Advanced Algorithms

Dr. Malek Mouhoub

Mathematics and Computer Science Department

University of Lethbridge

Spring 2001

2. Mathematical Fundations and Algorithm Analysis

- Mathematics Review
- Asymptotic notation and Growth of functions
- Algorithm analysis

Mathematical Review

Exponents

$$
\begin{aligned}
X^A X^B &= X^{A+B} \\
\frac{X^A}{X^B} &= X^{A-B} \\
(X^A)^B &= X^{AB} \\
X^N + X^N &= 2X^N \neq X^{2N} \\
2^N + 2^N &= 2^{N+1}
\end{aligned}
$$

Mathematical Review

Logarithms

- By default, logarithms used in this course are to the base 2.
- $X^A = B \Leftrightarrow \log_X B = A$
- $\log_A B = \frac{\log_C B}{\log_C A},\ A, B, C > 0,\ A \neq 1$
- $\log AB = \log A + \log B;\ A, B > 0$
- $\log A/B = \log A - \log B$
- $\log (A^B) = B \log A$
- $\log X < X\ \forall X > 0$
- $\log 1 = 0,\ \log 2 = 1, \log 1,024 = \quad , \log 1,048,576 =$

Mathematics Review

Summations

- $\sum_{i=1}^{N} a_i = a_1 + a_2 + \cdots + a_N$
- $lim_{N \to \infty} \sum_{i=1}^{N} a_i = \sum_{i=1}^{\infty} a_i = a_1 + a_2 + \cdots$ (infinite sum)
- **Linearity**
  - $\sum_{i=1}^{N}(ca_i + db_i) = c\sum_{i=1}^{N} a_i + d\sum_{i=1}^{N} b_i$
  - $\sum_{i=1}^{N} \Theta(f(i)) = \Theta(\sum_{i=1}^{N} f(i))$
- **General algebraic manipulations**:
  - $\sum_{i=1}^{N} f(N) = Nf(N)$
  - $\sum_{i=n_0}^{N} f(i) = \sum_{i=1}^{N} f(i) - \sum_{i=1}^{n_0-1} f(i)$

Mathematics Review

Summations

- **Geometric series** :
  - $\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$
  - if $0 < A < 1$ then $\sum_{i=0}^{N} A^i \leq \frac{1}{1-A}$
- **Arithmetic series** :
  - $\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = \frac{N^2+N}{2} \approx \frac{N^2}{2}$
  - $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$
  - $\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|} \quad k \neq -1$
    - if $k = -1$ then $H_N = \sum_{i=1}^{N} \frac{1}{i} \approx \log_e N$
    - error in approx : $\gamma \approx 0,57721566$

Mathematics Review

Products

- $\prod_{i=1}^{N} a_i = a_1 \times a_2 \times \cdots \times a_N$
- $\log(\prod_{i=1}^{N} a_i) = \sum_{i=1}^{N} \log a_i$

Mathematics Review

Proving statements

- **Proof by induction**

  1. Proving a base case : establishing that a theorem is true for some small values.
  2. Inductive hypothesis : the thoerem is assumed to be true for all cases up to some limit k.
  3. Given this assumption, show that the thoerem is true for $k + 1$

- **Proof by Counterexample** : find an example showing that the thoerem is not true.

- **Proof by Contradiction** : Assuming that the thoerem is false and showing that this assumption implies that some known property is false, and hence the original assumption was erroneous.

Asymptotic notation

- Running time of an algorithm almost always depends on the amount of input : more inputs means more time. Thus the running time T, is a function of the amount of input, N, or $T(N) = f(N)$ where N is in general a natural number.
- The exact value of the function depends on :
  - the speed of the machine;
  - the quality of the compiler and optimizer;
  - the quality of the program that implements the algorithm;
  - the basic fondamentals of the algorithm
- Typically, the last item is most important.

Worst-case versus Average-case

- Worst-case running time is a bound over all inputs of a certain size N. (Guarantee)
- Average-case running time is an average over all inputs of a certain size N. (Prediction)

Θ-notation

For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \exists\, c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}$$

We say that $g(n)$ is an *asymptotically tight bound* for $f(n)$.

Example : The running time of insertion sort is $T(n) = \Theta(n^2)$.

Ω-notation

For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions :

$$\Omega(g(n)) \;=\; \{f(n) \;:\; \exists\, c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

We say that $g(n)$ is an *asymptotic lower bound* for $f(n)$.

Big-Oh notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions :

$$O(g(n)) = \{f(n) \; : \; \exists \, c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

We say that $g(n)$ is an *asymptotic upper bound* for $f(n)$.

Note that O-notation is, in general, used informally to describe asymptotically tight bounds (Θ-notation).

Big-Oh notation

- Exponential : dominant term is some constant times 2^N.
- Cubic : dominant term is some constant times N^3. We say $O(N^3)$.
- Quadratic : dominant term is some constant times N^2. We say $O(N^2)$.
- $O(N \log N)$: dominant term is some constant times $N \log N$.
- Linear : dominant term is some constant times N. We say $O(N)$.
- Logarithmic : dominant term is some constant times $\log N$.
- Constant : c.

Note : Big-Oh ignores leading constants.

Dominant Term Matters

- Suppose we estimate $35N^2 + N + N^3$.
- For N=10000 :
  - Actual value is 1,003,500,010,000
  - Estimate is 1,000,000,000,000
  - Error in estimate is 0,35%, which is negligible.
- For large N, dominant term is usually indicative of algorithm's behavior.
- For small N, dominant term is not necessarily indicative of behavior, **BUT**, typically programs on small inputs run so fast we don't care anyway.

Example 1 : Computing the Minimum

- **Minimum item in an array**
  - Given an array of N items, find the smallest.
- Obvious algorithm is sequential scan.
- Running time is $O(N)$ (linear) because we repeat a fixed amount of work for each element in the array.
- A linear algorithm is a good as we can hope for because we have to examine every element in the array, a process that requires linear time.

Example 2 : Closest Points

- **Closest Points in the Plane**
  - Given N points in a plane (that is, an x-y coordinate system, find the pair of points that are closest together).
- Fundamental problem in graphics.
- Solution : Calculate the distance between each pair of points, and retain the minimum distance.
- $N(N-1)/2$ pairs of points, so the algorithm is quadratic.
- Better algorithms that use more subtle observations are known.

Example 3 : Co-linear Points in the Plane

- **Co-linear points in the plane**
  - Given N points in the plane, determine if any three form a straight line.
- Important in graphics : co-linear points introduce nasty degenerate cases that require special handling.
- Solution : enumerate all groups of three points; for each possible triplet of three points check if the points are co-linear. This is a cubic algorithm.

Case Study

- Examine a problem with several different solutions.
  - Will look at four algorithms
  - Some algorithms much easier to code than others.
  - Some algorithms much easier to prove correct than others.
  - Some algorithms much, much faster (or slower) than others.

The problem

- **Maximum Contiguous Subsequence Sum Problem**
  - Given (possibly negative integers) $A_1, A_2, \ldots, A_N$ find (and identify the sequence corresponding to) the maximum value of $(A_i + A_{i+1} + \cdots + A_j)$.
- The maximum contiguous subsequence sum is zero if all the integers are negative.
- Examples :
  - -2, **11, -4, 13**, -4, 2
  - 1, -3, **4, -2, -1, 6**

Brute Force 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;
}
```

Analysis

- Loop of size N inside of loop of size N inside of loop of size N means $O(N^3)$, or cubic algorithm.
- Slight over-estimate that results from some loops being of size less than N is not important.

Actual Running time

- For $N = 100$, actual time is 0,47 seconds on a particular computer.
- Can use this to estimate time for larger inputs :

$$T(N) = cN^3$$

$$T(10N) = c(10N)^3 = 1000cN^3 = 1000T(N)$$

- Inputs size increases by a factor of 10 means that running time increases by a factor of 1,000.
- For N=1000, estimate an actual time of 470 seconds. (Actual was 449 seconds).
- For N=10,000, estimate 449000 seconds (6 days).

How to improve

- Remove a loop; not always possible.
- Here it is : innermost loop is unnecessary because it throws away information.
- `ThisSum` for next `j` is easily obtained from old value of `ThisSum`:
  - Need $A_i + A_{i+1} + \cdots + A_{j-1} + A_j$
  - Just computed $A_i + A_{i+1} + \cdots + A_{j-1}$
  - What we need is what we just computed $+A_j$.

The Better 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;
}
```

Analysis

- Same logic as before : now the running time is quadratic, or $O(N^2)$.
- As we will see, this algorithm is still usable for inputs in the tens of thousands.
- Recall that the cubic algorithm was not practical for this amount of input.

Actual running time

- For $N = 100$, actual time is 0.0111 seconds on the same particular computer.
- Can use this to estimate time for larger inputs :

$$T(N) = cN^2$$

$$T(10N) = c(10N)^2 = 100cN^2 = 100T(N)$$

- Inputs size increases by a factor of 10 means that running time increases by a factor of 100.
- for $N = 1000$, estimate a running time of 1.11 seconds. (Actual was 1.12 seconds).
- For $N = 10,000$, estimate 111 seconds (=actual).

Linear Algorithms

- Linear algorithm would be best.
- Running time is proportional to amount of input. Hard to do better for an algorithm.
- If inputs increases by a factor of ten, then so does running time.

Recursive algorithm

- Use a divide-and-conquer approach.
- The maximum subsequence either
  - lies entirely in the first half
  - lies entirely in the second half
  - starts somewhere in the first half, goes to the last element in the first half, continues at the first element in the second half, ends somewhere in the second half.
- Compute all three possibilities, and use the maximum.
- First two possibilities easily computed recursively.

Computing the third case

- **Idea :**
  1. Find the largest sum in the first half, that includes the last element in the first half.
  2. Find the largest sum in the second half that includes the first element in the second half.
  3. Add the 2 sums together.
- **Implementation :**
  - Easily done with two loops.
  - For maximum sum that starts in the first half and extends to the last element in the first half, use a right-to-left scan starting at the last element in the first half.
  - For the other maximum sum, do a left-to-right scan, starting at the first half.

Analysis

- Let $T(N) =$ the time for an algorithm to solve a problem of size N.
- Then $T(1) = 1$ (1 will be the quantum time unit; constants don't matter).
- $T(N) = 2T(N/2) + N$
  - Two recursive calls, each of size $N/2$. The time to solve each recursive call is $T(N/2)$ by the above definition.
  - Case three takes $O(N)$ time; we use N, because we will throw out the constants eventually.

Bottom Line

$$
\begin{aligned}
T(1) &= 1 = 1 * 1 \\
T(2) &= 2 * T(1) + 2 = 4 = 2 * 2 = 2^1 * 2 \\
T(4) &= 2 * T(2) + 4 = 12 = 4 * 3 = 2^2 * 2 \\
T(8) &= 2 * T(4) + 8 = 32 = 8 * 4 = 2^3 * 4 \\
T(16) &= 2 * T(8) + 16 = 80 = 16 * 5 = 2^4 * 5 \\
T(32) &= 2 * T(16) + 32 = 192 = 32 * 6 = 2^5 * 6 \\
T(64) &= 2 * T(32) + 64 = 448 = 64 * 7 = 2^6 * 7 \\
T(N) &= 2^k * (k + 1) = N(1 + \log N) = O(N \log N)
\end{aligned}
$$

$\boxed{N \log N}$

- Any recursive algorithm that solves two half-sized problems and does linear non-recursive work to combine/split these solutions will always take $O(N \log N)$ time because the above analysis will always hold.

- This is a very significant improvement over quadratic.

- It is still not as good as $O(N)$, but is not that far away either. There is a linear-time algorithm for this problem. The running time is clear, but the correctness is non-trivial.

The Linear-time algorithm

```
/** Linear-time maximum contiguous subsequence.
     sum algorithm    */
        int maxSubSum4( const vector<int> & a )
        {
/* 1*/       int maxSum = 0, thisSum = 0;
/* 2*/       for( int j = 0; j < a.size( ); j++ )
             {
/* 3*/           thisSum += a[ j ];
/* 4*/           if( thisSum > maxSum )
/* 5*/               maxSum = thisSum;
/* 6*/           else if( thisSum < 0 )
/* 7*/               thisSum = 0;}
/* 8*/       return maxSum;}
```

The Logarithm

- Formal Definition
  - For any $B,\ N > 0,\ \log_B N = K$ if $B^K = N$
  - If the base B is omitted, it defaults to 2 in computer science.
- Examples :
  - $\log 32 = 5$ (because $2^5 = 32$)
  - $\log 1024 = 10$
  - $\log 1048576 = 20$
  - $\log 1 billion$ = about 30
- The logarithm grows much more slowly than N, and slower than the square of N.

Examples of the Logarithm

Bits in a binary number : how many bits are required to represent N consecutive integers ?

Repeated doubling : starting from $X = 1$, how many times should X be doubled before it is at least as large as $\dot{N}$?

Repeated halving : Starting from $X = N$, if N is repeatedly halved, how many iterations must be applied to make N smaller than or equal to 1 ? (Halving rounds up).

Answer to all of the above is $\log N$ (rounded up).

Why $\log N$

- B bits represents 2^B integers. Thus 2^B is at least as big as N, so B is at least $\log N$. Since B must be an integer, round up if needed.
- Same logic for the other examples.

Repeated Halving Principle

- An algorithm is $O(\log N)$ if it takes constant time to reduce the problem size by a constant fraction (which is usually 1/2).
- Reason : there will be $\log N$ iterations of constant work.

Static Searching

- Given an integer X and an array A, return the position of X in A or an indicator that it is not present. If X occurs more than once, return any occurrence. The array A is not altered.
- If input array is not sorted, solution is to use a sequential search. Running times :
  - Unsuccessful search : $O(N)$; every item is examined.
  - Successful search :
    - Worst case : $O(N)$; every item is examined.
    - Average case :$O(N/2)$; half the items are examined.
- Can we do better if we know the array is sorted ?

Binary Search

- Yes ! use a binary search.
- Look in the middle :

  Case 1: If X is less than the item in the middle, then look in the subarrayto the left of the middle.

  Case 2: If X is greater than the item in the middle, then look in the subarray to the right of the middle.

  Case 3 : If X is equal to the item in the middle, then we have a match.

  Base Case : If the subarray is empty, X is not found.

- This is logarithmic by the repeated halving principle.

Binary Search Continued

- Binary search is an example of a data structure implementation :
  - **Insert** : $O(N)$ time per operation, because we must insert and maintain the array in sorted order.
  - **Delete** : $O(N)$ time per operation, because we must slide elements that are to the right of the deleted element over one spot to maintain contiguity.
  - **Find** : $O(\log N)$ time per operation, via binary search.
- In this course we examine different data structures. Generally we allow Insert, Delete, and Find, but Find and Delete are usually restricted.