AMT 211 2.0 Design and Analysis of Algorithms

Dr. G.H.J. Lanel

Semester 1 - 2017

Outline

Outline

What is the Binomial Identity?

In general, a binomial identity is a formula expressing products of factors as a sum over terms, each including a binomial coefficient $\binom{n}{k}$.

The prototypical example is the binomial theorem;

For $n > 0$

$$(x + a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

What is the Binomial Identity?

In general, a binomial identity is a formula expressing products of factors as a sum over terms, each including a binomial coefficient $\binom{n}{k}$.

The prototypical example is the binomial theorem;

For $n > 0$

$$(x + a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

Many algorithms, particularly divide and conquer algorithms, have time complexities which are naturally modeled by recurrence relations.

A recurrence relation is an equation which is defined in terms of itself.

Why are recurrences good things?

1. Many natural functions are easily expressed as recurrences:

   $a_n = a_{n-1} + 1, a_1 = 1 \rightarrow a_n = n$ (polynomial)

   $a_n = 2 * a_{n-1}, a_1 = 1 \rightarrow a_n = 2^{n-1}$ (exponential)

   $a_n = n * a_{n-1}, a_1 = 1 \rightarrow a_n = n!$ (weird function)
2. It is often easy to find a recurrence as the solution of a counting problem. Solving the recurrence can be done for many special cases as we will see, although it is somewhat of an art.

Many algorithms, particularly divide and conquer algorithms, have time complexities which are naturally modeled by recurrence relations.

A recurrence relation is an equation which is defined in terms of itself.

Why are recurrences good things?

1. Many natural functions are easily expressed as recurrences:

   $a_n = a_{n-1} + 1, a_1 = 1 \rightarrow a_n = n$ (polynomial)

   $a_n = 2 * a_{n-1}, a_1 = 1 \rightarrow a_n = 2^{n-1}$ (exponential)

   $a_n = n * a_{n-1}, a_1 = 1 \rightarrow a_n = n!$ (weird function)
2. It is often easy to find a recurrence as the solution of a counting problem. Solving the recurrence can be done for many special cases as we will see, although it is somewhat of an art.

Recursion is Mathematical Induction!

In both we have general and boundary conditions, with the general condition breaking the problem into smaller and smaller pieces.

The initial or boundary condition terminate the recursion.

As we will see, induction provides a useful tool to solve recurrences- guess a solution and prove it by induction.

Example:

$T_n = 2 * T_{n-1} + 1, T_0 = 0$

n	0	1	2	3	4	5	6	7
T_n	0	1	3	7	15	31	63	127

Guess what the solution is?

Prove $T_n = 2^n - 1$ by induction:

Solution:

1. Show that the basis is true: $T_0 = 2^0 - 1 = 0$.
2. Now assume true for T_{n-1}.
3. Using this assumption show:

$$T_n = 2 * T_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$$

Solving Recurrences

No general procedure for solving recurrence relations is known which is why it is an art.

Asymptotic Analysis

1. The asymptotic analysis of an algorithm determines the running time in big-Oh notation.
2. To perform the asymptotic analysis
    1. We find the worst-case number of primitive operations executed as a function of the input size.
    2. We express this function with big-Oh notation.

Example of Asymptotic Analysis

An algorithm for computing prefix averages

Algorithm prefixAverages1(X):

Input: An n-element array X of numbers.

Output: An n -element array A of numbers such that A[i] is the average of elements X[0], ... , X[i].

```
Let A be an array of n numbers.
for i ← 0 to n - 1 do
    a ← 0
    for j ← 0 to i do
      a ← a + X[j]
    A[i] ← a/(i+ 1)
return array A
```

1 step

i iterations with i=0,1,2...n-1

n iterations

Algorithm prefixAverages1(X) "runs in $O(n)$ time"

Asymptotic Notation(Terminology)

1. Special classes of algorithms:

   logarithmic: $O(logn)$
   linear: $O(n)$
   quadratic: $O(n^2)$
   polynomial: $O(n^k), k \geq 1$
   exponential: $O(a^n), n > 1$

2. "Relatives" of the Big-Oh
   1. $\Omega(f(n))$: Big Omega–asymptotic lower bound
   2. $\Theta(f(n))$: Big Theta–asymptotic tight bound

Asymptotic Analysis of The Running Time

1. Use the Big-Oh notation to express the number of primitive operations executed as a function of the input size.
2. For example, we say that the **prefixAverages** algorithm runs in $O(n)$ time.
3. Comparing the asymptotic running time
   1. an algorithm that runs in $O(n)$ time is better than one that runs in $O(n^2)$ time.
   2. similarly, $O(logn)$ is better than $O(n)$.
   3. hierarchy of functions: $logn \leq n \leq n^2 \leq n^3 \leq 2n$.

Caution! Beware of very large constant factors.

An algorithm running in time $1,000,000$ n is still $O(n)$ but might be less efficient on your data set than one running in time $2n^2$, which is $O\left(n^2\right)$.

Outline

Algorithm

- An algorithm is a process to solve a problem manually in sequence with finite number of steps.
- It is a set of rules that must be followed when solving a specific problem.
- It is a well-defined computational procedure which takes some value or set of values as input and generates some set of values as output.
- So,an algorithm is defined as a finite sequence of computational steps, that transforms to given input into the output for a given problem.

- An algorithm is considered to be correct, if for every input instance, it generates the correct output and gets terminated.
- So, a correct algorithm solves a given computational problem and gives the desired output.
- The main objectives of algorithm are
  - To solve a problem manually in sequence with finite number of steps.
  - For designing an algorithm, we need to construct an efficient solution for a problem.

Algorithm Paradigm

It includes four steps.That is:

1. Design of Algorithm
2. Algorithm validation
3. Analysis of algorithms
4. Algorithm testing

1. Design of algorithm:

- Various designing techniques are available which yield good and useful algorithm.
- These techniques are not only applicable to only computer science, but also to other areas, such as operation research and electrical engineering.
- The techniques are: divide and conquer, incremental approach, dynamic programming...etc. By studying this we can formulate good algorithm.

2. Algorithm validation:

- Algorithm validation checks the algorithm result for all legal set of input.
- After designing, it is necessary to check the algorithm, whether it computes the correct and desired result or not for all possible legal set of input.
- Here the algorithm is not converted into the program.But after showing the validity of the method, a program is written.This is known as ”program providing” or ” program verification”.
- Here we check the program output for all possible set of input.
- It requires that, each statement should be precisely defined an all basic operations can be correctly provided.

3. Analysis of algorithms:

- The analysis of algorithm focuses on time complexity or space complexity.
- The amount of memory needed by program to run to completion is referred to as space complexity.
- The amount of time needed by an algorithm to run to completion is referred to as time complexity.
- For an algorithm time complexity depends upon the size of the input, thus is a function of input size 'n'.

- Usually, we deal with the best case time, average case time and worst case time for an algorithm.
- The minimum amount of time that an algorithm requires for an input size 'n', is referred to as Best Case Time Complexity.
- Average Case Time Complexity is the execution of an algorithm having typical input data of size 'n'.
- The maximum amount of time needed by an algorithm for an input size 'n' is referred to as Worst Case Time Complexity.

4. Algorithm testing:

- This phase involves testing of a program.It consists of two phases.That is: Debugging and Performance Measurement.
- Debugging is the process of finding and correcting the cause at variance with the desired and observed behaviors.
- Debugging can only point to the presence of errors, but not their absence.
- The performance measurement or profiling precise by described the correct program execution for all possible data sets and it takes time and space to compute results.

NOTES

- While designing and analyzing an algorithm, two fundamental issue to be considered.That is:
  1. Correctness of the algorithm
  2. Efficiency of the algorithm
- While designing the algorithm, it should be clear, simple and should be unambiguous.
- The characteristics of algorithm is: finiteness, definiteness, efficiency, input and output.

Outline

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.

```
// 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 previous 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 $\theta(n)$.

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 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.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

$$= \Theta(n)$$

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 $\theta(1)$

- Most of the times, we do **worst case analysis** to analyze algorithms. In the worst case 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 the mathematical distribution of all possible inputs.
- The **best case analysis** is bogus. Guaranteeing a lower bound on an algorithm doesnt provide any information as in the worst case, an algorithm may take years to run.