Algorithm Complexity Analysis: Big-O Notation (Chapter 10.4)

Dr. Yingwu Zhu

Measure Algorithm Efficiency

- Space utilization: amount of memory required
- Time efficiency: amount of time required to accomplish the task
- As space is not a problem nowadays
  - Time efficiency is more emphasized
  - But, in embedded computers or sensor nodes, space efficiency is still important

Time Efficiency

- Time efficiency depends on :
  - size of input
  - speed of machine
  - quality of source code
  - quality of compiler

These vary from one platform to another

So, we cannot express time efficiency meaningfully in real time units such as seconds!

Time Efficiency

- Time efficiency = the number of times instructions executed
  - A measurement of efficiency of an algorithm

- Measure computing time T(n) as
  - T(n) = number of times the instructions executed
  - T(n): computing time of an algorithm for input of size n

Example: calculating a mean

Task	# times executed
1. Initialize the *sum* to 0	1
2. Initialize index *i* to 0	1
3. While $i < n$ do following	n+1
4. a) Add x[i] to sum	n
5. b) Increment *i* by 1	n
6. Return *mean = sum/n*	1
Total	3n + 4

T(n): Order of Magnitude

- As number of inputs increases
  - T(n) = 3n + 4 grows at a rate proportional to *n*
  - Thus T(n) has the "order of magnitude" *n*

T(n): Order of Magnitude

- T(n)*on input of size n,*
  - If **there is some constant C such that**
    $T(n) \leq C \cdot f(n)$
    for all sufficiently large values of *n*
  - Then *T(n)* said to have ***order of magnitude f(n)***,
  - Written as: **T(n) = O(f(n))**
- Big-Oh notation

Big Oh Notation

Another way of saying this:

The ***complexity*** of the algorithm is *O(f(n))*.

Example: For the Mean-Calculation Algorithm:

Mean-Calculation Algorithm

```
[1] int sum = 0;
[2] int i = 0;
[3] While(i < n) {
[4]   sum += a[i];
[5]   i++;
    }
[6] return sum/n;
```

Big Oh Notation

Example: For the Mean-Calculation Algorithm:

T(n) is **O(n)**

Note that constants and multiplicative factors are ignored.

Simple function: f(n) = n

Measure T(n)

- Not only depends on the input size, but also depends on the arrangement of the input items
  - Best case: not informative
  - Average value of T: difficult to determine
  - Worst case: is used to measure an algorithm's performance

Simplifying the complexity analysis

Q: Do we need to examine all statements?

- Identify the statement executed most often and determine its execution count
- Ignore items with lower degree
- Only the highest power of n that affects Big-O estimate
- Big-O estimate gives an approximate measure of the computing time of an algorithm for large inputs

Get a Taste

```
int search (int a[n], int x) { // pseudocode for search x
  for (int i=0; i<n; i++)
    if (a[i]==x) return i;
  return -1;
} // assuming the elements in a[] are unique
```

Complexity analysis T(n):

-- best case: ?

-- average value: ?

-- worst case: ?

Get a Taste

- Answer:
  - Best case: O(1)
  - Average case: O(n)
  - Worst case: O(n)
- How to get this?

Simple Selection Sorting Algorithm,p554

```
// Algorithm to sort x[0]…x[n-1] into ascending order
1. for (int i=0; i<n-1; i++) {
   /*On the ith pass, first find the smallest element in the sublist x[i],…,x[n-1]. */
2.      int spos = i;
3.      int smallest = x[spos];
4.      for (int j=i+1; j<n; j++) {
5.          if (x[j] < smallest) {
6.              spos = j;
7.              smallest = x[j];
            } // end if
        }  // end for
8.      x[spos] = x[i];
9.      X[i] = smallest
   }// end for
```

What's the worst case T(n)?

- $T(n) = O(n^2)$
- How do we get that?
  - Let's try!

Exercises #1

```
for (int k=0; k < n; k++)
    for (int i = 0; i < n; i++)
        m[i][j] = a[i][j] + b[i][j];
```

Exercises #2

```
for (int k=0; k < n; k++)
    for (int i = k+1; i < n; i++)
        m[i][j] = a[i][j] + b[i][j];
```

Big-Oh notation

- f(n) is usually simple:

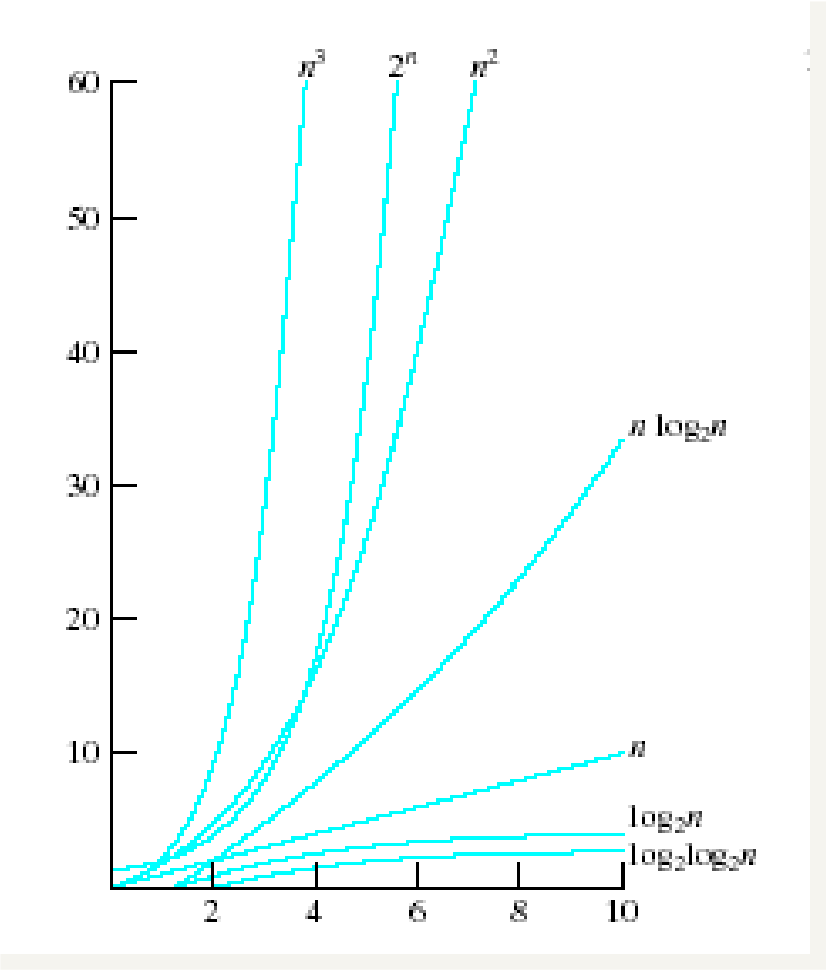
  n, n^2, n^3, ...

  2^n,

  1, $\log_2 n$

  n $\log_2 n$

  $\log_2 \log_2 n$



Exercises #3

```
n = 0;
sum = 0;
cin >> x;
while (x != -999) {
      n++;
      sum += x;
      cin >> x;
 }
mean = sum / n;
```

Binary Search Algorithm

```
int bin_search(int a[], int item) // a [0… n-1]
        bool found = false;
        int first = 0;
        int last = n-1;
        while (first <= last && !found) {
            int loc = (first + last) / 2;
            if (item < a[loc])
                last = loc – 1;
            else if (item > a[loc])
                first = loc + 1;
            else found = !found;
        } //end while
} //end bin_search
```

Binary Search Algorithm

See p556.

Step 1: identify the statement executed most often

Step 2: determine the execution count for that statement (the worst case!)

$T(n)= O(\log_2 n)$

Better approach: how binary search is performed?

How about recursive algorithms?

```
double power(double x, unsigned n)
{
    if (n==0)
        return 1.0;
    return x * power(x, n-1);
}
```

How to compute Big-Oh?

- Recurrence relation: expresses the computing time for input of size *n* in terms of smaller-sized inputs
- How to solve recurrence relations?
  - Using telescoping principle: repeatedly apply the relation until the anchor case is reached

Exercises #5

```
double fun(double x, unsigned  n) {
    if (n==0)
       return 1.0;
    return x*fun(x, n/2);
}
```

Exercises #6

```
for (int i = 0; i < n * n; ++i) {
    sum = sum/n;
    for (int j = 0; j < i; ++j)
        j >> cout;
}
```

Exercises #7

```
for (int j = 4; j < n; ++j) {
    cin >> val;
    for (int i = 0; i < j; ++i) {
        b = b * val;
        for (int k = 0; k < n; ++k)
            c = b + c;
    }
}
```

Exercises #8

```
for (int i = 1; i<n-1; i++) {
    temp = a[i];
    for (int j = i-1; j >= 0; j--)
      if (temp < a[j])
            a[j+1] = a[j];
      else
            break;
    a[j+1] = temp;
 }
```

Review of Lecture

- How to measure algorithm efficiency?
- How to compute time efficiency T(n)?
- Big-Oh notation
- For an algorithm, give the Big-Oh notation
  - Simplified analysis
  - Non-recursive
  - Recursive: telescoping principle