DESIGN AND ANALYSIS OF ALGORITHMS

CS 4120/5120

THE EFFICIENCY OF ALGORITHMS

AGENDA

- Model of Implementation
- The complexity of an algorithm
  - Running time
  - Analyze running using **cost** and **time**
- Derive the running time function

ANALYZING ALGORITHMS

- Analyzing an algorithm means *predicting the resources* that the algorithm requires.
  - Memory
  - Communication bandwidth
  - Computer hardware
  - ***Computational time***
    - Algorithms will be implemented by computer programs.
      - The computational time (performance) can be affected by the above aspects

CASE

- Consider computers A and B. Both computer executes one instruction per clock cycle of their respective CPUs.

Computer	Architecture	Clock cycle time
A	Reduced Instruction Set Computer (RISC)	80 ps
B	Complex Instruction Set Computer (CISC)	1000 ps

  - Computer B's instruction set includes an instruction for sorting integers
- Run the same sorting program on the same input data on both computers.
- Which computer is likely to finish sorting first?

IMPLEMENTATION TECHNOLOGY
THE RAM MODEL

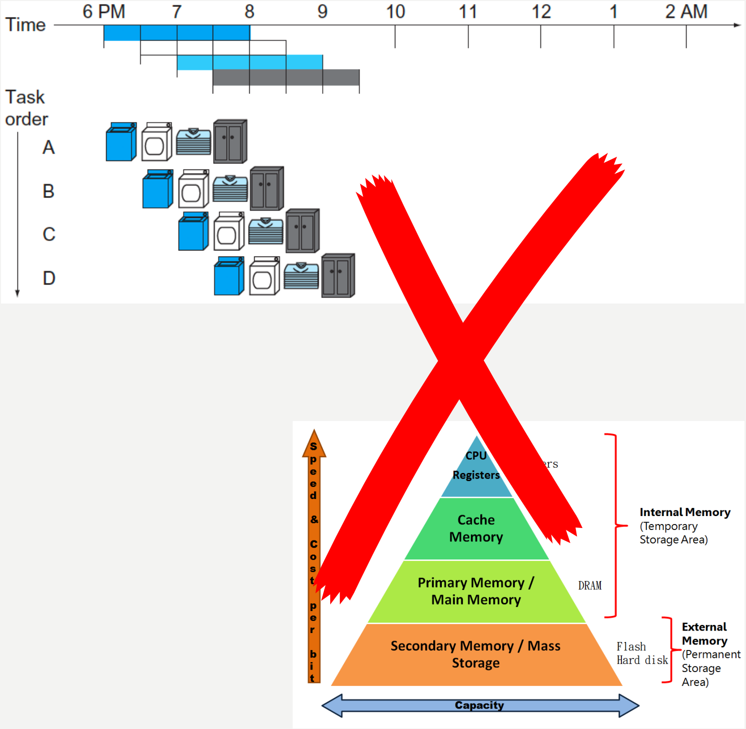
- A generic one-processor, ***R***andom-***A***ccess ***M***achine (***RAM***) model of computation
- Certain constraints on the ISA apply

RAMDOM-ACCESS MACHINE (RAM) INSTRUCTION CONSTRAINTS

- The instruction set of the RAM model contains **instructions commonly found in real computers**.
  - Arithmetic
    - $+,\ -, \times, \div, mod, floor\ (\lfloor\ \ \rfloor), ceiling\ (\lceil\ \ \rceil)$
  - Data movement
    - load, store, copy
  - Control
    - conditional and unconditional branches, subroutine call and return

RAMDOM-ACCESS MACHINE (RAM) INSTRUCTION EXECUTION

- The program instructions of the RAM model are executed one after another, with **no concurrent operations**.
- Each instruction takes a ***CONSTANT amount of time***.
  - The computational time of exponentiation is specified in the next slide.
- **Do NOT model memory hierarchy**



RAMDOM-ACCESS MACHINE (RAM)
EXPONENTIATION EXECUTION TIME

- In general, computing x^y when x and y are real numbers is **NOT a constant-time operation**.
  - Example: $1.5^{2.1}$, $\left(\sqrt{2}\right)^{3.5}$, $5^{0.5}$, 1.5^y, where $y \in R$. R is the set of real numbers.

- However, computing 2^k when k is a **small enough positive integer** is a constant-time operation.

CONSTANT-TIME OPERATION PRACTICE

CONSIDER THE FOLLOWING OPERATIONS CARRIED OUT BY A RAM.

- Determine whether each statement is a constant-time op.
- Then, determine whether each lettered item is a constant-time op.

A.
```
int x = 5;
```

B.
```
y = a + b;
```

C.
```
for i = 1 to n
    A [i] = A[i] + 1
```

D.
```
for i = 1 to 100
    A [i] = A[i] + 1
```

E. $n = 2^m$, where m is an integer, and $m \in (0,10]$

F. **if** $n > A.length$ //A is an array

G. $n = 2^k$, where k is a real number, and $k \in (0, 10]$.

H. $n = m^k$

CONSTANT-TIME OPERATION
PRACTICE

- Consider the algorithm. Is each statement a constant-time operation?

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j − 1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

THE EFFICIENCY OF AN ALGORITHM
COMPLEXITY

- The algorithm can solve the problem **with limited resources**.
  - Time complexity: the job could be done within *finite* time.
    - Also called the ***running time*** of an algorithm.
      - The ***running time*** (time complexity) of an algorithm, denoted by $\boldsymbol{T(n)}$, is the number of primitive operations or "steps" executed
  - Space complexity: the amount of available memory is determinate.
    - Compared to time complexity, space complexity is given less consideration in this course when designing an algorithm.

THE EFFICIENCY OF AN ALGORITHM
RUNNING TIME

- In other words, ***running time*** of an algorithm is the sum of running times for each statement executed.

- Example
  - Consider the code below

The computational *cost* of a single statement

The number of executions of a single statement

		Cost	Time
1	**for** $i = 1$ **to** n	c_1	$n+1$
2	$A[i]++$	c_2	n

  - The running time of the code is $T(n) = c_1 \cdot (n+1) + c_2 \cdot n$

RUNNING TIME PRACTICE

- Compute the running time of following (independent) code segments using the ***cost-time*** table.

A.

Code	Cost	Time	Running Time
for $i = 2$ **to** n			$\boldsymbol{T(n)} =$
$\quad A[i]++$			

B.

Code	Cost	Time	Running Time
for $j = 5$ **to** n			$\boldsymbol{T(n)} =$
$\quad i = j - 1$			
$\quad$ **while** $i > 0$			
$\qquad i--$			

THE EFFICIENCY OF AN ALGORITHM
ANALYZE INSERTION-SORT

- Derive the close-end form of the running time function $T(n)$ of INSERTION-SORT
  - Use $\sum$ and t_j to denote the executions of the **while**-loop at line 5 for a value of j
- See next slide.

$INSERTION\text{-}SORT(A)$	Cost	Time
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 $A[1..j-1]$	$c_3 = 0$	
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ **and** $A[i] > key$	c_5	$\sum t_j$
6 $A[i+1] = A[i]$	c_6	$\sum (t_j - 1)$
7 $i = i - 1$	c_7	$\sum (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

THE EFFICIENCY OF AN ALGORITHM
ANALYZE INSERTION-SORT (CONT'D)

- Derive the close-end form of the running time function $T(n)$ of INSERTION-SORT

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

$INSERTION-SORT(A)$	Cost	Time
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$	$c_3 = 0$	
$i = j - 1$	c_4	$n-1$
while $i > 0$ **and** $A[i] > key$	c_5	$\sum t_j$
$A[i+1] = A[i]$	c_6	$\sum (t_j - 1)$
$i = i - 1$	c_7	$\sum (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

THE EFFICIENCY OF AN ALGORITHM
SIMPLIFY THE FUNCTION

- Simplify the running time function

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

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

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

THE EFFICIENCY OF AN ALGORITHM
SIMPLIFY THE FUNCTION (CONT'D)

- Simplify the running time function

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

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

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

This term determines the running time is linear or quadratic.

THE EFFICIENCY OF AN ALGORITHM
BEST- VS WORST-CASE

- At this point, we have obtained the running time function in terms of n and t_j.

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

  - $\boldsymbol{t_j}$ depends on the actual input array A
  - More specifically, $\boldsymbol{t_j}$ depends on $\boldsymbol{j}$ and the **relationship between each element of subarray $\boldsymbol{A[1..j-1]}$ and $\boldsymbol{A[j]}$**.

THE EFFICIENCY OF AN ALGORITHM
BEST- VS WORST-CASE (CONT'D)

- Revisit the algorithm under best-case and worst-case scenarios
  - Complete the missing cells
    - Best-case scenario
      $t_j =$
    - Worst-case scenario
      $t_j =$

$INSERTION-SORT(A)$	Cost	Time: Best-case	Time: Worst-case
for $j = 2$ **to** $A.length$	c_1	n	$n-1$
$key = A[j]$	c_2	$n-1$	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$	$c_3 = 0$	NA	NA
$i = j - 1$	c_4	$n-1$	$n-1$
while $i > 0$ **and** $A[i] > key$	c_5		
$A[i+1] = A[i]$	c_6		
$i = i - 1$	c_7		
$A[i+1] = key$	c_8	$n-1$	$n-1$

THE EFFICIENCY OF AN ALGORITHM
RUNNING TIME OF INSERTION-SORT

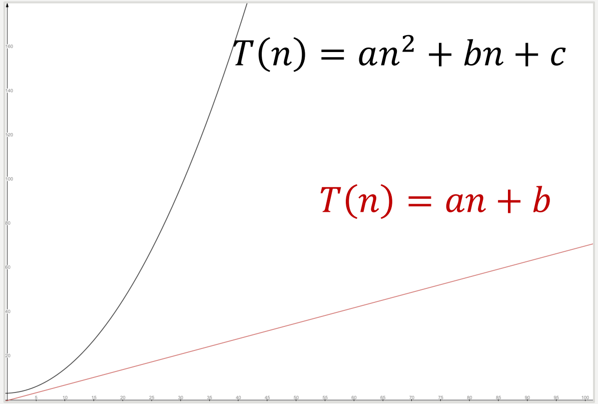
- **Best**-case scenario
  - Input A is already **sorted** in the desired order
  - $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)\boldsymbol{n} - (c_2 + c_4 + c_5 + c_8)$
    - In the form of $a\boldsymbol{n} + b$
  - Running time is a ***linear function***.

THE EFFICIENCY OF AN ALGORITHM
RUNNING TIME OF INSERTION-SORT

- **Worst**-case scenario
  - Input A is **reverse** sorted .
  - $T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)\boldsymbol{n^2} + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)\boldsymbol{n} - (c_2 + c_4 + c_5 + c_8)$
    - In the form of $a\boldsymbol{n^2} + b\boldsymbol{n} + c$
  - Running time is a ***quadratic function***.

THE EFFICIENCY OF AN ALGORITHM
RUNNING TIME OF INSERTION-SORT

- Compare the running time functions
  - Best-case: $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)\boldsymbol{n} - (c_2 + c_4 + c_5 + c_8)$
  - Worst-case: $T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)\boldsymbol{n^2} + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)\boldsymbol{n} - (c_2 + c_4 + c_5 + c_8)$



NEXT UP
GROWTH FUNCTIONS

REFERENCES