CSCI 335
Software Design and Analysis III
Lecture 5: Analysis of Algorithms

Professor Anita Raja

1

1

Agenda

- HW1 discussion
- Review: Recursion
- Fibonnaci
- Maximum Subsequence Problem
- Logarithmic Complexity
  - Binary Search
  - Exponentiation

2

2

Review: Model of computation

- algorithm analysis
- average machine; instructions are executed sequentially.
- one unit of time
- fixed-size (32-bit) integers.
- infinite memory

3

What to analyze

- Running time (Time complexity)
- Memory usage (Space complexity)
- What kind of input?
- Size of the input problem, N.

4

Useful Rules for Big-O

- When adding algorithmic complexities, the larger value dominates.

For any **polynomial** $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$, where $a_0, a_1, \dots, a_n$ are real numbers, $f(x)$ is $O(x^n)$.

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$

- If $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $O(g(x))$, then $(f_1 + f_2)(x)$ is $O(g(x))$.

5

Review: Useful Rules for Big-O

- **Loops**:
  - Running time of a for loop is at most the running time of the statements inside the for loop times the number of iterations.
- **Nested loops:**
  - Analyze inside out. Running time of statement multiplied by the product of the sizes of the loops.
- **Consecutive statements:**
  - Just add.
- **If/Else:**
  - Never more than the running time of the test plus the larger of the running times of S1 and S2.

6

Review: Recursion

- Factorial:

```
long Factorial(int n) {
  if (n <= 1)
     return 1;
  else
     return n * Factorial(n – 1);
}
```

- Running time?

7

7

Recursion

- Factorial:

```
long Factorial(int n) {
  if (n <= 1)                          1
     return 1;                         1
  else
     return n * Factorial(n – 1);    1 + ?
}
```

- Running time?
  - Factorial is a thinly veiled loop O(n)

8

8

Recursion

- Factorial running time:

T(n) = 1 + T(n-1) for n >1, T(1)=2

T(n) = 1 + T(n-1)
= 1 + (1 + T(n-2)) = … =
= 1+(1(+…(1+T(n-k))..) (k 1's)
=> T(n) = k + T(n-k)
= (n-1) + T(n-(n-1))
= (n-1) + T(1)
= (n-1)+2
= n+1
=> T(n) = n + 1 => T(n) = O(n).
Linear algorithm.

9

9

Recursion

- Fibonacci (bad example):

```
// @n: an integer.
// @return the Fibonacci number of n.
long Fibonacci(int n) {
    if (n <= 1)
       return 1;
   else
      return Fibonacci(n – 1) + Fibonacci(n – 2);
}
```

- Running time?

10

10

Recursion

- Fibonacci (bad example):
- Running time: T(n) = T(n-1) + T(n-2) + 2, T(0)=T(1)=1

- Since fib(n) = fib(n-1)+fib(n-2), we can prove (by induction) that T(n) >= fib(n)
- Section 1.2.5 proves that $T(n) < (5/3)^n$

$=> T(n) =? \ ((5/3)^n) \quad O/\Theta/\Omega/o$

Exponential: Really bad result! (Use iteration instead)

11

11

Fibonacci

N	T(n)	1.5^n
0	1	1.00
1	1	1.50
2	4	2.25
3	7	3.38
4	13	5.06

…

We can see that T(n) > 1.5^n for small values of n

12

12

Maximum Subsequence Problem

- Given (possibly negative) integers $A_1, \ldots, A_N$, find the maximum value of $\sum_{k=i}^{j} A_j$ for $i, j = 1,2, \ldots, N, i \leq j$

- Given a sequence of integers (possibly negative), find the subsequence whose sum is the maximum

-2	11	-4	13	-5	-2

- Many algorithms to solve this simple problem
- When input size is small, brute force works fine.

13

13

Algorithm 1 (brute force)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Brute force approach.
int MaxSubsequenceSum1(const vector<int> &a) {
    int max_sum = 0;
    for (size_t i = 0; i < a.size(); ++i)
      for (size_t j = i; j < a.size(); ++j) {
            int current_sum = 0;
            for (size_t k = i; k <= j; ++k)
                current_sum += a[k];
            if (current_sum > max_sum)
                max_sum = current_sum;
      }
    return max_sum;
}
```

14

14

Algorithm 1 (brute force)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Brute force approach.
int MaxSubsequenceSum1(const vector<int> &a) {
    int max_sum = 0;
    for (size_t i = 0; i < a.size(); ++i)
      for (size_t j = i; j < a.size(); ++j) {
            int current_sum = 0;
            for (size_t k = i; k <= j; ++k)
                current_sum += a[k];
            if (current_sum > max_sum)
                max_sum = current_sum;
      }
    return max_sum;
}
```

- Check all possible subsequences
- $O(N^3)$. Redundant work.
- For N=1,000,000 this is 10^{18}, about 58 days at 2Ghz (assuming 2billion operations per second)

15

15

Algorithm 2 (brute force)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Brute force approach (slightly better).
int MaxSubsequenceSum2(const vector<int> &a) {
    int max_sum = 0;
    for (size_t i = 0; i < a.size(); ++i) {
      int current_sum = 0;
      for (size_t j = i; j < a.size(); ++j) {
          current_sum += a[j];
          if (current_sum > max_sum)
             max_sum = current_sum;
      }
    }
    return max_sum;
}
```

16

16

Algorithm 2 (brute force)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Brute force approach (slightly better).
int MaxSubsequenceSum2(const vector<int> &a) {
    int max_sum = 0;
    for (size_t i = 0; i < a.size(); ++i) {
      int current_sum = 0;
      for (size_t j = i; j < a.size(); ++j) {
          current_sum += a[j];
          if (current_sum > max_sum)
             max_sum = current_sum;
      }
    }
    return max_sum;
}
```

- Removed one loop
- $O(N^2)$.
- About 8 seconds at 2Ghz

17

17

Algorithm 3 (Recursive)

- The maximum subsequence is either entirely in the first half, entirely in the second half, or it crosses the middle and is in both halves
- Example:

First half				Second half			
4	-3	5	-2	-1	2	6	-2

- Best first half: ?
- Best second half: ?
- Best last part of first half: ?
- Best first part of second half: ?
- Result: ?

18

```
// @a: a vector of integers.
// @left: a left index.
// @right: a right index.
// It is assumed that left <= right. It is also assumed that right is not
//  outside of the boundaries of the vector.
// @return the maximum positive subsequence sum of items
//     a[left], a[left + 1], … , a[right].
// Recursive solution.
int MaxSubsequenceSum3(const vector<int> &a, size_t left, size_t right) {
    if (left > right) abort();  // Invalid.
    if (right >= a.size()) abort();  // Invalid.

    if (left == right)  // Base case.
       return a[left] > 0 ? a[left]: 0;

    const size_t center = (left + right) / 2;
    const int max_left_sum = MaxSubsequenceSum3(a, left, center);
    const int max_right_sum = MaxSubsequenceSum3(a, center + 1, right);

    // Compute maximum of left part when you always end at center.
    int max_left_border_sum = 0, left_border_sum = 0;
    for (int i = center; i >= left; --i) {
       left_border_sum += a[i];
       if (left_border_sum > max_left_border_sum)
            max_left_border_sum = left_border_sum;
    }
    // continues in next page…
```

19

```
// Compute maximum of right part when you always start at center + 1.
 int max_right_border_sum = 0, right_border_sum = 0;
 for (int i = center + 1; i <= right; ++i) {
       right_border_sum += a[i];
       if (right_border_sum > max_right_border_sum)
           max_right_border_sum = right_border_sum;
    }

 return MaxOfThree(max_left_sum, max_right_sum,
                   max_right_border_sum + max_left_border_sum);
}  // End of MaxSubsequenceSum3.

// Driver routine, that calls the recursive one.
int MaxSubsequenceSum3Driver(const vector<int> &a) {
  return MaxSubsequenceSum3(a, 0, a.size() – 1);
}
```

20

Algorithm 3 Running time analysis

- $T(1) = 1$ (base case)
- $T(N) = 2\,T(N/2) + O(N)$ (why ?)
- Simpler (note that we always ignore constants)
- $T(N) = 2\,T(N/2) + N$ (assume N = 2^k => k= logN)
- T(N) = N*(k+1)
  = NlogN + N
  = O(NlogN)

21

Running time analysis (details cont.)

T(N) = 2 T(N/2) + N (assume N = 2^k)
T(N/2) = 2 T(N/4) + N/2
…
T(N/2^i) = 2 T(N/2^(i+1)) + N/2^i
…
T(N/2^(k-1)) = 2 T(N/2^k) + N/2^k-1
T(2) = 2 T(1) + 2

2^(k) = N => k = log(N)

So T(N) = 2 * (2*T(N/4) + N/2) + N = 2^2T(N/4)+N + N=
2^2 (2*T(N/8)+N/4) + N + N = 2^3 T(N/8)+N+N+N= 2^3T(N/2^3)+3N …..
T(N) = 2^kT(N/2^k)+kN, but 2^k=N and k=log(N)
Therefore
T(N) = N*T(1)+log(N)*N = N + N*log(N) =**O (N*logN)**
(0.003 secs on earlier example)

22

Algorithm 4 (Linear Version)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Linear version.
int MaxSubsequenceSum4(const vector<int> &a) {
   int max_sum = 0, current_sum = 0;

   for (size_t i = 0; i < a.size(); ++i) {
     current_sum += a[i];

     if (current_sum > max_sum)
      max_sum = current_sum;
     else if (current_sum < 0)
      current_sum = 0;
   }

   return max_sum;
}
// Run on example: -2, 10, 3, -4, -8, -2, 10, 2, -5
```

23

Algorithm 4 (Linear Version)

- Why does it work?
- Running time is obvious, but correctness is not.
- 0.0005 secs on the earlier example

24

Algorithm 4 (example)

- Consider this sequence:

-2, 10, 3, -4, -8, -2, 10, 2, -5, …

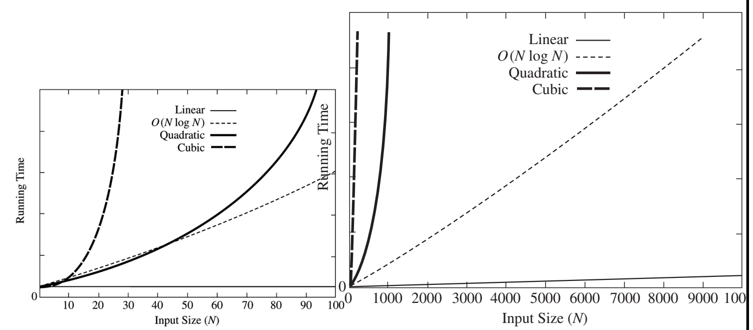
Append 3, 20 to the above sequence and continue…

- Algorithm 4 is an example of an **on-line algorithm**
  - Data can be read sequentially with no need to store it in main memory (i.e. you could apply the algorithm to data from a disk, or as it arrives from the internet, or through a sensor)
  - At any point in time the algorithm provides the current best solution to the problem.

25

25

Four algorithms (graphs)



26

26

Four different algorithms

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

Note, that time required to read input is included in the above analysis.

27

27

Logarithms in running time

- divide and conquer algorithms
- general rule:
  - An algorithm is O(logN) if it takes constant (O(1)) time to cut the problem size by a fraction (which is usually half).
  - On the other hand, if constant time is required to merely reduce the problem by a constant amount (such as to make the problem smaller by 1), then the problem is O(N)

28

28

Binary Search

- Given an integer X and integers $A_0, \ldots, A_{N-1}$, which are **presorted** and already in memory, find i such that $A_i = X$, or return $i = -1$ if X is not in the input
- Running time?

29

```
// Performs the standard binary search using two comparisons per level.
// @a: input vector of elements. Assumes sorted vector.
// @x: item we are searching for.
// @return index where item is found or -1 if not found.
template <typename Comparable>
int BinarySearch(const vector<Comparable> &a, const Comparable &x) {
    int low  = 0;
    int high = a.size() - 1;
    while (low <= high) {
      const int mid = (low + high) / 2;
      if (a[mid] == x) return mid;  // Found.
      if (a[mid] < x)
            low = mid + 1;
      else
          high = mid - 1;
    }
    return -1;
}
```

30

Binary Search Running Time

- Running time?
  - $T(N) = 1 + T(N/2), T(1) = 1$
- -----------------------------------

$$
\begin{aligned}
T(N) &= 1 + T\left(\frac{N}{2}\right) \\
&= 1 + 1 + T\left(\frac{N}{2^2}\right) \\
&= 2 + T\left(\frac{N}{2^2}\right) \\
&= 2 + 1 + T\left(\frac{N}{2^3}\right) \\
&= 3 + T\left(\frac{N}{2^3}\right)
\end{aligned}
$$

$$= \cdots = \boldsymbol{k + T(N/2^k)} = \ldots$$

If N is a power of 2 (i.e. $N = 2^k$ with $k = \log(N)$) we will have:

$$T(N) = \boldsymbol{k + T\left(\frac{N}{2^k}\right)} = k + T(1) = \log(N) + 1$$

$$=> \boldsymbol{T(N) = O(logN)}$$

31

Exponentiation

- Compute X^N, for positive N
- Naïve algorithm: (N-1) multiplications
- Smarter algorithm ?
- How?

32

Fast Exponentiation

$$X^0 = 1$$
$$X^1 = X$$
$$X^N = \left(X^{N/2}\right)^2, \text{if } N \text{ is even}$$
$$X^N = X\left(X^{(N-1)/2}\right)^2, \text{if } N \text{ is odd}$$

$$T(N) = \begin{cases} T\left(\frac{N}{2}\right) + 1, & \text{even } N \\ T\left(\frac{N-1}{2}\right) + 2, & \text{odd } N \end{cases}$$

33

```
// @x: a number.
// @n: a positive integer.
// @return x^n.
long Power(long x, unsigned int n) {
    if (n == 0)
         return 1;

    if (n == 1)
         return x;

    if (n % 2 == 0) // even.
       return Power(x * x, n / 2);

    else // odd.
       return x * Power(x * x, n / 2);
}
```

34

Exponentiation

```
// Can we replace recursive call
//      Power(x * x, n / 2)
// by one of the following?
return Power(Power(x, 2), n / 2);
return Power(Power(x, n / 2), 2);
return Power(x, n / 2) * Power(x, n / 2);
```

35

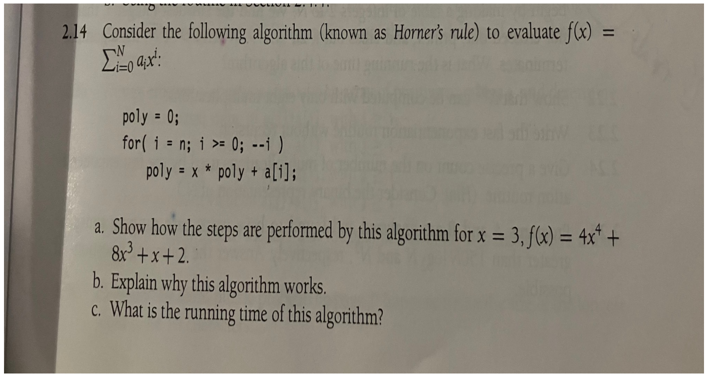
Exponentiation

```
// Endless loop ?
return Power(Power(x, 2), n /2);
return Power(Power(x, n / 2), 2);

// Non-logarithmic runtime. Why?
Return Power(x, n / 2) * Power(x, n / 2);
```

36

Exercise 2.14



2.14 Consider the following algorithm (known as *Horner's rule*) to evaluate $f(x) = \sum_{i=0}^{N} a_i x^i$:

```
poly = 0;
for( i = n; i >= 0; --i )
    poly = x * poly + a[i];
```

a. Show how the steps are performed by this algorithm for $x = 3$, $f(x) = 4x^4 + 8x^3 + x + 2$.

b. Explain why this algorithm works.

c. What is the running time of this algorithm?

37

37

Homework Exercises

- Order the following functions by growth rate:

N, $N^{1/2}$, $N^{1.5}$, N^2, $N \log N$, $N \log \log N$, $N \log^2 N$, $N \log(N^2)$, $2/N$, 2^N, $2^{N/2}$, 37, $N^2 \log N$, N^3

- Find 2 functions f(N) and g(N) such that neither f(N) = O((g(N)) nor g(N)= O(f(N)

38

38