dynamic Programming algorithm solves every subproblem just once and then saves its answer in a table. A dynamic programming algorithm remembers past results and uses them to find new results.

Dynamic programming is generally used for optimization problems in which:

- o Multiple solutions exist, need to find the best one
- o Requires optimal substructure and overlapping subproblem
- o Optimal substructure: Optimal solution contains optimal solutions to subproblems
- o Overlapping subproblems: Solutions to subproblems can be stored and reused in a bottom-up fashion

16] Give the control Abstraction for Divide-and-Conquer. Find out the solution to the Equation

$T(n) = aT(n/b) + \Theta(n^k)$ where $a \geq 1$ and $b \geq 1$ with if $a < b^k$.

Soln:

Control abstraction Divide & Conquer

1. Algorithm **DAndC**(P)
2. {
3. If Small(P) then return S(P);
4. else
5. {
6. divide P into smaller instances , $P_1, P_2, \dots, P_k, k \geq 1$
7. Apply **DAndC** to each of these subproblems;
8. Return Combine (**DAndC**(P_1), **DAndC**(P_2), ..., **DAndC**(P_k))
9. }
10. }

→ Master's theorem :-

it is used to solve the
recurrence equations.

$$T(n) = a \cdot T(n/b) + O(n^k \log^p n)$$

$a \geq 1, b > 1, k \geq 0$ & p is real number

i) if $a > b^k$ then $T(n) = O(n^{\log_b a})$

ii) if $a = b^k$

a) if $p > -1$ then $T(n) = O(n^{\log_b a} \cdot \log^{p+1} n)$

b) if $p = -1$ then $T(n) = O(n^{\log_b a} \cdot \log \log n)$

c) if $p < -1$ then $T(n) = O(n^{\log_b a})$

(iii) if $a < b^k$

a) if $p \geq 0$, then $T(n) = O(n^k \log^p n)$

b) if $p < 0$, then $T(n) = O(n^k)$

Q(17) What is an absolute approximation algorithm? Suggest an absolute approximation algorithm for the planner graph colouring?

Solution:

Absolute approximation: **A** is an absolute approximation algorithm for problem **P** if and only if for every instance **I** of **P**,

$$| \text{FO}(\mathbf{I}) - \text{FA}(\mathbf{I}) | \leq k \text{ for some constant } k.$$

Theorem 1 *A 2-absolute approximation algorithm exists for planar graph coloring.*

Proof: By the Five Color Theorem, every planar graph is 5-colorable. Further, note that empty graphs (graphs without edges) are 1-colorable, bipartite graphs are 2-colorable, while all other graphs require at least 3 colors. These observations lead to the following algorithm:

1. If the graph is empty or bipartite, color it optimally.
2. Otherwise, color it with 5 colors.

Since this algorithm only uses 5 colors when the optimum number of colors is at least 3, it is a 2-absolute approximation algorithm. ■

Q (18) What do you mean by ρ approximation algorithm? Suggest a ratio-2 approximation algorithm for the vertex cover problem?

Solution:

ρ -approximation algorithm.

- An algorithm **A** for problem **P** that runs in polynomial time.
- For every problem instance, **A** outputs a feasible solution within ratio ρ of true optimum for that instance.

A 2-Approximation Algorithm for Vertex Cover

- **The Algorithm:** Find a maximal matching M of the graph and output the set V' of matched vertices
 - **Correctness:**
 - Edges belonging in M are all covered by V'
 - Since M is a maximal matching, any other edge $e \in E \setminus M$ will share at least one endpoint v with some $e' \in M$. So v is in V' and guards e .
 - **Analysis:**
 - Any vertex cover should pick at least one endpoint of each matched edge $\rightarrow |M| \leq \text{OPT}$
 - $|V'| = 2|M|$
- Thus $\text{SOL} = |V'| = 2|M| \leq 2\text{OPT} \Rightarrow \text{SOL} \leq 2\text{OPT}$

~ Vertex Cover is in APX

Q (19)

Define 0/1 Knapsack problem as optimization problem, explain the application of Dynamic Programming, and Greedy algorithm to find an optimal solution. Give the two different representation of problem space and compare the time complexity of the algorithms under the said paradigm.

Solution:

The Knapsack Problem is an example of a combinatorial optimization problem, which seeks for a best solution from among many other solutions. It is concerned with a knapsack that has positive integer volume (or capacity) V . There are n distinct items that may potentially be placed in the knapsack. Item i has a positive integer volume V_i and positive integer benefit B_i . In addition, there are Q_i copies of item i available, where quantity Q_i is a positive integer satisfying $1 \leq Q_i \leq \infty$.

Let X_i determines how many copies of item i are to be placed into the knapsack. The goal is to:

$$\begin{aligned} &\text{Maximize} \\ &\quad \sum_{i=1}^N B_i X_i \\ &\text{Subject to the constraints} \\ &\quad \sum_{i=1}^N V_i X_i \leq V \\ &\text{And} \\ &\quad 0 \leq X_i \leq Q_i. \end{aligned}$$

We implemented and tested all three of the strategies. We got the best results with the third strategy - choosing the items with as high value-to-weight ratios as possible.

ALGORITHM GreedyAlgorithm (Weights [1 ... N], Values [1 ... N])

// Input: Array Weights contains the weights of all items

Array Values contains the values of all items

// Output: Array Solution which indicates the items are included in the knapsack ('1') or not ('0')

Integer CumWeight

Compute the value-to-weight ratios $r_i = v_i / w_i$, $i = 1, \dots, N$, for the items given

Sort the items in non-increasing order of the value-to-weight ratios

for all items do

if the current item on the list fits into the knapsack then
place it in the knapsack

else
proceed to the next one

Complexity

1. Sorting by any advanced algorithm is $O(N \log N)$

2. $\sum_{i=1}^N 1 = [1+1+1 \dots 1]$ (N times) $= N \approx O(N)$

From (1) and (2), the complexity of the greedy algorithm is, $O(N \log N) + O(N) \approx O(N \log N)$. In terms of memory, this algorithm only requires a one dimensional array to record the solution string.

ALGORITHM Dynamic Programming (Weights [1 ... N], Values [1 ... N],
Table [0 ... N, 0 ... Capacity])

// Input: Array Weights contains the weights of all items
Array Values contains the values of all items
Array Table is initialized with 0s; it is used to store the results from the dynamic programming algorithm.
// Output: The last value of array Table (Table [N, Capacity]) contains the optimal solution of the problem for the given Capacity

Complexity

$$\sum_{i=0}^N \sum_{j=0}^{\text{Capacity}} 1 = \sum_{i=0}^N [1+1+1+\dots+1] \text{ (Capacity times)}$$
$$= \text{Capacity} * [1+1+1+\dots+1] \text{ (N times)}$$
$$= \text{Capacity} * N$$
$$= O(N * \text{Capacity})$$

Thus, the complexity of the Dynamic Programming algorithm is $O(N * \text{Capacity})$. In terms of memory, Dynamic Programming requires a two dimensional array with rows equal to the number of items and columns equal to the capacity of the knapsack. This algorithm is probably one of the easiest to implement because it does not require the use of any additional structures.

1.b) What is the role of a criteria function in problem solving process with back tracking Algorithm?

Solution - Backtracking is a type of algorithm that is a refinement of brute force search. In backtracking, multiple solutions can be eliminated without being explicitly examined, by using specific properties of the problem. It can be a strategy for finding solutions to constraint satisfaction problems.

There are two spaces related to backtracking -

Problem space - human's way to characterizing the function

Solutions space - Range of possible potential solution

In backtracking a solution is expressed as n-tuple (x_1, \dots, x_n) .

Solution space is actually the criterion function $P(x_1, x_2, \dots, x_n)$, also called bounding functions to test whether the vector formed has any chance of success?

2.a) Define and differentiate between performance analysis and performance measurement of an algorithm?

Ans - Performance analysis estimates space and time complexity in advance, while performance measurement measures the space and time taken in actual runs.

3) Define optimization problem? Explain the use of greedy algorithm to find sub-optimal solution for optimization problems? Define subset paradigm and ordering paradigm in the context of greedy approach? Write the control abstraction for greedy algorithm in ordering paradigm?

Ans - Optimization Problem: The class of problem with some objective function based on the problem instance or, an optimization problem is the problem of finding the best solution from all feasible solutions. Objective function is generally given in terms of minimization or maximization.

Subset paradigm - The greedy method suggests that one can devise an algorithm that works in stage. At each stage a decision is made whether a particular input is in the optimal solution and whether the given input should be chosen or not. This is called subset paradigm. E.g – MST problem, etc.

ordering paradigm - For problems that do not call for the selection for an optimal subset, in the greedy method we make decisions by considering the input in some order. Such version of greedy method is called ordering problem. E.g - single source shortest path, etc.

9.) You are given a collection of n bolts of different widths and n corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average case efficiency of $(n \log n)$.

Ans - This one is like randomized quick-sort. Divide the nuts and bolts into two sub-problems as follows: pick a random nut – call it n . Now find the matching bolt, call it b . Partition the bolts into two parts: those which are larger than b and those smaller than b – we can do this by comparing the bolts with n – note that we cannot just compare two bolts and say which one is larger. Similarly, by comparing with b , partition the nuts into two – those smaller than n , and those larger than n . This gives two sub-problems. As in the quick-sort algorithm, with probability at least $1/3$, both the sub-problems will have at most $2n/3$ nuts (or bolts). If this event happens, solve the two sub-problems recursively, else repeat the random selection process. The analysis is as for the quick-sort algorithm.

10) Consider a possible DIVIDE-AND-CONQUER approach to finding a minimal spanning tree in a connected, weighted, graph G . Suppose that we recursively divide the vertices of G into two disjoint subsets V_1 and V_2 . We then find a minimal spanning tree T_1 for V_1 and a minimal spanning tree T_2 for V_2 . Finally, we find a minimum weight edge e connecting T_1 and T_2 . We then let T be the graph obtained by combining T_1 , T_2 , and e .

a) Is T always a spanning tree?

b) If T is a spanning tree, is it always a minimal spanning tree? Explain.

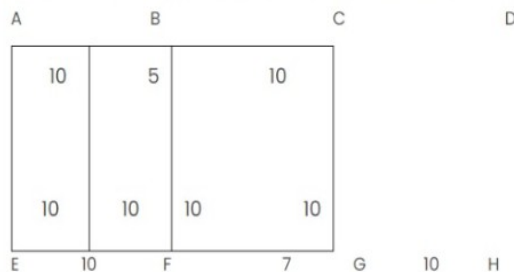
Solution:

1. It is correct

2. It is true only in case, if all the edges have same cost. but in general it is not correct. Please find the below explanation for this:

It's correct that T will always be a spanning tree but you won't always get a minimal spanning tree by just connecting T1 and T2. Please find the following example:

Let us consider the graph given below $G(A,B,C,D,H,G,F,E)$



Suppose Edge (B,C) has cost 5 and (F,G) has cost 7, all other edges have cost 10.

Then you divide it in $T_1(A,B,E,F)$

and $T_2(C,D,H,G)$

so the minimum spanning tree for T_1 and T_2 will be cost you 30 each.

If you now connect them with the minimal possible edge you get a spanning tree with cost 65:

i.e. $T = T_1 + T_2$ (you have to choose from BC or FG to combine those subgraphs, so we will have to take BC as it costs 5)

so you can see T is obviously a spanning tree but it is not a minimum spanning as tree. As in our case the Minimum spanning tree will cost you 62

i.e. $G(A,B,C,D,H,G,F,E)$ $(10+5+10+10+10+7+10=62)$

so $62 < 65$. so T will be Minimal spanning tree only if you have same cost for all the edges. if it varies then it won't be be a MST.

Q11: Given a set P of n points in 2D, a point $p \in P$ is said to be central if the x-coordinate of p has rank between $0.2n$ and $0.8n$ among the x-coordinates of P and the y-coordinate of p has rank between $0.2n$ and $0.8n$ among the y-coordinates of P. (Recall that the rank of an element z among a set refers to the number of elements smaller than z in the set. You may assume that coordinates are all distinct.). Design and analyze a Las Vegas linear-time algorithm for finding a central point in P.

Solution:

<https://www.geeksforgeeks.org/randomized-algorithms-set-1-introduction-and-analysis/>

GENETIC ALGO Questions

(1) Name the main features of GA.

Genetic Algorithms (GA) use principles of natural evolution. There are five important features of GA:

- Encoding possible solutions of a problem are considered as individuals in a population. If the solutions can be divided into a series of small steps (building blocks), then these steps are represented by genes and a series of genes (a chromosome) will encode the whole solution. This way different solutions of a problem are represented in GA as chromosomes of individuals.
- Fitness Function represents the main requirements of the desired solution of a problem (i.e. cheapest price, shortest route, most compact arrangement, etc). This function calculates and returns the fitness of an individual solution.
- Selection operator defines the way individuals in the current population are selected for reproduction. There are many strategies for that (e.g. roulette-wheel, ranked, tournament selection, etc), but usually the individuals which are more fit are selected.
- Crossover operator defines how chromosomes of parents are mixed in order to obtain genetic codes of their offspring (e.g. one-point, two-point, uniform crossover, etc). This operator implements the inheritance property (offspring inherit genes of their parents).
- Mutation operator creates random changes in genetic codes of the offspring. This operator is needed to bring some random diversity into the genetic code. In some cases GA cannot find the optimal solution without mutation operator (local maximum problem).

(2) What two requirements should a problem satisfy in order to be suitable for solving it by a GA?

GA can only be applied to problems that satisfy the following requirements:

- The fitness function can be well-defined.
- Solutions should be decomposable into steps (building blocks) which could be then encoded as chromosomes.

(3) Show, in pseudo code, a simple genetic algorithm with a brief description of each of the main elements.

Algorithm:

```
GA( $n$ ,  $\chi$ ,  $\mu$ )
// Initialise generation 0:
 $k := 0$ ;
 $P_k :=$  a population of  $n$  randomly-generated individuals;
// Evaluate  $P_k$ :
Compute fitness( $i$ ) for each  $i \in P_k$ ;
do
{ // Create generation  $k + 1$ :
  // 1. Copy:
  Select  $(1 - \chi) \times n$  members of  $P_k$  and insert into  $P_{k+1}$ ;
  // 2. Crossover:
  Select  $\chi \times n$  members of  $P_k$ ; pair them up; produce offspring;
  insert the offspring into  $P_{k+1}$ ;
  // 3. Mutate:
  Select  $\mu \times n$  members of  $P_{k+1}$ ; invert a randomly-selected bit
  in each;
  // Evaluate  $P_{k+1}$ :
  Compute fitness( $i$ ) for each  $i \in P_{k+1}$ ;
  // Increment:
   $k := k + 1$ ;
}
while fitness of fittest individual in  $P_k$  is not high enough;
return the fittest individual from  $P_k$ ;
```

n is the number of individuals in the population; χ is the fraction of the population to be replaced by crossover in each iteration; and μ is the mutation rate.

(4) Find the defining length and order of the following schemata of the form $*1*1*0*$, $*11****$, $111111*$, $**1***1$.

Schema Order $o(H)$ is the number of non '*' genes in schema H.

Example, $o(* * * 0 * * *) = 1$

Schema Defining Length, $d(H)$, is the distance between first and last non '*' gene in schema H.

Example, $\delta(* * * 0 * * *) = 4 - 4 = 0$

(5) What is a Schema. Show that "The number of a given schema in the next generation is a product of its fitness and its ability to survive crossover"? Find A search space consists of 10^6 points. Find the total number of schemata for binary and decimal coded genetic algorithm.

Schema is a subset of the space of all possible individuals for which all the genes match the template for schema H. The schema $H = [0 \ 1 \ * \ 1 \ *]$ identifies the chromosome set,

0 1 0 1 0

0 1 0 1 1

0 1 1 1 0

0 1 1 1 1

Needless to say, not all schema are created equal, thus, $1 * * * * *$ does not tell us as much as $0 \ 1 \ * \ 1 \ 0$.

Moreover, schema $1 * * * * 0$ spans the entire length of an individual whereas $1 * 1 * * * *$ does not.

(6) Consider a function maximization application of a GA. Suppose the 5 chromosomes at a given generation have fitness values listed below. Construct the “roulette wheel” for selection of parents for crossover.

$f(x_1) = 55$, $f(x_2) = 24$, $f(x_3) = 8$, $f(x_4) = 19$, $f(x_5) = 42$

$55/148=37\%$

$24/148=16\%$

$8/148=5\%$

$19/148=13\%$

$42/148=28\%$

(7) Describe briefly the Travelling Salesman Problem? Illustrate your description by explicitly finding the shortest tour given the distances (or costs) between cities shown in the following Table.

Travelling Salesman Problem: the problem of finding the shortest route through several cities, such that each city is visited only once and in the end return to the starting city.

Distances between cities.

To	A	B	C	D
From				
A	-	4	6	12
B	3	-	6	8
C	7	10	-	10
D	11	7	9	-

(8) Describe briefly how a solution to the Travelling Salesman Problem might be obtained using a genetic algorithm, indicating how features of the problem map to the elements needed to use a genetic algorithm.

To apply genetic algorithm for the TSP:

- o each chromosome represents the sequence through which cities have to be traversed and each gene

represent the number/character assigned to represent a city.

- o The criteria in the TSP for good chromosome are its length. The fitness function will be the total cost of the route represented by each chromosome. This can be calculated as the sum of the distances traversed in each travel segment. Lesser the sum, fitter is the solution.

o

Algorithm

1. Initialization: Generate N random candidate routes and

calculate fitness value for each route.

2. Repeat following steps Number of iteration times:

a) Selection: Select two best candidate routes.

b) Crossover: Reproduce two routes from the best routes.

c) Mutate the resultant off-springs if required, thus giving rise to a new tours

3. Return the best result

(9) Consider genetic algorithms using only selection and mutation operators (no crossover). Is this equivalent to another search algorithm? Justify your answer?

If you use a genetic algorithm without crossovers, you end up with a rather bad local search algorithm. Crossover is the mechanism that lets a GA share information about different parts of the solution space between different candidate solutions.

If you removed crossover, you're left with just mutation, and the algorithm essentially becomes

randomised local search, done in parallel for each starting candidate solution.

You'd need to change your selection mechanism between generations too, since without crossover there's no way to generate new candidate solutions at each generation. The usual criteria of keeping the "best N" candidates from the previous generation would result in keeping all of the candidates from the last generation, regardless of whether they got better or worse. Done this way, this approach is essentially a random "guess and check" search.

There are better heuristics for a randomised search - for example GRASP (Greedy randomized adaptive search procedure) GRASP uses randomness to avoid being trapped by any one local minimum, and uses a greedy local search procedure from a random start point to avoid wasting time on solutions that aren't locally optimal (and therefore aren't globally optimal either).

As always, the best algorithm to use depends on your problem, but a crossover-less GA doesn't have much going for it.

(9) Consider the problem of finding the shortest route through several cities, such that each city is visited only once and in the end return to the starting city (the Travelling Salesman problem). Suppose that in order to solve this problem we use a genetic algorithm, in which genes represent links between pairs of cities. For example, a link between London and Paris is represented by a single gene 'LP'. Let also assume that the direction in which we travel is not important, so that $LP = PL$.

a) How many genes will be used in a chromosome of each individual if the number of cities is 10?

Answer: Each chromosome will consist of 10 genes. Each gene representing the path between a pair of cities in the tour.

b) How many genes will be in the alphabet of the algorithm?

Answer: The alphabet will consist of 45 genes. Indeed, each of the 10 cities can be connected with 9 remaining. Thus, $10 \times 9 = 90$ is the number of ways in which 10 cities can be grouped in pairs. However, because the direction is not important (i.e. London–Paris is the same as Paris–London) the number must be divided by 2. So, we shall need $90/2 = 45$ genes in order to encode all pairs. In general the formula for n cities is: $n(n - 1)/ 2$.

(10) A budget airline company operates 3 plains and employs 5 cabin crews. Only one crew can operate on any plain on a single day, and each crew cannot work for more than two days in a row. The company uses all planes every day. A Genetic Algorithm is used to work out the best combination of crews on any particular day.

a) Suggest what chromosome could represent an individual in this algorithm?

Answer: On each day, a solution is a combination of 5 cabin crews assigned to 3 airplanes. Thus, we could encode different crews by different genes:

Genes (crews) : A, B, C, D, E

and then a chromosome representing a solution would be a chain of 3 genes (i.e. 3 cabin crews):

Solution 1 : A B C

Solution 2 : A B D

.....

Solution 10 : C D E

In this case, each position in the chromosome corresponds to a plane.

Another encoding is possible for the same problem. For example, we could encode if a crew is assigned on a plane by number 1 (if it does not matter which plane it is) and 0 if the crew has a day off:

Genes (work, day off) : 1, 0

Then we can use a chromosome of 5 genes each corresponding to a particular crew with 0 or 1 representing if a crew is at work or has a day off:

Solution 1: 1 1 1 0 0

Solution 2: 1 1 0 1 0

.....

Solution 10: 0 0 1 1 1

As you can see, the same problem can be encoded in many different ways.

b) Suggest what could be the alphabet of this algorithm? What is its size?

Answer: This depends on encoding used. In the first case, when genes represent the crews, the alphabet consists of 5 letters. In the second case, when binary representation is used, only two genes are required.

c) Suggest a fitness function for this problem.

Answer: You may come up with different versions, but it is important for the fitness to take into account the condition that cabin crews cannot work more than 2 days in a row.

For example, the fitness function can take into account how many days each crew has left before a day off (e.g. 1 or 0). The fitness could be calculated as the sum of these numbers for all drivers in the

chromosome. So, if d is the number of days an employee has before a day off, and m is the number of employees in the selected crews, then the fitness can be computed as

$$\text{Fitness} = d_1 + d_2 + \dots + d_m = \sum_{i=1}^m d_i$$

d) How many solutions are in this problem? Is it necessary to use Genetic Algorithms for solving it? What if the company operated more plains and employed more crews?

Answer:

The number of solutions is the number of times 3 crews can be selected out of 5 without replacement and without taking into account their order. The first crew can be selected in 5 different ways, the second in 4 ways and the third in 3 different ways. These numbers multiplied together will give us total number times how 3 crews can be selected randomly out of 5: $5 \times 4 \times 3 = 60$ times. However, there are 6 possible combinations in which 3 crews can be ordered, and because the order does not matter the answer is $60/6 = 10$. Thus, there are 10 possible solutions for this problem.

It is not really necessary to use GA for a problem with such a small population, because solutions can be checked explicitly. However, as the number of crews and airplanes increases, so does the number of solutions, and the use of GA can be the only option. In fact, if n is the number of cabin crews and $k \leq n$ is the number of airplanes, then the number of solutions is

$$\frac{n!}{k!(n-k)!}$$

For example, if the company operated 10 airplanes and employed 20 cabin crews, then the number of solutions would be

$$\frac{20!}{10!(20-10)!} = 184,756$$

(11) Give an example of combinatorial problem. What is the most difficult in solving these problems?

Answer: One classical example is the Travelling Salesman problem (TSP), described in the lecture notes. Another example is the timetable problem. The main difficulty is that the number of combinations (and, hence, the number of possible solutions) grows much faster than the number of items involved in the problem (i.e. the number of cities in TSP, the number of time-slots, etc). This problem is known as combinatorial explosion.

(12) Suppose a genetic algorithm uses chromosomes of the form

$x = abcdefgh$ with a fixed length of eight genes. Each gene can be any digit between 0 and 9.

Let the fitness of individual x be calculated as: $f(x) = (a + b) - (c + d) + (e + f) - (g + h)$

and let the initial population consist of four individuals with the following chromosomes:

$x_1 = 6\ 5\ 4\ 1\ 3\ 5\ 3\ 2$

$x_2 = 8\ 7\ 1\ 2\ 6\ 6\ 0\ 1$

$x_3 = 2\ 3\ 9\ 2\ 1\ 2\ 8\ 5$

$x_4 = 4\ 1\ 8\ 5\ 2\ 0\ 9\ 4$

a) Evaluate the fitness of each individual, showing all your workings, and arrange them in order with the fittest first and the least fit last.

Answer:

$$f(x_1) = (6 + 5) - (4 + 1) + (3 + 5) - (3 + 2) = 9$$

$$f(x_2) = (8 + 7) - (1 + 2) + (6 + 6) - (0 + 1) = 23$$

$$f(x_3) = (2 + 3) - (9 + 2) + (1 + 2) - (8 + 5) = -16$$

$$f(x_4) = (4 + 1) - (8 + 5) + (2 + 0) - (9 + 4) = -19$$

The order is x2, x1, x3 and x4.

b) Perform the following crossover operations:

i) Cross the fittest two individuals using one-point crossover at the middle point.

Answer: One-point crossover on x2 and x1:

$$x_2 = \begin{matrix} 8 & 7 & 1 & 2 & 6 & 6 & 0 & 1 \end{matrix}$$

$$x_1 = \begin{matrix} 6 & 5 & 4 & 1 & 3 & 5 & 3 & 2 \end{matrix}$$

$$\Rightarrow \quad o_1 = \begin{matrix} 8 & 7 & 1 & 2 & 3 & 5 & 3 & 2 \end{matrix}$$

$$\quad \quad o_2 = \begin{matrix} 6 & 5 & 4 & 1 & 6 & 6 & 0 & 1 \end{matrix}$$

ii) Cross the second and third fittest individuals using a two-point crossover (points b and f).

Answer: Two-point crossover on x1 and x3

$$x_1 = \begin{matrix} 6 & 5 & 4 & 1 & 3 & 5 & 3 & 2 \end{matrix}$$

$$x_3 = \begin{matrix} 2 & 3 & 9 & 2 & 1 & 2 & 8 & 5 \end{matrix}$$

$$\Rightarrow \quad o_3 = \begin{matrix} 6 & 5 & 9 & 2 & 1 & 2 & 3 & 2 \end{matrix}$$

$$\quad \quad o_4 = \begin{matrix} 2 & 3 & 4 & 1 & 3 & 5 & 8 & 5 \end{matrix}$$

iii) Cross the first and third fittest individuals (ranked 1st and 3rd) using a uniform crossover.

Answer: In the simplest case uniform crossover means just a random exchange of genes between two parents.

For example, we may swap genes at positions a, d and f of parents x2 and x3:

x2 = 8 7 1 2 6 6 0 1

x3 = 2 3 9 2 1 2 8 5

⇒ 05 = 2 7 1 2 6 2 0 1

06 = 8 3 9 2 1 6 8 5

c) Suppose the new population consists of the six offspring individuals received by the crossover operations in the above question. Evaluate the fitness of the new population, showing all your workings. Has the overall fitness improved?

Answer: The new population is:

01 = 8 7 1 2 3 5 3 2

02 = 6 5 4 1 6 6 0 1

03 = 6 5 9 2 1 2 3 2

04 = 2 3 4 1 3 5 8 5

05 = 2 7 1 2 6 2 0 1

06 = 8 3 9 2 1 6 8 5

Now apply the fitness function $f(x) = (a+b)-(c+d)+(e+f)-(g+h)$:

$$f(01) = (8 + 7) - (1 + 2) + (3 + 5) - (3 + 2) = 15$$

$$f(02) = (6 + 5) - (4 + 1) + (6 + 6) - (0 + 1) = 17$$

$$f(03) = (6 + 5) - (9 + 2) + (1 + 2) - (3 + 2) = -2$$

$$f(04) = (2 + 3) - (4 + 1) + (3 + 5) - (8 + 5) = -5$$

$$f(05) = (2 + 7) - (1 + 2) + (6 + 2) - (0 + 1) = 13$$

$$f(06) = (8 + 3) - (9 + 2) + (1 + 6) - (8 + 5) = -6$$

The overall fitness has improved

d) By looking at the fitness function and considering that genes can only be digits between 0 and 9 find the chromosome representing the optimal solution (i.e. with the maximum fitness). Find the value of the maximum fitness.

Answer: The optimal solution should have a chromosome that gives the maximum of the fitness function

$$\max f(x) = \max [(a + b) - (c + d) + (e + f) - (g + h)] .$$

Because genes can only be digits from 0 to 9, the optimal solution should be:

$$x_{\text{optimal}} = 9 \ 9 \ 0 \ 0 \ 9 \ 9 \ 0 \ 0 ,$$

and the maximum fitness is

$$f(x_{\text{optimal}}) = (9 + 9) - (0 + 0) + (9 + 9) - (0 + 0) = 36$$

e) By looking at the initial population of the algorithm can you say whether it will be able to reach the optimal solution without the mutation operator?

Answer: No, the algorithm will never reach the optimal solution without mutation. The optimal solution is $x_{\text{optimal}} = 9 \ 9 \ 0 \ 0 \ 9 \ 9 \ 0 \ 0$. If mutation does not occur, then the only way to change genes is by applying the crossover operator. Regardless of the way crossover is performed, its only outcome is an exchange of genes of parents at certain positions in the chromosome. This means that the first gene in the chromosomes of children can only be either 6, 8, 2 or 4 (i.e. first genes of x_1 , x_2 , x_3 and x_4), and because none of the individuals in the initial population begins with gene 9, the crossover operator alone will never be able to produce an offspring with gene 9 in the beginning. One can easily check that a

similar problem is present at several other positions. Thus, without mutation, this GA will not be able to reach the optimal solution.

(13) Show, in pseudo code, a simple genetic algorithm with a brief description of each of the main elements. Using an example, show why it is important to have a mutation operator in a genetic algorithm.

Answer from previous questions.

Hashing Questions

(1) Suppose there is a hash table of size H . If α be the load factor and ω be the word length for a key value, then find the total storage space required for the following cases: (i) Open hashing (assume the one word is required for a link field), and (ii) Closed hashing.

(i) If α be the load factor and ω be the word length for a key value, then find the total storage space required for Open hashing (Open hashing = chaining)

Ans: ...??

(ii) If α be the load factor and ω be the word length for a key value, then find the total storage space required for Closed hashing.

(Closed hashing = open addressing)

Closed hashing total storage space = # of entries * entry length

#entries = load factor * hash table size = $\alpha * H$

\Rightarrow Closed hashing total storage space = $\alpha * H * \omega$

(2) Hash table of size 10 with 2 slots, contains bucket address from 0 through 9 and following keys are to be mapped into the hash table from a master file with maximum 15 records.

98, 18, 32, 45, 15, 59, 29, 21, 79, 93, 92, 22, 42.

[i]Construct Open addressing hash table using linear probing with $f(x) = x \pmod{10}$ and find how many collisions occurred?

Bucket	Slot0	Slot1
0		
1	21	
2	32	92
3	93	
4		
5	45	15
6		
7		
8	98	18
9	59	29

collisions : 18,15,29,79,79,92,22,22,42,42 #collisions = 10

[ii]Construct Open addressing hash table using quadratic probing with $f(x) = x \pmod{10}$ and find how many collisions occurred?

Bucket	Slot0	Slot1
0	79	
1	21	
2	32	92
3	93	22
4		
5	45	15
6	42	
7		
8	98	18
9	59	29

**collisions : 18,15,29,79,79,92,22,22,42,42,22,42,42
#collisions = 13**

[iii] Define and calculate identifier density and loading density of the above hash table.

The identifier density of a hash table is the ratio $\frac{n}{T}$, where **n** is the number of identifiers in the table and **T** is the total number of possible identifiers.

#possible identifiers = 15, #identifiers = 13

$$n/T = 13/15$$

The loading density or loading factor of a hash table is

$$\alpha = n/(sb).$$

$$\alpha = n/(sb) = 13/(2*10) = 13/20$$

(3) Suppose that an open-address hash table has a capacity of 811 and it contains 81 elements. What is the table's load factor? (An approximation is fine.)

The purpose of the [load factor](#) is to give an idea of how likely (on average) it is that you will need collision resolution if a new element is added to the table. A collision happens when a new element is assigned a bucket that already has an element. The chance that a given bucket already has an element depends on how many elements are in the container.

load factor = # of elements / # of buckets

(4) Suppose a hash table with capacity $M=31$ gets to be over $3/4$ ths full. We decide to rehash. What is a good size choice for the new table to reduce the load factor below 0.5 and also avoid collisions?

$$M = 31, \text{ } 3/4 \text{ full}$$

$$N = 3*31/4 = 24$$

Build a second table twice as large as the original and rehash there all the keys of the original table.

Expensive operation, running time $O(N)$

However, once done, the new hash table will have good performance.

$$\text{New table size } M = 31*2 = 62$$

$$N = 24$$

$$\text{Load factor} = 24/62 = 0.38 < 0.5$$

(5) Define load factor in a hash Table. A large number of deletions in an open hash table can cause the table to be fairly empty, which wastes spaces. In this case, we can rehash to a table half as large. Assume that we rehash to a larger table when there are twice as

many elements as table size. How empty should an open table to be before we rehash to a smaller table.

We must be careful not to rehash too often. Let p be the threshold (fraction of table size) at which we rehash to a smaller table. Then, if the new table has size N , it contains $2Np$ elements. This table will require rehashing after either $2N - 2Np$ insertions or pN deletions. Then, we don't want to do rehashing either after a few insertions or a few deletions. A good strategy is to set $2N - 2Np$ equals to pN and we get $p = 2/3$. For instance, suppose we have a table of size 300. If we rehash at 200 elements, then the new table size is $N = 150$, and we can do either 100 insertions or 100 deletions until a new rehash is required.

If we know that insertions are more frequent than deletions, then we might choose p to be somewhat larger. All in all, we play around the relation between $2N - 2Np$ and pN depends on which operation we favorite.

(6) Given the values { 2341, 4234, 2839, 430, 22, 397, 3920}, a hash table of size 7, and hash function $h(x) = x \bmod 7$, show the resulting tables after inserting the values in the given order with each of these collision strategies: separate chaining, linear probing, quadratic probing, and double hashing with hash function $h'(x) = (2x - 1) \bmod 7$.

{2341, 4234, 2839, 430, 22, 397, 3920} $h(x) = x \bmod 7$

$$2341 \% 7 = 3$$

$$4234 \% 7 = 6$$

$$2839 \% 7 = 4$$

$$430 \% 7 = 3$$

$$22 \% 7 = 1$$

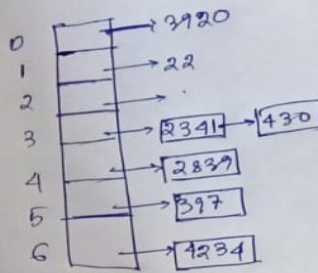
$$397 \% 7 = 5$$

$$3920 \% 7 = 0$$

chaining

mod 7

2341	→ 3
4234	→ 6
2839	→ 4
430	→ 3
22	→ 1
397	→ 5
3920	→ 0



Linear Probing

0	3920 ✓
1	22 ✓
2	3920 ✓
3	2341 ✓
4	2839 ✓
5	430 ✓
6	4234 ✓

Quadratic

0	430 ✓
1	22 ✓
2	3920 ✓
3	2341 ✓
4	2839 ✓
5	397 ✓
6	4234 ✓

430 → 3
~~3920~~ → 4 ×
 $3+1 \rightarrow 4 \times$
 $3+4 \rightarrow 7 \% 7 = 0$

3920 → 0
 $0+1 \rightarrow 1 \times$
 $0+4 \rightarrow 4 \times$
 $0+9 \rightarrow 9 \% 7 = 2$

double hashing

0	3920 ✓
1	430 ✓
2	22 ✓
3	2341 ✓
4	2839 ✓
5	397 2341 ✓
6	4234 ✓

$(2 \times 397 - 1) \% 7$
 $793 \% 7 = 2$

$h'(x)$

$(2 \times 22 - 1) \% 7$
 $(2 \times 2341 - 1) \% 7 = 5$
 $(2 \times 4234 - 1) \% 7 = 0$
 $(2 \times 430 - 1) \% 7 = 1$
 $(2 \times 22 - 1) \% 7$
 $= 43 \% 7 = 1$

lower bound Questions

(1) Define lower bound of a problem? What is the difference between worst case lower bound and average case lower bound? Find out the lower bound of heap sort algorithm.

A lower bound for a problem is the worst-case running time of the best possible algorithm for that problem. It is a big-Omega bound on the worst-case running time of any algorithm that solves the problem. Lower bound: an estimate on a minimum amount of work needed to solve a given problem. Lower bound can be an exact count an efficiency class (Ω).

Worst Case Lower Bound: A function that is a boundary below the algorithm's run time function, when that algorithm is given the input that maximizes the algorithm's runtime. Eg: comparison based sorting which is known to have $\Omega(n \log n)$ in worst case one cant make an algorithm that beats that in worst case.

Average case lower bound: The average time complexity of a sorting algorithm is = (the external path length of the binary tree) / $n!$

The external path length is minimized if the tree is balanced. (all leaf nodes on level d or level $d-1$)

Average case lower bound of sorting: $\Omega(n \log n)$

Algorithm 2 - 7 Heapsort

Input : $A(1), A(2), \dots, A(n)$

where each $A(i)$ is a node of a heap already constructed.

Output : The sorted sequence of $A(i)$'s.

For $i := n$ down to 2 do

Begin

Output $A(1)$

$A(1) := A(i)$

Delete $A(i)$

Restore(1, $i-1$)

End

Output $A(1)$

Time complexity

Phase 1: construction

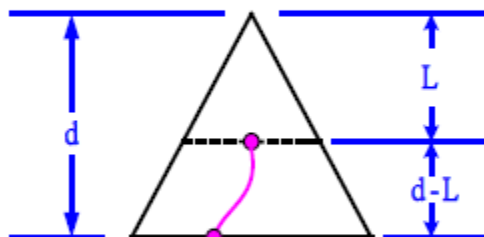
$d = \lfloor \log n \rfloor$: depth

of comparisons is at most:

$$\begin{aligned} & \sum_{L=0}^{d-1} 2(d-L)2^L \\ &= 2d \sum_{L=0}^{d-1} 2^L - 4 \sum_{L=0}^{d-1} L2^{L-1} \\ & \left(\sum_{L=0}^k L2^{L-1} = 2^k(k-1)+1 \right) \\ &= 2d(2^d-1) - 4(2^{d-1}(d-1-1)+1) \end{aligned}$$

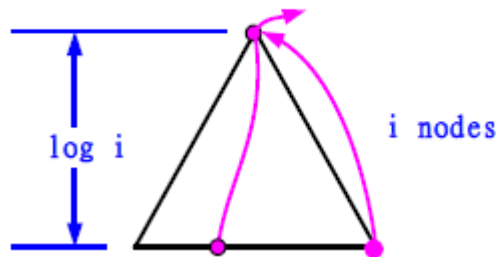
:

$$= cn - 2\lfloor \log n \rfloor - 4, \quad 2 \leq c \leq 4$$



Phase 2: output, Time complexity

$$\begin{aligned} & 2 \sum_{i=1}^{n-1} \lfloor \log i \rfloor \\ & = \dots \\ & = 2n \lfloor \log n \rfloor - 4cn + 4, \quad 2 \leq c \leq 4 \\ & = O(n \log n) \end{aligned}$$



(2) Define and differentiate between P, NP and NP-complete problems with examples? What is reducibility in the context of NP-completeness?

Three classes of problems

- P corresponds to a class of problems that can be solved in polynomial time. Those problems for which there is a polynomial time-algorithm that will produce a solution to all instances of the problem of input size n with a worst-case running time that is $O(n^k)$ for some constant k . The class of problems for which there is a polynomial-time solution.
 - Examples: Binary Search, Sorting, Matrix Multiplication, Shortest Path Problems, Minimal Spanning Tree, etc
- NP corresponds to a class of problems that contains problems that can be verified in polynomial-time. The class P is a sub-class of NP (since any problem that can be solved in polynomial time can also be verified in polynomial-time, if only by solving the problem and comparing solutions). But NP also contains many problems that can be verified in polynomial-time but for

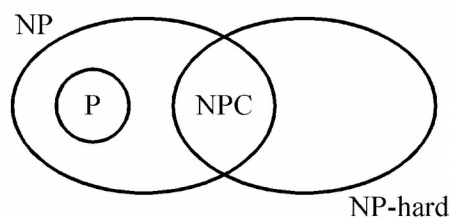
which there is no known polynomial-time solution. Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms.

- o Example: problems include the Traveling Salesman Problem and the Hamiltonian Circuit Problem.
- NP corresponds to a class of problems in NP that is also NP-hard. A problem is NP-hard if every problem in NP is reducible to it in polynomial time. Thus, L is NP-complete problem if: 1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution). 2) Every problem in NP is reducible to L in polynomial time.

- o Example: SAT, CSAT, 3SAT, TSP, Hamiltonian Path, graph colouring

View of Theoretical Computer Scientists on P, NP, NPC

$$\bullet \quad P \subseteq NP, NPC \subseteq NP, P \cap NPC = \emptyset$$



$P \subseteq NP$ (Sure)

$NPC \subseteq NP$ (sure)

$P = NP$ (or $P \subset NP$, or $P \neq NP$) ???

$NPC = NP$ (or $NPC \subset NP$, or $NPC \neq NP$) ???

$P \neq NP$: one of the deepest, most perplexing open research problems in (theoretical) computer science since 1971.

reducibility in the context of NP-completeness :Among all the problems known to be NP, there is a subset known as NP-complete problems, which contains the hardest. An NP-complete problem has a property that any problem in NP can be *polynomially* reduced to it.

A problem P1 can be reduced to P2 as follows:

- Provide a mapping so that any instance of P1 can be transformed to an instance of P2.
- Solve P2, and then map the answer back to the original.

Example: In a pocket calculator, the decimal numbers are converted to binary and all calculations are performed in binary, the final answer is converted back to decimal display

P1 to be polynomially reducible to P2 means that , all the work associated with the transformation must be performed in polynomial time.

(3) How you prove a problem to be NP-Complete?

Given a problem U, the steps involved in proving that it is NP-complete are the following:

Step 1: Show that U is in NP.

Step 2: Select a known NP-complete problem V.

Step 3: Construct a reduction from V to U.

Step 4: Show that the reduction requires polynomial time.

(4) Give the algorithm of Binary search. Explain how it functions? Devise a ternary search algorithm that first tests the element at position $n/3$ for equality with some value x, and then checks the element at $2n/3$ and either

discovers x or reduces the set size to one-third the size of the original. Compare this with binary search? Comment on time and space complexity?

Binary Search:

Algorithm:

```
Begin
  if start <= end then
    mid := start + (end - start) / 2
    if array[mid] = key then
      return mid location
    if array[mid] > key then
      call binarySearch(array, mid+1, end, key)
    else when array[mid] < key then
      call binarySearch(array, start, mid-1, key)
  else
    return invalid location
End
```

How it works:

Assumption: Binary search

- Input array to be in sorted order.
- All the element in the array must be unique.

Binary search compares the target value to the item in the middle of the list.

- If the target is smaller than the middle item, the search examines the items in the left half of the list.
- If the target is bigger than the middle item, the search continues in the right half of the list.
- If the target matches the middle item, the search is finished.

Each time the algorithm compares the target to the middle item, it divides the list into approximately two halves.

Ternary Search:

Algorithm:

```

Begin
  if start <= end then
    midFirst := start + (end - start) / 3
    midSecond := midFirst + (end - start) / 3
    if array[midFirst] = key then
      return midFirst
    if array[midSecond] = key then
      return midSecond
    if key < array[midFirst] then
      call ternarySearch(array, start, midFirst-1, key)
    if key > array[midSecond] then
      call ternarySearch(array, midFirst+1, end, key)
    else
      call ternarySearch(array, midFirst+1, midSecond-1, key)
  else
    return invalid location
End

```

Compare binary and ternary search:

Recursive formula for counting comparisons in worst case of Binary Search.

$$T(n) = T(n/2) + 2, \quad T(1) = 1$$

Recursive formula for counting comparisons in worst case of Ternary Search.

$$T(n) = T(n/3) + 4, \quad T(1) = 1$$

In binary search, there are $2\log_2 n + 1$ comparisons in worst case. In ternary search, there are $4\log_3 n + 1$ comparisons in worst case.

Time Complexity for Binary search = $2\log_2 n + O(1)$

Time Complexity for Ternary search = $4\log_3 n + O(1)$

Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions $2\log_3 n$ and $\log_2 n$. The value of $2\log_3 n$ can be written as $(2 / \log_2 3) * \log_2 n$. Since the value of $(2 / \log_2 3)$ is

more than one, Ternary Search does more comparisons than Binary Search in worst case.

Space complexity for both = $O(1)$

(5) Suppose we have an unsorted array A of n elements, and we want to know if the array contains any duplicate elements. Clearly outline an efficient method for solving this problem. By efficient, I mean your method should use $O(n \log n)$ key comparisons in the worst case.

First perform a comparison based sorting on the n elements of the array. The best comparison-based sorting algorithms (e.g., merge sort) use roughly $n \log n$ comparisons in the worst case.

Then simply walk the sorted list and compares the current element to the next element, looking for duplicates. This takes $(n - 1)$ comparisons.

Thus method will use $O(n \log n)$ comparisons in worst case.

Approximation Algo Questions

1[a] Present the definition of an approximation ratio to measure the approximation quality of the greedy algorithm?

Solution: An algorithm has an **approximation ratio** of $\rho(n)$ if, for any input of size n , the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution: $\max(C/C^*, C^*/C) \leq \rho(n)$.

In Maximization problems: $0 < C \leq C^*, \rho(n) = C^*/C$

In Minimization Problems: $0 < C^* \leq C, \rho(n) = C/C^*$

$\rho(n)$ is never less than 1.

An algorithm that achieves an approximation ratio of $\rho(n)$ is called a $\rho(n)$ -approximation algorithm.

$\rho(n) \geq 1$, 1-approximation algorithm produces an optimal solution.

Approximation ratio can be a constant or grow as function of input size n .

Theorem 1: Algorithm Greedy resource allocation is a 2-approximation algorithm.

Algorithm 1 Greedy resource allocation

Require: $ETC(MaxTask, MaxNode)$

Ensure: *makespan*

- 1: $T_j \leftarrow 0$ for all node M_j
 - 2: $A(j) \leftarrow \phi$ for all node M_j
 - 3: **for** $i = 1$ to $MaxTask$ **do**
 - 4: Let M_j be a node with minimum T_j
 - 5: Allocate task t_i to Node M_j
 - 6: $A(j) \leftarrow A(j) \cup \{t_i\}$
 - 7: $T_j \leftarrow T_j + t_{ij}$
 - 8: **end for**
 - 9: $T \leftarrow \max_j T_j$
-

1[d] What are PTAS and FPTAS? How the value of ϵ affects the running time in PTAS and FPTAS?

Which one is better among PTAS and FPTAS? Why?

Solution:

Polynomial Time Approximation Scheme (PTAS) is a type of approximate algorithms that provide user to control over accuracy which is a desirable feature. These algorithms take an additional parameter $\epsilon > 0$ and provide a solution that is $(1 + \epsilon)$ approximate for minimization and $(1 - \epsilon)$ for maximization.

In PTAS algorithms, the exponent of the polynomial can increase dramatically as ϵ reduces, for example if the runtime is $O(n^{(1/\epsilon)!})$.

Ideally, if ϵ decreases by a constant factor, the running time should not increase by more than a constant factor. We would like the running time to be polynomial in $1/\epsilon$ as well as in n .

We say that an approximation scheme is a fully polynomial-time approximation scheme if it is an approximation scheme and its running time is polynomial in both $1/\epsilon$ and the size " n " of the input instance. For example, the scheme might have a running time of $O((1/\epsilon)^{2n^3})$. With such a scheme, any constant-factor decrease in ϵ comes with a corresponding constant-factor increase in the running time.

2[a] What are the different approaches to prove the correctness of an algorithm? Explain Loop Invariants associated with an algorithm? Use binary search algorithm to discuss the process of proving correctness?

2[b] Define convex hull in R^2 ? Show that solution space of finding a 2-dimensional convex hull is exponential? Write Approximation algorithm for convex hulls for a set of n points with time complexity $O(n+k)$?

The **Convex Hull** is the line completely enclosing a set of points in a plane so that there are no concavities in the line. More formally, we can describe it as the smallest convex polygon which encloses a set of points such that each point in the set lies within the polygon or on its perimeter.

- **Input:** A set of n points.
 - **Output:** An approximate convex hull of S .
1. Find the leftmost and right most points of S , denoted as A_1 and A_2 , respectively. (with minimum and maximum x-coordinates respectively).
 2. Divide the area bounded by A_1 and A_2 into k equally spaced strips and for each strip, select the points with the minimum and maximum y-coordinates. Denote the set of points selected in this step together with A_1 and A_2 as set P .
 3. Construct the convex hull of P and use that as the approximate convex hull of S .
 4. time complexity: $O(n+k)$
 5. Step 1: $O(n)$

6. Step 2: $O(n)$

7. Step 3: $O(k)$

4[a] Why should an approximation algorithm be polynomial? What are the main steps for designing an approximation algorithm? How does lower bound of a problem play role in deriving approximation ratio?

- The goal of an **approximation algorithm** is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time.

4[b] Discuss the concept of pattern matching algorithm? What is the advantage of Boyer-Moore method over Brute-Force method to match a pattern P in a given string T. Explain the concept of looking glass heuristic and character-jump heuristic used in Boyer-Moore algorithm?

Pattern matching is the process of checking whether a specific sequence of characters/tokens/data exists among the given data. Regular programming languages make use of regular expressions ([regex](#)) for pattern matching.

Pattern matching is used to determine whether source files of high-level languages are syntactically correct. It is also used to find and replace a matching pattern in a text or code with another text/code. Any application that supports search functionality uses pattern matching in one way or another.

Boyer-Moore algorithm uses reverse order search logic that is in the right-to-left order. The movement of sliding window is determined by using bad character rule and good suffix rule (S. S. Sheik et al., 2004). The algorithm performs a preprocessing over the pattern P and creates two tables, which are known as Boyer-Moore bad-character (bmBc) and Boyer-Moore good-suffix (bmGs) tables respectively. The Boyer-Moore bad-character table has an entry for each character in the alphabet set which gives the shift value based on the occurrence of the character in the pattern P. On the other hand, the Boyer-Moore good-suffix table gives the matching shift value for each and every character in the pattern P.

The Boyer-Moore algorithm is sub-linear as the number of references per character decreases as the patterns get longer. When the alphabet set is sufficiently large and the pattern is sufficiently long, the algorithm executes

less than one instruction per character pass. While on the other hand Brute-Force algorithm shows linear time complexity.

Behaviors	Boyer-Moore	Brute-Force
Order of comparison	Right to left	Left to right
Shifting rules	Both bad character rule and good suffix rule	One by one character shift
Preprocessing time complexity	$O(m+n)$	No pre-processing

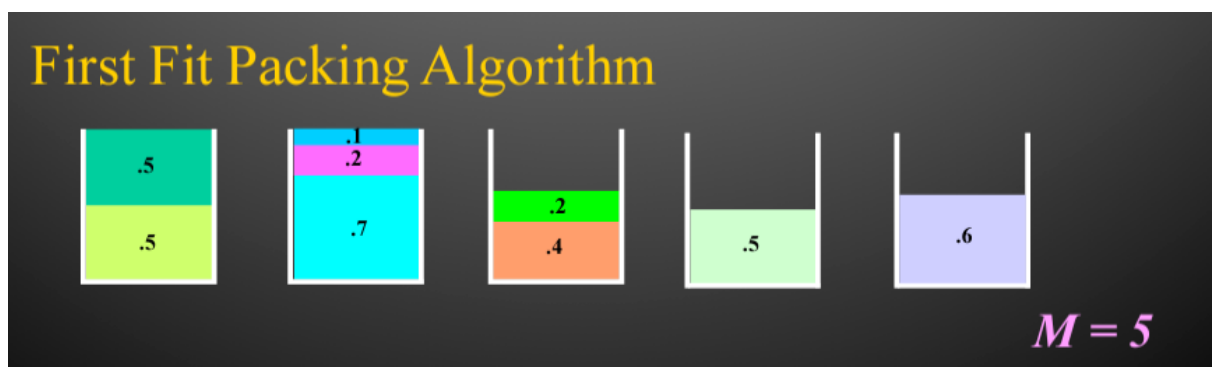
1. Looking-Glass heuristic enables to get to the destination faster by going backward if there is a mismatch during the consideration of P at a certain location in T.
2. Character-Jump heuristic enables to avoid lots of needless comparisons by significantly shifting P relative to T.

7[b] Explain how to implement first-fit and best-fit heuristic based approximation algorithm for bin packing algorithm in $O(n \log n)$ time.

First Fit:

Scan the bins in order and place the new item in the first bin that is large enough to hold it. A new bin is created only when an item does not fit in the previous bins.

Example:



Running Time :

the Boyer-Moore algorithm is sub-linear as the number of references per character decreases as the patterns get longer. When the alphabet set is sufficiently large and the pattern is sufficiently long, the algorithm executes less than one instruction per character pass. While on the other hand Brute-Force algorithm shows linear time complexity.

Easily implemented in $O(n^2)$ time.

Can be implemented in $O(n \log n)$ time:

- Idea: Use a balanced search tree with height $O(\log n)$.

- Each node has three values: index of bin, remaining capacity of bin, and best (largest) in all the bins represented by the subtree rooted at the node.

-The ordering of the tree is by bin index.

Let M be the optimal number of bins required to pack a list I of items. Then First Fit never uses more than $\lceil 1.7M \rceil$.

BEST FIT:

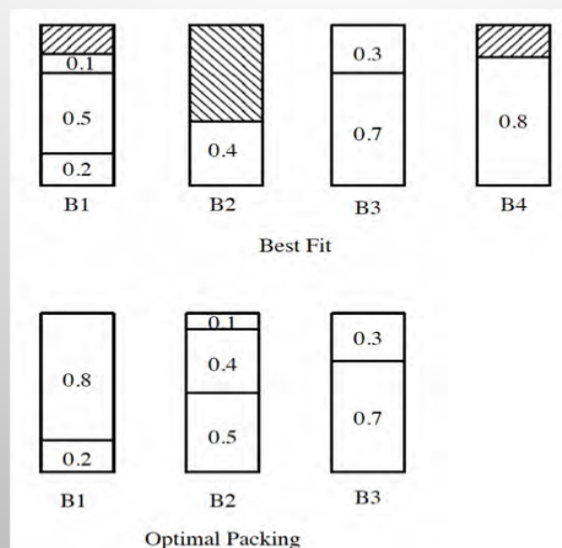
New item is placed in a bin where it fits the tightest. If it does not fit in any bin, then start a new bin.

- Can be implemented in $O(n \log n)$ time, by using a balanced binary tree storing bins ordered by remaining capacity.

Example:

Example for Best Fit (BF)

- $I = (0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8)$



8[e] Why approximation algorithms are used to solve NP-hard problem?
What do you mean by polynomial-time approximation algorithm?

In computer science and operations research, approximation algorithms are efficient algorithms that find approximate solutions to optimization problems (in particular NP-hard problems) with provable guarantees on the distance of the returned solution to the optimal one.[1] Approximation algorithms naturally arise in the field of theoretical computer science as a consequence of the widely believed $P \neq NP$ conjecture. Under this conjecture, a wide class of optimization problems cannot be solved exactly in polynomial time. The field of approximation algorithms, therefore, tries to understand how closely it is possible to approximate optimal solutions to such problems in polynomial time. In an overwhelming majority of the cases, the guarantee of such algorithms is a multiplicative one expressed as an approximation ratio or approximation factor i.e., the optimal solution is always guaranteed to be within a (predetermined) multiplicative factor of the returned solution.

NP-hard problems vary greatly in their approximability; some, such as the knapsack problem, can be approximated within a multiplicative factor $1+\epsilon$, for any fixed $\epsilon > 0$, and therefore produce solutions arbitrarily close to the optimum (such a family of approximation algorithms is called a polynomial time approximation scheme or PTAS).

