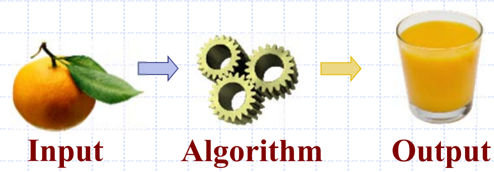
Ch 01. Analysis of Algorithms



Acknowledgement: Parts of slides in this presentation come from the materials accompanying the textbook Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

What's an Algorithm?

- Computer Science is about problem-solving using computers.
- Software is a solution to some problems.
- Algorithm is a recipe/design inside a software.
- Informally, an algorithm is

  a method for solving a well-specified computational problem.

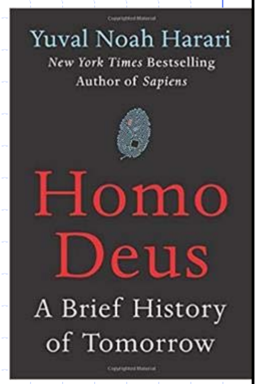
- Algorithms become more and more important in digital age.

Homo Deus: A Brief History of Tomorrow

A 2016 top seller book by Historian Yuval Noah Harari

Central thesis:

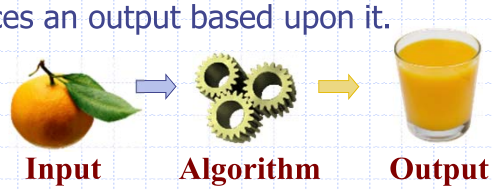
- Organisms are algorithms, and as such homo sapiens (today's human) may not be dominant in the future.
- Computers will do much better than organisms. Many professions will be out-of-date and labors become less worth.
- Harari believes that humanism will push humans to search for immortality, happiness, and power.
- Harari suggests the possibility of the replacement of humankind with a super-man, i.e. "homo deus", endowed with abilities such as eternal life and artificial intelligence.



3

Algorithms and Data Structures

- An **algorithm** is a step-by-step procedure for performing some task in a finite amount of time.
  - Typically, an algorithm takes input data and produces an output based upon it.



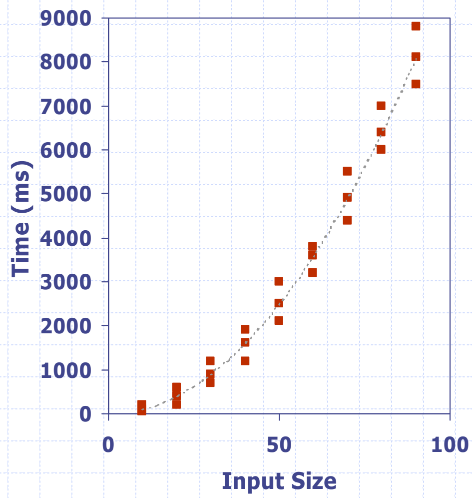
- A **data structure** is a systematic way of organizing and accessing data.

4

4

Experimental Studies of Algorithms

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:
- Plot the results



5

5

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.

6

6

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- High-level description of an algorithm
  - More structured than English prose
  - Less detailed than a real program
- Preferred notation for describing algorithms
- Easy map to real programming languages, or to primitive operations of CPU

Algorithm arrayMax(A, n):
 Input: An array A storing $n \geq 1$ integers.
 Output: The maximum element in A.

currentMax $\leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if *currentMax* $< A[i]$ **then**
   currentMax $\leftarrow A[i]$
return *currentMax*

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …

- Method call

  method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions:

  $\leftarrow$ Assignment

  $=$ Equality testing

  n^2 Superscripts and other mathematical formatting allowed

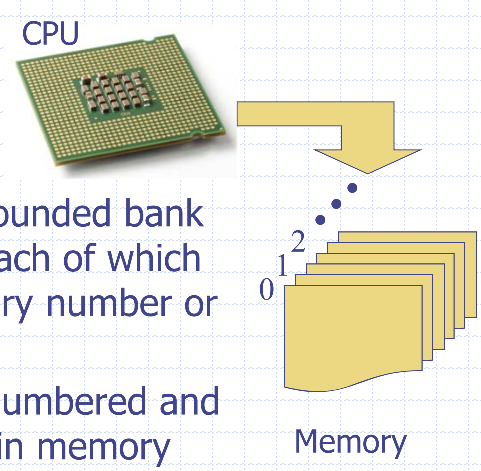
9

9

The Random Access Machine (RAM) Model

A **RAM** consists of

- A CPU
- An potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time



10

10

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Arithmetic operations
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

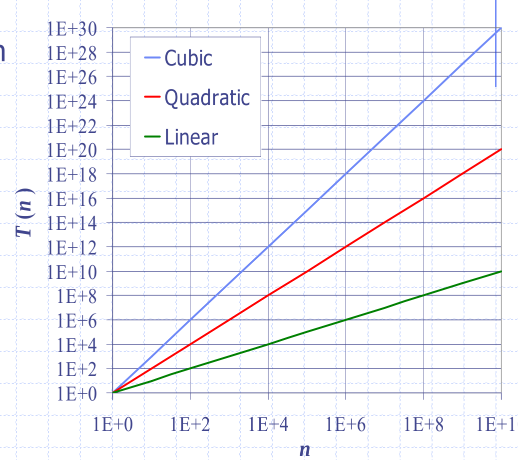
11

11

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ **1**
  - Logarithmic ≈ log ***n***
  - Linear ≈ ***n***
  - N-Log-N ≈ ***n*** log ***n***
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n

- In a log-log chart, the slope of the line corresponds to the growth rate

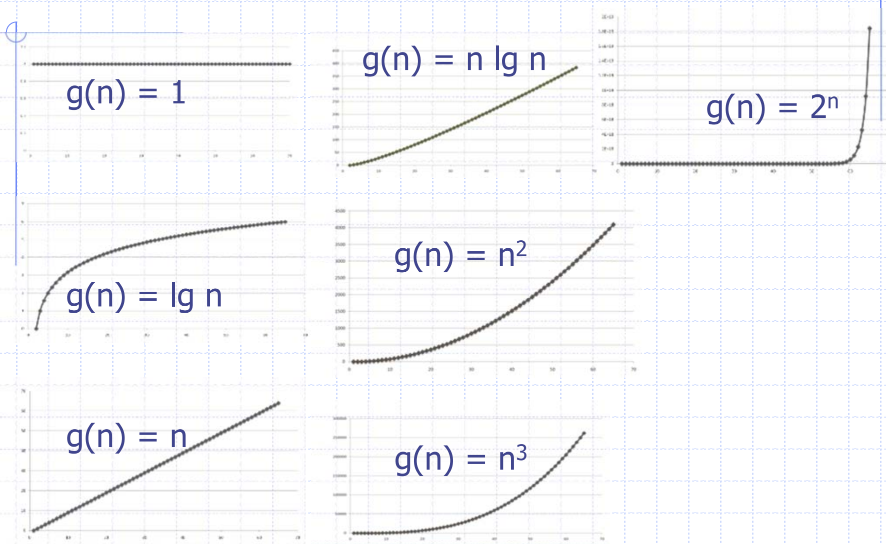


$n = 10^x$, $T(n) = 10^y$ ➔ $x = \log n$, $y = \log(T(n))$

12

12

Functions Graphed Using "Normal" Scale



13

Counting Primitive Operations

- Example: By inspecting the pseudocode, we can determine the minimum and maximum number of primitive operations executed by an algorithm, as a function of the input size

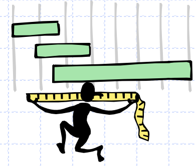
Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

Pseudocode	How many primitive operations at each line?
currentMax $\leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n-1$ **do**	3n-1
if *currentMax* $< A[i]$ **then**	2(n-1)
currentMax $\leftarrow A[i]$	0 to 2(n-1)
return *currentMax*	1

Minimum: 2 + 3n-1 + 2(n-1) + 1 = 5n
Maximum: 2 + 3n-1 + 4(n-1) + 1 = 7n - 2

14

Estimating Running Time



- Algorithm arrayMax executes $7n - 2$ primitive operations in the worst case, $5n$ in the best case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of arrayMax. Then

$$a(5n) \le T(n) \le b(7n - 2)$$

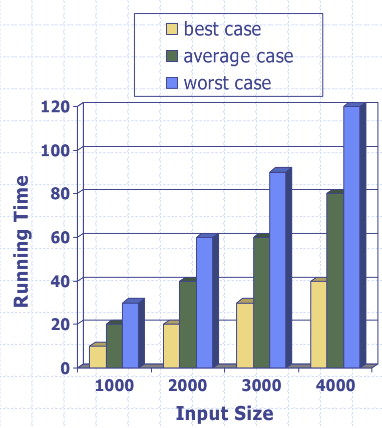
- Hence, the running time $T(n)$ is bounded by two linear functions

15

15

Running Time

- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus primarily on the **worst case running time**.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

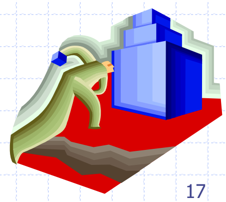


16

16

Growth Rate of Running Time

- Changing the hardware/software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax



17

17

Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
$c\ n^2$	$\sim c\ n^2 + 2c\ n$	**$4c\ n^2$**	$16c\ n^2$
$c\ n^3$	$\sim c\ n^3 + 3c\ n^2$	$8c\ n^3$	$64c\ n^3$
$c\ 2^n$	$c\ 2^{n+1}$	$c\ 2^{2n}$	$c\ 2^{4n}$

runtime quadruples when problem size doubles

18

18

Analyzing Recursive Algorithms

- Use a function, T(*n*), to derive a **recurrence relation** that characterizes the running time of the algorithm in terms of smaller values of *n*.

Algorithm recursiveMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.
if $n = 1$ **then**
  return $A[0]$
return $\max\{$recursiveMax$(A, n-1),\ A[n-1]\}$

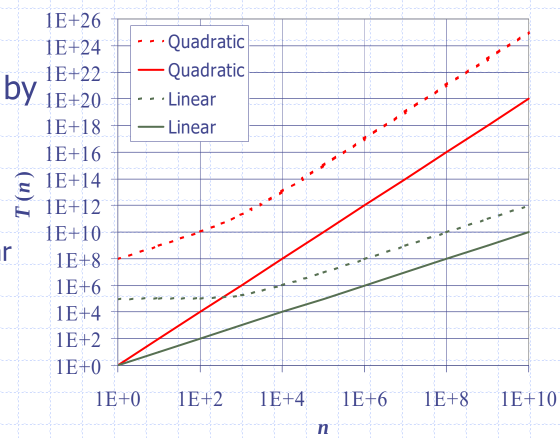
$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise,} \end{cases}$$

19

19

Constant Factors

- The growth rate is minimally affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^2 n^2 + 10^5 n$ is a quadratic function



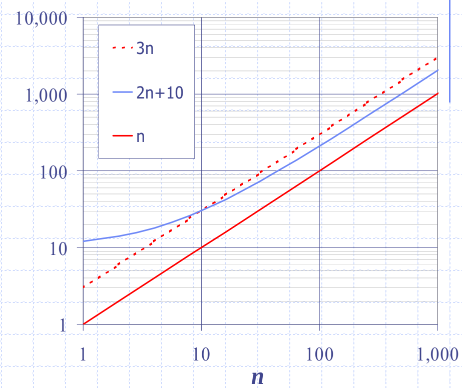
20

20

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \le cg(n)$ for $n \ge n_0$
- We also say $g(n)$ is an ***asymptotic upper bound*** for $f(n)$.



Example: $2n + 10$ is $O(n)$

$2n + 10 \le cn$

$(c - 2)\, n \ge 10$

$n \ge 10/(c - 2)$

Pick $c = 3$ and $n_0 = 10$

21

21

Relatives of Big-Oh

big-Omega

- f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \ge 1$ such that

  $f(n) \ge c\, g(n)$ for $n \ge n_0$

big-Theta

- f(n) is Θ(g(n)) if there are constants c′ > 0 and c″ > 0 and an integer constant $n_0 \ge 1$ such that

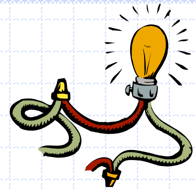
  $c'g(n) \le f(n) \le c''g(n)$ for $n \ge n_0$

Theorem: Θ is an equivalence relation.
(reflexive, symmetric, and transitive)

22

22

Intuition for Asymptotic Notation



big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)

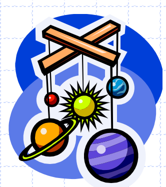
big-Omega
- f(n) is Ω(g(n)) if f(n) is asymptotically greater than or equal to g(n)

big-Theta
- f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)

23

23

Example Uses of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$
- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$
- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c\, g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

24

24

Big-Oh, Big-Theta, Big Omega Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(n)$"

25

25

Examples

$\Theta(n^3)$:

n^3

$5n^3 + 4n$

$105n^3 + 4n^2 + 6n$

$\Theta(n^2)$:

n^2

$5n^2 + 4n + 6$

$n^2 + 5$

$\Theta(\log n)$:

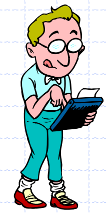
$\log n$

$\log n^2$

$\log (n + n^3)$

26

Math you need to Review



- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:
  - $a^{(b+c)} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $a^b / a^c = a^{(b-c)}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c \log_a b}$
- Properties of logarithms:
  - $\log_b(xy) = \log_b x + \log_b y$
  - $\log_b (x/y) = \log_b x - \log_b y$
  - $\log_b x^a = a \log_b x$
  - $\log_b a = \log_x a / \log_x b$

27

27

Functions in the order of faster growth rate

- c_0, $(\log n)^{c_1}$, n^{c_2}, c_3^n
  - c_0, c_1, c_2, are positive constants;
  - c_3 is a constant greater than 1.

28

28

Little oh

f(n) grows slower than g(n) (or g(n) grows faster than f(n)) if

$$\lim_{n \to \infty} (f(n) / g(n)) = 0,$$

Notation: $f(n) = o(g(n))$
pronounced "little oh"

29

Little omega

f(n) grows faster than g(n) (or g(n) grows slower than f(n)) if

$$\lim_{n \to \infty} (f(n) / g(n)) = \infty,$$

Notation: $f(n) = \omega (g(n))$
pronounced "little omega"

30

Relation Summary:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} \infty & \Rightarrow f(n) = \omega(g(n)) \\ C & \Rightarrow f(n) = \Theta(g(n)) \\ 0 & \Rightarrow f(n) = o(g(n)) \end{cases}$$

$f(n) = \omega(g(n))$ → $f(n) = \Omega(g(n))$

$f(n) = \Theta(g(n))$ → $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$

$f(n) = o(g(n))$ → $f(n) = O(g(n))$

Example: Which function grows faster?

$(\log n)^n$ and $n^{\log n}$

Example: Some functions are not comparable asymptotically.

$f(n) = n(1 - \sin(90^o n))$

$g(n) = n(1 - \cos(90^o n))$

31

Possible Quiz Problem

Decide the asymptotical relation of the following function pairs f and g, i.e., $f = O(g)$, or $f = \Omega(g)$, or both?

- $f = 10n^2 + n(\log n)$, $g = 100n(\log n)^2$
- $f = 100n + 3n^{2.5}$, $g = n^2(\log n)$

32

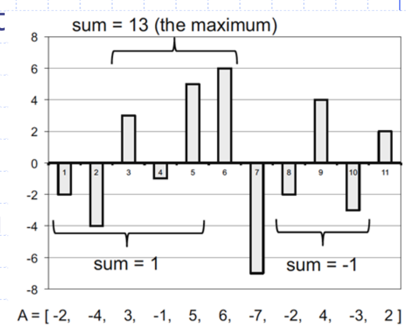
32

A Case Study in Algorithm Analysis

- Given an array of *n* integers, find the subarray, A[j..k] that maximizes the sum

$$s_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i.$$

- In addition to being an interview question for testing the thinking skills of job candidates, this **maximum subarray problem** also has applications in pattern analysis in digitized images.



33

A First (Slow) Solution

Compute the maximum of every possible subarray summation A[j, k] of the array *A* separately.

Algorithm MaxsubSlow(A):
  Input: An n-element array A of numbers, indexed from 1 to n.
  Output: The maximum subarray sum of array A.

```
m ← 0    // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s ← 0    // the next partial sum we are computing
        for i ← j to k do
            s ← s + A[i]
        if s > m then
            m ← s
return m
```

- The outer loop, for index j, will iterate n times, its middle-inner loop, for index k, will iterate j ~ n times, and the inner-most loop, for index i, will iterate j ~ k times.
- Thus, the running time of the MaxsubSlow algorithm is $O(n^3)$.

34

An Improved Algorithm

- A more efficient way to calculate these summations is to consider **prefix sums**

$$S_t = a_1 + a_2 + \cdots + a_t = \sum_{i=1}^{t} a_i$$

- If we are given all such prefix sums (and assuming S_0=0), we can compute any summation $s_{j,k}$ in constant time as

$$s_{j,k} = S_k - S_{j-1}$$

Example:

i =	0	1	2	3	4	5	6	7	8	9	10	11
A=		-2	-4	3	-1	5	6	-7	-2	4	-3	2
S=	0	-2	-6	-3	-4	1	7	0	-2	2	-1	1

Max = $s_{6,3} = S_6 - S_2 = 7 - (-6) = 13$.

35

An Improved Algorithm, cont.

- Compute all the prefix sums -- O(n), time and space
- Then compute all the subarray sums -- O(n^2)

Algorithm MaxsubFaster(A):
    Input: An n-element array A of numbers, indexed from 1 to n.
    Output: The maximum subarray sum of array A.

```
S_0 ← 0      // the initial prefix sum
for i ← 1 to n do                          i:  n iterations
    S_i ← S_{i-1} + A[i]
m ← 0        // the maximum found so far
for j ← 1 to n do                          j:  n iterations
    for k ← j to n do                      k:  j ~ n iterations
        s = S_k − S_{j−1}
        if s > m then
            m ← s
return m
```

36

36

A Linear-Time Algorithm

- Instead of computing prefix sum $S_t = s_{1,t}$, let us compute a maximum suffix sum, M_t, which is the maximum of any subarray (including the empty one) ending at t:

$$M_t = \max\{0, \max_{j=1,\cdots,t}\{s_{j,t}\}\}$$

- If $M_t > 0$, then it is the summation value for a maximum subarray that ends at t, and if $M_t = 0$, then we can safely ignore any subarray that ends at t.
- If we know all the M_t values, for t = 1, 2, …, n, then the solution to the maximum subarray problem would simply be the maximum of all these values.

37

37

A Linear-Time Algorithm, cont.

- If t = 0, then $M_t = 0$.
- For t ≥ 1, to compute M_t, the maximum subarray that ends at t, we can add A[t] to M_{t-1}. If the result is a positive sum, then we are done; if it is negative, we let M_t be 0, i.e., take the empty subarray, for there is no non-empty subarray that ends at t with a positive summation.
- So we can define $M_0 = 0$ and recursively

$$M_t = \max\{0, M_{t-1} + A[t]\}$$

Example:

t =	0	1	2	3	4	5	6	7	8	9	10	11
A=		-2	-4	3	-1	5	6	-7	-2	4	-3	2
M=	0	0	0	3	2	7	13	6	4	8	5	7

Max = M_6 = 13.

38

38

A Linear-Time Algorithm, cont.

Algorithm MaxsubFastest(A):
  Input: An n-element array A of numbers, indexed from 1 to n.
  Output: The maximum subarray sum of array A.

$M_0 \leftarrow 0$ // the initial prefix maximum
for $t \leftarrow 1$ **to** n **do**
  $M_t \leftarrow \max\{0, M_{t-1} + A[t]\}$
$m \leftarrow 0$ // the maximum found so far
for $t \leftarrow 1$ **to** n **do**
  $m \leftarrow \max\{m, M_t\}$
return m

- The MaxsubFastest algorithm consists of two loops, which each iterate exactly n times and take O(1) time in each iteration. Thus, the total running time of the MaxsubFastest algorithm is O(n), time and space.

Possible Quiz Problem

Algorithm MaxsubFastest(A):
  Input: An n-element array A of numbers, indexed from 1 to n.
  Output: The maximum subarray sum of array A.

$M_0 \leftarrow 0$ // the initial prefix maximum
for $t \leftarrow 1$ **to** n **do**
  $M_t \leftarrow \max\{0, M_{t-1} + A[t]\}$
$m \leftarrow 0$ // the maximum found so far
for $t \leftarrow 1$ **to** n **do**
  $m \leftarrow \max\{m, M_t\}$
return m

- How to use only a constant number of space, instead of storing M_t for all t?
- How to find the values of j and k if $A[j, k]$ contains the maximum of every possible subarray summation of the array A **in linear time?**

Summations

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \cdots + f(n-1) + f(n)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6}$$

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \text{ for } a > 1$$

41

Summations

$$\sum_{i=1}^{n} 1/i = O(\ln n)$$

using Integral of 1/x.

$$\sum_{i=1}^{n} \log i = O(n \log n)$$

using Stirling's approximation

$$n! \approx \sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n$$

42

The Factorial Function

Definition:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

Stirling's approximation:

$$n! \approx \sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n$$

or log(n!) = O(n log n)

43

Bounds of Factorial Function

Let $\log n! = \sum_{x=1}^{n} \log x.$

then

$$\int_1^n \log x \, dx \le \sum_{x=1}^{n} \log x \le \int_0^n \log(x+1)\, dx$$

which gives

$$n \log\left(\frac{n}{e}\right) + 1 \le \log n! \le (n+1)\log\left(\frac{n+1}{e}\right) + 1.$$

So

$$e\left(\frac{n}{e}\right)^n \le n! \le e\left(\frac{n+1}{e}\right)^{n+1}.$$

Similar to

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

44

Average Case Analysis

- In worst case analysis of time complexity we select the maximum cost among all possible inputs of size n.
- In average case analysis, the running time is taken to be the average time over all inputs of size n.
  - Unfortunately, there are infinite inputs.
  - It is necessary to know the probabilities of all input occurrences.
  - The analysis is in many cases complex and lengthy.

45

What is the average case of executing "*currentMax* ← *A*[*i*]"?

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

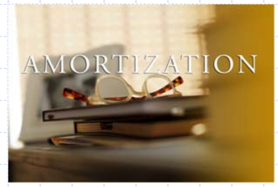
$currentMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $currentMax < A[i]$ **then**
    $currentMax \leftarrow A[i]$
return $currentMax$

Number of Assignments: the worst case is n. If numbers are randomly distributed, then the average case is 1+1/2 + 1/3 + 1/4 + … + 1/n = O(log n).

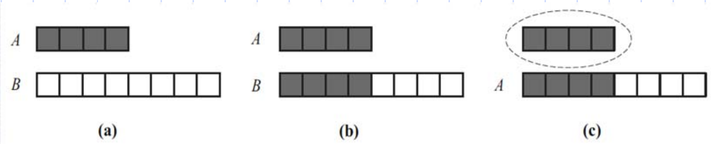
This is because A[i] has only 1/i probability to be the max of A[1], A[2], …, A[i], under the assumption that all numbers are randomly distributed.

46

Amortized Analysis



- The **amortized running time** of an operation within a series of operations is the worst-case running time of the series of operations divided by the number of operations.
- Example: A growable array, S. When needing to grow:
  - a. Allocate a new array B of larger capacity.
  - b. Copy A[i] to B[i], for i = 0, . . . , n − 1, where n is size of A.
  - c. Let A = B, that is, we use B as the array now supporting A.



Dynamic Array Description

- Let add(e) be the operation that adds element e at the end of the array
- When the array is full, we replace the array with a larger one
- But how large should the new array be?
  - Incremental strategy: increase the size by a constant *c*
  - Doubling strategy: double the size

```
Algorithm add(e)
  if n = A.length then
      B ← new array of
          size …
      for i ← 0 to n−1 do
          B[i] ← A[i]
      A ← B
  n ← n + 1
  A[n−1] ← e
```

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add operations
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of an add operation the average time taken by an add operation over the series of operations, i.e., $T(n)/n$.

49

49

Incremental Strategy Analysis

- Over n add operations, we replace the array $k = n/c$ times, where c is a constant
- The total time $T(n)$ of a series of n add operations is proportional to

$$T(n) = n + c + 2c + 3c + 4c + \dots + kc$$
$$= n + c(1 + 2 + 3 + \dots + k)$$
$$= n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of an add operation, $T(n)/n$, is $O(n)$.

50

50

Doubling Strategy Analysis: The Aggregate Method

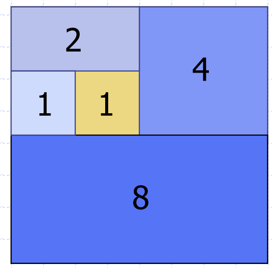
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

  $n + 1 + 2 + 4 + 8 + \ldots + 2^k =$

  $n + 2^{k+1} - 1 =$

  $3n - 1$
- $T(n)$ is $O(n)$.
- The amortized time of an add operation is $O(1)$.

geometric series



51

51

Doubling Strategy Analysis: The Accounting Method

- We view the computer as a coin-operated appliance that requires one **cyber-dollar** for a constant amount of computing time.
- For this example, we shall pay each add operation 3 cyber-dollars.
- Set a saving account with $s_0 = 0$ initially.
- The i^{th} operation has a budget cost of $a_i = 3$, which is the amortized cost of each operation.
- The account value after the i^{th} add operation is

  $s_i = s_{i-1} + a_i - c_i$ where c_i is the actual cost.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
Array size	1	2	4		8				16								32				
c_i	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	1	1	...
s_i	2	3	3	5	3	5	7	9	3	5	7	9	11	15	17	19	5	7	9	11	...

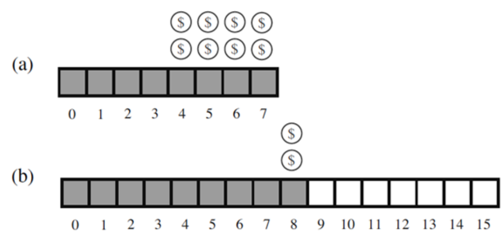
Note: the account value s_i never goes under 0.

52

52

Doubling Strategy Analysis: The Accounting Method

- We shall pay each add operation a_i= 3 cyber-dollars, that is, it will have an amortized O(1) amortized running time.
  - We over-pay each add operation not causing an overflow 2 cyber-dollars.
  - An overflow occurs when the array A has 2^i elements.
  - Thus, doubling the size of the array will require 2^i cyber-dollars.
  - These cyber-dollars are at the elements stored in cells 2^{i-1} through 2^i-1.



53

Possible Quiz Problem

For dynamic arrays, instead of doubling the size of the current array, the current array size is tripled when it is full. What will be amortized cost of *add*(e)?

What will is the answer when the new array size is only 50% more than the current array?

54

54

Summary

- Worst-case complexity: Given an upper bound at the worst case
- Average complexity: Assume a probability distribution of all inputs, give the complexity under this distribution.
- Amortized complexity: Compute the worst case of the sum of a sequence of operations, and then divide it by the number of operations.