|  |
| --- |
|  |
| **Running Time Calculations**   1. **Basic****operations in algorithm**    * + 1. An algorithm to solve a particular task employs some set of basic operations.        2. When we estimate the amount of work done by an algorithm we usually do not consider all the steps such as e.g. initializing certain variables.        3. Generally, the total number of steps is roughly proportional to the number of the basic operations.        4. Thus, we are concerned mainly with the basic operations -  how many times the basic operations have to be performed depending on the size of input.   **The work done by an algorithm, i.e. its complexity, is determined by the number of the basic operations (Primitive operations) necessary to solve the problem**.  **Note:**The complexity of a program that implements an algorithm is roughly the same,  but not exactly the same as the complexity of the algorithm.  Here we are talking about algorithms independent on any particular implementation -  programming language or computer.   1. **Size of****input**   Some algorithms are not dependent on the size of the input - the number of the operations they perform is fixed. Other algorithms depend on the size of the input, and these are the algorithms that might cause problems. Before implementing such an algorithm, we have to be sure that the algorithm will finish the job in reasonable time.  What is size of input? We need to choose some reasonable measure of size. Here are some examples:   |  |  | | --- | --- | | **Problem** | **Size of input** | | Find x in an array | The number of the elements in the array | | Multiply two matrices | The dimensions of the matrices | | Sort an array | The number of elements in the array | | Traverse a binary tree | The number of nodes | | Solve a system of linear equations | The number of equations, or the number of the unknowns, or both |   **Counting the number of operations**  There are four rules to count the operations:  **Rule 1: *for* loops**  The running time of a ***for* loop** is at most the running time of the statements inside the loop  times the number of iterations.  **for( i = 0; i < n; i++)**  **sum = sum + i;**  **Find how many times each statement is executed:**  for( i = 0; i < n; i++) // i = 0; executed only once: O(1)  // i < n; n + 1 times : O(n)  // i++; n times: O(n)  **total time of the loop heading:**  O(1) + O(n) + O(n) = **O(n)**    sum = sum + i; // executed n times: **O(n)**  **The loop heading plus the loop body will give: O(n) + O(n) = O(n). Loop running time is: O(n)**  Hence we can say: **If**   * + 1. the size of the loop is **n** (loop variable runs from 0 or some fixed constant, to **n**) and     2. the body has constant running time (no nested loops)   **then the time is O(n)**  **Rule 2: Nested loops**  The total running time is the running time of the inside statements **times** the product of the sizes of all the loops  ***sum = 0; for( i = 0; i < n; i++)***  ***for( j = 0; j < n; j++)***  ***sum++;***  **Applying Rule 1** for the nested loop **(the 'j' loop) we get O(n)** for the body of the outer loop.  The outer loop **(the 'i' loop)** runs n times so we get **O(n),**  therefore the total time for the nested loops will be : **O(n) \* O(n) = O(n\*n) = O(n2).**  ***What happens if the inner loop does not start from 0?***  sum = 0; for( i = 0; i < n; i++)  for( j = i; j < n; j++)  sum++;  Here, the number of the times the inner loop is executed **depends on the value of i:**  i = 0, inner loop runs n times  i = 1, inner loop runs (n-1) times  i = 2, inner loop runs (n-2) times  ...  i = n - 2, inner loop runs 2 times  i = n - 1, inner loop runs 1 (once)  Adding the right column, we get: ( 1 + 2 + … + n) = n\*(n+1)/2 = **O(n2)**  **General rule for loops:**  **Running time is the product of the size of the loops times the running time of the body.**  Example:  ***sum = 0; for( i = 0; i < n; i++)***  ***for( j = 0; j < 2n; j++)***  ***sum++;***  We have one operation inside the loops, and the product of the sizes is 2n2. Hence the running time is O(2n2) = **O(n2)**  **Rule 3: Consecutive program fragments**  The total running time is the maximum of the running time of the individual fragments  ***sum = 0; for( i = 0; i < n; i++)***  ***sum = sum + i;***  ***sum = 0; for( i = 0; i < n; i++)***  ***for( j = 0; j < 2n; j++)***  ***sum++;***  The first loop **runs in O(n) time,** the **second - O(n2) time,** the maximum is **O(n2)**  **Rule 4: If statement**  **if Condition**  **S1;**  **else**  **S2;**  The running time is the maximum of the running times of **S1** and **S2**.    **Examples**   * 1. **Search in an unordered array of elements**.   for (i = 0; i < n; i++)  if (a[i] == x) return 1; // 1 means succeed  return -1; // -1 means failure, the element is  // not found  The basic operation in this problem is comparison, so we are interested in how the number of comparisons depends on **n**.  Here we have a loop that runs at most **n** times:  If the element is not there, the algorithm needs **n** comparisons. If the element is at the end, we need **n** comparisons. If the element is somewhere in between, we need less than **n** comparisons.  In the **worst case** (element not there, or located at the end), we have **n**comparisons to make. Hence the number of operations is **O(n)**.  To find what is the **average case**, we have to consider the possible cases:  element is at the first position: 1 comparison element is at the second position: 2 comparisons etc.  We compute the sum of the comparisons for the possible cases and then divide by the number of the cases, assuming that all cases are equally likely to happen:  1 + 2 + … + n = n(n+1)/2  Thus in the average case the number of comparisons would be (n+1)/2 = **O(n).**  Generally, the algorithm for finding an element in an unsorted array needs **O(n)** operations.   * 1. **Search in a table n x m**   for (i = 0; i < n; i++)  for (j = 0; j < m; j++)  if (a[i][j] == x) return 1 ; // 1 means succeed  return -1; // -1 means failure - the element  // is not found.  Here the inner loop runs at most **m** times and it is located in another loop that runs at most **n** times, so in total there would be at most **nm** operations. Strictly speaking, **nm** is less than **n2**when **m < n.** However, for very large values of **n** and **m** the difference is negligible,  the amount of work done is roughly the same.  Thus we can say that the complexity here is **O(n2)**.   * 1. **Finding the greatest element in an array**     amax = a[0];  for (i = 1; i < n; i++)  if (a[i] > amax)  amax = a[i];  Here the number of operations is always **n-1**. The amount of work depends on the size of the input, but does not depend on the particular values.  The running time is **O(n)**, we disregard "-1" because the difference for large n is negligible.  The complexity here is **O(n)**, we disregard "-1" because the difference for large **n**is negligible.     |  | | --- | | **Intro to Algorithms** | | **Problems on loop complexity**   * **Problem 1**   sum = 0; for( i = 0; i < n; i++)  sum++;  **Sol:** The running time for the operation *sum++* is a constant. The loop runs *n* times, hence the complexity of the loop would be **O(n)**   * **Problem 2**   sum = 0; for( i = 0; i < n; i++)  for( j = 0; j < n; j++)  sum++;  **Sol:** The running time for the operation *sum++* is a constant.  The outer loop runs *n* times, The nested loop also runs *n* times, hence the complexity would be  **O(n2)**   * **Problem 3**   sum = 0; for( i = 0; i < n; i++)  for( j = 0; j < n \* n; j++)  sum++;  **Sol:** The running time for the operation *sum++* is a constant.  The outer loop runs *n* times, The nested loop runs *n \* n* times, hence the complexity would be  **O(n3)**   * **Problem 4**   sum = 0; for( i = 0; i < n; i++)  for( j = 0; j < i; j++)  sum++;  **Sol:** The running time for the operation *sum++* is a constant.  The outer loop runs *n* times. For the first execution of the outer loop the inner loop runs only once. For the second execution of the outer loop the inner loop runce twice, for the third execution - three times, etc. Thus the inner loop will be executed 1 + 2 + ... + (n-1) + n times.  1 + 2 + ... + (n-1) + n = n(n+1) / 2, which gives (n+1) / 2 on average.  Thus the total running time would be O(n\*(n+1)/2) = O(n\*n) = **O(n2)**   * **Problem 5**   sum = 0; for( i = 0; i < n; i++)  for( j = 0; j < i\*i; j++)  for( k = 0; k < j; k++)  sum++;  **Sol:** The running time for the operation *sum++* is a constant.  The most inner loop runs at most *n\*n* times, the middle loop also runs at most *n\*n* times, and the outer loop runs *n* times, thus the overall complexity would be **O(n5)**   * **Problem 6**   sum = 0; for( i = 0; i < n; i++)  for( j = 0; j < i\*i; j++)  if (j % i ==0)  for( k = 0; k < j; k++)  sum++;  **Sol:** Compare this problem with [Problem 5](http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L03-BigOhSolutions.htm" \l "PR5). Obviously the most inner loop will run less times than in Problem 5, and a refined analysis is possible, we are usually content to neglect the *if* statement and consider its running time to be O(n2), yielding an overall running time of **O(n5)**   * **Problem 7**   sum = 0; for( i = 0; i < n; i++)  sum++;  val = 1; for( j = 0; j < n\*n; j++)  val = val \* j;  **Sol:** First, we asume that the running time to compute an arithmetic expression without function calls is negligible. Then, we have two consecutive loops with running times O(n) and O(n2). We take the maximum complexity, hence the overall runing time would be **O(n2)**   * **Problem 8**   sum = 0; for( i = 0; i < n; i++)  sum++;  for( j = 0; j < n\*n; j++)  compute\_val(sum,j);  The complexity of the function *compute\_val(x,y)*is given to be O(nlog*n*)  **Sol:** The difference between this problem and the [Problem 7](http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L03-BigOhSolutions.htm" \l "PR7) consists in the presence of a function in the loop body, whose complexity is not a constant - it depends on *n* and is given to be O(nlog*n*)  The second loop runs *n\*n* times, so its complexity would be O(n2 \* nlog*n*) = O(n3log*n*).  The first loop has less running time - O(n), we take the maximum and conclude that the overall running time would be **O(n3log*n*)** | |  | |