Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University

a.khan@innopolis.ru

Recap

- MAP ADT
- Hashmap
- Time complexity of a hashmap

Objectives

- What is an algorithmic strategy?

- Learn about commonly used Algorithmic Strategies
  - Brute-force
  - Divide-and-conquer
  - Master Theorem

- You will also see an example of how classical algorithmic problems can appear in daily life

Algorithm Classification

- Based on problem domain
- Based on algorithmic strategy

Algorithmic Strategies

- Approach to solving a problem
- Algorithms that use a similar problem solving approach can be grouped together
- Classification scheme for algorithms

Brute Force

Brute Force

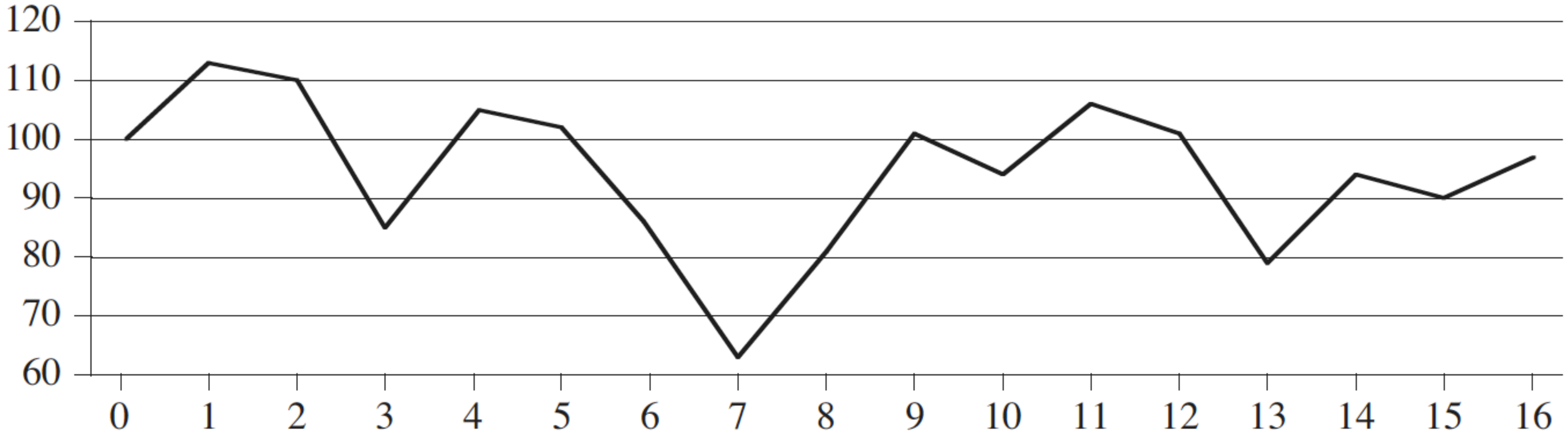
- Straightforward approach to solving a problem based on the simple formulation of the problem
- Often, does not require deep analysis of the problem
- Perhaps the easiest approach to apply and is useful for solving problems of small-size
- May results in naïve solutions with poor performance

Example Algorithmic Problem

- Maximum subarray problem
  - Given a sequence of integers $i_1, i_1, \dots, i_n$ find the sub-sequence with the maximum sum
  - Example:
    - 1, -3, 4, -2, -1, 6   gives the solution ?

Max Subarray in Real-life

- Information about the price of stock in a Chemical manufacturing company after the close of trading over a period of 17 days

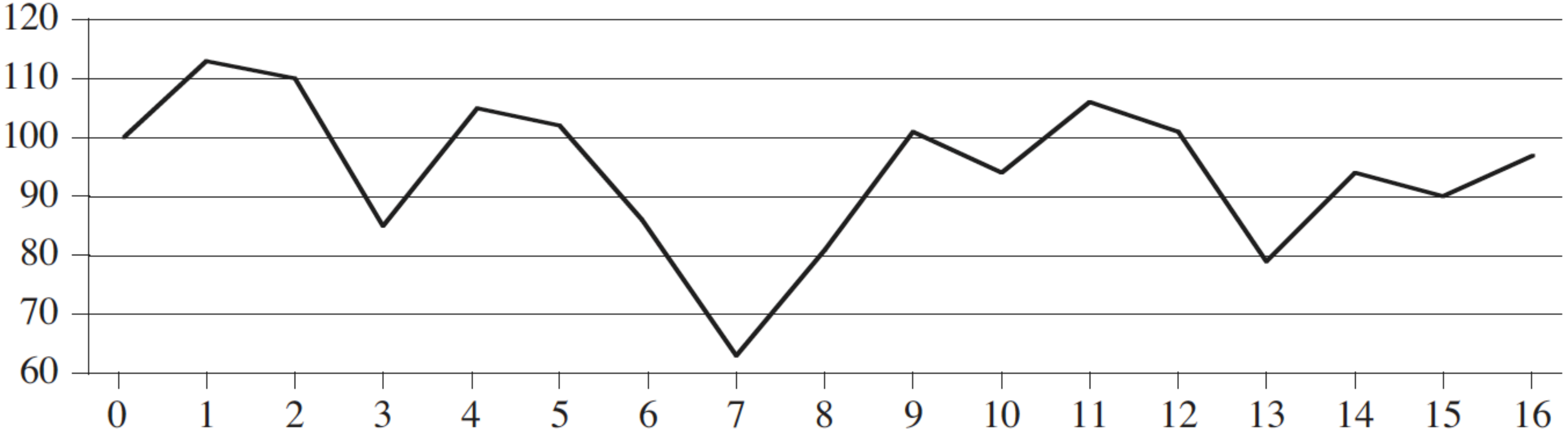


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

- Goal: When to buy the stock and when to sell it to maximize the profit?

Max Subarray in Real-life

- Transformation to convert this problem into the max-subarry problem



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- When to buy the stock and when to sell it to maximize the profit? *Now we can answer this by finding the sequence of days over which the net change is maximum*

Brute Force Approach to Our Problem

- We can easily devise a brute-force solution to this problem
- Begin with a max-sum of 0
- For each index in the sequence
  - Compute the sum for sequences of all lengths and compare them with the max-sum
  - Update max-sum if you find a sum that is greater than max-sum
- Time Complexity ?

Brute Force

- The most straightforward and the easiest of all approach
- Often, does not required deep analysis of the problem
- May results in naïve solutions with poor performance, but easy to implement

Divide-and-Conquer

Divide-and-Conquer

- Solving a problem *recursively*, applying three steps at each level of *recursion*
  - **Divide** the problems into a number a sub-problems that are smaller instances of the same problem
  - **Conquer** the sub-problems by solving them recursively. If the sub-problems size is small enough, just solve it in a straightforward manner
  - **Combine** the solutions to the sub-problems into the solution for the original problem

Recursion and Recurrence Relations

- Recursion

  - A function is said to be recursive if it calls itself – usually with "*smaller or simpler*" inputs

  - Two properties:
    a) A problem should be solvable by utilizing the solutions to the smaller versions of the same problem,
    b) The smaller versions should reduce to easily solvable cases

Recursion Example

```
long power(long x, long n)
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
```

Recursion Example

```
long power(long x, long n)
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
```

Base case!

Recursive case!

Recursion and Recurrence Relations

- Recurrence Relations
  - Are used to determine the running time of recursive algorithms
  - Let $T(n)$ = Time required to solve the problem of size n

$$T(0) = \text{time to solve problem of size 0 – Base Case}$$

$$T(n) = \text{time to solve problem of size } n \text{ – Recursive Case}$$

Recursion and Recurrence Relations

- Recurrence Relations

```
long power(long x, long n)
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
```

$$T(0) = c_1 \qquad \text{for some constant } c_1$$
$$T(n) = c_2 + T(n-1) \qquad \text{for some constant } c_2$$

Recursion and Recurrence Relations

- Recurrence Relations

$$T(0) = c_1$$
$$T(n) = T(n-1) + c_2$$

If we knew $T(n-1)$, we could solve $T(n)$.

Recursion and Recurrence Relations

- Recurrence Relations

$$T(0) = c_1$$
$$T(n) = T(n-1) + c_2$$

If we knew $T(n-1)$, we could solve $T(n)$.

$$\begin{aligned} T(n) &= T(n-1) + c_2 & \qquad T(n-1) = T(n-2) + c_2 \\ &= T(n-2) + c_2 + c_2 & \\ &= T(n-2) + 2c_2 & \qquad T(n-2) = T(n-3) + c_2 \\ &= T(n-3) + c_2 + 2c_2 & \\ &= T(n-3) + 3c_2 & \qquad T(n-3) = T(n-4) + c_2 \\ &= T(n-4) + 4c_2 & \\ &= \ldots & \\ &= \boxed{T(n-k) + kc_2} & \end{aligned}$$

Recursion and Recurrence Relations

- Recurrence Relations

$$T(0) = c_1$$
$$T(n) = T(n-k) + k * c_2 \quad \text{for all } k$$

If we set $k = n$, we have:

$$\begin{aligned} T(n) &= T(n-n) + nc_2 \\ &= T(0) + nc_2 \\ &= c_1 + nc_2 \end{aligned}$$

Solving Max-SubArray with Divide-and-Conquer

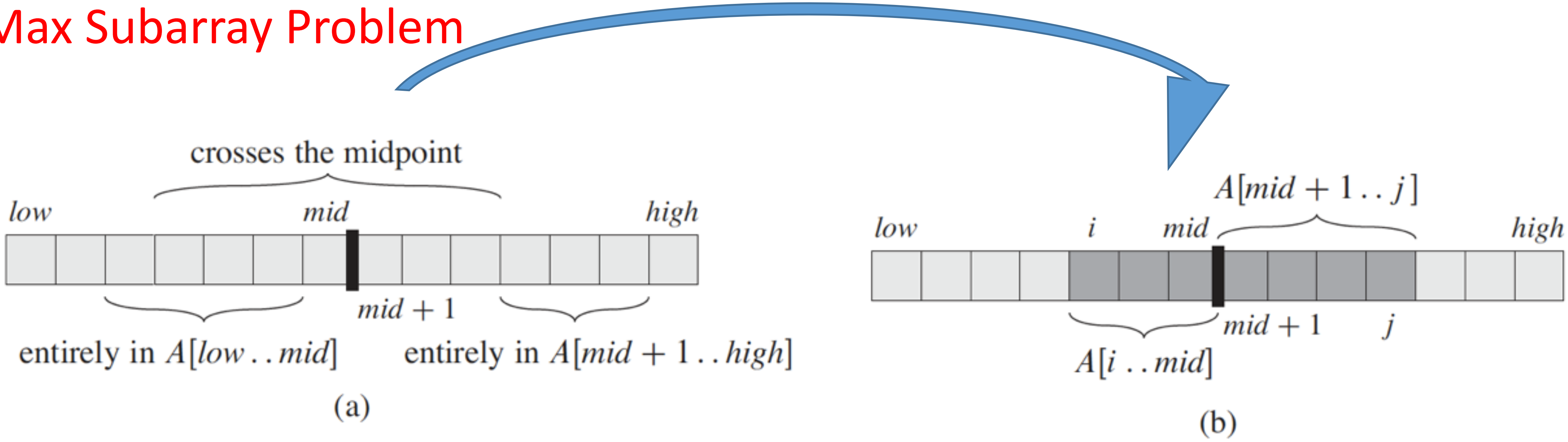
Divide-and-Conquer

- Max Subarray Problem

    - *Divide the problem into sub-problems*
    - *Solve the sub problems*
    - *Merge the solution of the sub problems*

Divide-and-Conquer

- Max Subarray Problem



- The solution lies in exactly one of the following places

entirely in the subarray $A[low \,..\, mid]$, so that $low \le i \le j \le mid$,

entirely in the subarray $A[mid + 1 \,..\, high]$, so that $mid < i \le j \le high$, or

crossing the midpoint, so that $low \le i \le mid < j \le high$.

Divide-and-Conquer

- Max Subarray Problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

- Max Subarray Problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

- Max Subarray Problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

- Max Subarray Problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

- Max Subarray Problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])            // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide

- Max Subarray Problem

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

```
FIND-MAXIMUM-SUBA
if high == low
    return (low, hi
else mid = ⌊(low -
    (left-low, left-h
        FIND-MA
    (right-l
        FIND-MA
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Example:

mid =5

	1	2	3	4	5	6	7	8	9	10
A	13	-3	-25	20	-3	-16	-23	18	20	-7

$S[5 .. 5] =$ -3

$S[4 .. 5] =$ 17 $\Leftarrow$ (max-left = 4)

$S[3 .. 5] =$ -8

$S[2 .. 5] =$ -11

$S[1 .. 5] =$ 2

mid =5

	1	2	3	4	5	6	7	8	9	10
A	13	-3	-25	20	-3	-16	-23	18	20	-7

$S[6 .. 6] =$ -16

$S[6 .. 7] =$ -39

$S[6 .. 8] =$ -21

$S[6 .. 9] =$ (max-right = 9) $\Rightarrow$ -1

$S[6..10] =$ -8

$\Rightarrow$ maximum subarray crossing *mid* is $S[4..9] = 16$

Divide-and-Conquer

- Max Subarray Problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])            // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

- Max Subarray Problem – Time Complexity

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

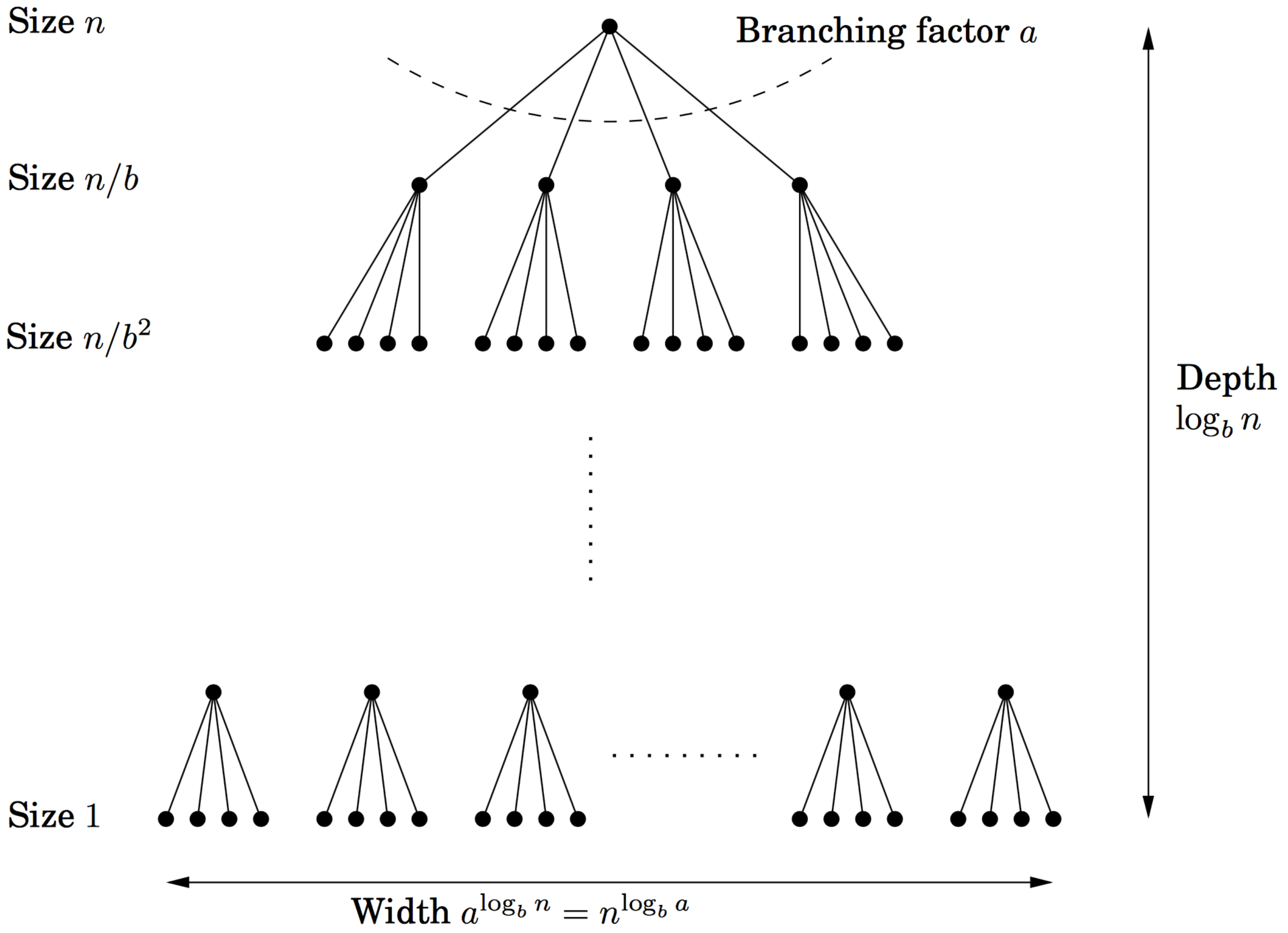
- This type of recurrence is called "*Divide-and-Conquer*" recurrence

We can solve this recurrence using the **"Master Theorem"** --
Cormen's, Chapter 4

Divide-and-Conquer

- Master Theorem

$$T(n) = \boxed{aT(n/b)} + \boxed{f(n)}$$



Divide-and-Conquer

- Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Divide-and-Conquer

- Max Subarray Problem – Time Complexity

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

$f(n)$

$a = 2$

$b = 2$

$log_b a = ?$

Divide-and-Conquer

- Max Subarray Problem – Time Complexity

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

$a = 2$
$b = 2$
$log_b a = 1$

- Which case of Master Theorem applies?

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Divide-and-Conquer

- Max Subarray Problem – Time Complexity

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \,. \end{cases}$$

- Case 2 from Master Theorem applies, thus we have the solution

$$T(n) = \Theta(n \lg n) .$$

Master Theorem

- You will get back to it in your tutorial today
- With some examples