UIT2201 Programming and Data Structures

Analysis of Algorithms

Chandrabose Aravindan
<AravindanC@ssn.edu.in>

Professor of Information Technology
SSN College of Engineering

May 11, 2022

Outline

1. Introduction

Outline

Outline

Outline

Outline

Outline

Introduction

- $T(n)$: Time / Steps taken by an algorithm for input of size n
- $S(n)$: Space (memory cells) required by an algorithm for input of size n

Introduction

- $T(n)$: Time / Steps taken by an algorithm for input of size n
- $S(n)$: Space (memory cells) required by an algorithm for input of size n
- It does not make much sense to express $T(n)$ is standard time units (Why?)

Introduction

- $T(n)$: Time / Steps taken by an algorithm for input of size n
- $S(n)$: Space (memory cells) required by an algorithm for input of size n
- It does not make much sense to express $T(n)$ is standard time units (Why?)
- Does the time taken by an algorithm depend on the input size n alone?

Introduction

- $T(n)$: Time / Steps taken by an algorithm for input of size n
- $S(n)$: Space (memory cells) required by an algorithm for input of size n
- It does not make much sense to express $T(n)$ is standard time units (Why?)
- Does the time taken by an algorithm depend on the input size n alone?
- What would be useful to us is to talk about the "growth rate" of $T(n)$
- And express that growth rate in terms of known simple functions — $T(n) \propto f(n)$

Introduction

- $T(n)$: Time / Steps taken by an algorithm for input of size n
- $S(n)$: Space (memory cells) required by an algorithm for input of size n
- It does not make much sense to express $T(n)$ is standard time units (Why?)
- Does the time taken by an algorithm depend on the input size n alone?
- What would be useful to us is to talk about the "growth rate" of $T(n)$
- And express that growth rate in terms of known simple functions — $T(n) \propto f(n)$
- Normally we consider the time / steps taken in the worst-case

Introduction

- $T(n)$: Time / Steps taken by an algorithm for input of size n
- $S(n)$: Space (memory cells) required by an algorithm for input of size n
- It does not make much sense to express $T(n)$ is standard time units (Why?)
- Does the time taken by an algorithm depend on the input size n alone?
- What would be useful to us is to talk about the "growth rate" of $T(n)$
- And express that growth rate in terms of known simple functions — $T(n) \propto f(n)$
- Normally we consider the time / steps taken in the worst-case
- Other analysis include best-case and average-case

Introduction

- $T(n)$: Time / Steps taken by an algorithm for input of size n
- $S(n)$: Space (memory cells) required by an algorithm for input of size n
- It does not make much sense to express $T(n)$ is standard time units (Why?)
- Does the time taken by an algorithm depend on the input size n alone?
- What would be useful to us is to talk about the "growth rate" of $T(n)$
- And express that growth rate in terms of known simple functions — $T(n) \propto f(n)$
- Normally we consider the time / steps taken in the worst-case
- Other analysis include best-case and average-case
- Amortized Analysis: Worst-case analysis of a sequence of operations — cost for individual operation is then amortized — total cost divided by number of operations

Example: Linear Search

- Let us consider the linear search algorithm as an example

Example: Linear Search

- Let us consider the linear search algorithm as an example

```
def linsearch(obj, lst):
    index = 0
    while (index < len(lst) and lst[index] != obj):
        index += 1
    return index
```

Example: Linear Search

- Let us consider the linear search algorithm as an example

```
def linsearch(obj, lst):
    index = 0
    while (index < len(lst) and lst[index] != obj):
        index += 1
    return index
```

- How many comparisons are made in the worst case?

Example: Linear Search

- Let us consider the linear search algorithm as an example

```
def linsearch(obj, lst):
    index = 0
    while (index < len(lst) and lst[index] != obj):
        index += 1
    return index
```

- How many comparisons are made in the worst case? $T(n) = n$

Example: Linear Search

- Let us consider the linear search algorithm as an example

```
def linsearch(obj, lst):
    index = 0
    while (index < len(lst) and lst[index] != obj):
        index += 1
    return index
```

- How many comparisons are made in the worst case? $T(n) = n$
- How many comparisons are made in the best case?

Example: Linear Search

- Let us consider the linear search algorithm as an example

```
def linsearch(obj, lst):
    index = 0
    while (index < len(lst) and lst[index] != obj):
        index += 1
    return index
```

- How many comparisons are made in the worst case? $T(n) = n$
- How many comparisons are made in the best case? $T(n) = 1$

Example: Linear Search

- Let us consider the linear search algorithm as an example

```
def linsearch(obj, lst):
    index = 0
    while (index < len(lst) and lst[index] != obj):
        index += 1
    return index
```

- How many comparisons are made in the worst case? $T(n) = n$
- How many comparisons are made in the best case? $T(n) = 1$
- How about average case?

Example: Linear Search

- Let p be the probability that the object is present in the list.

Example: Linear Search

- Let p be the probability that the object is present in the list. If it is present, then it has equal probability to be present at any index.

Example: Linear Search

- Let p be the probability that the object is present in the list. If it is present, then it has equal probability to be present at any index.

$$T(n) = p\left[1.\frac{1}{n} + 2.\frac{1}{n} + \cdots + n.\frac{1}{n}\right] + (1-p).n$$

- Let p be the probability that the object is present in the list. If it is present, then it has equal probability to be present at any index.

$$T(n) = p\left[1.\frac{1}{n} + 2.\frac{1}{n} + \cdots + n.\frac{1}{n}\right] + (1-p).n$$

$$T(n) = \frac{p}{n}.\left[1 + 2 + \cdots + n\right] + (1-p).n$$

Example: Linear Search

- Let p be the probability that the object is present in the list. If it is present, then it has equal probability to be present at any index.

$$T(n) = p\left[1.\frac{1}{n} + 2.\frac{1}{n} + \cdots + n.\frac{1}{n}\right] + (1-p).n$$

$$T(n) = \frac{p}{n}.\left[1 + 2 + \cdots + n\right] + (1-p).n$$

$$T(n) = \frac{p}{n}.\frac{n.(n+1)}{2} + (1-p).n$$

Example: Linear Search

- Let p be the probability that the object is present in the list. If it is present, then it has equal probability to be present at any index.

$$T(n) = p\left[1.\frac{1}{n} + 2.\frac{1}{n} + \cdots + n.\frac{1}{n}\right] + (1-p).n$$

$$T(n) = \frac{p}{n}.\left[1 + 2 + \cdots + n\right] + (1-p).n$$

$$T(n) = \frac{p}{n}.\frac{n.(n+1)}{2} + (1-p).n$$

$$T(n) = p.\frac{(n+1)}{2} + (1-p).n$$

Big-Oh Notation

Big-Oh Definition

$T(n)$ is $O(f(n))$ if there exist constants c and n_0 such that $T(n) \leq cf(n)$ $\forall n \geq n_0$

Big-Oh Example

- Suppose $T(n) = 3n^3 + 2n^2$

Big-Oh Example

- Suppose $T(n) = 3n^3 + 2n^2$
- Claim: For $n_0 = 1$ and $c = 5$, $3n^3 + 2n^2 \leq 5n^3 \quad \forall n \geq 1$

Big-Oh Example

- Suppose $T(n) = 3n^3 + 2n^2$
- Claim: For $n_0 = 1$ and $c = 5$, $3n^3 + 2n^2 \leq 5n^3 \quad \forall n \geq 1$
- Such a claim may be proved using Mathematical Induction

Big-Oh Example

- Suppose $T(n) = 3n^3 + 2n^2$
- Claim: For $n_0 = 1$ and $c = 5$, $3n^3 + 2n^2 \leq 5n^3 \quad \forall n \geq 1$
- Such a claim may be proved using Mathematical Induction
- If this claim is true, then the complexity (growth rate) can be expressed as $O(n^3)$

Big-Oh Example

- Suppose $T(n) = 3n^3 + 2n^2$
- Claim: For $n_0 = 1$ and $c = 5$, $3n^3 + 2n^2 \leq 5n^3 \quad \forall n \geq 1$
- Such a claim may be proved using Mathematical Induction
- If this claim is true, then the complexity (growth rate) can be expressed as $O(n^3)$
- Technically, we can also say that this $T(n)$ is $O(n^4)$!

Big-Oh Example

- Suppose $T(n) = 3n^3 + 2n^2$
- Claim: For $n_0 = 1$ and $c = 5$, $3n^3 + 2n^2 \leq 5n^3 \quad \forall n \geq 1$
- Such a claim may be proved using Mathematical Induction
- If this claim is true, then the complexity (growth rate) can be expressed as $O(n^3)$
- Technically, we can also say that this $T(n)$ is $O(n^4)$!
- But, that is a weak statement, and it is understood that we need to find the "least upper bound"

Big-Oh Example

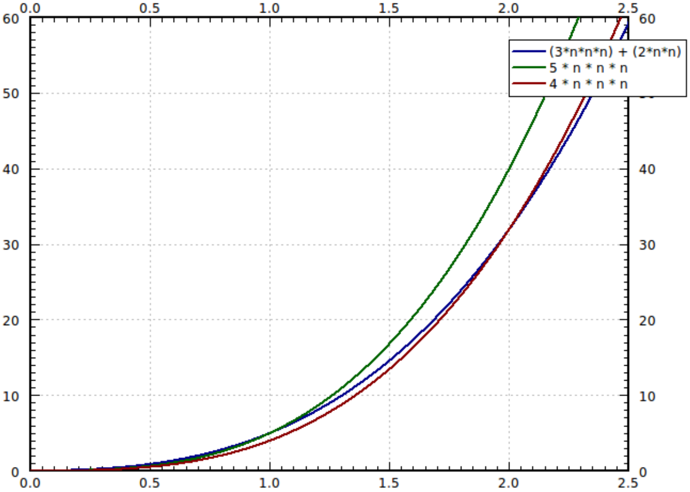

Figure: Big-Oh Illustration

Big-Omega Notation

Big-Omega Definition

$T(n)$ is $\Omega(g(n))$ if there exists a positive constant c such that $T(n) \geq cg(n)$ infinitely often

Big-Omega Notation

Big-Omega Definition

$T(n)$ is $\Omega(g(n))$ if there exists a positive constant c such that $T(n) \geq cg(n)$ infinitely often

- Like in the case of Big-Oh, $g(n)$ is expected to be a tight lower bound for $T(n)$

Big-Theta Notation

Big-Theta Definition

$T(n)$ is $\Theta(f(n))$ if there exist positive constants c_1, c_2, n_0 such that $c_1 f(n) \leq T(n) \leq c_2 f(n)\ \forall n \geq n_0$

Big-Theta Notation

Big-Theta Definition

$T(n)$ is $\Theta(f(n))$ if there exist positive constants c_1, c_2, n_0 such that $c_1 f(n) \leq T(n) \leq c_2 f(n)\ \forall n \geq n_0$

- $f(n)$ provides both upper and lower bounds for $T(n)$, and hence Big-Theta is a much preferred notation

Big-Theta Notation

Big-Theta Definition

$T(n)$ is $\Theta(f(n))$ if there exist positive constants c_1, c_2, n_0 such that $c_1 f(n) \leq T(n) \leq c_2 f(n) \; \forall n \geq n_0$

- $f(n)$ provides both upper and lower bounds for $T(n)$, and hence Big-Theta is a much preferred notation
- In other words, for any $T(n)$ and $f(n)$, $T(n)$ is $\Theta(f(n))$ if and only if $T(n)$ is $O(f(n))$ and $T(n)$ is $\Omega(f(n))$

Big-Theta Example

- Consider $T(n) = \frac{n^2}{2} - 3n$

Big-Theta Example

- Consider $T(n) = \frac{n^2}{2} - 3n$
- This can be shown to be in $\Theta(n^2)$

Big-Theta Example

- Consider $T(n) = \frac{n^2}{2} - 3n$
- This can be shown to be in $\Theta(n^2)$

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$$

Big-Theta Example

- Consider $T(n) = \frac{n^2}{2} - 3n$
- This can be shown to be in $\Theta(n^2)$

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Big-Theta Example

- Consider $T(n) = \frac{n^2}{2} - 3n$
- This can be shown to be in $\Theta(n^2)$

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- Note that $\frac{1}{2} - \frac{3}{n}$ approaches $\frac{1}{2}$ as n grows larger, and hence $c_2 = \frac{1}{2}$

Big-Theta Example

- Consider $T(n) = \frac{n^2}{2} - 3n$
- This can be shown to be in $\Theta(n^2)$

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- Note that $\frac{1}{2} - \frac{3}{n}$ approaches $\frac{1}{2}$ as n grows larger, and hence $c_2 = \frac{1}{2}$
- Note that $\frac{1}{2} - \frac{3}{n}$ becomes positive $(\frac{1}{14})$ when $n = 7$. Hence, we can conclude that for any $n \geq 7$, $\frac{1}{2} - \frac{3}{n} \geq \frac{1}{14}$

Big-Theta Example

- Consider $T(n) = \frac{n^2}{2} - 3n$
- This can be shown to be in $\Theta(n^2)$

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- Note that $\frac{1}{2} - \frac{3}{n}$ approaches $\frac{1}{2}$ as n grows larger, and hence $c_2 = \frac{1}{2}$
- Note that $\frac{1}{2} - \frac{3}{n}$ becomes positive $(\frac{1}{14})$ when $n = 7$. Hence, we can conclude that for any $n \geq 7$, $\frac{1}{2} - \frac{3}{n} \geq \frac{1}{14}$
- Thus, by selecting $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$, and $n_0 = 7$, we can show that $T(n)$ is $\Theta(n^2)$

Big-Theta Example

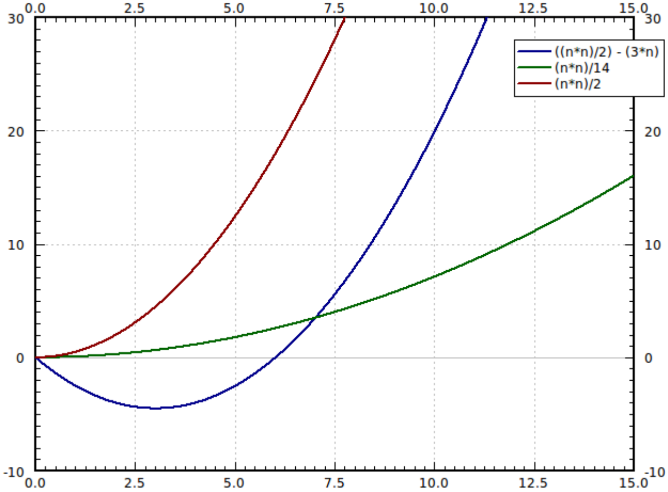

Figure: Big-Theta Illustration

Big-Theta Theorem

Theorem

Any polynomial $T(n) = \sum_{i=0}^{d} a_i n^i$ *with* $a_d > 0$ *is* $\Theta(n^d)$

Big-Theta Theorem

Theorem

Any polynomial $T(n) = \sum_{i=0}^{d} a_i n^i$ *with* $a_d > 0$ *is* $\Theta(n^d)$

Theorem

As a special case, when $d = 0$, $T(n)$ *is a constant and can be expressed as* $\Theta(1)$

Small Notations

small-oh Definition

$T(n)$ is $o(f(n))$ if for any positive constant $c > 0$, $\exists n_0$ such that $T(n) < cf(n)\ \forall n > n_0$

Small Notations

small-oh Definition

$T(n)$ is $o(f(n))$ if for any positive constant $c > 0$, $\exists n_0$ such that $T(n) < cf(n)\ \forall n > n_0$

small-omega Definition

$T(n)$ is $\omega(f(n))$ if for any positive constant $c > 0$, $\exists n_0$ such that $T(n) > cf(n)\ \forall n > n_0$

Typical Growth Rates

- $O(1)$
- $O(log\ n)$
- $O(log^2\ n)$
- $O(n)$
- $O(n\ log\ n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
- $O(n!)$

Typical Growth Rates

- $O(1)$ Constant Time
- $O(log\ n)$
- $O(log^2\ n)$
- $O(n)$
- $O(n\ log\ n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
- $O(n!)$

Typical Growth Rates

- $O(1)$ Constant Time
- $O(log\ n)$ Logarithmic Time
- $O(log^2\ n)$
- $O(n)$
- $O(n\ log\ n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
- $O(n!)$

Typical Growth Rates

- $O(1)$ Constant Time
- $O(log\ n)$ Logarithmic Time
- $O(log^2\ n)$
- $O(n)$ Linear Time
- $O(n\ log\ n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
- $O(n!)$

Typical Growth Rates

- $O(1)$ Constant Time
- $O(log\ n)$ Logarithmic Time
- $O(log^2\ n)$
- $O(n)$ Linear Time
- $O(n\ log\ n)$
- $O(n^2)$ Polynomial Time
- $O(n^3)$
- $O(2^n)$
- $O(n!)$

Typical Growth Rates

- $O(1)$ — Constant Time
- $O(log\ n)$ — Logarithmic Time
- $O(log^2\ n)$
- $O(n)$ — Linear Time
- $O(n\ log\ n)$
- $O(n^2)$ — Polynomial Time
- $O(n^3)$
- $O(2^n)$ — Exponential Time
- $O(n!)$

Typical Growth Rates

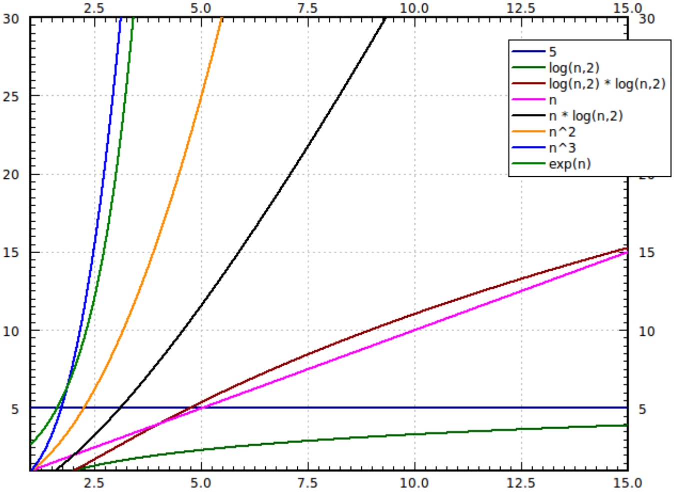

Figure: Typical Growth Rates

Typical Growth Rates

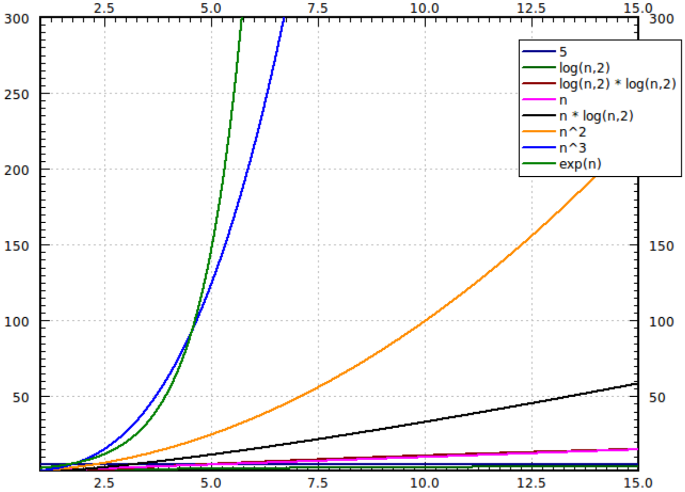

Figure: Typical Growth Rates

Typical Growth Rates

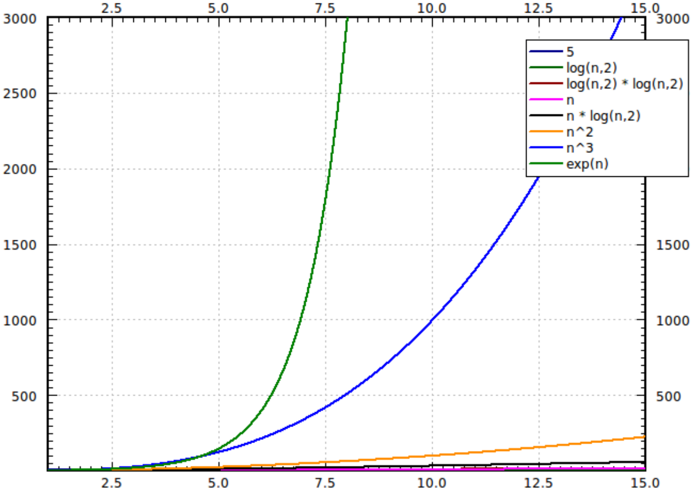

Figure: Typical Growth Rates

Typical Growth Rates

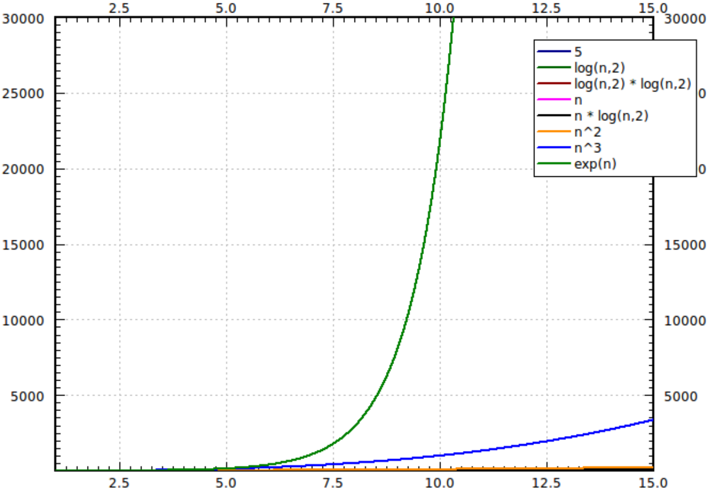

Figure: Typical Growth Rates

Rule of Sums

Sequential Segments

$<$

. . . Program Segment 1

. . .

$>$

$<$

. . . Program Segment 2

. . .

$>$

Rule of Sums

Sequential Segments

$<$

. . . Program Segment 1 $T_1(n)$

. . .

$>$

$<$

. . . Program Segment 2 $T_2(n)$

. . .

$>$

Rule of Sums

Sequential Segments

$<$

. . . Program Segment 1 $\qquad T_1(n)$

. . .

$>$

$<$

. . . Program Segment 2 $\qquad T_2(n)$

. . .

$>$

$T(n) = T_1(n) + T_2(n)$

Rule of Sums

Sequential Segments

$<$

. . . Program Segment 1 $\quad T_1(n)$

. . .

$>$

$<$

. . . Program Segment 2 $\quad T_2(n)$

. . .

$>$

$T(n) = T_1(n) + T_2(n)$

Sum Rule

Suppose $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T(n)$ is $O\left(max\left(f(n), g(n)\right)\right)$

Rule of Products

Iteration

<

. . . Program Segment

. . .

>

executed $T_2(n)$ times

Rule of Products

Iteration

<

. . . Program Segment

. . .

>

executed $T_2(n)$ times

$T_1(n)$

Rule of Products

Iteration

$<$

. . . Program Segment

. . .

$>$

executed $T_2(n)$ times

$T_1(n)$

$$T(n) = T_1(n) \times T_2(n)$$

Rule of Products

Iteration

$<$

. . . Program Segment

. . .

$>$

executed $T_2(n)$ times

$T_1(n)$

$T(n) = T_1(n) \times T_2(n)$

Product Rule

Suppose $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T(n)$ is $O(f(n)g(n))$

Rule of Products

Iteration

<

. . . Program Segment

. . .

>

executed $T_2(n)$ times

$T_1(n)$

$$T(n) = T_1(n) \times T_2(n)$$

Product Rule

Suppose $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T(n)$ is $O\left(f(n)g(n)\right)$

Note that $O(cf(n))$ is same as $O(f(n))$

Example: Maximum Sub-sequence Sum

- Given a sequence of integers $\langle a_1, a_2, \cdots, a_N \rangle$, find the maximum value of
$$\sum_{k=i}^{j} a_k$$
for some (i, j) range within the bounds of the sequence.

Example: Maximum Sub-sequence Sum

- Given a sequence of integers $\langle a_1, a_2, \cdots, a_N \rangle$, find the maximum value of
$$\sum_{k=i}^{j} a_k$$
for some (i, j) range within the bounds of the sequence.
- You may assume that sum may be reported as 0 whenever it goes negative

Example: Maximum Sub-sequence Sum

- Given a sequence of integers $\langle a_1, a_2, \cdots, a_N \rangle$, find the maximum value of
$$\sum_{k=i}^{j} a_k$$
for some (i, j) range within the bounds of the sequence.
- You may assume that sum may be reported as 0 whenever it goes negative
- An instance: $\langle -2, 11, -4, 13, -5, -2 \rangle$
- Answer?

Example: Maximum Sub-sequence Sum

- Given a sequence of integers $\langle a_1, a_2, \cdots, a_N \rangle$, find the maximum value of
$$\sum_{k=i}^{j} a_k$$
for some (i, j) range within the bounds of the sequence.
- You may assume that sum may be reported as 0 whenever it goes negative
- An instance: $\langle -2, 11, -4, 13, -5, -2 \rangle$
- Answer? 20

Example: Maximum Sub-sequence Sum

- Given a sequence of integers $\langle a_1, a_2, \cdots, a_N \rangle$, find the maximum value of
$$\sum_{k=i}^{j} a_k$$
for some (i, j) range within the bounds of the sequence.
- You may assume that sum may be reported as 0 whenever it goes negative
- An instance: $\langle -2, 11, -4, 13, -5, -2 \rangle$
- Answer? 20
- Algorithm idea?

Example: Maximum Sub-sequence Sum

- Given a sequence of integers $\langle a_1, a_2, \cdots, a_N \rangle$, find the maximum value of
$$\sum_{k=i}^{j} a_k$$
for some (i, j) range within the bounds of the sequence.
- You may assume that sum may be reported as 0 whenever it goes negative
- An instance: $\langle -2, 11, -4, 13, -5, -2 \rangle$
- Answer? 20
- Algorithm idea? Brute-force approach — exhaustive search — examine all the sub-sequences and choose the maximum

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
      ThisSum += A[k]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum;
return MaxSum
```

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
       ThisSum += A[k]
    if (ThisSum > MaxSum):
       MaxSum = ThisSum;
return MaxSum
```

- "k" – loop

$$\sum_{k=i}^{j} 1 =$$

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
      ThisSum += A[k]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum;
return MaxSum
```

- "k" – loop

$$\sum_{k=i}^{j} 1 = j - i + 1$$

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
      ThisSum += A[k]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum;
return MaxSum
```

- "j" – loop

$$\sum_{j=i}^{N-1}(j-i+1) =$$

- "k" – loop

$$\sum_{k=i}^{j} 1 = j - i + 1$$

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
      ThisSum += A[k]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum;
return MaxSum
```

- "j" – loop

$$\sum_{j=i}^{N-1}(j-i+1) = \frac{(N-i)(N-i+1)}{2}$$

- "k" – loop

$$\sum_{k=i}^{j} 1 = j-i+1$$

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
      ThisSum += A[k]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum;
return MaxSum
```

- "i" – loop

$$\sum_{i=0}^{N-1} \frac{(N-i)(N-i+1)}{2} =$$

- "j" – loop

$$\sum_{j=i}^{N-1} (j-i+1) = \frac{(N-i)(N-i+1)}{2}$$

- "k" – loop

$$\sum_{k=i}^{j} 1 = j-i+1$$

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
      ThisSum += A[k]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum;
return MaxSum
```

- "i" – loop

$$\sum_{i=0}^{N-1} \frac{(N-i)(N-i+1)}{2} = \frac{N(N+1)(N+2)}{6}$$

- "j" – loop

$$\sum_{j=i}^{N-1} (j-i+1) = \frac{(N-i)(N-i+1)}{2}$$

- "k" – loop

$$\sum_{k=i}^{j} 1 = j-i+1$$

Example

```
MaxSum = 0
for i in range(N):
    for j in range(i, N):
        ThisSum = 0
        for k in range(i, j+1):
            ThisSum += A[k]
        if (ThisSum > MaxSum):
            MaxSum = ThisSum;
return MaxSum
```

- "i" – loop

$$\sum_{i=0}^{N-1} \frac{(N-i)(N-i+1)}{2} = \frac{N(N+1)(N+2)}{6}$$

- "j" – loop

$$\sum_{j=i}^{N-1} (j-i+1) = \frac{(N-i)(N-i+1)}{2}$$

- "k" – loop

$$\sum_{k=i}^{j} 1 = j-i+1$$

Time complexity is $O(n^3)$

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
       ThisSum += A[k]
    if (ThisSum > MaxSum):
       MaxSum = ThisSum;
return MaxSum
```

- "i" – loop

$$\sum_{i=0}^{N-1} \frac{(N-i)(N-i+1)}{2} = \frac{N(N+1)(N+2)}{6}$$

- "j" – loop

$$\sum_{j=i}^{N-1} (j-i+1) = \frac{(N-i)(N-i+1)}{2}$$

- "k" – loop

$$\sum_{k=i}^{j} 1 = j - i + 1$$

Time complexity is $O(n^3)$

Is it possible to reduce the work done?

Example

```
MaxSum = 0
for i in range(N):
  for j in range(i, N):
    ThisSum = 0
    for k in range(i, j+1):
      ThisSum += A[k]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum;
return MaxSum
```

- "i" – loop

$$\sum_{i=0}^{N-1} \frac{(N-i)(N-i+1)}{2} = \frac{N(N+1)(N+2)}{6}$$

- "j" – loop

$$\sum_{j=i}^{N-1} (j-i+1) = \frac{(N-i)(N-i+1)}{2}$$

- "k" – loop

$$\sum_{k=i}^{j} 1 = j-i+1$$

Time complexity is $O(n^3)$

Is it possible to reduce the work done? Dynamic Programming!

Example

```
MaxSum = 0
for i in range(N):
   ThisSum = 0
   for j in range(i, N):
      ThisSum += A[j]
      if (ThisSum > MaxSum):
         MaxSum = ThisSum
return MaxSum
```

Example

```
MaxSum = 0
for i in range(N):
  ThisSum = 0
  for j in range(i, N):
    ThisSum += A[j]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum
return MaxSum
```

- "j" – loop

$$\sum_{j=i}^{N-1} 1 = (N-1) - i + 1 = N - i$$

Example

```
MaxSum = 0
for i in range(N):
  ThisSum = 0
  for j in range(i, N):
    ThisSum += A[j]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum
return MaxSum
```

- "i" – loop

$$\sum_{i=0}^{N-1} N-i = N+(N-1)+\cdots+1 = \frac{N(N+1)}{2}$$

- "j" – loop

$$\sum_{j=i}^{N-1} 1 = (N-1) - i + 1 = N - i$$

Example

```
MaxSum = 0
for i in range(N):
  ThisSum = 0
  for j in range(i, N):
    ThisSum += A[j]
    if (ThisSum > MaxSum):
      MaxSum = ThisSum
return MaxSum
```

Time complexity is $O(n^2)$

- "i" – loop

$$\sum_{i=0}^{N-1} N-i = N+(N-1)+\cdots+1 = \frac{N(N+1)}{2}$$

- "j" – loop

$$\sum_{j=i}^{N-1} 1 = (N-1) - i + 1 = N - i$$

```
MaxSum = 0
for i in range(N):
  ThisSum = 0
  for j in range(i, N):
    ThisSum += A[j]
    if (ThisSum > MaxSum):
       MaxSum = ThisSum
return MaxSum
```

- "i" – loop

$$\sum_{i=0}^{N-1} N-i = N+(N-1)+\cdots+1 = \frac{N(N+1)}{2}$$

- "j" – loop

$$\sum_{j=i}^{N-1} 1 = (N-1)-i+1 = N-i$$

Time complexity is $O(n^2)$
Is it possible to find the maximum sum without examining all the sub-sequences!?

Example

```
MaxSum = ThisSum = 0
for j in range(N):
  ThisSum += A[j]
  if (ThisSum > MaxSum):
    MaxSum = ThisSum
  elif (ThisSum < 0):
    ThisSum = 0
return MaxSum
```

Example

```
MaxSum = ThisSum = 0
for j in range(N):
   ThisSum += A[j]
   if (ThisSum > MaxSum):
      MaxSum = ThisSum
   elif (ThisSum < 0):
      ThisSum = 0
return MaxSum
```

- Try this on the instance $\langle -2, 11, -4, 13, -5, -2 \rangle$

Example

```
MaxSum = ThisSum = 0
for j in range(N):
  ThisSum += A[j]
  if (ThisSum > MaxSum):
    MaxSum = ThisSum
  elif (ThisSum < 0):
    ThisSum = 0
return MaxSum
```

- Try this on the instance $\langle -2, 11, -4, 13, -5, -2 \rangle$
- $O(n) \times O(1)$

Example

```
MaxSum = ThisSum = 0
for j in range(N):
  ThisSum += A[j]
  if (ThisSum > MaxSum):
    MaxSum = ThisSum
  elif (ThisSum < 0):
    ThisSum = 0
return MaxSum
```

- Try this on the instance $\langle -2, 11, -4, 13, -5, -2 \rangle$
- $O(n) \times O(1)$
- Time complexity is $O(n)$

Example

```
MaxSum = ThisSum = 0
for j in range(N):
  ThisSum += A[j]
  if (ThisSum > MaxSum):
    MaxSum = ThisSum
  elif (ThisSum < 0):
    ThisSum = 0
return MaxSum
```

- Try this on the instance $\langle -2, 11, -4, 13, -5, -2 \rangle$
- $O(n) \times O(1)$
- Time complexity is $O(n)$
- Note that all the sub-sequences are not examined!

Example

```
MaxSum = ThisSum = 0
for j in range(N):
  ThisSum += A[j]
  if (ThisSum > MaxSum):
    MaxSum = ThisSum
  elif (ThisSum < 0):
    ThisSum = 0
return MaxSum
```

- Try this on the instance $\langle -2, 11, -4, 13, -5, -2 \rangle$
- $O(n) \times O(1)$
- Time complexity is $O(n)$
- Note that all the sub-sequences are not examined! — Greedy!?

Empirical Verification: Ratio Analysis

- Is it possible to empirically verify if the running time of an algorithm is $O(f(n))$?

Empirical Verification: Ratio Analysis

- Is it possible to empirically verify if the running time of an algorithm is $O(f(n))$?
- Implement the algorithm and note down the actual running time $T(n)$ for different values of n (you may have to take average of several runs for each n)

Empirical Verification: Ratio Analysis

- Is it possible to empirically verify if the running time of an algorithm is $O(f(n))$?
- Implement the algorithm and note down the actual running time $T(n)$ for different values of n (you may have to take average of several runs for each n)
- Now find the ratio $\frac{T(n)}{f(n)}$, for those different values of n.

Empirical Verification: Ratio Analysis

- Is it possible to empirically verify if the running time of an algorithm is $O(f(n))$?
- Implement the algorithm and note down the actual running time $T(n)$ for different values of n (you may have to take average of several runs for each n)
- Now find the ratio $\frac{T(n)}{f(n)}$, for those different values of n.
- $f(n)$ is a tight bound if this ratio converges to a positive constant

Empirical Verification: Ratio Analysis

- Is it possible to empirically verify if the running time of an algorithm is $O(f(n))$?
- Implement the algorithm and note down the actual running time $T(n)$ for different values of n (you may have to take average of several runs for each n)
- Now find the ratio $\frac{T(n)}{f(n)}$, for those different values of n.
- $f(n)$ is a tight bound if this ratio converges to a positive constant
- If the ratio converges to 0, then $f(n)$ is an over-estimate

Empirical Verification: Ratio Analysis

- Is it possible to empirically verify if the running time of an algorithm is $O(f(n))$?
- Implement the algorithm and note down the actual running time $T(n)$ for different values of n (you may have to take average of several runs for each n)
- Now find the ratio $\frac{T(n)}{f(n)}$, for those different values of n.
- $f(n)$ is a tight bound if this ratio converges to a positive constant
- If the ratio converges to 0, then $f(n)$ is an over-estimate
- $f(n)$ is an under-estimation, if this ratio diverges

Summary

- Time / Space complexity of an algorithm are expressed in notations such as Big-Oh and Big-Theta
- These notations bring out the growth rate of time / space wrt the size of the input
- These notations enable us to avoid exact calculations of number of "basic steps" or memory space required — overall growth rate can be estimated based on growth rates of components
- It is possible to come up with several designs of algorithms for a problem, and analysis is important to choose the appropriate one
- It is also possible to perform empirical ratio analysis to determine / verify time complexity of an algorithm

What next?

- We will take up analysis of recursive algorithms in the next lecture
- Meanwhile, start reading about complexity analysis of algorithms from standard books