Intro to Algorithm Analysis

Slides

1. Table of Contents
2. Analysis Metrics
3. Exact Analysis Rules
4. Simple Summation
5. Summation Formulas
6. Order of Magnitude
7. Big-O Notation
8. Big-O Theorems
9. Complexity Classes
10. Practical Complexity Classes
11. Big-O Simple Summation
12. Big-O Analysis Rules
13. Big-O Array Summation
14. Array Summation (exact count)
15. Practical Applications
16. Hardware Speedup
17. Algorithm Behavior

Algorithm Analysis == Complexity Analysis

Analysis Metrics

Program Running (Execution) Time Factors

- Machine Speed (not just CPU speed)
- Programming Language and Implementation
- Compiler Code Generation (optimization)
- Input Data Size
- Time Complexity of Algorithm
    - Number of executed statements: **T(n)**
    - Function of the size of the input (termed **n**)

Running Time Factor Implications

- Compiler code generation & processor speed differences are too great to be used as a basis for impartial algorithm comparisons.
- Overall system load may cause inconsistent timing results, even if the same compiler and hardware are used.
- Hardware characteristics, such as the amount of physical memory and the speed of virtual memory, can dominate timing results.
- In any case, those factors are irrelevant to the complexity of the algorithm.

Exact Analysis Rules

When attempting an exact time analysis:

1. We assume an arbitrary time unit.
2. Running of each of the following operations takes time T(1):
   a) assignment operations
   b) I/O operations
   c) Boolean operations
   d) arithmetic operations
   e) function return
3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loop execution time is the time for the loop setup (initialization & setup) + the sum, (over the number of times the loop is executed), of the body time + time for the loop check and update operations. (Loop setup will include the termination check on pre-test loops.)

   †Always assume that the loop executes the maximum number of iterations possible
5. Running time of a function call is T(1) for setup + the time for any parameter calculations + the time required for the execution of the function body.

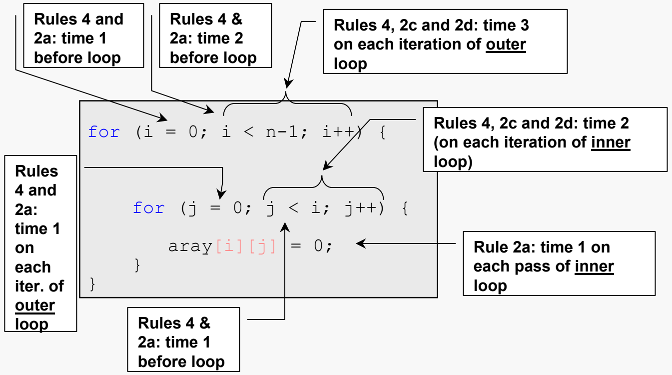
Non-executable statements, (e.g., declarations), are not counted. Only executable statements are analyzed.

Simple Summation

Given:

```
for (i = 0; i < n-1; i++) {

    for (j = 0; j < i; j++) {
       aray[i][j] = 0;
    }
}
```



Rules 4 and 2a: time 1 before loop

Rules 4 & 2a: time 2 before loop

Rules 4, 2c and 2d: time 3 on each iteration of outer loop

Rules 4, 2c and 2d: time 2 (on each iteration of inner loop)

Rules 4 and 2a: time 1 on each iter. of outer loop

Rule 2a: time 1 on each pass of inner loop

Rules 4 & 2a: time 1 before loop

```
for (i = 0; i < n-1; i++) {

    for (j = 0; j < i; j++) {

       aray[i][j] = 0;
    }
}
```

- † Outer loop will execute n -1 times -- this is the external sum.
- † Inner loop will execute i times -- this is the internal sum.

So, the total time T(n) is given by:

$$T(n) = 3 + \sum_{i=1}^{n-1}\left(5 + \sum_{j=1}^{i} 3\right)$$

For ease of summation count loop repetitions starting at 1.

Summation from applying Rule 4 to outer loop

Summation from applying Rule 4 to inner loop

Summation Formulas

Let N ≥ 0, let A, B, and C be constants, and let *f* and *g* be any functions. Then:

$$\sum_{k=1}^{N} Cf(k) = C\sum_{k=1}^{N} f(k)$$

S1: factor out constant

$$\sum_{k=1}^{N} (f(k) \pm g(k)) = \sum_{k=1}^{N} f(k) \pm \sum_{k=1}^{N} g(k)$$

S2: separate summed terms

$$\sum_{k=1}^{N} C = NC$$

S3: sum of constant

$$\sum_{k=1}^{N} k = \frac{N(N+1)}{2}$$

S4: sum of *k*

$$\sum_{k=1}^{N} k^2 = \frac{N(N+1)(2N+1)}{6}$$

S5: sum of *k* squared

The summation formulas above can be used to evaluate the expressions obtained when analyzing the complexity of an algorithm:

$$T(n) = 3 + \sum_{i=1}^{n-1}\left(5 + \sum_{j=1}^{i} 3\right)$$

From analysis on previous slide.

$$= 3 + \sum_{i=1}^{n-1}(5 + 3i)$$

Apply S3 to the inner sum.

$$= 3 + \sum_{i=1}^{n-1}(5) + 3\sum_{i=1}^{n-1}(i)$$

Apply S2 and S1 to the outer sum.

$$= 3 + 5(n-1) + 3\frac{(n-1)(n)}{2}$$

Apply S3 to the first sum, and apply S4 to the second sum.

$$= \frac{3}{2}n^2 + \frac{7}{2}n - 2$$

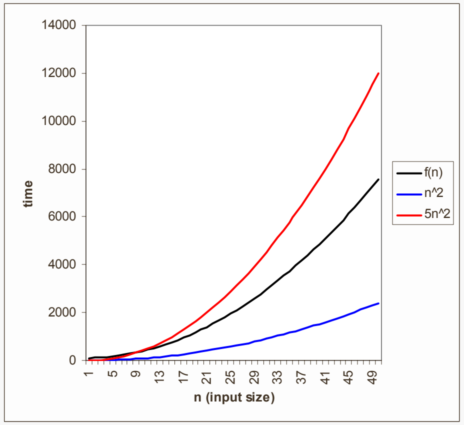
Simplify and combine terms.

Order of Magnitude

Function Estimation

Given an algorithm that takes time: $f(n) = 3n^2 + 5n + 100$

Graphically:



Algebraically:

If $n \geq 10$ then $n^2 \geq 100$
If $n \geq 5$ then $n^2 \geq 5n$
Therefore, if $n \geq 10$ then:
$f(n) = 3n^2 + 5n + 100 < 3n^2 + n^2 + n^2 = 5n^2$

So $5n^2$ forms an “upper bound” on f(n) if n is 10 or larger (asymptotic bound). In other words, f(n) doesn't grow any faster than $5n^2$ “in the long run”.

Big-O Notation

Big-O notation is used to express the asymptotic growth rate of a function.

- Formally suppose that $f(n)$ and $g(n)$ are functions of n. Then we say that $f(n)$ is in $O(g(n))$ provided that there are constants $C > 0$ and $N > 0$ such that for all $n > N$ then $f(n) \leq Cg(n)$.

- We often say that "f is big-O of g" or that "f is in O(g)".
- By the definition above, demonstrating that a function f is big-O of a function g requires that we find specific constants C and N for which the inequality holds (and show that the inequality does, in fact, hold).

Example: from the previous slide, if $n > 10$ then

$$f(n) = 3n^2 + 5n + 100 \leq 5n^2$$

C = 5
N = 10

So, by the definition above, $f(n)$ is in $O(n^2)$.

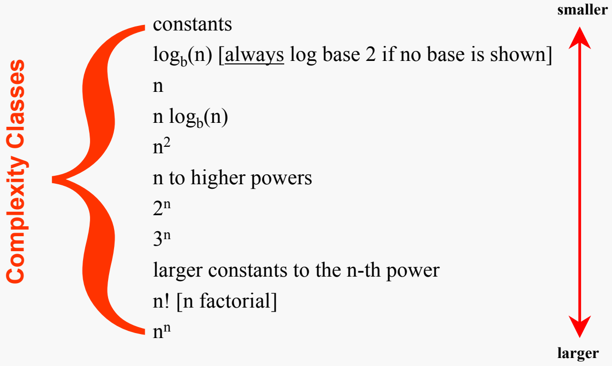
Note that $5n^2 \leq 9n^2$ (for all n), so we could also conclude that $f(n)$ is $O(9n^2)$. Usually, we're interested in the tightest (smallest) upper bound, so we'd prefer $5n^2$ to $9n^2$.

Big-O Theorems

The analysis given on slide 4 of this chapter is typical of big-O analysis, and is somewhat tricky. In order to simplify our work, we state the following theorems about big-O:

Assume that $f(n)$ is a function of n and that K is an arbitrary constant.

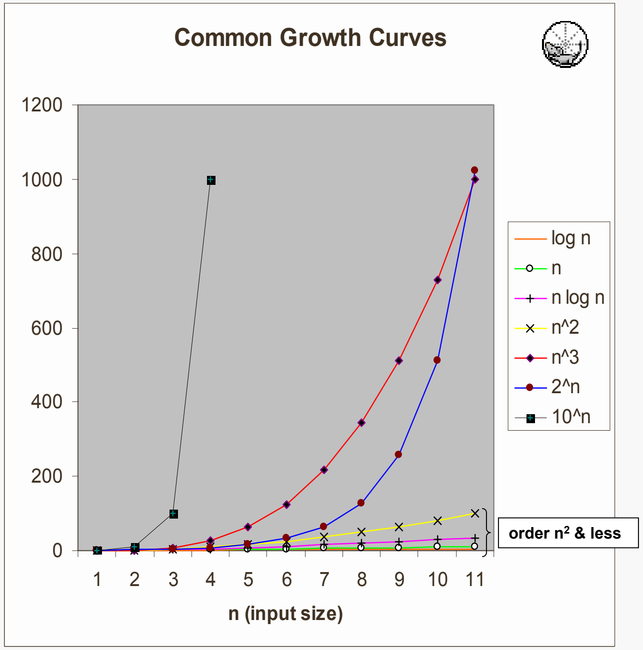
Thm 1: K is O(1)
Thm 2: A polynomial is O(the term containing the highest power of n)
Thm 3: $K*f(n)$ is $O(f(n))$ [i.e., constant coefficients can be dropped]
Thm 4: In general, $f(n)$ is big-O of the dominant term of $f(n)$, where "dominant" may usually be determined by the following list:



constants
$\log_b(n)$ [always log base 2 if no base is shown]
n
$n \log_b(n)$
n^2
n to higher powers
2^n
3^n
larger constants to the n-th power
n! [n factorial]
n^n

Thm 5: For any base b, $\log_b(n)$ is $O(\log(n))$.

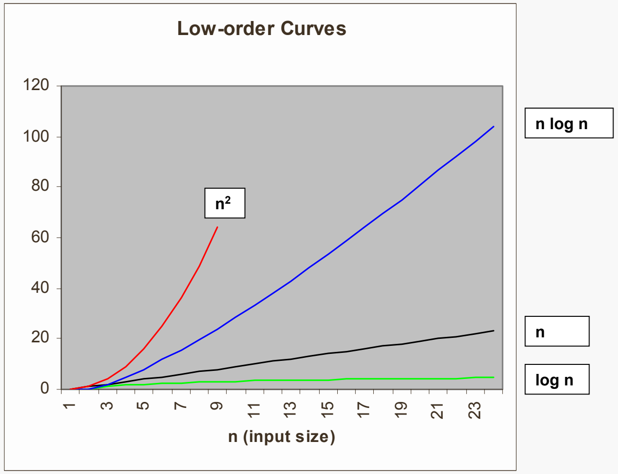
Complexity Classes



Observations

- Algorithms with Order > n^2 require FAR more time than algorithms with **O**rder n^2 or less, even for fairly small input sizes.
- For small n, there's not much practical difference between Order n^2 and order log n.

Practical Complexity Classes



Observations

- Even for moderately small input sizes, Order n2 algorithms will require FAR more time than Order n log(n) algorithms.
- **constants of proportionality**, (coefficients & lesser terms), have very little effect for large values of n (between complexity classes).
- Large problems with **O**rder > n log(n) cannot practically be executed
  - † For n = 1000 (medium problems) n^2 algorithms can still be used

Big-O Simple Summation

Recall:

```
for (i = 0; i < n-1; i++) {

    for (j = 0; j < i; j++) {
       aray[i][j] = 0;
    }
}
```

had a total time complexity T(n) is given by (slide 4):

$$T(n) = 3 + \sum_{i=1}^{n-1}\left(5 + \sum_{j=1}^{i} 3\right)$$

…and that T(n) reduced to (slide 5): $T(n) = \frac{3}{2}n^2 + \frac{7}{2}n - 3$

Now by Theorem 2: $T(n) \in O\left(\frac{3}{2}n^2\right)$

…and then by Theorem 3: $T(n) \in O\left(n^2\right)$

So we'd say that the given code fragment is of order n^2 complexity.

Of course, this involved some unnecessary work. Big-O analysis provides a gross indication of the complexity of an algorithm, and that can be obtained without first doing an exact analysis (as we did on slides 4 and 5 for this code fragment).

Big-O Analysis Rules

When attempting an approximate big-O time analysis:

1. We assume an arbitrary time unit.
2. Running of each of the following type of statement takes time T(1): [*omitting the arithmetic operators*]
   a) assignment statement
   b) I/O statement
   c) Boolean expression evaluation
   d) function return

Change

Ignore individual Boolean operations & arithmetic operations

3. Running time of a selection statement (if, switch) is T(1) for the condition evaluation + the maximum of the running times for the individual clauses in the selection.

Change

4. Loop execution time is the time for the loop setup (initialization & setup) + the sum, over the number of times the loop is executed, of the body time + time for the loop check and update operations.

   †Always assume that the loop executes the maximum number of iterations possible

5. Running time of a function call is T(1) for function setup + the time required for the execution of the function body.

Change

Ignore parameter computations

New

6. Running time of a sequence of statements is the largest time of any statement in the sequence.

Change **indicates changes from the rules for exact analysis stated earlier.**

Big-O Array Summation

```
typedef int rayType[N];

void sumItoN(rayType ray, int n) {
   int i, j, t;

   i = 0;                      // a
   while (i <= n) {            // b (outer loop)
      j = t = 0;               // c
      while (j <= i) {         // d (inner loop)
         t = t + ray[j];       // e
         j++;                  // f
      }
      ray[i] = t;              // g
      i++;                     // h
   }
}
```

Analysis will deal with the statements labeled a .. h; executable statements only.

Inner Loop: Sum from 0..i (or 1..i+1) of the loop body.
Body Rule 6: Maximum of {loop condition, e, f}
Rule 2: loop condition, e, and f each take 1
So, the inner loop body takes time O(1) and so the inner loop is

$$O\left(\sum_{j=1}^{i+1} 1\right) = O(i+1) = O(i)$$

Outer Loop: Sum from 0..n (or 1..n+1) of the loop body.
Body Rule 6: Maximum of {loop condition, c, inner loop, g, h}
Rule 2: loop condition, c, g, and h each take time 1
So, the outer loop body takes time O(i) and so the outer loop is

$$O\left(\sum_{i=1}^{n+1} i\right) = O\left(\frac{(n+1)(n+2)}{2}\right) = O\left(\frac{1}{2}n^2 + \frac{3}{2}n + 1\right) = O\left(n^2\right)$$

Finally by Rule 6, the big-O complexity of the function is the maximum of the outer loop and statement 1, which is O(1); so the function is $O(n^2)$.

Array Summation (exact count)

```
typedef int rayType[N];

void sumItoN(rayType ray, int n) {
   int i, j, t;

   i = 0;                      // a
   while (i <= n) {            // b (outer loop)
      j = t = 0;               // c
      while (j <= i) {         // d (inner loop)
         t = t + ray[j];       // e
         j++;                  // f
      }
      ray[i] = t;              // g
      i++;                     // h
   }
}
```

Analysis will deal with the statements labeled a .. h; executable statements only.

Inner Loop: Sum from 0..i (or 1..i+1) of the loop body.
Body Rule 4: Sum of {loop condition, e, f}
Rule 2: loop condition and f each take time 1; e takes time 2
So, the inner loop body takes time 4 and so the time for the inner loop is

$$\sum_{j=1}^{i+1} 4 = 4(i+1)$$

Outer Loop: Sum from 0..n (or 1..n+1) of the loop body.
Body Rule 6: Sum of {loop condition, c, inner loop cond, inner loop, g, h}
Rule 2: loop condition, g, and h each take time 1; c takes time 2
So, the outer loop body takes 4(i+1) + 6 or 4i + 10, and so the outer loop plus statement a is

$$T(n) = 2 + \sum_{i=1}^{n+1}(4i+10) = 2 + 4\frac{(n+1)(n+2)}{2} + 10(n+1) = 2n^2 + 16n + 12$$

Practical Applications

Assume:

- 1 day ≈ 100,000 sec. ≈ 10^5 sec. (actually 86, 400)
- Input size $n = 10^6$
- A computer that executes 1,000,000 Inst/sec
  - † **C/C++** statement instructions

Algorithm Complexity Class Comparison

Order: n^2
$(10^6)^2$ Instructions
10^{12} Instructions
$10^{12} / 10^6$ secs to run
10^6 secs to run
$10^6 / 10^5$ days to run
10 days to run

Order: $n \log_2 n$
$10^6 \log_2 10^6$ Instructions
$20 (10^6) = 2 (10^7)$
$2 (10^7) / 10^6$ secs to run
20 sec to run

Internal Class Comparisons

- Within complexity classes the differences between algorithms due to **constants of proportionality**, (coefficients & lesser terms), are not significant enough to warrant reporting except for certain (high usage) applications (e.g., sorting, searching)

Hardware Speedup

Does the fact that hardware is always becoming faster hardware mean that algorithm complexity doesn't really matter?

Suppose we could obtain a machine that was capable of executing 10 times as many instructions per second (so roughly 10 times faster than the machine hypothesized on the previous slide).

How long would the order n^2 algorithm take on this machine with an input size of 10^6?

Order: n^2
instructions: $(10^6)^2 = 10^{12}$
seconds to run: $10^{12} / 10^7 = 10^5$
days to run: $10^5 / 10^5 = 1$

Impressed?

You shouldn't be. That's still 1 day versus 20 seconds if an algorithm of order n log(n) were used.

What about 100 times faster hardware? 2.4 hours.

Algorithm Behavior

Categories

- Algorithms must be examined under different situations to correctly determine their efficiency for accurate comparisons.

Best Case Analysis

- Assumes the input, data etc. are arranged in the most advantageous order for the algorithm, i.e. causes the execution of the fewest number of instructions.
- E.g., sorting - list is already sorted; searching - desired item is located at first accessed position.

Worst Case Analysis

- Assumes the input, data etc. are arranged in the most disadvantageous order for the algorithm, i.e. causes the execution of the largest number of statements.
- E.g., sorting - list is in opposite order; searching - desired item is located at the last accessed position or is missing.

Average Case Analysis

- Determines the average of the running times over all possible permutations of the input data.
- E.g., searching - desired item is located at every position, for each search), or is missing.

Caveats

- Algorithms may have quite different **O**rders for the analysis categories, e.g., **O(**1**)**, **O(**n^2**)**, **O(**nlogn**)**, respectively.