Algorithm Analysis CISC4080 CIS, Fordham Univ.

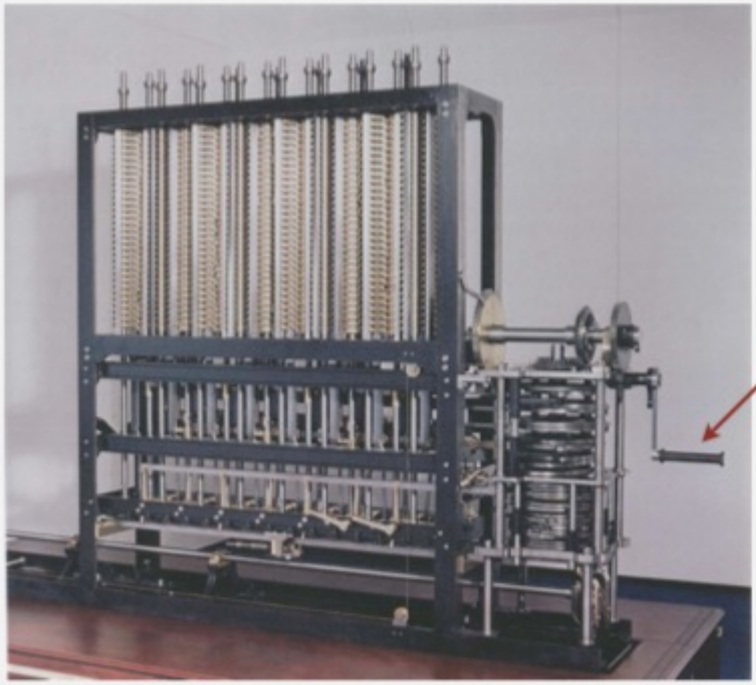
Instructor: X. Zhang

Outline

- Resource usage of algorithms: space and time
- Characterize resource usage of algorithms
  - Experimental approach
  - Running time Analysis: based upon code or pseudocode
    - count number of “computer steps”
    - write recursive formula for running time of recursive algorithm
- math help: math. induction

Running time

" *As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?* " — *Charles Babbage (1864)*



Analytic Engine

In pursuit of better algorithms

- We want to solve problems using less resource:
  - Space: how much memory is needed?
  - Time: how fast can we get the result?
- Observation: Given an algorithm, its resource usage depends on the size of input
  - It takes longer to sort a long list
  - It takes longer to calculate Fn, n-th term in Fibonacci sequence for larger n
  - …
- When the input is small, it doesn't matter. So we focus on problem of very large size.

Algorithm Analysis: bubble sort

bubblesort (a[0…n-1])

input: a list of numbers a[0…n-1]

output: a sorted version of this list

```
for endp=n-1 to 1:
    swapOP=0;
    for i=0 to endp-1:
        if a[i] > a[i+1]:  swap (a[i], a[i+1]); swapOP++;
    if (swapOP==0): return  //no need to continue
return
```

Q: Does it take same amount of time to sort following input lists?

1. list1[0…5]={1, 4, 5, 6, 7, 9}
2. list2[0…5]={9, 1, 4, 7, 6, 9}
3. list3[0…5]={9, 7, 6, 5, 4, 1}

Resource usage of algorithms

Running time/space requirement of an algorithm depends upon

- Size of the input, usually denoted as n
- often times, also upon the particular input, e.g. if the list is sorted already or not

For a given problem size, running time often varies:

- best case: some problem instances/inputs yield shortest possible running time (e.g. the value searched for is in first place)
- worst case: some problem instances yield longest possible running time (e.g., linear search: value searched for is in last place)
- average case: if we assume all instances are equally likely (or any other prob. distribution), the expected running time…

- We usually focus on worst case running time.

Outline

- Resource usage of algorithms: space and time
- Characterize resource usage of algorithms
    - Experimental approach
    - Running time Analysis: based upon code or pseudocode
        - count number of “computer steps”
        - write recursive formula for running time of recursive algorithm
- math help: math. induction

Experimental Approach

Insight: [knuth 1970] Use scientific method to understand performance

1. Observe some features of the natural world (here measure running time or memory usage)
2. Hypothesize a model that is consistent with the observation
3. Predict using the hypothesis
4. Verify the prediction by making further observation
5. Validate by repeating until the hypothesis and observation agrees

Observe how long does it take…

you to finish a homework assignment?

- Right before beginning to work on the homework, check the clock, and it’s Wednesday, Sept 9, 10:15:00 am
- Work on the homework (without taking any break).
- Right after finish, check the clock again and it’s Wednesday, Sept 9, 1:20:00pm

- How long did it take you to finish the homework?

How long does it take…

an algorithm to run when given some input?

- Get current time of the clock, and store it in start_time
- Call your algorithm with some input
- Get current time of the clock, and store it in finish_time
- Running time of the algorithm: finish_time - start_time

System Time

All computer systems maintain a system time/clock.

- It keeps track number of seconds and nanoseconds (10^{-9} second) passed since some starting time, called the *epoch*.

In Unix and POSIX-compliant systems, the epoch used is Jan 1st, 1970 00:00:00

Unix time/clock tells you how many seconds and nanoseconds has passed since the Unix epoch at January 1st 1970 00:00:00

clock_gettime()

A library function, **clock_gettime** retrieves the time of a clock

Usage example

```
#include <time.h>
/* timespec type:
  struct timespec {
    time_t   tv_sec;       /* seconds */
    long     tv_nsec;      /* nanoseconds */
};  */

struct timespec t; // declare a variable t of type timespec

clock_gettime(CLOCK_REALTIME, &t);
// Retrieve the time of specified clock, CLOCK_REALTIME
```

Measuring running time

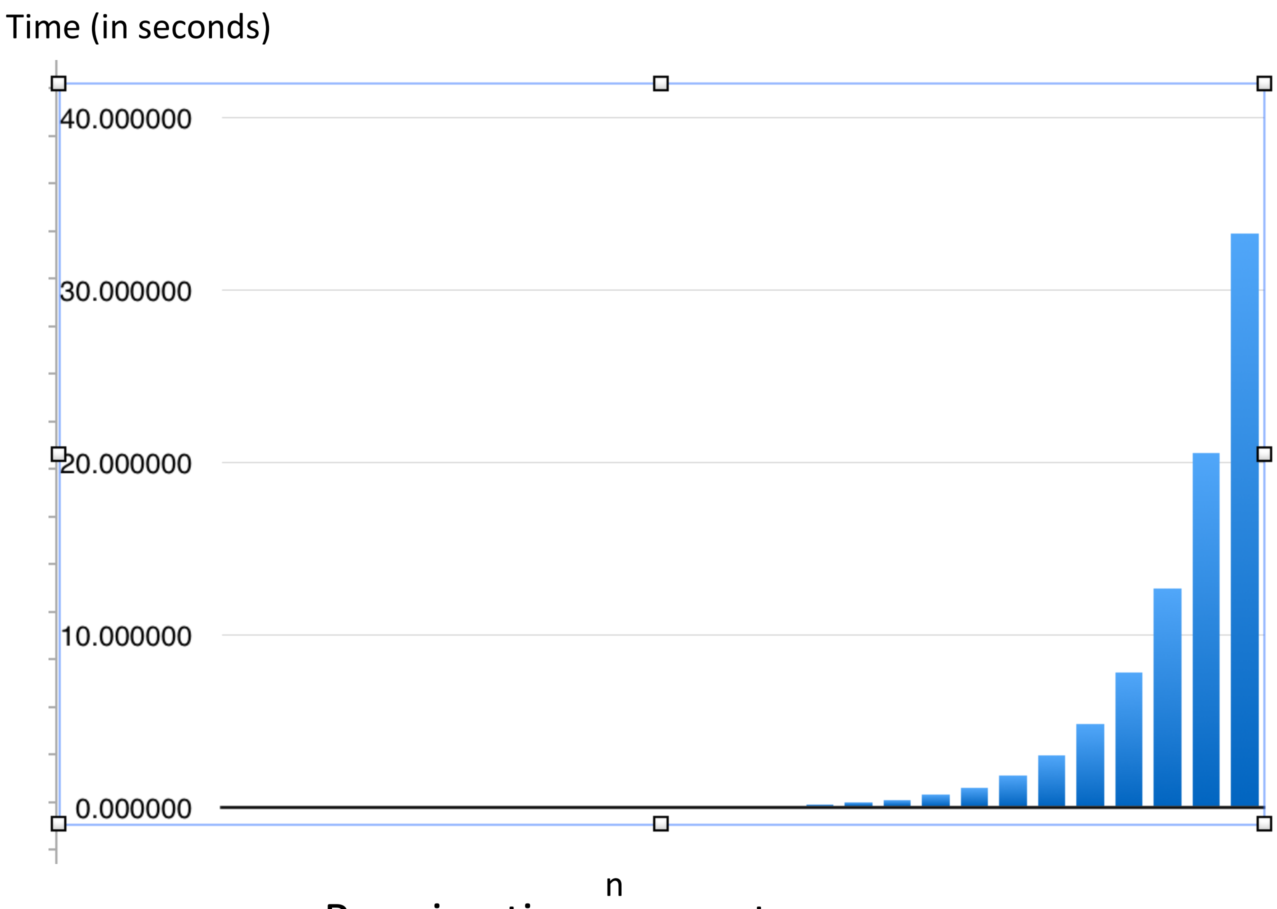
```
#include <time.h>
…
struct timespec t1, t2; //t1, t2 are two variables of type timeval
…
clock_gettime(CLOCK_REALTIME, &t1);
result = Fib(100);  // or any other algorithm we want to measure
running time of …
clock_gettime (CLOCK_REALTIME, &t2);

// calculate how many seconds have passed between t1 and t2
double timeInSeconds = (t2.tv_sec-t1.tv_sec) +
      (t2.tv_nsec-t2.tv_nsec)/1e9;
```

* Question: How the running time of Fib() grows when with n?

Example (Fib1: recursive)

```
Recursive Fib calculator
n   F(n)       T(n)ofFib1 (s)
1   1   2.72e-07
2   1   1.67e-07
3   2   2.49e-07
4   3   3e-07
5   5   3.95e-07
6   8   3.98e-07
7   13  5.46e-07
8   21  7.25e-07
…
13  233 4.256e-06
14  377 6.848e-06
15  610 1.0964e-05
16  987 1.7656e-05
17  1597    2.8207e-05
18  2584    4.5365e-05
19  4181    7.3375e-05
…
33  3524578    0.0195626
34  5702887    0.0318968
35  9227465    0.0516566
36  14930352   0.0839258
…
40  102334155  0.605248
41  165580141  0.932493
42  267914296  1.51373
43  433494437  2.40282
44  701408733  3.89735
45  1134903170 6.31143
46  1836311903 10.2351
47  -1323752223    16.5711
48  512559680  26.8665
49  -811192543 43.6285
```



Running time seems to grows exponentially as n increases

How long does it take to Calculate F(100)?

Observation

Observation:

- Time to calculate n-th term is sum of time to calculate (n-1), (n-2) terms
- Explanation?

Hypothesis: running time of recursive Fib grows exponentially with n,

- $T(n)=c*a^n+c_2$, for $a > 1$
- Possible to model fitting to obtain the constants…

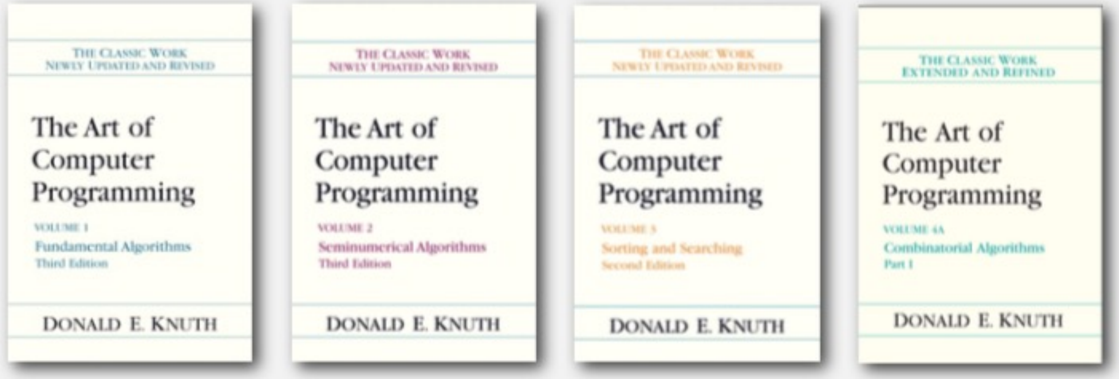
Outline

- Resource usage of algorithms: space and time
- How to understand resource usage of algorithms?
  - Experimental approach
  - Running time Analysis: based upon code or pseudocode
    - count number of “computer steps”
    - write recursive formula for running time of recursive algorithm
- math help: math. induction

Analytic Approach

Total running time: sum of cost x frequency for all operations

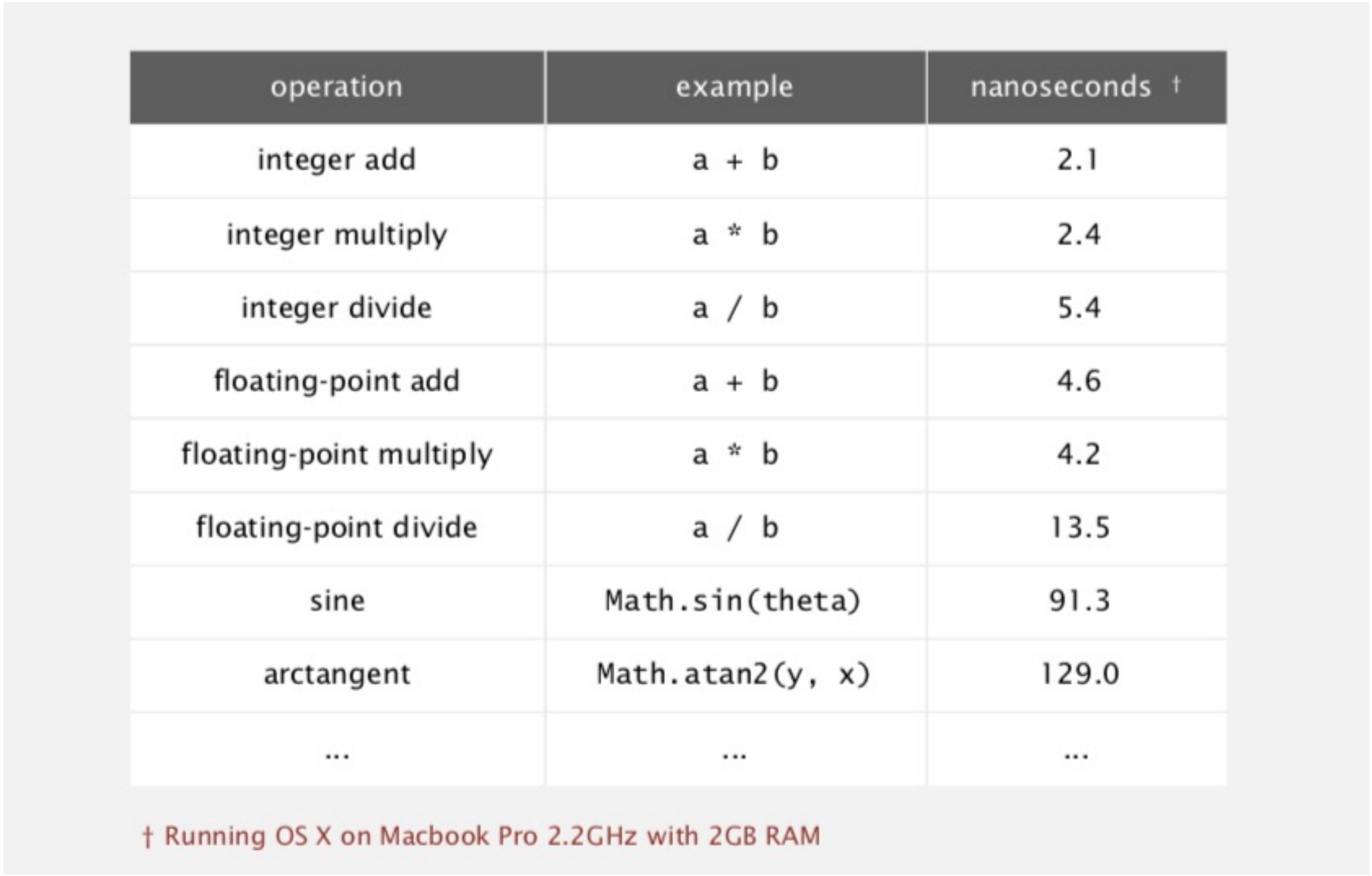
- Need to analyze program to determine set of operations
- Cost of operation: depends on machine, compiler
- Frequency depends on algorithm, input data



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

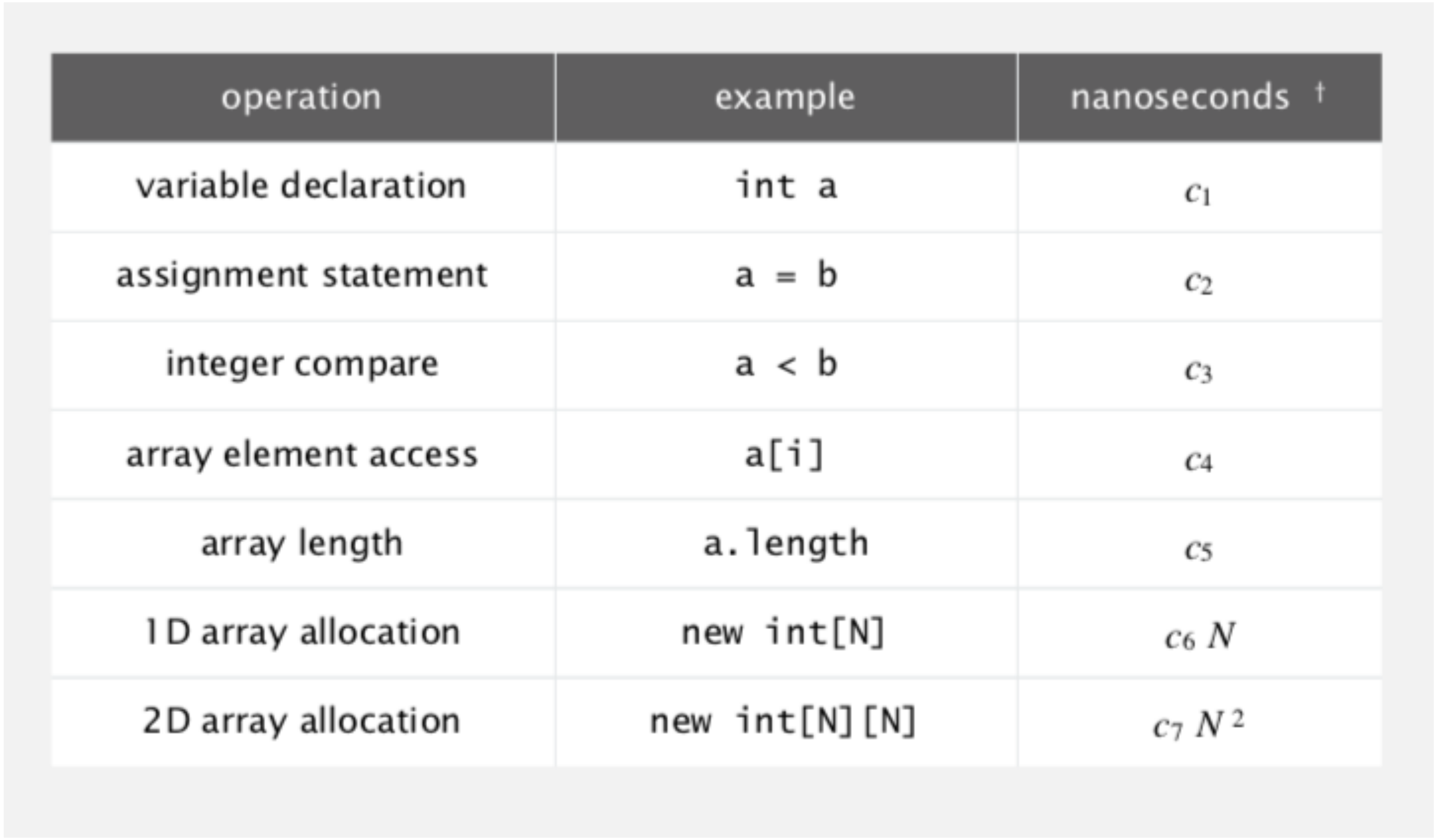


operation	example	nanoseconds †
integer add	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating-point add	`a + b`	4.6
floating-point multiply	`a * b`	4.2
floating-point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

Observation: most primitive operations takes constant time

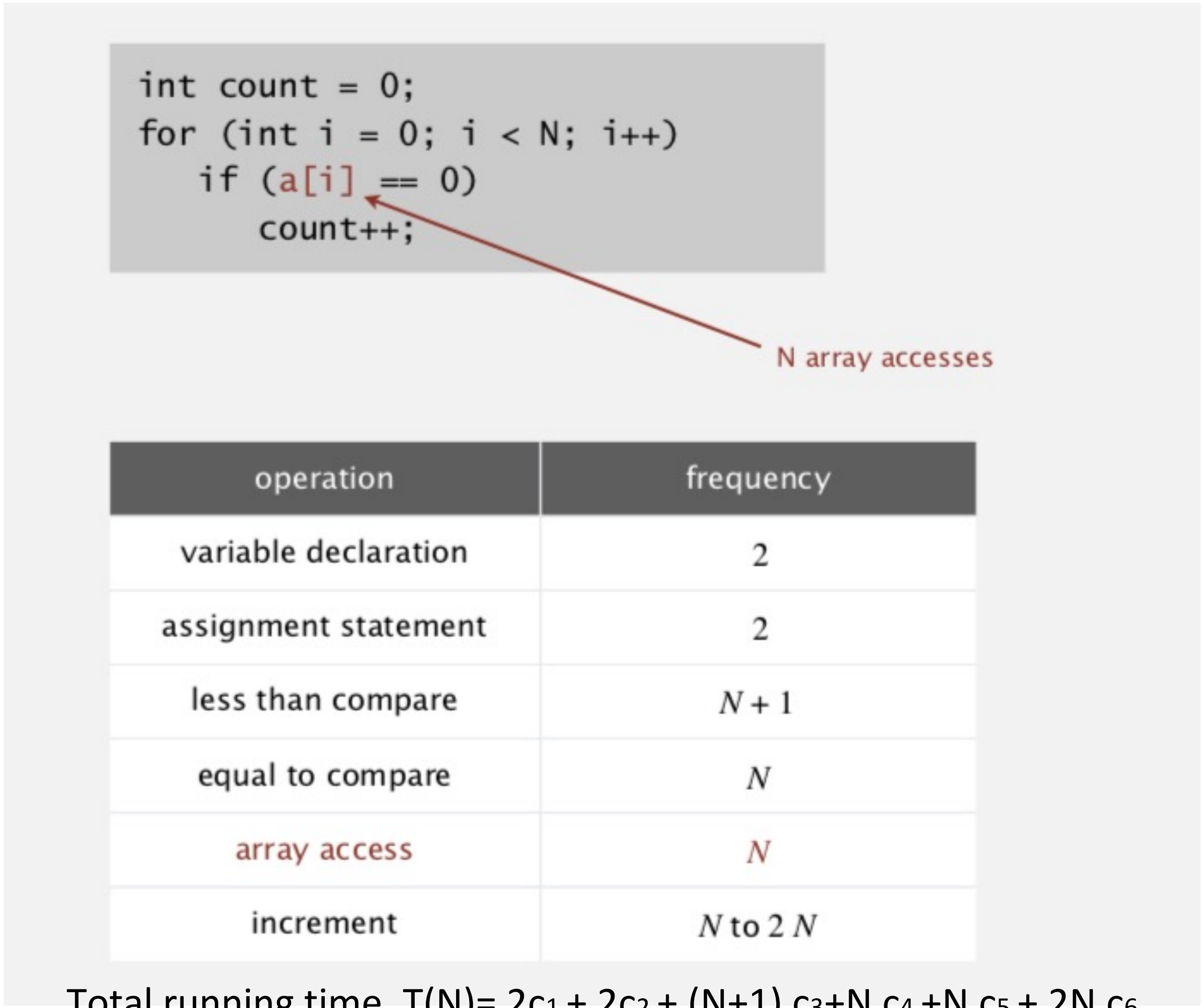


operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6 N$
2D array allocation	`new int[N][N]`	$c_7 N^2$

Caveat. Non-primitive operations take more time constant time
e.g. BST search takes log N time (C++, ordered_map look up)…

Example: how many 0’s?

Q. Frequency as a function of input size N?



```
int count = 0;
for (int i = 0; i < N; i++)
   if (a[i] == 0)
      count++;
```

N array accesses

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2N$

Total running time, $T(N) = 2c_1 + 2c_2 + (N+1)\,c_3 + N\,c_4 + N\,c_5 + 2N\,c_6$

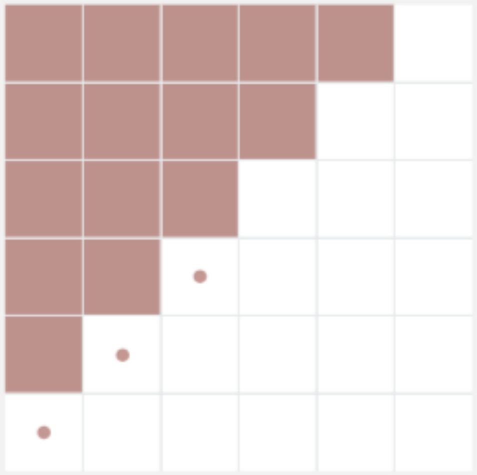
Example: Sum to 0

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

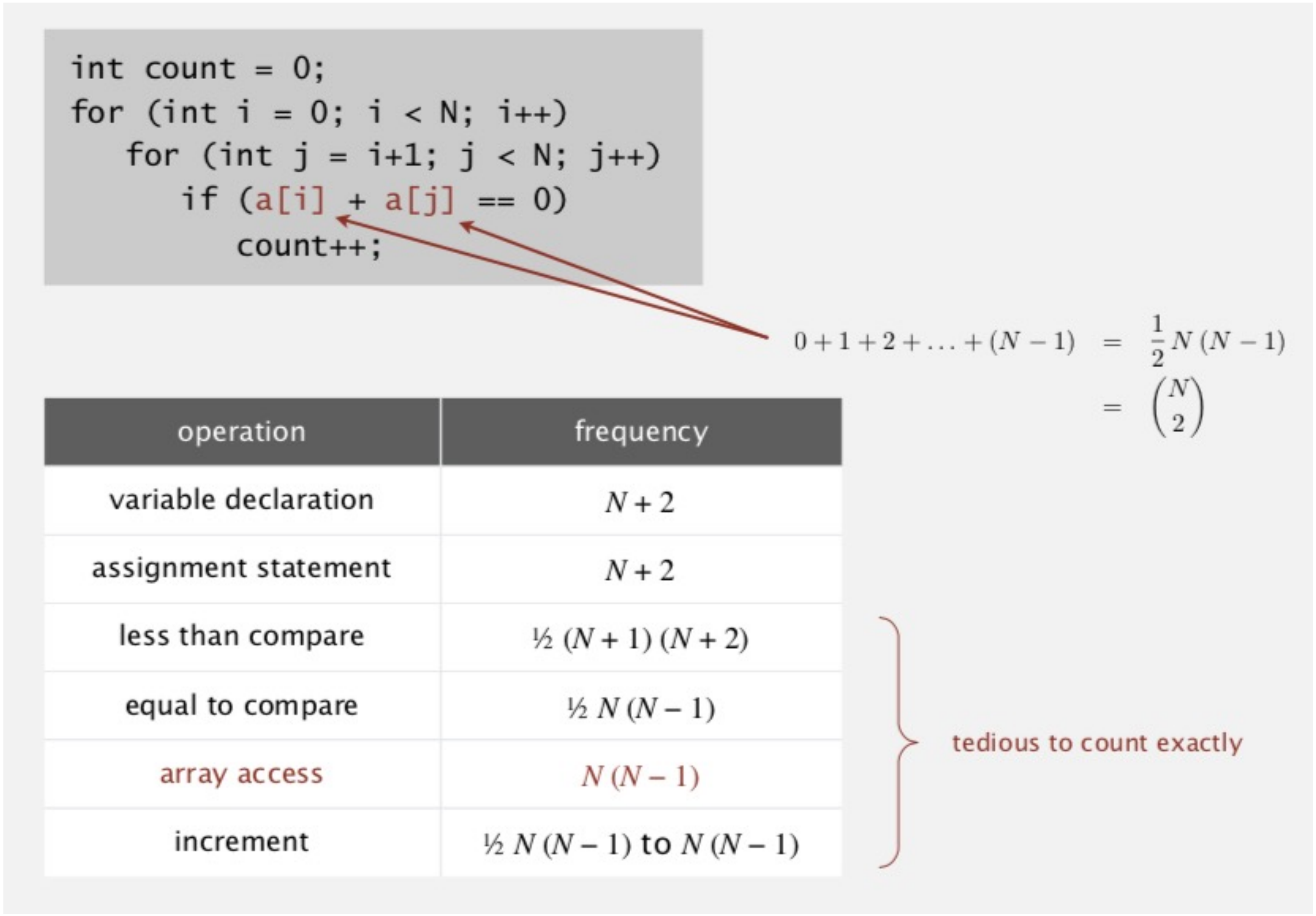
Pf. [n even]



$$0 + 1 + 2 + \ldots + (N-1) = \underbrace{\frac{1}{2}N^2}_{\text{half of square}} - \underbrace{\frac{1}{2}N}_{\text{half of diagonal}}$$

Example: 2-Sum to 0

Q. How many instructions as a function of input size N?



```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

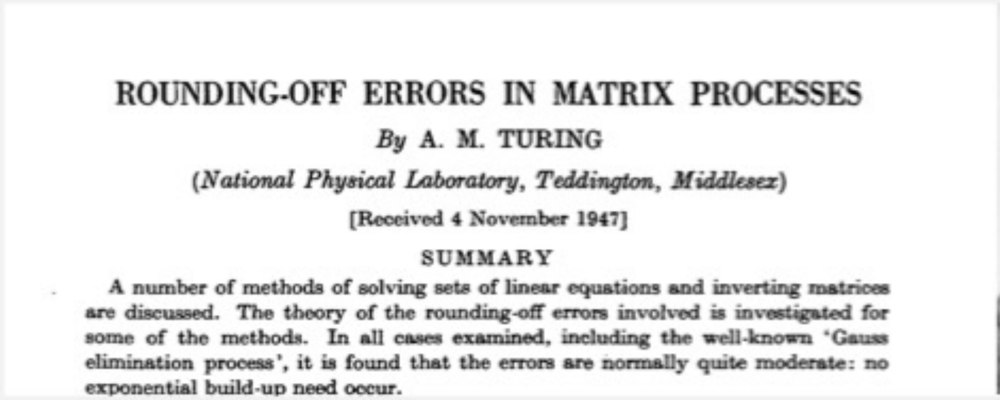
$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	½ $(N + 1)(N + 2)$
equal to compare	½ $N (N - 1)$
array access	$N (N - 1)$
increment	½ $N (N - 1)$ to $N (N - 1)$

tedious to count exactly

Simplifying the calculations

“ It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. ” — Alan Turing



ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

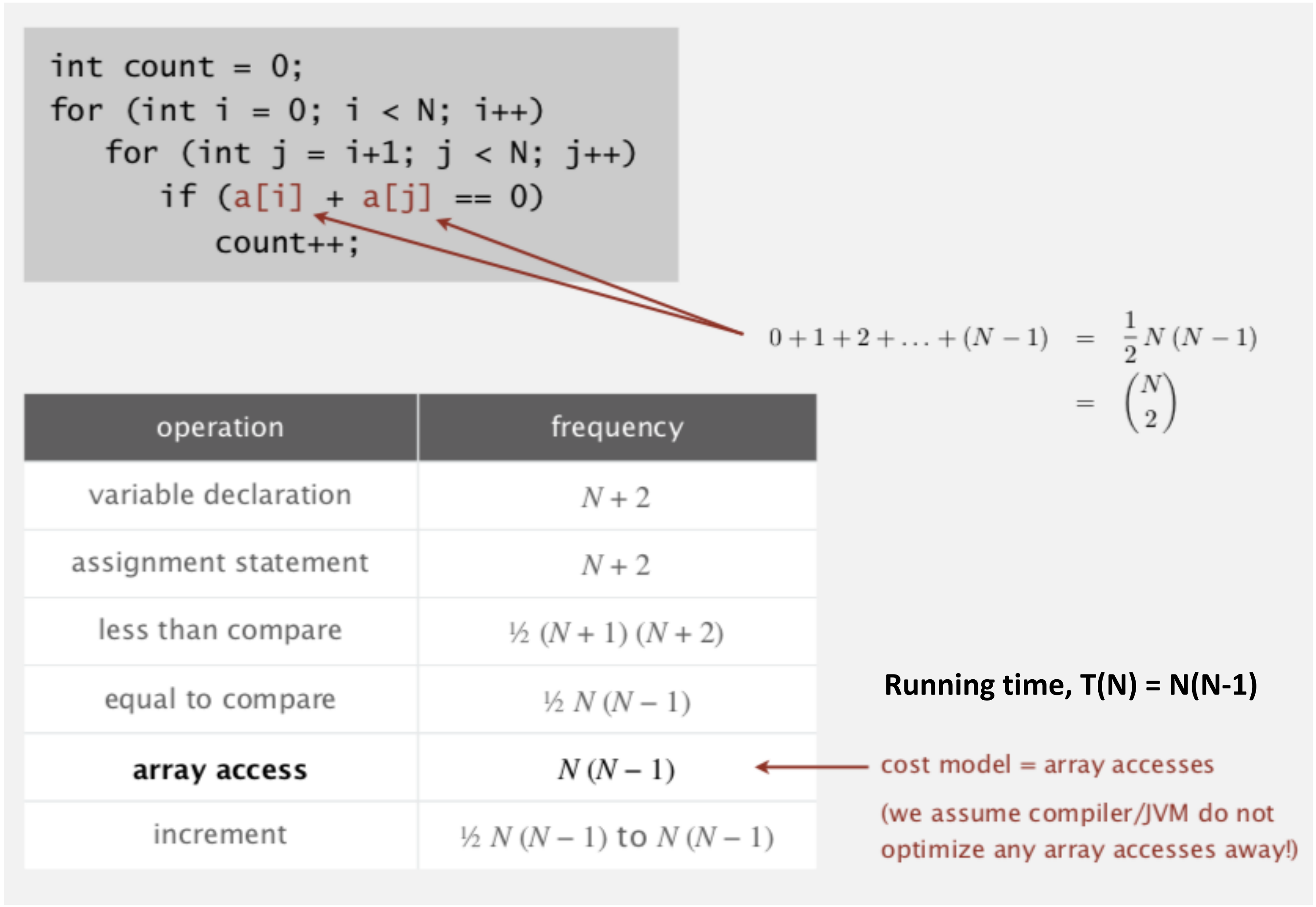
[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

Simplification

Use frequency of basic operation in the deepest loop to represent/capture running time



```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2} (N+1)(N+2)$
equal to compare	$\frac{1}{2} N (N-1)$
array access	$N (N-1)$
increment	$\frac{1}{2} N (N-1)$ to $N (N-1)$

Running time, T(N) = N(N-1)

cost model = array accesses

(we assume compiler/JVM do not optimize any array accesses away!)

Running time fib2(n)

Characterize running time of fib2(n), by choosing frequency of array access operation

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

Note: in homework/quizzes, the representative operations will be specified for you.

Example: checking cross-pairs

Q. How many **array access** operations?

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        if (a[i] + b[j] == 0)
            count++;
```

Steps

1. List all values for outer loop variables I
2. For each value of i, write the inner loop header
   - How many times is the inner loop body iterates?
3. Sum all frequency up

bubble sort: worst case

bubblesort (a[0…n-1])

input: an array of numbers a[0…n-1]

output: a sorted version of this array

```
for endp=n-1 to 1:
        swapOP=0;
        for i=0 to endp-1:
                if a[i] > a[i+1]:  swap (a[i], a[i+1]);

return
```

endp=n: inner loop (for j=1 to endp-1) repeats for n-1 times

endp=n-1:   inner loop repeats for n-2 times

endp=n-2: inner loop repeats for n-3 times

…

endp=2: inner loop repeats for 1 times

Total # of comparison operation: $T(n) = (n-1)+(n-2)+(n-3)+\ldots+1$

How big is T(n)?

T(n) = (n-1)+(n-2)+(n-3)+…+1

Can you write big sigma notation for T(n)?

$$1 + 2 + 3 + ... + (n-2) + (n-1) = \Sigma_{i=1}^{n-1} i$$

Can you write simplified formula for T(n)?

$$1 + 2 + 3 + ... + (n-2) + (n-1) = \frac{n(n-1)}{2}, \text{ for } n \geq 2$$

Can you prove the above using math. induction?

1) when n=2, left hand size =1, right hand size is $\frac{2(2-1)}{2} = 1$

2) if the equation is true for n=k, then it's also true for n=k+1

Outline

- Resource usage of algorithms: space and time
- How to understand resource usage of algorithms?
    - Experimental approach
    - Running time Analysis: based upon code or pseudocode
        - count number of "computer steps"
        - write recursive formula for running time of recursive algorithm
- math help: math. induction

Running Time analysis

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

Analyze running time of recursive algorithm

- first, define T(n) to denote the running time (or total number of operations carried out) during fib1(n)'s execution
- write a recursive formula for T(n)
- then, either derive a closed formula for T(n) or come up some bound for T(n)

Our first exponential running time algorithm

- T(n): number of computer steps to compute fib1(n),
  - **T(0)=1**
  - **T(1)=2**
  - **T(n)=T(n-1)+T(n-2)+3, n>1**
- We can see for any n, T(n) > F_n
- Given $F_0=0$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$, we can prove

$$F_n \geq 2^{\frac{n}{2}} = 2^{0.5n}$$

- In fact, there is a tighter lower bound: T(n) >= $2^{0.694n}$
- So, running time of Fib1 grows exponentially fast:

$$T(n) > F_n \geq 2^{0.694n}$$

How many comparison operations?

```
SelectionSort_Recursive(l, first, last)
{
        if first==last
            return
        s=first
        for i=first+1 to last
            if l[i]<l[s]
                s=i
        if s!=first
             Swap (l[s],l[first])
        SelectionSort_Recursive(l,first+1, last)
}
```

Summary: Running time analysis

- For an input of size n, *how many total number of computer steps are executed?*
  - Size of input: size of an array, polynomial degree, # of elements in a matrix, vertices and edges in a graph, # of bits in the binary representation of input, …
  - Computer steps: arithmetic operations, data movement, control, decision making (if /then), comparison,…
- For non-recursive algorithm: unpack loops
- For recursive algorithm: Write recursive formula for T(n)