Contents

Chapter 1

Analysis of Algorithms | Set 1 (Asymptotic Analysis)

Why performance analysis?
There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?
The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!

Given two algorithms for a task, how do we find out which one is better?
One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.
1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.
For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Does Asymptotic Analysis always work?
Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.
Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be

possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

We will covering more on analysis of algorithms in some more posts on this topic.

References:
MIT's Video lecture 1 on Introduction to Algorithms.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

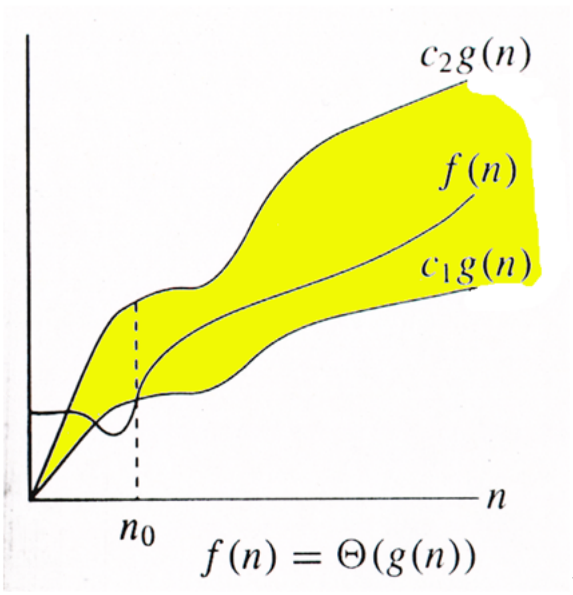
Source

http://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/

Chapter 2

Analysis of Algorithms | Set 3 (Asymptotic Notations)

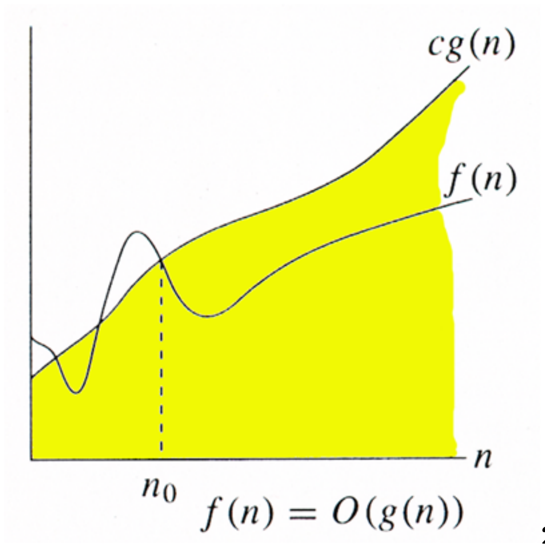
We have discussed Asymptotic Analysis, andWorst, Average and Best Cases of Algorithms. The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.



1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.
$3n^3 + 6n^2 + 6000 = \Theta(n^3)$
Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ beats $\Theta n^2)$ irrespective of the constants involved.
For a given function g(n), we denote Θ(g(n)) is following set of functions.

```
Θ((g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that
                  0 = n0}
```

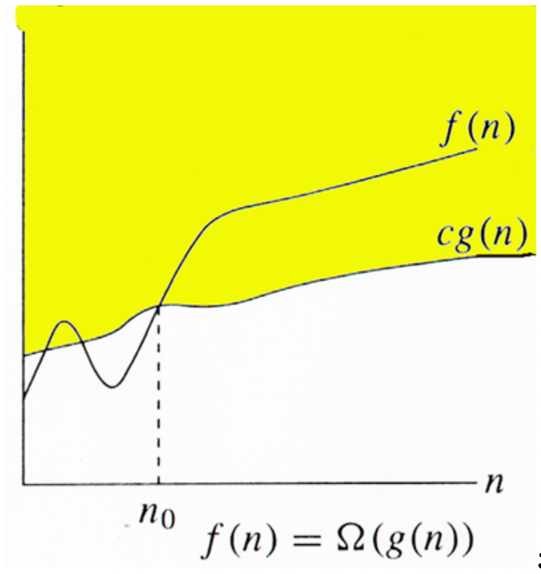
The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(nˆ2). Note that O(nˆ2) also covers linear time.
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
1. The worst case time complexity of Insertion Sort is Θ(nˆ2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

```
O(g(n)) = { f(n): there exist positive constants c and n0 such that
            0 = n0}
```



3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notationbest case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

```
Ω (g(n)) = {f(n): there exist positive constants c and n0 such that
                                 0 = n0}.
```

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Exercise:
Which of the following statements is/are valid?
1. Time Complexity of QuickSort is Θ(nˆ2)
2. Time Complexity of QuickSort is O(nˆ2)
3. For any two functions f(n) and g(n), we have f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
4. Time complexity of all computer algorithms can be written as Ω(1)

References:

Lec 1 | MIT (Introduction to Algorithms)

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/

Chapter 3

Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)

In the previous post, we discussed how Asymptotic analysis overcomes the problems of naive way of analyzing algorithms. In this post, we will take an example of Linear Search and analyze it using Asymptotic analysis.

We can have three cases to analyze an algorithm:
1) Worst Case
2) Average Case
3) Best Case

Let us consider the following implementation of Linear Search.

```
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
       if (arr[i] == x)
         return i;
    }
    return -1;
}

/* Driver program to test above functions*/
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)
In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n).

Average Case Analysis (Sometimes done)
In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

```
Average Case Time =

                  =

                  = Θ(n)
```

Best Case Analysis (Bogus)
In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1)

Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example,Merge Sort. Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

References:
MIT's Video lecture 1 on Introduction to Algorithms.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/

Chapter 4

Analysis of Algorithm | Set 4 (Solving Recurrences)

In the previous post, we discussed analysis of loops. Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.

```
 For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction
to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true
for values smaller than n.

T(n) = 2T(n/2) + n
    <= cn/2Log(n/2) + n
    =  cnLogn - cnLog2 + n
    =  cnLogn - cn + n
    <= cnLogn
```

2) Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

```
 For example consider the recurrence relation
T(n) = T(n/4) + T(n/2) + cn2

           cn2
```

```
         /       \
     T(n/4)     T(n/2)

If we further break down the expression T(n/4) and T(n/2),
we get following recursion tree.

                cn2
           /           \
       c(n2)/16      c(n2)/4
      /      \          /     \
  T(n/16)     T(n/8)  T(n/8)    T(n/4)
Breaking down further gives us following
                 cn2
            /            \
       c(n2)/16          c(n2)/4
       /      \            /      \
c(n2)/256   c(n2)/64  c(n2)/64    c(n2)/16
 /    \      /    \    /    \       /    \

To know the value of T(n), we need to calculate sum of tree
nodes level by level. If we sum the above tree level by level,
we get the following series
T(n)  = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ....
The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.
We get the sum as (n2)/(1 - 5/16) which is O(n2)
```

3) Master Method:
Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

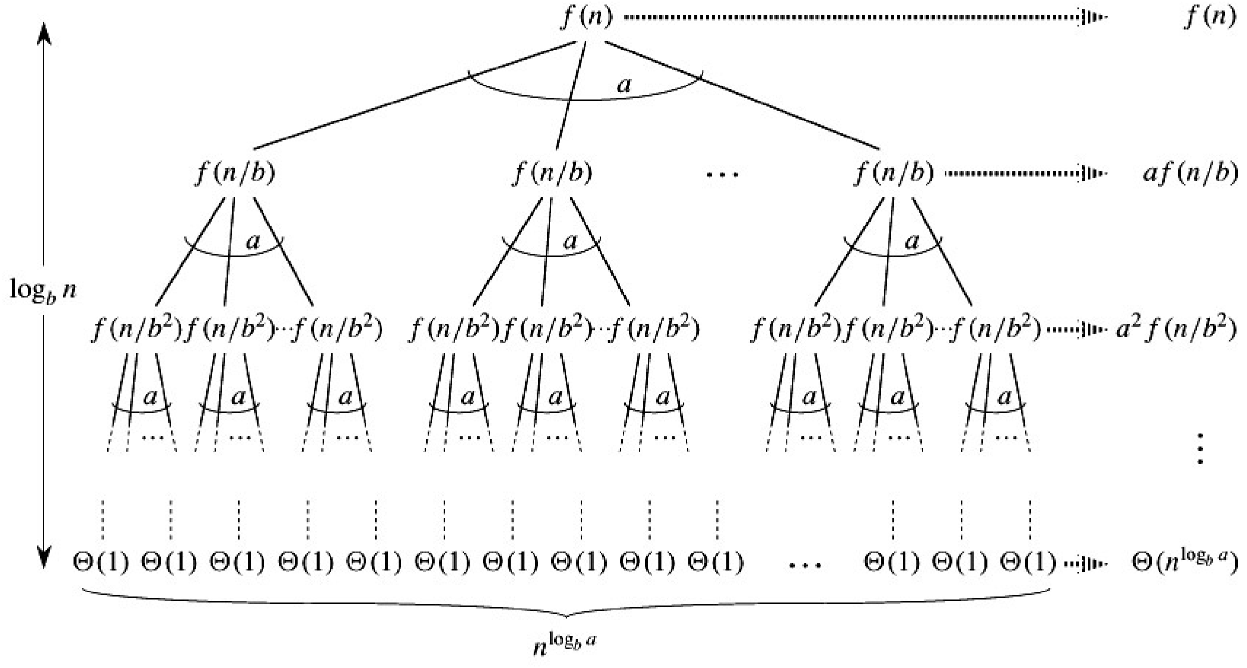
```
 T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
```

There are following three cases:
1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$

3.If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?
Master method is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and Log$_b$a] is also 1. So the solution is Θ(n Logn)

Binary Search: T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and Log$_b$a is also 0. So the solution is Θ(Logn)

Notes:

1) It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence T(n) = 2T(n/2) + n/Logn cannot be solved using master method.

2) Case 2 can be extended for f(n) = Θ(n^{c}Log^{k}n)
If f(n) = Θ(n^{c}Log^{k}n) for some constant k >= 0 and c = Log$_b$a, then T(n) = Θ(n^{c}Log^{k+1}n)

Practice Problems and Solutions on Master Theorem.

References:

http://en.wikipedia.org/wiki/Master_theorem
MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

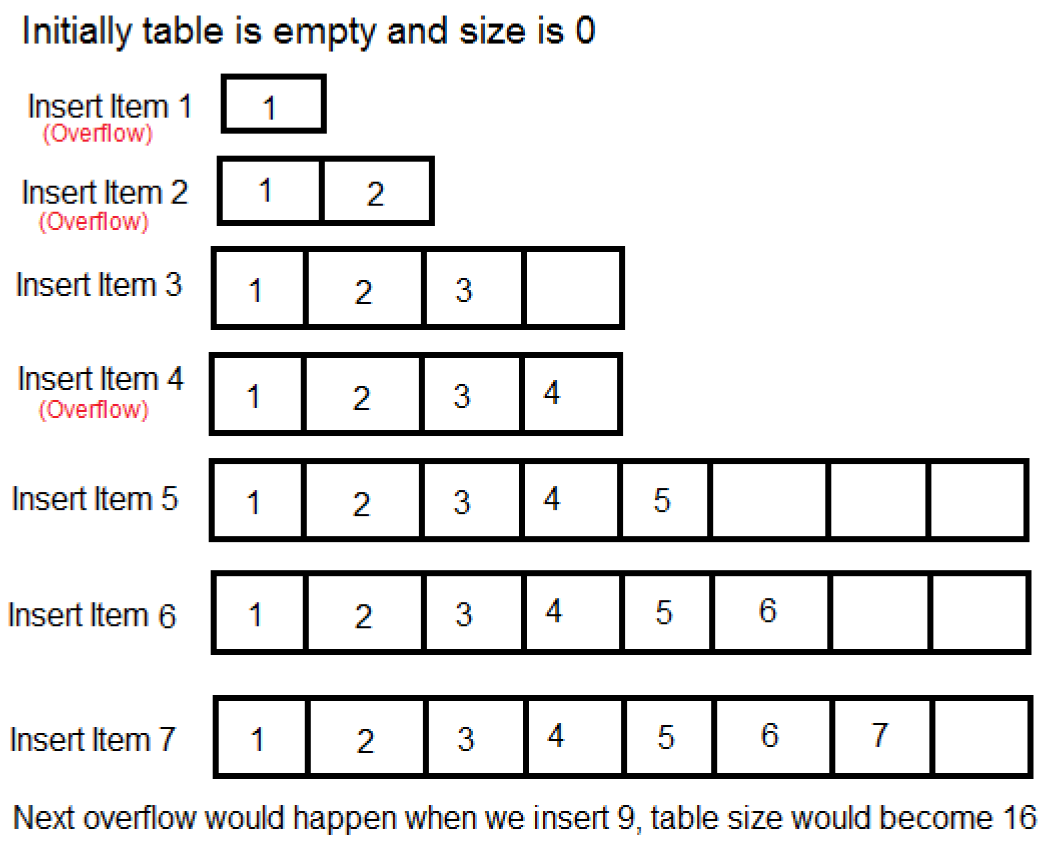
http://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/

Chapter 5

Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.
The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

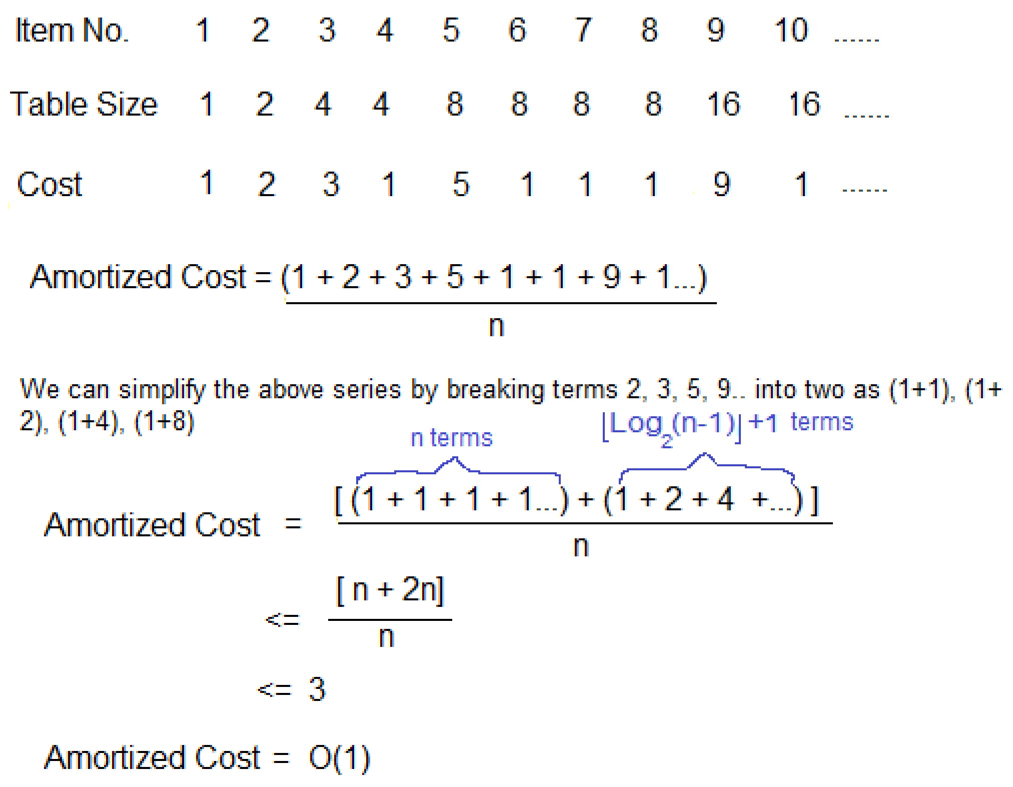
Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.



The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.
1) Allocate memory for a larger table of size, typically twice the old table.
2) Copy the contents of old table to new table.
3) Free the old table.

If the table has space available, we simply insert new item in available space.

What is the time complexity of n insertions using the above scheme?
If we use simple analysis, the worst case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n * O(n) which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take $\Theta(n)$ time.



Item No.	1	2	3	4	5	6	7	8	9	10	......
Table Size	1	2	4	4	8	8	8	8	16	16	......
Cost	1	2	3	1	5	1	1	1	9	1	......

$$\text{Amortized Cost} = \frac{(1+2+3+5+1+1+9+1\ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\,\overbrace{(1+1+1+1\ldots)}^{n\text{ terms}} + \overbrace{(1+2+4+\ldots)}^{\lfloor \mathrm{Log}_2(n-1)\rfloor+1\text{ terms}}\,]}{n}$$

$$<= \frac{[n+2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in vectors in C++, ArrayList in Java.

Following are few important notes.
1) Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

2) The above Amortized Analysis done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called ***Accounting Method*** and ***Potential Method***. We will be discussing the other two methods in separate posts.

3) The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

Sources:
Berkeley Lecture 35: Amortized Analysis
MIT Lecture 13: Amortized Algorithms, Table Doubling, Potential Method

http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/

Chapter 6

Analysis of Algorithms | Set 4 (Analysis of Loops)

We have discussed Asymptotic Analysis, Worst, Average and Best Cases and Asymptotic Notations in previous posts. In this post, analysis of iterative programs with simple examples is discussed.

1) O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
    // set of non-recursive and non-loop statements
```

For example swap() function has O(1) time complexity.
A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
    // Here c is a constant
   for (int i = 1; i
```

```
2) O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by
    // Here c is a positive integer constant
   for (int i = 1; i  0; i -= c) {
       // some O(1) expressions
   }
```

3) O(n^c): Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have O(n^2) time complexity

```
   for (int i = 1; i <=n; i += c) {
       for (int j = 1; j <=n; j += c) {
          // some O(1) expressions
       }
   }

   for (int i = n; i > 0; i += c) {
       for (int j = i+1; j <=n; j += c) {
          // some O(1) expressions
   }
```

For example Selection sort and Insertion Sort have O(n^2) time complexity.

4) O(Logn) Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
    for (int i = 1; i <=n; i *= c) {
       // some O(1) expressions
   }
   for (int i = n; i > 0; i /= c) {
       // some O(1) expressions
   }
```

For example Binary Search(refer iterative implementation) has O(Logn) time complexity.

5) O(LogLogn) Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

```
    // Here c is a constant greater than 1
   for (int i = 2; i <=n; i = pow(i, c)) {
       // some O(1) expressions
   }
   //Here fun is sqrt or cuberoot or any other constant root
   for (int i = n; i > 0; i = fun(i)) {
       // some O(1) expressions
   }
```

See thisfor more explanation.

How to combine time complexities of consecutive loops?
When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
   for (int i = 1; i <=m; i += c) {
        // some O(1) expressions
   }
   for (int i = 1; i <=n; i += c) {
        // some O(1) expressions
   }
   Time complexity of above code is O(m) + O(n) which is O(m+n)
   If m == n, the time complexity becomes O(2n) which is O(n).
```

How to calculate time complexity when there are many if, else statements inside loops?
As discussed here, worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum

number of statements to be executed.
For example consider the linear search function where we consider the case when element is present at the end or not present at all.
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

How to calculate time complexity of recursive functions?
Time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences. We will soon be discussing recurrence solving techniques as a separate post.

Quiz on Analysis of Algorithms

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/

Chapter 7

What does 'Space Complexity' mean?

Space Complexity:
The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

Auxiliary Space is the extra space or temporary space used by an algorithm.

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/g-fact-86/

Category: Misc

Chapter 8

A Time Complexity Question

What is the time complexity of following function fun()? Assume that log(x) returns log value in base 2.

```
void fun()
{
   int i, j;
   for (i=1; i<=n; i++)
      for (j=1; j<=log(i); j++)
         printf("GeeksforGeeks");
}
```

Time Complexity of the above function can be written as $\Theta(\log 1) + \Theta(\log 2) + \Theta(\log 3) + \ldots + \Theta(\log n)$ which is $\Theta(\log n!)$

Order of growth of 'log n!' and 'n log n' is same for large values of n, i.e., $\Theta(\log n!) = \Theta(n \log n)$. So time complexity of fun() is $\Theta(n \log n)$.

The expression $\Theta(\log n!) = \Theta(n \log n)$ can be easily derived from following Stirling's approximation (or Stirling's formula).

```
      log n! = n log n - n + O(log(n))
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Sources:
http://en.wikipedia.org/wiki/Stirling%27s_approximation

Source

http://www.geeksforgeeks.org/a-time-complexity-question/

Category: Misc

Chapter 9

Time Complexity of building a heap

Consider the following algorithm for building a Heap of an input array A.

```
BUILD-HEAP(A)
    heapsize := size(A);
    for i := floor(heapsize/2) downto 1
        do HEAPIFY(A, i);
    end for
END
```

What is the worst case time complexity of the above algo?
Although the worst case complexity looks like O(nLogn), upper bound of time complexity is O(n). See following links for the proof of time complexity.

http://www.cse.iitk.ac.in/users/sbaswana/Courses/ESO211/heap.pdf/
http://www.cs.sfu.ca/CourseCentral/307/petra/2009/SLN_2.pdf

Source

http://www.geeksforgeeks.org/g-fact-85/

Category: Misc

Chapter 10

Time Complexity where loop variable is incremented by 1, 2, 3, 4 ..

What is the time complexity of below code?

```
void fun(int n)
{
   int j = 1, i = 0;
   while (i < n)
   {
       // Some O(1) task
       i = i + j;
       j++;
   }
}
```

The loop variable 'i' is incremented by 1, 2, 3, 4, … until i becomes greater than or equal to n.

The value of i is x(x+1)/2 after x iterations. So if loop runs x times, then x(x+1)/2

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/time-complexity-where-loop-variable-is-incremented-by-1-2-3-4/

Category: Articles

Chapter 11

Time Complexity of Loop with Powers

What is the time complexity of below function?

```
void fun(int n, int k)
{
    for (int i=1; i<=n; i++)
    {
      int p = pow(i, k);
      for (int j=1; j<=p; j++)
      {
          // Some O(1) work
      }
    }
}
```

Time complexity of above function can be written as $1^k + 2^k + 3^k + \dots n1^k$.

Let us try few examples:

```
k=1
Sum = 1 + 2 + 3 ... n
    = n(n+1)/2
    = n2 + n/2

k=2
Sum = 12 + 22 + 32 + ... n12.
    = n(n+1)(2n+1)/6
    = n3/3 + n2/2 + n/6

k=3
Sum = 13 + 23 + 33 + ... n13.
    = n2(n+1)2/4
    = n4/4 + n3/2 + n2/4
```

In general, asymptotic value can be written as $\mathbf{(n^{k+1})/(k+1) + \Theta(n^k)}$

Note that, in asymptotic notations like Θ we can always ignore lower order terms. So the time complexity is $\mathbf{\Theta(n^{k+1} / (k+1))}$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/time-complexity-of-loop-with-powers/

Category: Articles Tags: time complexity

Chapter 12

NP-Completeness | Set 1 (Introduction)

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

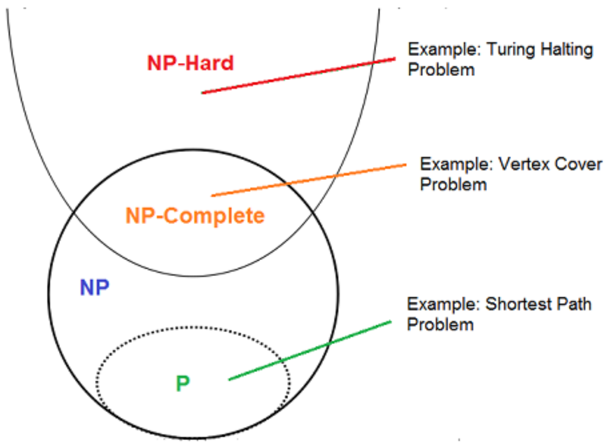
Can all computational problems be solved by a computer? There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source Halting Problem).
Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

What are NP, P, NP-complete and NP-Hard problems?
P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).
Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete 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 (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.


NP-Hard
Example: Turing Halting Problem
Example: Vertex Cover Problem
NP-Complete
NP
Example: Shortest Path Problem
P

This diagram assumes that P != NP

Decision vs Optimization Problems

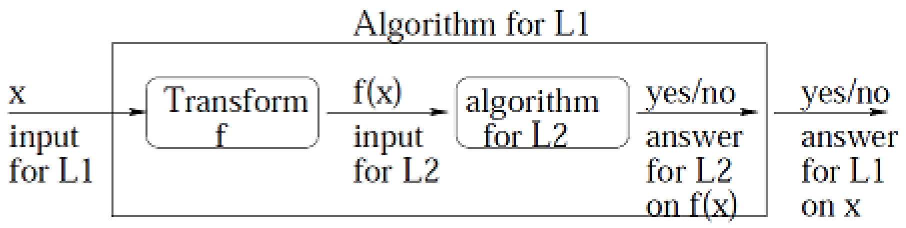
NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not.

The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?
There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem)is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

We will soon be discussing more NP-Complete problems and their proof for NP-Completeness.

References:
MIT Video Lecture on Computational Complexity

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
http://www.ics.uci.edu/~eppstein/161/960312.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/np-completeness-set-1/