Part I: Complexity Analysis of Non-recursive Algorithms

AUBURN UNIVERSITY

Hugh Kwon

Slides adapted from Dr. Debswapna Bhattacharya's class

Topics

1. Complexity Notations

   What is the technical meaning of Big-Oh (O)? How about the other notations (small-oh o, theta Θ, omega Ω and small-omega ω)?

2. Approximate Big-Oh analyses of non recursive algorithms
3. Detailed complexity analyses of nonrecursive algorithms
4. Approximate Big-Oh analyses of recursive algorithms
5. Detailed complexity analyses of recursive algorithms
   1. Characterizing recursive algorithms by developing recurrence relations
   2. Analyzing their complexity by solving recurrence relations

Approximate Big-Oh Analysis of Complexity Nonrecursive Algorithms

1. For each step that is not a loop statement, decide whether it has a constant cost or not.
   - If constant cost, use a cost of O(1) for the entire step.
   - If not constant cost, estimate maximum (worst case) cost as a function of n and state the appropriate Big-Oh complexity.

2. For loops, start from the innermost loop and work outwards, updating the complexity as each loop is considered. For each loop:
   - Estimate the maximum (worst case or upper bound) number of times the loop statement (for, while, etc.) will execute as a function of n and state the appropriate Big-Oh complexity.
   - Calculate the single execution cost of the loop body as the maximum of the costs of each step in the body.
   - Multiply the Big-Oh estimate of the number of times the loop will be executed with the Big-Oh estimate of the loop body's cost.

3. Complexity of the algorithm is the same as the largest Big-Oh complexity of any step or loop after all estimations in steps (1) and (2) are done.

Insertion Sort

```
Insertion-sort(A: array [1…n] of number, n≥1)
1  for j = 2 to n
2      key=A[j]
3      i = j - 1
4      while i>0 and A[i]>key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = key
```

What is the inherent complexity of sorting?

What does approximate Big-Oh analysis of Insertion Sort yield?

Detailed Analysis of Non-recursive Algorithm Complexity

- Complexity or efficiency of an algorithm is characterized by an equation T(n) that represents the time taken by the algorithm to solve a problem of size n.

Computing T(n)

- Arithmetic operations, assignment, single or multi-dimensional array reference, execution of statements such as return, print, read-from-file etc. take CONSTANT time.

Consecutive Steps

$$
\begin{array}{ll}
temp = i+1 & c_1 \\
A[i] = A[i-1]-3 & c_2 \\
A[i] = A[i-1]*3 & c_3
\end{array}
$$

$$\text{Total time} = c_1 + c_2 + c_3$$

$$\text{Total time} = c = \mathrm{T(n)}$$

Conditional Step

```
if k==2 then                $c_1$
    temp = i+1              $c_2$
    A[i] = A[i-1]-3         $c_3$
else if k==0 then           $c_4$
    temp = 0                $c_5$
else
    A[i] = A[i-1]+3         $c_6$
end if
```

Total time $T(n) < c_1 + c_2 + c_3 + c_4 + c_5 + c_6$

In fact, a **tighter upper bound** is
$T(n) \leq max(c_1 + c_2 + c_3, c_1 + c_4 + c_5, c_1 + c_4 + c_6)$

But we will generally add up all the costs even though that is a looser upper bound.

Loop

	cost of a step	*# times executed*
for i=1 to n	c_1	$n+1$
if A[i]>m then	c_2	n
m = A[i]	c_3	n

Total cost = $(n+1)c_1 + nc_2 + nc_3 = (c_1+c_2+c_3)n + c_1$

$T(n) = nc + c_1$ *where* $c=c_1+c_2+c_3$

Independent Nested Loops

for i=1 to n	n+1 times	$c_1(n+1)$
for j=1 to n	n(n+1) times	$c_2n(n+1)$
temp=i+j+1	n^2 times	c_3n^2
A[i,j]=temp	n^2 times	c_4n^2

$T(n)=c_5n^2 + c_6n+c_1$

Dependent Nested Loops

```
1  **for** $i$ =1 **to** n
2      key=A[$i$]
3       **for** $j$ =1 **to** i
4           $A[j+1] = A[j]$
```

$$T(n) = c_1(n+1) + c_2 n + c_3 \sum_{i=1}^{n} (i+1) + c_4 \sum_{i=1}^{n} i$$

learn to expand these summations   to obtain a polynomial

Example

```
Max (A: Array [1..n] of integer)
m: integer
begin
1   m = A[1]
2    for i = 2 to n
3           if A[i]>m then
4                     m = A[i]
5    return m
```

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5$$

$$T(n) = c_6 n + c_7$$

Example

Insertion-sort(A)	cost	times
1 **for** j =2 **to** A.length	c_1	n
2 key=A[j]	c_2	$n-1$
3 //Insert $A[j]$ into the sorted sequence $A[1..j-1]$		
4 $i = j - 1$	c_4	$n-1$
5 **while** i>0 and $A[i]$>key	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = \text{key}$	c_8	$n-1$

t_j : the number of times the while loop test in line 5 is executed for the value of j.

the running time

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j +$$
$$c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- $t_j = 1$ for j = 2,3,…,n; best case – why?

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

the running time

$t_j = j$ for $j = 2,3,\ldots,n$; quadratic function on n; worst case – why?

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) +$$

$$c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5 + c_6 + c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8\right)n$$

$$-(c_2 + c_4 + c_5 + c_8)$$

Best-case, worst-case and average-case analysis

- Usually, we calculate only the *worst-case running time*
- Reason:
  - It is an upper bound on the running time
  - The worst case occurs fairly often
- The average case is often as bad as the worst case. The best case is usually particular kinds of inputs (which may be as few as one or as large as infinite) amongst the large (possibly infinite) number of all possible inputs.

Reading & Thinking Assignments

- Chapter 2:
  - Read Section 2.2
  - Do problems 2.2-1 through 2.2-4

Determining the constants

- Count a cost of 1 for
  - reading from a variable
  - writing a value to a variable (i.e., assigning a value to a variable)
  - using an array index to locate the corresponding memory location (or accessing a memory location such as a tree or linked list node using a pointer)
  - reading from or writing into that location (i.e. reading or writing an array cell or a tree or linked list node)
  - an arithmetic operation: + - / *
  - a comparison: =, <, >, !=
  - note that ≤ and ≥ involve two comparisons
  - boolean operations: and, or, not, xor

Determining the constants

- Examples:

  m = 12: cost 1

  m = n: cost 2

  if position ≥ m: cost 4

  A[5] = A[1]: cost 4

  A[i] = A[j]: cost 6

  m = A[i+1]: cost 5

  A[i] = A[i+1]: cost 7

What about loops?

- a loop statement containing one or more of: loop variable initialization, loop variable update and exit condition:
  - for loop
  - while loop
  - repeat-until loop

What about loops?

– for loop: it encapsulates multiple basic operations for loop variable initialization, update and exit condition checking implicitly.

```
for i = 2 to n
    <loop body>
```

is equivalent to

```
i=2
while i≤n
    <loop body>
    i=i+1
```

But we will assume a cost of 1 for the for loop statement for simplicity

What about loops?

– while loop and repeat-until loop: its execution cost is the cost of the exit condition checking.

```
while $i < n$                    cost 3
while $i<5$ and $p \leq q$           cost 7

repeat                    no cost
        <loop body>
until $i<n$                   cost 3
```

Thinking Assignment

```
Loopy()
1 outerloop=middleloop=innerloop=0
2  for $i$ =1 to n
3          outerloop=outerloop+1
4        for $j$ =n down to i
5                  middleloop=middleloop+1
6                  for $k$ =3 to (n−j)
7                          innerloop=innerloop+1
8 print (outerloop, middleloop, innerloop)
```

1. What are the three positive integers this algorithm will print?
2. Write the number of times each statement 4,5,6 & 7 will execute using the summation (sigma) notation
3. Calculate T(n) for this algorithm

What about calls to the same or other algorithms?

- if you encounter a call to a different algorithm, find out its cost first
- count cost of explicit operations in the calling step; ignore cost of implicit operations
- parameter passing is implicit:
    - variables/constants by value (make a copy)
    - data objects by reference (pass a pointer)

E.g.: Mystery (a[i+1…j], temp, 5)

cost = cost of Mystery + 2 (for computing "i+1")

•we will deal with recursive calls to the same algorithms later

T(n) calculation

- How detailed should the T(n) calculation be?
- As detailed as needed for your purpose!
- Approximate vs detailed analyses

Example

```
algorithm A1
1 for $i$ =1 to n
2     key=A[ $i$ ]
3         for $j$ =1 to i
4             $A[ j+1 ] = A[ j ]$
```

Suppose you want to compare this “algorithm” with another A3 that you know has three nested loops each of which executes n times. This means that A3’s approx. complexity is $O(n^3)$.

The above algorithm’s approx. complexity, on the other hand, is $O(n^2)$; so A1 is a more efficient algorithm.

Example

But suppose you want to compare the previous algorithm A1 with algorithm A2:

```
for $i$ =1 to n
    key=A[ $i$ ]
        for $j$ =1 to n
            $A[j+1] = A[j]$
```

By approximate analysis, this and the algorithm A1 on the previous slide are both O(n^2) algorithms. But you can tell A2 will be less efficient than A1. Why?

But suppose you also want to know how much more inefficient this algorithm is. Then you need to do a detailed calculation.

Example

```
1 for $i$ =1 to n
2     key=A[ $i$ ]
3         for $j$ =1 to i
4             $A[j+1]=A[j]$
```

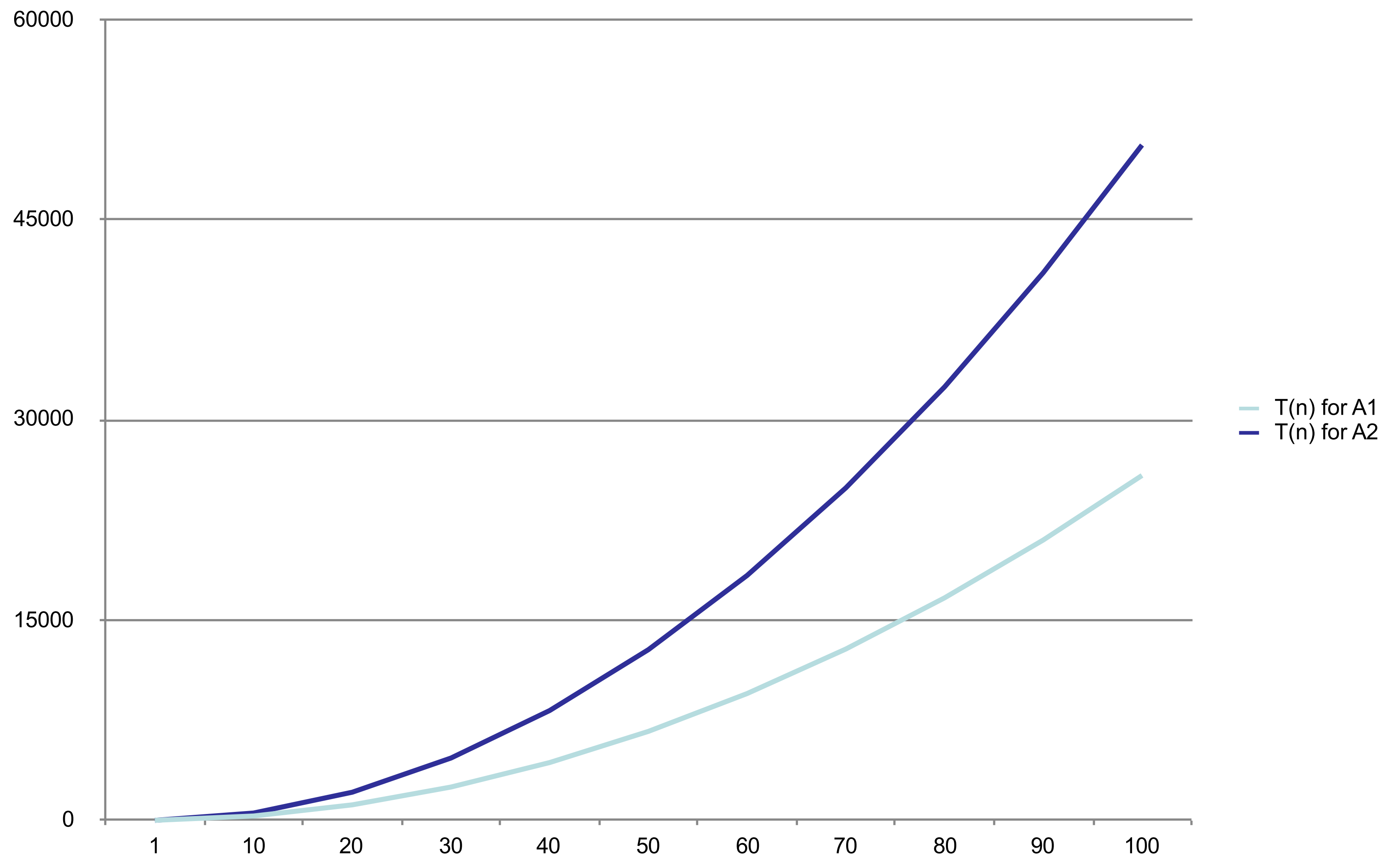
- T(n) = $4n^2+10n+1$

algorithm A2:

```
1 for $i$ =1 to n
2     key=A[ $i$ ]
3         for $j$ =1 to n
4             $A[j+1]=A[j]$
```

- T(n) = $8n^2+6n+1$

The Growth Rate of A1 vs. A2



Example

- You can see that though the number of steps executed or the complexity of both algorithms are close to each other for very small values of n, as the input becomes larger the efficiency gap between them increases. Only a detailed complexity calculation would alert you that for large inputs A1 will be significantly faster.

Thinking Assignment

Do an approximate complexity analysis of the Merge procedure in the next two slides and show that it is O(n).

Then calculate the exact expression T(n) for the Merge procedure of Merge Sort, given in the following slides.

1. Assume that the size of the input array A = r-p+1 = n.
2. The variables n_1 and n_2 should not appear in T(n).
3. Assume that step 3 is not an executable statement, so has no cost.
4. The correct answer has the form $T(n) = c_1n+c_2$ where c_1 and c_2 should be specific integers.

Merge(A,p,q,r)

```
1   $n_1 = q - p + 1$
2   $n_2 = r - q$
3   create array L[ 1 .. $n_1 + 1$ ] and R[ 1 .. $n_2 + 1$ ]
4   **for** $i = 1$ ***to*** $n_1$
5       L[ $i$ ] =A[ $p + i - 1$ ]
6   **for** $j = 1$ ***to*** $n_2$
7       R[ $j$ ] =A[ $q + j$ ]
8   L[$n_1 + 1$] = $\infty$
9   R[$n_2 + 1$] = $\infty$
```

Merge(A,p,q,r)

```
$i = 1$
$j = 1$
**for** $k = p$ ***to*** $r$
    ***if*** $L[i] \le R[j]$
        **then** $A[k] = L[i]$
            $i = i + 1$
    **else** $A[k] = R[j]$
            $j = j + 1$
```

Reading Assignment

At this point, you should finish reading all of chapter 2 except for section 2.3.2. Pay special attention to the Loop Invariant proofs of Insertion Sort and the Merge procedure.

Thinking Assignment

Do problems 2.2.1-2.2-4, p.29

AUBURN UNIVERSITY