BM267 - Introduction to Data Structures

3. Principles of Algorithm Analysis

Ankara University

Computer Engineering Department

Bulent Tugrul

Analysis Framework

- There are two kinds of algorithm efficiency: “time effinciency” and “space effinciency”. **Time efficiency** indicates that how fast an algorithm runs; **space efficiency** deals with amount of space it needs.
- Nowadays the amount of extra space required by an algorithm is typically not of as much concern, due to the improvements in memory capacity.
- We are going to concentrate on time efficiency but analytical framework is applicable to analyzing space efficiency as well.

Measuring an Input's size

- It is logical to investigate an algorithm's effinciency as a function of some parameter **"n"** indicating the algorithm's input size.

- The input size will be the size of the lists or arrays for problems such as sorting, searching, or finding the smallest element.

Units for Measuring Running time

- First approach: We can use time measurement (second, or millisecond) to measure an algorithm's running time.
- There are drawbacks of this approach, such as speed of the computer, quality of programmer, the compiler used in generating the machine code.
- Second approach: We can count the number of times each algorithm's operations is executed. The thing to do is to identify the most important operation of the algorithm, called the ***basic operation***, the operation contributing the most of the total running time, and compute the number of times the basic operaion is executed.
- As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop.

Units for Measuring Running time(2)

- Ex: Sequential Search

```
int search( int a[0...n-1], int K)
{
   int i;                          executes only once
   for( i=0;  i < n; i++ )         executes n times
        if( a[i] = = K )            executes at most n times
                return i;           at most once
        return –1;                  at most once
}
```

Units for Measuring Running time(3)

- Most sorting algorithms work by comparing elements of a list with each other; for such algorithms the basic operation is a key comparision.
- As another example algorithms for matrix multiplication require two operations; multiplication and addition. multiplication takes much more longer time than addition so we can use the total number of multiplication as our unit for algorithm measure.
- MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])

```
for( i=0; i < n; i++)
        for( j=0; j < n; j++)
                C[i,j]=0;
                for( k=0; k < n; k++)
                        C[i,j] = C[i,j] + A[i,k]*B[k,j];
```

Units for Measuring Running time(4)

- Efficiency T(*n*) is investigated as a function of some parameter *n* indicating problem's size.
- T(*n*) is computed as the number of times the algorithm's "basic operation" is executed.

- For matrix multiplication $T(n)=n^3$

Values of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Worst, Best, and Average Case

•There are many algorithms for which running time depends not only on an input size but also on a particular input.

•The running time of search algorithm can be quite different for the same list size n. In the ***worst-case***, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$$T(n)_{worst} = n.$$

•The worst-case efficiency of an algorithm is its efficiency for the worst case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size.

Worst, Best, and Average Case(2)

- In order to find the worst case efficiency of an algorithm, we analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count among all possible inputs of size n.
- The worst case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size 'n' the running time will not exceed $T(n)_{worst}$.
- The ***best-case*** efficiency of an algorithm is its efficiency for the best case input of size n, which is an input of size n for which the algorithm runs the fastest.
- The best case does not mean that the smallest input; it means the input of size n for which the algorithm runs the fastest.

Worst, Best, and Average Case(3)

- For example, for search best case inputs will be of size n with their first elements equal to search key. $T(n)_{best}=1$
- The best case efficiency is not as important as worst case efficiency.
- Neither worst-case nor best-case efficiency yields the necessary information about an algorithms behavior on a random input. The **average-case efficiency** provide us this kind of information.
- To analyze the algorithm's average-case efficiency' we must make some assumptions about possible inputs of size n.
- The assumptions are:
  - The probability of a successful search is equal to p ($0 \leq p \leq 1$)
  - The probability of the first match occurring in the *i* th position of the list is the same for every *i*.

Worst, Best, and Average Case(4)

- In the case of a successful search, the probability of the first match occurring in the *i* th position of the list is p/n for every *i*.
- The number of comparisons made by the algorithm is *i*.
- In the case of unsuccessful search, the number of comparisons is n with the probability of such a search being (1-p).
- $T_{avg}(n)= [1*p/n + 2*p/n+\ldots+n*p/n] + n*(1-p)$

  $= p/n*[1 + 2+\ldots+n] + n*(1-p)$

  $= p*(n+1)/2 + n(1-p)$

  If p=1(successful search) $T_{avg}(n)= (n+1)/2$

  If p=0(unsuccessful search) $T_{avg}(n)= n$

Mathematical Analysis of Nonrecursive Algorithms

- General Plan for Analyzing Efficiency on Nonrecursive Algorithms
  - Decide on parameters indicating an input's size
  - Identify the algorithms basic operations(As a rule, it is located in its inner loop)
  - Check whether the number of times the basic operation is executed. Investigate worst, best, and average case efficiency.
  - Set up a sum expressing the number of times the algorithm's basic operation is executed.
  - Using standard formulas and rules of sum manipulation, find a closed form for the count

Mathematical Analysis of Nonrecursive Algorithms(2)

- Ex 1:

```
Algorithm MaxElement( A[0…n-1])
{  //Determines the largest element of the array
   maxval=A[0];
   for( i=1; i < n; i++)
         if( A[i]>maxval)
                  maxval=A[i];
   return maxval;
}
```

- The input size is the number of element in the array; n
- The operation that are going to be executed most often are in the algorithm's for loop.

Mathematical Analysis of Nonrecursive Algorithms(3)

- There are two operations in loop's body;the comparisons *A[i]>maxval* and the assignment *maxval=A[i]*. We consider the comparison as the basic operation because it is executed in each repetition.
- The number of comparisons will be ***n-1*.**
- The worst, best and average case efficiency will be same.
- The algorithm makes one comparisons on each execution of the loop; therefore

$$T(n) = \sum_{i=1}^{n-1} 1 = n-1$$

Mathematical Analysis of Nonrecursive Algorithms(4)

- Ex 1:

```
Algorithm UniqueElements( A[0…n-1] )
{   //checks whether all the elements in a given array are distinct
    for( i=0; i ≤  n-2; i++)
          for( j=i+1; j ≤  n-1; j++)
                if( A[i] = = A[j])
                      return false;
    return true;
}
```

- The input's size is the number of elements in the array: ***n***
- The basic operation is comparison: ***A[i] = = A[j]***

Mathematical Analysis of Nonrecursive Algorithms(5)

- Worst case occurs in two situations: arrays with no equal elements and the last two elements are the only equal pairs.
- For such inputs, one comparison is made for each repetition of innermost loop therefore:

$$T_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-2} (n-1+i)$$

$$= (n-1)*n / 2$$

In the worst case algorithm has to make $(n-1)*n / 2$ comparisons.

What is efficiency of matrix multiplication?

$$\text{Solution: } = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3$$

Mathematical Analysis of Recursive Algorithms

- General Plan for Analyzing Efficiency on Recursive Algorithms
  - Decide on parameters indicating an input's size
  - Identify the algorithms basic operations(As a rule, it is located in its inner loop)
  - Check whether the number of times the basic operation is executed. Investigate worst, best, and average case efficiency.
  - Set up a recurrence relation, with an initial condition, for the number of times the basic operation is executed.
- Ex 1: Computing the factorial function F(n) = n!

  n! = n*(n-1)*(n-2)*…*3*2*1 = n*(n-1)! for $n \geq 1$ and 0!=1

  we can compute F(n)= F(n-1)*n

Mathematical Analysis of Recursive Algorithms(2)

- Algorithm F(n)

```
{//Computes n! recursively
    If ( n = = 0 )
            return 1;
    else
            return F(n-1)*n; }
```

- We consider "***n***" as the algorithm input size.
- The basic operation of the algorithm is again multiplication.
- We denote T(n) the number of multiplications needed to compute F(n).

$$T(n) = \underbrace{T(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n}$$

Mathematical Analysis of Recursive Algorithms(3)

- How to solve a recurrence equation
  - We need a ***initial condition*** that tells us the value which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls.

    if (n = = 0) return 1;

    This tells us two things. First the calls stop when n=0. Second we can see that when n=0 the algorithm performs no multiplication.There fore

    $$T(0) = 0$$

  - We use ***backward substitution*** to solve the equation

Mathematical Analysis of Recursive Algorithms(4)

$T(n) = T(n-1) + 1$ substitute T(n-1) with T(n-2) +1

$= [T(n-2) +1] +1 = T(n-2) +2$ substitute T(n-2) with T(n-3) +1

$= [T(n-3) +1] +2 = T(n-3) +3$

….

$T(n) = T(n-i) +i$

When i = n

$T(n) = T(n-n) + n = T(0) + n = 0 + n = n$

So we need ***n*** multiplication to compute F(n)