|  |  |
| --- | --- |
| Brute Force - Computing an – String Matching -Closest-Pair and Convex-Hull Problems-Exhaustive Search -Traveling Salesman Problem -Knapsack Problem -Assignment problem. Divide and conquer methodology –Merge sort –Quick sort–Heap Sort-Binary search –Multiplication of Large Integers –Strassen‟s Matrix Multiplication-Closest-Pair and Convex-Hull Problems | |
| **UNIT II**  **PART A** | |
|  | **What is Brute Force?**  Brute Force is a straightforward approach to solving a problem, usually directly based on the problem’s statement and definitions of the concepts involved. |
|  | **What would be the most straightforward method for solving the sorting problem?**  Selection sort and bubble sort are the two better algorithms and it implements brute-force approach more clearly. |
|  | **What are the advantages and disadvantages of brute force approach?**  **Advantages:**   * applicable to a wide variety of problems * for important problems it yields reasonable algorithms of at least some practical value with no limitation on instance size * It can solve few instances of a problem with acceptable speed * simple to design and may be useful to solve small problem instances   **Disadvantage:**   * inefficient in general case |
|  | **When is sorting method said to be stable?**  A sorting method is said to be stable when it has minimum number of swaps (less than ‘n’ number of comparisons) i.e. if the two data items of matching value are guaranteed not to be rearranged with respect to each other when the algorithm progresses. |
|  | **List out some of the stable and unstable sorting techniques.**  **Stable sorting techniques includes**   * Bubble sort * Insertion sort * Selection sort * Merge sort   **Unstable sorting techniques includes**   * Shell sort * Quick sort * Radix sort * Heap sort |
|  | **Define unstable sort.**  A sorting method is said to be Unstable when it has maximum number of swaps (greater than ‘n’ number of comparisons) i.e. if the two data items of matching value are guaranteed not to be rearranged with respect to each other when the algorithm progresses. |
|  | **Define Closet –pair problem. *(May 16,17)***  Given n points in the points in the plane, find the closet pair among them. Brute force approach |
|  | **Define Convex.**  A set of points (finite or infinite) on the plane is called convex if for any two points P and Q in the set, the entire line segment with the end points at P and Q belongs to the set. |
|  | **Define Convex-Hull and Convex Hull Problem.**  The Convex-Hull of a set S of points is the smallest convex set containing S. The smallest requirements means that the convex-hull of S must be a sub set of any convex set containing S. The problem of constructing the convex-hull for a given set S of n points. |
|  | **Define Extreme points.**  The extreme point of a convex set is a point of set S that is not a middle point of any line segment with end points in the set. |
|  | **Define Exhaustive search. (April/May 2018)**  Exhaustive search is simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem’s domain, selecting those of them that satisfy all the constraints, and then finding a desired element. |
|  | **Define travelling salesman problem.**  Travelling salesman problem finds the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it is started. |
|  | **Define Hamiltonian circuit. (Nov 2018)**  It is defined as a cycle that passes through all the vertices of a graph exactly once. |
|  | **Define Knapsack problem*.(Dec 14)***  Given n items of known weights w1…wn and values v1…vn and knapsack of capacity W. The aims is to find the most valuable subset if the items that fit into the knapsack. The exhaustive search approach to knapsack problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset to identify feasible subsets and finding a subset of the largest value among them. |
|  | **Define assignment problem. *(May 16)***  There are n people who need to be assigned to execute n jobs, one person per job. The cost that would accrue if the ith person is assigned to the jth job is a known quantity c [i,j] for each pair I, j=1,…,n. The problem is to find an assignment with the smallest total cost. |
|  | **Define merge sort.**  The merge sort algorithm divides a given array A[0..n-1] by dividing it into two halves A[0.. └ n/2┘-1] and A[└ n/2┘-..n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.  If the list has even length, split the list into two equal sub lists. If the list has odd length, divide the list in two by making the first sub list one entry greater than the second sub list. Then spilt both the sub lists in to two and go on until each of the sub lists are of size one. Finally start merging the individual sub lists to obtain a sorted list. |
|  | **Define quick sort.**  Quick sort employs a divide-and-conquer strategy. It starts by picking an element from the list to be the "pivot." It then reorders the list so that all elements with values less than the pivot come before the pivot, and all elements with values greater than the pivot come after it (a process often called "partitioning"). It then sorts the sub-lists to the left and the right of the pivot using the same strategy, continuing this process recursively until the whole list is sorted. |
|  | **What is a pivot element?**  The pivot element is the chosen number which is used to divide the unsorted data into two halves. The lower half contains less than value of the chosen number i.e. pivot element. The upper half contains greater than value of the chosen number i.e. pivot element. So the chosen number is now sorted. |
|  | **Give the general plan of divide and conquer algorithms. *(May 13,16, Nov 17)***   * A problem’s instance is divided into several smaller instances of the same problem, ideally of about the same size. * The smaller instances are solved * If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance. |
|  | **What is median-of-three –portioning method?**  The median-of-three-portioning method is employed in quick sort to find the pivot value. This is done by randomly choosing three elements in the list and finding the median of the elements gives the pivot value. |
|  | **Define Binary Search. *(Dec 07)***  Binary search is a efficient algorithm for searching in a sorted array. It works by comparing a search key K with the array’s middle element A[m]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if K<A[m] and for the second half if K>A[m]. |
|  | **Define Stassen’s matrix multiplication. *(Dec 07)***  Strassen showed that 2x2 matrix multiplications can be accomplished in 7 multiplication and 18 additions or subtractions. (2log27 =22.807). This reduce can be done by Divide and Conquer Approach. |
|  | **How many multiplications are performed in two n-digit multiplication?**  There is a divide-and-conquer algorithm for multiplying two n-digit integers that requires about n1.585 one-digit multiplications. |
|  | **Derive the complexity of binary search algorithm. *(May 15)***  Let us find the number of key comparisons in the worst case *Cworst(n).* The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for *Cworst(n)*:    For the initial condition *Cworst(*1*)* = 1*,* we obtain    For *n* = 2*k* can be tweaked to get a solution valid for an arbitrary positive integer *n*: |
|  | **Devise an algorithm to make a change for 1655 using the Greedy strategy. The coins available are {1000, 500, 100, 50, 20, 10, 5}. (May 17)**  Algorithm:  While(there are more coins and the instance is not solved)  {  grab the largest remaining coin; // selection procedure  if(adding the coin makes the change exceed the amount owed )  reject the coin; // feasibility check  else  add the coin to the change;  if( the total value of the change equals the amount owed )  // solution check the instance is solved;  } Solution for the given instance 1655 = 1000 + 500 +100 + 50 + 5. |
|  | **What are the advantages of insertion sort?(Nov 2017)**  The main advantage of the insertion sort is its simplicity.   * It also exhibits a good performance when dealing with a small list. * The insertion sort is an in-place sorting algorithm so the space requirement is minimal. |
|  | **Define Heap sort.**  A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:   * The ***shape property***—the binary tree is ***essentially complete*** (or simply ***complete***), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing. * The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children. |
|  | **Explain String matching problem.**  String-matching problem: Given a string of *n* characters called the ***text*** and a string of *m* characters *(m* ≤ *n)* called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching substring in the text—such that |
| **UNIT II**  **PART B** | |
|  | **Explain how brute force approach is applied to solve closest-Pair and convex-Hull problem. (Nov 2017)**  The closest-pair problem calls for finding the two closest points in a set of *n* points. It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces.  **Applications:**   * Points in question can represent such physical objects as airplanes or post offices as well as database records, statistical samples, DNA sequences, and so on. * An air-traffic controller might be interested in two closest planes as the most probable collision candidates. * A regional postal service manager might need a solution to the closest pair problem to find candidate post-office locations to be closed.   For simplicity, we consider the two-dimensional case of the closest-pair problem. We assume that the points in question are specified in a standard fashion by their *(x, y)* Cartesian coordinates and that the distance between two points *pi(xi,yi)* and *pj(xj, yj )* is the standard Euclidean distance    The brute-force approach to solving this problem leads to the following obvious algorithm: compute the distance between each pair of distinct points and find a pair with the smallest distance. Of course, we do not want to compute the distance between the same pair of points twice. To avoid doing so, we consider only the pairs of points *(pi, pj )* for which *i < j*.  **Pseudocode:**    **Analysis:**  The basic operation of the algorithm is computing the square root. For starters, even for most integers, square roots are irrational numbers that therefore can be found only approximately. Moreover, computing such approximations is not a trivial matter. Then the basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:    **CONVEX-HULL PROBLEM**  Finding the convex hull for a given set of points in the plane or a higher dimensional space is one ofthe most important problems in computational geometry  **Definition:** A set of points (finite or infinite) in the plane is called ***convex*** if for any two points *p* and *q* in the set, the entire line segment with the endpoints at *p* and *q* belongs to the set.    All the sets depicted in Figure (a) are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon, a circle, and the entire plane. On the other hand, the sets depicted in Figure b, any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.  **Definition:** The ***convex hull*** of a set *S* of points is the smallest convex set containing *S*. (The “smallest” requirement means that the convex hull of *S* must be a subset of any convex set containing *S*.)    If *S* is convex, its convex hull is obviously *S* itself. If *S* is a set of two points, its convex hull is the line segment connecting these points. If *S* is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on the same line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For an example of the convex hull for a larger set, see Figure  .  The ***convex-hull problem*** is the problem of constructing the convex hull for a given set *S* of *n* points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. The vertices of such a polygon is called “extreme points.” By definition, an ***extreme point*** of a convex set is a point of this set that is not a middle point of any line segment with endpoints in the set. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in above Figure are *p*1*, p*5*, p*6*, p*7*,* and *p*3.  How can we solve the convex-hull problem in a brute-force manner?  The convex hull problem is one with no obvious algorithmic solution. Nevertheless, there is a simple but inefficient algorithm that is based on the following observation about line segments making up the boundary of a convex hull: a line segment connecting two points *pi* and *pj* of a set of *n* points is a part of the convex hull’s boundary if and only if all the other points of the set lie on the same side of the straight line through these two points. Repeating this test for every pair of points yields a list of line segments that make up the convex hull’s boundary.  **Time efficiency:**  It is in O *(n*3*)*: for each of *n (n* − 1*)/*2 pairs of distinct points, we may need to find the sign of *ax* + *by* – *c* for each of the other *n* − 2 points. There are much more efficient algorithms for this important problem |
|  | **Explain how brute force approach is applied for computing an and string matching problem.**  **Computing an using brute force technique**  Brute forceis a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. As an example, consider the exponentiation problem: compute *an* for a nonzero number *a* and a nonnegative integer *n.* By the definition of exponentiation    This suggests simply computing *an* by multiplying 1 by *a n* times. We have already encountered at least two brute-force algorithms: the consecutive integer checking algorithm for computing gcd *(m, n)* and the definition-based algorithm for matrix multiplication  We consider a straightforward approach to two well-known problems dealing with a finite set of points in the plane. These problems, aside from their theoretical interest, arise in two important applied areas: computational geometry and operations research  **String Matching using Brute Force technique**  String-matching problem: Given a string of *n* characters called the ***text*** and a string of *m* characters *(m* ≤ *n)* called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching substring in the text—such that    Brute-force algorithm for the string-matching problem:  Align the pattern against the first *m* characters of the text and start matching the corresponding pairs of characters from left to right until either all the *m* pairs of the characters match or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text that can still be a beginning of a matching substring is *n* – *m* (provided the text positions are indexed from 0 to *n* − 1). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.  **ALGORITHM** *BruteForceStringMatch(T* [0*..n* − 1]*, P*[0*..m* − 1]*)*  //Implements brute-force string matching  //Input: An array *T* [0*..n* − 1] of *n* characters representing a text and  // an array *P*[0*..m* − 1] of *m* characters representing a pattern  //Output: The index of the first character in the text that starts a  // matching substring or −1 if the search is unsuccessful  **for** *i* ←0 **to** *n* − *m* **do**  *j* ←0  **while** *j <m***and** *P*[*j* ]= *T* [*i* + *j* ] **do**  *j* ←*j* + 1  **if** *j* = *m* **return** *i*  **return** −1  Example of brute-force string matching:    An operation of the algorithm is illustrated above Figure. The pattern’s characters that are compared with their text counterparts are in bold type.  **Brute Force-Complexity**  • Given a pattern M characters in length, and a text N characters in length...  • Worst case: compares pattern to each substring of text of length M. For example, M=5.   1. AAAAAAAAAAAAAAAAAAAAAAAAAAAH   AAAAH 5 comparisons made   1. AAAAAAAAAAAAAAAAAAAAAAAAAAAH   AAAAH 5 comparisons made   1. AAAAAAAAAAAAAAAAAAAAAAAAAAAH   AAAAH 5 comparisons made   1. AAAAAAAAAAAAAAAAAAAAAAAAAAAH   AAAAH 5 comparisons made   1. AAAAAAAAAAAAAAAAAAAAAAAAAAAH   AAAAH 5 comparisons made  ....  N) AAAAAAAAAAAAAAAAAAAAAAAAAAAH  5 comparisons made AAAAH  • Total number of comparisons: M (N-M+1)  • Worst case time complexity: Ο (MN)  **Brute Force-Complexity**  • Given a pattern M characters in length, and a text N characters in length...  • Best case if pattern found: Finds pattern in first M positions of text. For example, M=5.   1. AAAAAAAAAAAAAAAAAAAAAAAAAAAH   AAAAA 5 comparisons made  • Total number of comparisons: M  • Best case time complexity: Ο (M)  **Brute Force-Complexity**  • Given a pattern M characters in length, and a text N characters in length...  • Best case if pattern not found: Always mismatch on first character. For example, M=5.   1. **A**AAAAAAAAAAAAAAAAAAAAAAAAAAH   **O**OOOH 1 comparison made   1. A**A**AAAAAAAAAAAAAAAAAAAAAAAAAH   **O**OOOH 1 comparison made   1. AA**A**AAAAAAAAAAAAAAAAAAAAAAAAH   **O**OOOH 1 comparison made   1. AAA**A**AAAAAAAAAAAAAAAAAAAAAAAH   **O**OOOH 1 comparison made   1. AAAA**A**AAAAAAAAAAAAAAAAAAAAAAH   **O**OOOH 1 comparison made  ...  N) AAAAAAAAAAAAAAAAAAAAAA**A**AAAAH  1 comparison made **O**OOOH  • Total number of comparisons: N  • Best case time complexity: Ο (N) |
|  | **Explain the concept of assignment problem and knapsack problem using exhaustive search.**  Exhaustive searchis simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function). Note that although the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects.  We illustrate exhaustive search by applying it to two important problems:   * Knapsack problem, and * Assignment problem.   There are *n* people who need to be assigned to execute *n* jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the *i*th person is assigned to the *j*th job is a known quantity *C*[*i, j* ] for each pair *i, j* = 1*,* 2*, . . . , n*. The problem is to find an assignment with the minimum total cost.    We can describe feasible solutions to the assignment problem as *n*-tuples *j*1*, . . . , jn* in which the *i*th component, *i* = 1*, . . . , n*, indicates the column of the element selected in the *i*th row (i.e., the job number assigned to the *i*th person). For example, for the cost matrix above, 2, 3, 4, 1 indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1. The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first *n* integers. Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1*,* 2*, . . . , n,* computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.      The optimal solution is: Person 1 to Job 2, Person 2 to Job 1, Person 3 to Job 3, and Person 4 to Job 4, with the total (minimal) cost of the assignment being 13.  Since the number of permutations to be considered for the general case of the assignment problem is *n*!, exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for this problem called the ***Hungarian method***  **KNAPSACK PROBLEM**  Given *n* items of known weights *w*1*, w*2*, . . . , wn* and values *v*1*, v*2*, . . . , vn* and a knapsack of capacity *W*, find the most valuable subset of the items that fit into the knapsack.  The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them. As an example, the solution to the instance of Figure a is given in Figure b.    (a) Instance of the knapsack problem. (b) Its solution by exhaustive search.  The information about the optimal selection is in bold.  Since the number of subsets of an *n*-element set is 2*n*, the exhaustive search leads to a *Ω(*2*n)* algorithm, no matter how efficiently individual subsets are generated. Thus, for both the traveling salesman and knapsack problems considered above, exhaustive search leads to algorithms that are extremely inefficient on every input. |
|  | **Explain in detail about merge sort. Illustrate the algorithm with a numeric example*. (Dec 06,Dec 13, Dec 14, May 16, April/May 2018)***  Merge sort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array *A*[0*..n* − 1] by dividing it into two halves. *A* [0*..n/*2 − 1] and *A* [*n/*2*..n* − 1]*,* sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.  **ALGORITHM** *Merge sort(A*[0*..n* − 1]*)*  //Sorts array *A*[0*..n* − 1] by recursive merge sort  //Input: An array *A*[0*..n* − 1] of orderable elements  //Output: Array *A*[0*..n* − 1] sorted in non decreasing order  **if** *n >* 1  copy *A*[0*..n/*2 − 1] to *B*[0*..n/*2− 1]  copy *A*[\_*n/*2\_*..n* − 1] to *C*[0*..n/*2 − 1]  *Merge sort (B* [0*..*\_*n/*2\_ − 1]*)*  *Merge sort(C* [0*..*\_*n/*2\_ − 1]*)*  *Merge (B, C, A)*  The merging of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.  **ALGORITHM** *Merge (B* [0*..p* − 1]*, C*[0*..q* − 1]*, A*[0*..p* + *q* − 1]*)*  //Merges two sorted arrays into one sorted array  //Input: Arrays *B* [0*..p* − 1] and *C*[0*..q* − 1] both sorted  //Output: Sorted array *A*[0*..p* + *q* − 1] of the elements of *B* and *C*  *i* ←0; *j* ←0; *k*←0  **while** *i <p* **and** *j <q* **do**  **if** *B*[*i*]≤ *C*[*j* ]  *A*[*k*]←*B*[*i*]; *i* ←*i* + 1  **else** *A*[*k*]←*C*[*j* ]; *j* ←*j* + 1  *k*←*k* + 1  **if** *i* = *p*  copy *C*[*j..q* − 1] to *A*[*k..p* + *q* − 1]  **else** copy *B*[*i..p* − 1] to *A*[*k..p* + *q* − 1]  The operation of the algorithm on the list 8*,* 3*,* 2*,* 9*,* 7*,* 1*,* 5*,* 4 is illustrated in Figure. |
|  | **Write the algorithm for Quick sort. Provide a complete analysis of quick sort for the given set of numbers 5, 3, 1, 9, 8, 2, 4, 7 *(June 06, May 13, May 15, May 2017, Nov 2018, )***  Quicksort is the other important sorting algorithm that is based on the divide-and conquer Partition approach.. A partition is an arrangement of the array’s elements so that all the elements to the left of some element *A*[*s*] are less than or equal to *A*[*s*]*,* and all the elements to the right of *A*[*s*] are greater than or equal to it:  *A*[0] *. . . A*[*s* − 1] *A*[*s*] *A*[*s* + 1] *. . . A*[*n* − 1]  all are ≤*A*[*s*] all are ≥*A*[*s*]  Obviously, after a partition is achieved, *A*[*s*] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of *A*[*s*] independently  Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value Note the difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.        **Improvements:**   * Better pivot selection methods such as ***randomized quicksort*** that uses a random element or the ***median-of-three*** method that uses the median of the leftmost, rightmost, and the middle element of the array * Switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array * Modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot |
|  | **Write an algorithm for binary search using divide and conquer and analyze the time complexity*. (June 06, Dec 14, May 17, ) (or)* What is divide and conquer strategy and explain the binary search with suitable example. *(May 17)***  Binary search is an efficient algorithm for searching in a **sorted array**. It compares a search key *K* with the array’s middle element *A*[*m*]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if *K <A*[*m*]*,* and for the second half if *K >A*[*m*]:    As an example, let us apply binary search to searching for *K* = 70 in the array  3 14 27 31 39 42 55 70 74 81 85 93 98  The iterations of the algorithm are given in the following table:  1 2 3 4 5 6 7 8 9 10 11 12 13   |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |   Iteration 1 l m r  Iteration 2 l m r  Iteration 3 l,m r    **Efficiency of binary search**  The efficiency of binary search is to count the number of times the search key is compared with an element of the array. The number of such comparisons does the algorithm make on an array of *n* elements depends not only on *n* but also on the specifics of a particular instance of the problem.  **Worst Case efficiency of binary search:**  Let us find the number of key comparisons in the worst case *Cworst(n).* The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for *Cworst(n)*:  *Cworst(n)* = *Cworst(n/*2*)* + 1 for *n >* 1*, Cworst(*1*)* = 1*.*    It implies that the worst-case time efficiency of binary search is in θ *(*log *n)* |
|  | **Explain in detail about Heap sort with an example.**  A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:   * The ***shape property***—the binary tree is ***essentially complete*** (or simply ***complete***), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing. * The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children.     For example, consider the trees of Figure. The first tree is a heap. The second one is not a heap, because the tree’s shape property is violated. And the third one is not a heap, because the parental dominance fails for the node with key 5.  Properties of heaps:   * There exists exactly one essentially complete binary tree with *n* nodes. Its height is equal to log2 *n* * The root of a heap always contains its largest element. * A node of a heap considered with all its descendants is also a heap. * A heap can be implemented as an array by recording its elements in the topdown, left-to-right fashion. It is convenient to store the heap’s elements in positions 1 through *n* of such an array*,* leaving *H*[0] either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,   + - the parental node keys will be in the first \_*n/*2\_ positions of the array, while the leaf keys will occupy the last \_*n/*2\_ positions;     - the children of a key in the array’s parental position *i (*1≤ *i* ≤ \_*n/*2\_*)* will be in positions 2*i* and 2*i* + 1, and, correspondingly, the parent of a key in position *i (*2 ≤ *i* ≤ *n)* will be in position \_*i/*2\_*.*   **Heapsort**  Now we can describe ***heapsort***. This is a two-stage algorithm that works as follows.   * **Stage 1** (heap construction): Construct a heap for a given array. * **Stage 2** (maximum deletions): Apply the root-deletion operation *n* − 1 times to the remaining heap.   As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order.  **Heap Construction:** There are two principal alternatives for construct a heap for a given list of keys doing this. The first is the ***bottom-up heap construction*** algorithm illustrated in Figure.    **Figure:** Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double headed arrows show key comparisons verifying the parental dominance.  It initializes the essentially complete binary tree with *n* nodes by placing keys in the order given and then “heapifies” the tree as follows. Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node. If it does not, the algorithm exchanges the node’s key *K* with the larger key of its children and checks whether the parental dominance holds for *K* in its new position. This process continues until the parental dominance for *K* is satisfied. After completing the “heapification” of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node’s immediate predecessor. The algorithm stops after this is done for the root of the tree.  **ALGORITHM** *HeapBottomUp(H*[1*..n*]*)*  //Constructs a heap from elements of a given array  // by the bottom-up algorithm  //Input: An array *H*[1*..n*] of orderable items  //Output: A heap *H*[1*..n*]  **for** *i* ←\_*n/*2\_ **downto** 1 **do**  *k*←*i*; *v*←*H*[*k*]  *heap*←**false**  **while not** *heap* **and** 2 ∗ *k* ≤ *n* **do**  *j* ←2 ∗ *k*  **if** *j <n* //there are two children  **if** *H*[*j* ]*<H*[*j* + 1] *j* ←*j* + 1  **if** *v* ≥ *H*[*j* ]  *heap*←**true**  **else** *H*[*k*]←*H*[*j* ]; *k*←*j*  *H*[*k*]←*v*  **Maximum Key Deletion** from a heap   * Exchange the root’s key with the last key *K* of the heap. * Decrease the heap’s size by 1. * “Heapify” the smaller tree by sifting *K* down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for *K*: if it holds, we are done; if not, swap *K* with the larger of its children and repeat this operation until the parental dominance condition holds for *K* in its new position.     Figure: Deleting the root’s key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is “heapified” by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied. |
|  | **Explain how divide and conquer method is applied in multiplication of large integers? *(May 16)***  With divide-and-conquer multiplication, we split each of the numbers into two halves, each with *n*/2 digits. I'll call the two numbers we're trying to multiply *a* and *b*, with the two halves of *a* being *a*L (the left or upper half) and *a*R (the right or lower half) and the two halves of *b* being *b*L and *b*R.  Basically, we can multiply these two numbers as follows.  aL aR  x bL bR  -----------------  aLbR aRbR  + aLbL aRbL  --------------------------  aLbL (aLbR + aRbL) aRbR  ab = (aL 10n/2 + aR) (bL 10n/2 + bR)  = aL bL 10n + aL bR 10n/2 + aR bL 10n/2 + aR bR  = aL bL 10n + (aL bR + aR bL) 10n/2 + aR bR  Thus, in order to multiply a pair of *n*-digit numbers, we can recursively multiply four pairs of *n*/2-digit numbers. The rest of the operations involved are all *O*(*n*) operations. But there turns out to be a very clever approach, we permits us to reduce the number of *n*/2-digit multiplications from four to *three*! This clever idea yields a better result.  aL bL 10n + (aL bR + aR bL) 10n/2 + aR bR  What we'll do is compute the following three products using recursive calls.  x1 = aL bL  x2 = aR bR  x3 = (aL + aR) (bL + bR)  These have all the information that we want, since the following is true.  x1 10n + (x3 - x1 - x2) 10n/2 + x2  = aL bL 10n + ((aL bL + aL bR + aR bL + aR bR) - aL bL - aR bR) 10n/2 + aR bR  = aL bL 10n + (aL bR + aR bL) 10n/2 + aR bR  Pseudocode:  BigInteger multiply(BigInteger a, BigInteger b) {  int n = max(number of digits in a, number of digits in b)  if(n == 1) {  return a.intValue() \* b.intValue();  } else {  BigInteger aR = bottom n/2 digits of a;  BigInteger aL = top remaining digits of a;  BigInteger bR = bottom n/2 digits of b;  BigInteger bL = top remaining digits of b;  BigInteger x1 = multiply (aL, bL);  BigInteger x2 = multiply (aR, bR);  BigInteger x3 = multiply (aL + bL, aR + bR);  return x1 \* pow(10, n) + (x3 - x1 - x2) \* pow(10, n / 2) + x2;  }  }  Recursion tree: labeling the edges with the final values computed by each node of the tree.  http://www.cburch.com/csbsju/cs/160/notes/31/karatex.gif  **Time analysis:**  Since multiplication of *n*-digit numbers requires three multiplications of *n/*2-digit numbers, the recurrence for the number of multiplications *M (n)* is  *M (n)* = 3*M (n/*2*)* for *n >* 1*, M (*1*)* = 1*.*  Solving it by backward substitutions for *n* = 2*k* yields  *M (*2*k)* = 3*M (*2*k*−1*)* = 3[3*M (*2*k*−2*)*] = 32*M (*2*k*−2*)*  = 3*iM (*2*k*−*i)*  = 3*kM (*2*k*−*k)*  = 3*k.*  Since *k* = log2 *n,*  *M (n)* = 3log2 *n* = *n*log2 3 ≈ *n*1*.*585 |
|  | **Explain how divide and conquer method is applied to solve closest-Pair and convex-Hull problem (May 15)**  Let *P* be a set of *n >* 1 points in the Cartesian plane. For the sake of simplicity, we assume that the points are distinct. We can also assume that the points are ordered in nondecreasing order of their *x* coordinate. It will also be convenient to have the points sorted in a separate list in nondecreasing order of the *y* coordinate; we will denote such a list *Q.*  If 2 ≤ *n* ≤ 3*,* the problem can be solved by the obvious brute-force algorithm. If *n >* 3*,* we can divide the points into two subsets *Pl* and *Pr* of \_*n/*2\_ and \_*n/*2\_ points, respectively, by drawing a vertical line through the median *m* of their *x* coordinates so that \_*n/*2\_ points lie to the left of or on the line itself, and \_*n/*2\_ points lie to the right of or on the line. Then we can solve the closest-pair problem recursively for subsets *Pl* and *Pr .* Let *dl* and *dr* be the smallest distances between pairs of points in *Pl* and *Pr ,* respectively, and let *d* = min{*dl, dr*}*.*  Note that *d* is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line. Therefore, as a step combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width 2*d* around the separating line, since the distance between any other pair of points is at least *d*    Algorithm EfficientClosestPair(P,Q)  // Solves the closest-pair problem by divide and conquer  // input: An array P of n>=2 points in the Cartesian plane sorted in  //nondecreasing order of their x coordinates and an array Q of the same  //points sorted in nondecreasing order of the y coordinates  //output: Euclidean distance between the closest pair of points  If n<=3  Return the minimal distance found by the brute force algorithm  Else    **Efficiency:**  The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that *n* is a power of 2, we have the following recurrence for the running time of the algorithm:  *T (n)* = 2*T (n/*2*)* + *f (n),*  where *f (n)* ∈ *\_(n).* Applying the Master Theorem (with *a* = 2*, b* = 2*,* and *d* = 1*)*, we get *T (n)* ∈ *\_(n* log *n).*  **CONVEX-HULL PROBLEM**  We revisit the convex-hull problem, here a divide-and-conquer algorithm called ***quickhull*** because of its resemblance to quicksort. Let *S* be a set of*n>*1 points *p*1*(x*1*, y*1*), . . . , pn(xn, yn)* in the Cartesian plane. We assume that the points are sorted in nondecreasing order of their *x* coordinates, with ties resolved by increasing order of the *y* coordinates of the points involved. The leftmost point *p*1 and the rightmost point *pn* are two distinct extreme points of the set’s convex hull    Let *p*1*pn* be the straight line through points *p*1 and *pn* directed from *p*1 to *pn*. This line separates the points of *S* into two sets: *S*1 is the set of points to the left of this line, and *S*2 is the set of points to the right of this line.The points of *S* on the line *p*1*pn,* other than *p*1 and *pn*, cannot be extreme points of the convex hull and hence are excluded from further consideration.  The boundary of the convex hull of *S* is made up of two polygonal chains: an “upper” boundary and a “lower” boundary. The “upper” boundary, called the ***upper hull***, is a sequence of line segments with vertices at *p*1*,* some of the points in *S*1 (if *S*1 is not empty) and *pn.* The “lower” boundary, called the ***lower hull***, is a sequence of line segments with vertices at *p*1*,* some of the points in *S*2 (if *S*2 is not empty) and *pn.* The fact that the convex hull of the entire set *S* is composed of the upper and lower hulls, which can be constructed independently and in a similar fashion, is a very useful observation exploited by several algorithms for this problem.  How quickhull proceeds to construct the upper hull; the lower hull can be constructed in the same manner. If *S*1 is empty, the upper hull is simply the line segment with the endpoints at *p*1 and *pn.* If *S*1 is not empty, the algorithm identifies point *p*max in *S*1*,* which is the farthest from the line *p*1*pn* (Figure ).    If there is a tie, the point that maximizes the angle *p*max*ppn* can be selected. (Note that point *p*max maximizes the area of the triangle withtwo vertices at *p*1 and *pn* and the third one at some other point of *S*1.) Then thealgorithm identifies all the points of set *S*1 that are to the left of the line *p*1*p*max;these are the points that will make up the set *S*1*,*1*.* The points of *S*1 to the left of the line *p*max*pn* will make up the set *S*1*,*2*.*  **Efficiency:**  Quickhull has the same θ*(n*2*)* worst-case efficiency as quicksort In the average case, however, we should expect a much better performance. |
|  | **Solve the following using Brute-Force algorithm:**  **Find whether the given string follows the specified pattern and return 0 or 1 accordingly.*(May 15,17)***  **Examples:**   1. **Pattern: “abba”, input:”redblueredblue” should return 1** 2. **Pattern: “aaaa”, input:”asdasdasdasd” should return 1** 3. **Pattern: “aabb”, input:”xyzabcxzyabc” should return 0**   **Solution:**  Pattern is abba [2 a’s and 2 b’s ] and string = redbluebluered[14 char]  Let number of chars in ‘a’ = x and ‘b’ = y  3x +4y = 14 find all possibilities of x and y  Here it came: x =2 and y = 2  Loop over all possibilities of x and y  Check in one more loop if string is following that pattern or not.  The approach is  Example: Pattern [a b a b], given string = redblueredblue (14 characters in total)  |a| (length of a) = 1, then that makes 2 characters for as and 12 characters is left for bs, i.e. |b| = 6, Divided string = r edblue r edblue. This matches right away.  Example2: Pattern = [a b a b], string = redbluebluered (14 characters in total)  |a| = 1, |a| = 6 - > divided string = r edblue b luered -> no match  |a| = 2, |a| = 5 -> divided string = r edblue b luered -> no match  |a| = 3, |a| = 4 -> divided string = r edblue b luered -> no match  Like this, it should be trial and error for |a| |b| length.  Algorithm: function brute\_force(text[], pattern[])  {  // let n be the size of the text and m the size of the pattern  For (i=0; i<n; i++)  {  For (j=0;j<m &&i+j<n; j++)  If (text [i+j]!=pattern[j]) break;  //mismatch found, break the inner loop  If(j==m) // match found  }  } |
|  | **Find all the solution to the travelling salesman problem (cities and distances shown below) by exhaustive search. Give the optimal solution. *(May 16)***  **TRAVELING SALESMAN PROBLEM**  The ***traveling salesman problem (TSP)***, the problem asks to find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started. The problem can be modeled by a weighted graph, with the graph’s vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.)  We assume, with no loss of generality, that all circuits start and end at one particular vertex. Thus, we can get all the tours by generating all the permutations of *n* − 1 intermediate cities, compute the tour lengths, and find the shortest among them. Figure presents a small instance of the problem and its solution by this method.    Figure reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half. The total number of permutations needed is still 1/2 *(n* − 1*)*!*,* which makes the exhaustive-search approach impractical for all but very small values of *n*. |
|  | **Solve travelling salesman problem using brute force approach for the given example. How the solution can be obtained using branch and bound method.(Nov 2018)**    In the following example, we will illustrate the steps to solve the travelling salesman problem.  We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are (n - 1)! number of possibilities.  Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.  Let us consider a graph G = (V, E), where V is a set of cities and E is a set of weighted edges. An edge e(u, v) represents that vertices u and v are connected. Distance between vertex u and v is d(u, v), which should be non-negative.  Suppose we have started at city 1 and after visiting some cities now we are in city j. Hence, this is a partial tour. We certainly need to know j, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.  For a subset of cities S Є {1, 2, 3, ... , n} that includes 1, and j Є S, let C(S, j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.  When |S| > 1, we define C(S, 1) = ∝ since the path cannot start and end at 1.  Now, let express C(S, j) in terms of smaller sub-problems. We need to start at 1 and end at j. We should select the next city in such a way that    **Algorithm:** Traveling-Salesman-Problem  C ({1}, 1) = 0  for s = 2 to n do  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1  C (S, 1) = ∞  for all j Є S and j ≠ 1  C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}  Return minj C ({1, 2, 3, …, n}, j) + d(j, i)  **Analysis**  There are at the most 2n .n sub-problems and each one takes linear time to solve. Therefore, the total running time is O(2n .n2).  From the above graph, the following table is prepared.   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 1 | 2 | 3 | 4 | | 1 | 0 | 10 | 15 | 20 | | 2 | 5 | 0 | 9 | 10 | | 3 | 6 | 13 | 0 | 12 | | 4 | 8 | 8 | 9 | 0 |  S = Φ Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5  Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6  Cost(4,Φ,1)=d(4,1)=8 S = 1 Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}  Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15  Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18  Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18  Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20  Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15  Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13 S = 2S = 3   The minimum cost path is 35.  Start from cost {1, {2, 3, 4}, 1}, we get the minimum value for d [1, 2]. When s = 3, select the path from 1 to 2 (cost is 10) then go backwards. When s = 2, we get the minimum value for d [4, 2]. Select the path from 2 to 4 (cost is 10) then go backwards.  When s = 1, we get the minimum value for d [4, 2] but 2 and 4 is already selected. Therefore, we select d [4, 3] (two possible values are 15 for d [2, 3] and d [4, 3], but our last node of the path is 4). Select path 4 to 3 (cost is 9), then go to s = Φ step. We get the minimum value for d [3, 1] (cost is 6).  **Branch and bound** is an algorithm that enhances the idea of generating a state space tree with idea of estimating the best value obtainable from a current node of the decision tree. If such an estimate is not superior to the best solution seen up to that point in the processing, the node is eliminated from further consideration. |