Algorithm Analysis

Contents

- Algorithms, the concept
- Why running time is important: Maximum Subsequence Problem
- Emperical evaluation of runnning time
- Computation model, time complexity and big-Oh notations
- Rules for time complexity evaluation
- Worst case and average case time complexity

Contents

- Algorithms, the concept
- Why Running time is important: Maximum Subsequence Problem
- Emperical evaluation of runnning time
- Computation model and big-Oh Notations
- Rules for running time evaluation
- Worst case and average case running time

Algorithms

- Data structures
  - Methods of organizing data
- What is Algorithm?
  - a set of well-defined instructions on the data to be followed to solve a problem
    - Takes a set of values, as input and
    - produces a value, or set of values, as output
  - May be specified
    - In English
    - As a computer program using some programming lang.
    - As a pseudo-code
- Algorithms + Data Structures = Programs

Why Algorithm Analysis Matter?

- Why algorithm analysis matter?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

Why Algorithm Analysis Matter?

- Maximum subsequence sum Problem. Given integers $A_1, A_2, \ldots, A_N$, find the maximum value of $\sum_{k=i}^{j} A_k$
- Example: for input -2,11,-4,13,-5,-2, the maximum is 20(A_2through A_4).
- There are various solutions to this problem and their running times vary drastically.
- Some solutions can return the results in a few seconds while other can take days or years.

A simple solution

Algorithm MSS([a_1,a_2,…,a_n])
Input: a_1,…,a_n is a sequence of numbers ($n \geq 0$)
Output: the maximum subsequence sum

```
maxSum ← 0
for i ← 1 to n
  for j ← i to n
    thisSum ← sum of [a_i, …, a_j]
    if thisSum > maxSum
      maxSum ← thisSum
return maxSum
```

Algorithm 1

```
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); i++ )
        for( int j = i; j < a.size( ); j++ )
        {
            int thisSum = 0;

            for( int k = i; k <= j; k++ )
                thisSum += a[ k ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }

    return maxSum;
}
```

Repeated computation Could be improved

Algorithm 2

```
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); i++ )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); j++ )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

A recursive solution based on divide and coquer

Algorithm MSS_rec([a_1,a_2,…,a_n])
Input: a_1,…,a_n is a sequence of numbers ($n \geq 0$)
Output: the maximum subsequence su

```
if n=0  return 0
if n =1 return a_1
leftMax = MSS_rec ([a_1,a_2,…,a_{n/2}])
rightMax = MSS_rec ([a_{n/2 +1}, a_{n/2 +2},…,a_n])
leftBorderMax ← MSS of [a_1,a_2,…,a_{n/2}] with a_{n/2}
rightBorderMax ← MSS of [a_{n/2+1}, a_{n/2 +2},…,a_n] with a_{n/2+1}
rossBorderMax ← leftBorderMax + rightBorderMax
return max(leftMax, rightMax, crossBorderMax)
```

Complexity:
$T(n)= 2T(n/2) + n$
$T(n) = O(n \log n)$

Algorithm 3

```
/**
 * Recursive maximum contiguous subsequence sum algorithm.
 * Finds maximum sum in subarray spanning a[left..right].
 * Does not attempt to maintain actual best sequence.
 */
int maxSumRec( const vector<int> & a, int left, int right )
{
    if( left == right )  // Base case
        if( a[ left ] > 0 )
            return a[ left ];
        else
            return 0;

    int center = ( left + right ) / 2;
    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    int maxRightBorderSum = 0, rightBorderSum = 0;
    for( int j = center + 1; j <= right; j++ )
    {
        rightBorderSum += a[ j ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum );
}

/**
 * Driver for divide-and-conquer maximum contiguous
 * subsequence sum algorithm.
 */
int maxSubSum3( const vector<int> & a )
{
    return maxSumRec( a, 0, a.size( ) - 1 );
}
```

Algorithm 2

```
/**
 * Quadratic maximum contiguous subsequence
 */
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); i++ )
    {
        int thisSum = 0;
        for( int j = i; j < a.size
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

The simple algorithm could be impoved: if sum of $[a_i..a_j]$ is negative, then this segment cannot start a MSS, we can skip to a_{j+1}.

Algorithm 4

```
/**
 * Linear-time maximum contiguous subsequence sum algorithm.
 */
int maxSubSum4( const vector<int> & a )
{
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 )
            thisSum = 0;
    }

    return maxSum;
}
```

Running times of maxSum Algorithms

- ***Demo of maxSumTest:***The running times of four algorithms (in seconds)

Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 10$	0.000009	0.000004	0.000006	0.000003
$N = 100$	0.002580	0.000109	0.000045	0.000006
$N = 1{,}000$	2.281013	0.010203	0.000485	0.000031
$N = 10{,}000$	NA	1.2329	0.005712	0.000317
$N = 100{,}000$	NA	135	0.064618	0.003206

Not Applicable for brute force algorithms.

Algorithm Analysis

- To evaluate and predict the amount of resources (such as time and storage) necessary to execute an algorithm.
- Time complexity: Running time of an algorithm means the time it takes from start to end, which is stated as a function of the input size;
- Space complexity: storage required to run an algorithm

Different approaches

- Empirical: Implement the algorithm and run the program. This depends on many factors such as hardware and software.
- Analytical (Formal): use theoretic-model data with a theoretical machine model.
- Machine model assumed
  - Instructions are executed one after another, with no concurrent operations ⇒ Not parallel computers
  - A simple instruction (not fancy, not depend on the size of the input) takes one time unit or one step to be executed.

Time complexity

- Formal analysis: count the number of steps (*time units)* to run an algorithm, T(n), where n is the size of the input. Call T(n) the *time complexity* of the algorithm.

Example

- Calculate $\sum_{i=1}^{N} i^3$

```
int sum(int n)
{
    int partialSum;

1   partialSum=0;                    1
2   for (int i=1;i<=n;i++)           2N+2
3     partialSum += i*i*i;           N
4   return partialSum;               1
}
```

- Lines 1 and 4 count for one unit each
- Line 3: executed N times, each time one unit
- Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2
- total cost: 3N + 4 ⇒ O(N)

Rules for Counting Running Time

- Rule for consecutive statements

These just add.

- Rule for if/else

Never more than the test plus the large of the running time of two branches.

- Rule for for loops

Running time of loop body times the number of iteration.

Time complexity Examples

- Example: nested loops

```
for (i=0;i<n; i++){
  for(j=I;j<n;j++){
    x +=y;
  }
}
```

- Example: a for loop program

```
for (i=0; i<n;i++) {
    x = sum(i); // sum(i) is not a simple instruction
}
```

Time Complexity Examples

- Example:

```
i=n;
while (i>0){
  i=i/2;
}
```

T(n) = 1 + 2log n +1

- Example:

```
if (a[mid]<x)
    low  = mid+1;
else if (a[mid]>x)
        high = mid -1;
 else
       return mid;
```

```
procedure bubbleSort( A : list of sortable items )
  n = length(A)
  for (i = 0; i < n; i++)
     for (j = n-1; j > i; j--)
        if A[j-1] > A[j] then
           swap(A[j-1], A[j])
        end if
     end for
  end for
end procedure
```

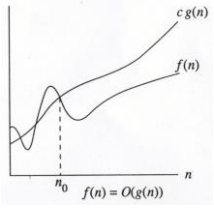
T(n) <= $1+\sum_{i=0}^{n-1}(\sum_{j=n-1}^{i+1}2)$ = n(n-1)

Which is the worst case time complexity.

Worst- / average- / best-case

- Worst-case running time of an algorithm
  - The longest running time for **any** input of size n
  - An upper bound on the running time for any input
    ⇒ guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the data is in decreasing order
  - The worst case can occur fairly often
    - E.g. in searching a database for a particular piece of information
- Best-case running time
  - sort a set of numbers in increasing order; and the data is already in increasing order
- Average-case running time
  - May be difficult to define what “average” means

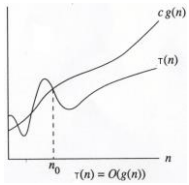
Growth Rate



- The idea is to establish a relative order among functions for large n
- $\exists$ c , $n_0 > 0$ such that $f(n) \leq c\, g(n)$ when $n \geq n_0$
- f(n) grows no faster than g(n) for “large” n

Big-Oh Notations (Asymptotic Notations)

- If for some g(n) there is a constant c and some n_0 such that $T(n) \leq cg(n)$ for all $n \geq n0$, then we denote $T(n) = O(f(n))$ and we call T(n) the asymptotic time complexity of the algorithm.



Big-Oh notations

- Example:

$T(n) = 3n^2 + 100n + 4 = O(n^2)$, reads "order n-squared" or "Big-Oh n-squared" ,

- For $T(n) = 3n^2 + 100n + 4$, both $T(n)=O(n^2)$ and $T(n)=O(n^3)$ are mathematically correct. However, a tight upper bound is preferred, so $T(n) = O(n^2)$ is the correct answer.
- Notice that we don't write $T(n) = O(2n^2)$ or $O(n^2 + n)$.
- $T(n) = O(f(n))$ has another reading: $T(n) \in O(f(n)) = \{g(n) | \exists c, n_0\ g(n) \leq cf(n)$ when $n \geq n_0\}$

Some rules for big-Oh

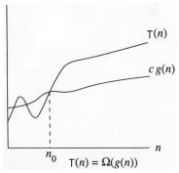
- Ignore the lower order terms
- Ignore the coefficients of the highest-order term
- No need to specify the base of logarithm
  - Changing the base from one constant to another changes the value of the logarithm by only a constant factor

If T1(n) = O(f(n)) and T2(n) = O(g(n)),

- $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$,
- $T_1(n) * T_2(n) = O(f(n) * g(n))$
- $\log^k n = O(n)$ for any positive constant k
- $\log n = O(n^k)$ for any positive constant k

Big-Omega, Lower bound

- Definition $T(n) = \Omega(g(n))$ if there are positive constants c and n_0 such that $T(n) \geq cg(n)$ when $n \geq n_0$.



- Let $T(n) = 2n^2 + 100n$. Then
  - $T(n) = \Omega(n)$
  - $T(n) = \Omega(n^2)$ (best answer)

Big-theta

- Definition $T(n) = \Theta(g(n))$ if $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$.
- The growth rate of T(n) *equals* the growth rate of g(n)
- Big-Theta means the bound is the tightest possible.
- Example: Let $T(n)=2n^2+100n$
  - Since $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$, thus $T(n) = \Theta(n^2)$

General Rules

- Loops
  - at most the running time of the statements inside the for-loop times the number of iterations.
- Nested loops: rule for loops apply
- Consecutive statements

```
for (i=0;i<n;i++)
    a[i]=0;
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        a[i] += a[j]+i+j;
```

  - These just add
  - $O(n) + O(n^2) = O(n^2)$
- Conditional: `If S1 else S2`
  - never more than the running time of the test plus the larger of the running times of S1 and S2.
- If $T(n)$ is a polynomial of degree k, then $T(n) = \Theta(n^k)$

Evaluating recursive algorithms

- Using recurrence relations;
- Using recursive trees.
- Example: factorial and Fibonacci, see Weiss.
- Hanoi tower: the solution uses divide & conquer strategy.

Task: move the tower from 1 to 3 using 2 as temporary storage.

Rules: 1. move only one disk at one time;

2. No larger disk can be placed on a smaller disk.

The Solution--Recursive Function

```
void move(int count, int start, int finish, int temp);
```

Pre: There are at least count disks on the tower start. The top disk (if any) on each of towers temp and finish is larger than any of the top count disks on tower start.

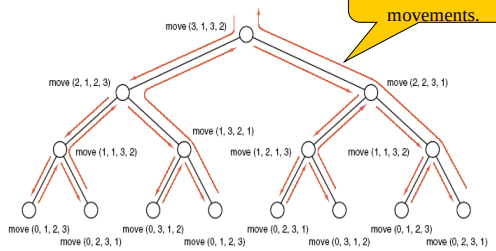
Post: The top count disks on start have been moved to finish; temp (used for temporary storage) has been returned to its starting position.

```
void move(int count, int start, int finish, int temp)
{
  if (count > 0) {
    move(count – 1, start, temp, finish);
    cout ≪ "Move disk " ≪ count ≪ " from " ≪ start
         ≪ " to " ≪ finish ≪ "." ≪ endl;
    move(count – 1, temp, finish, start);
  }
}
```

The base case is count == 0

Recursion tree for three disks

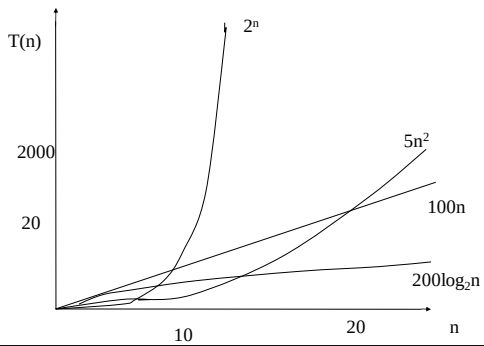
Number of nodes is the number of disk movements.



Solving Recurrence Relations

- Example: Assuming the complexity of an algorithm satisfies the recurrence relation T(n) = 4T(n/2) + n. Prove that T(n) = $O(n^3)$.
- Exercise: Prove that T(n) = $O(n^2)$.

•Some time complexities：O(1), O($\log_2$ n), O(n), O(n$\log_2$ n) , O(n^2), O(2^n), O(n!)



Assuming the running times of two algorithms:
$T_1(n) = 100n\log n$, $T_2(n) = n^2$

Size n	$T_1(n)$ (seconds)	$T_2(n)$ (seconds)
100	0.06	0.01
1000	1	1
10000	13	100
100000	2	166
1000000	33	16666 minutes=11.5days

Fast machines or efficient algorithms

- Assuming using more powerful machine which is 100 times faster

T(n)	n	n'		n'/n
10n	**1000**	**100000**	**n'=10n**	**100**
20n	**500**	**50000**	**n'=10n**	**100**
5nlogn	**251**	**14415**	**3n<n'<10n**	**57**
$2n^2$	**70**	**707**	**$n'=10^{1/2}n$**	**10**
2^n	**13**	**19**	**n'=n+6**	

n:input size original machine can solve in a hour (a 10000steps/h machine)，n':input size the new machine can solve in one hour.

Space complexity

- The amount of storage required to run an algorithm S(n) is called the space complexity, which is denoted as a function of the input size.
- S(n) includes: storage for variables and dynamic storage allocation for recursion, not counting the storage for input. One space unit for a variable.
- Analysis techniques use asymptotic notations.
- Example: for bubble sort S(n) = #vars = O(1), for hanoi move S(n) = #vars + recursion depth = O(n).

Requirements

- Writing algorithms in pseudocode
- Evaluating the running time of algorithms in big-Oh notaions.
- Understanding the meaning of big-Oh notations.

Reading and assignments

- Read Weiss Chapter 2
- Understand time complexity and its asymptotic notations.
- Homework (see webpage).