DSC40B: Theoretical Foundations of Data Science II

Lecture 5: *Binary search and recurrence*

Instructor: Yusu Wang

Previously

- Different types of time complexity analysis
  - Equipping us with ways to analyze performance of algorithms

- Today and onwards
  - Algorithm design
  - Start with the Search problem in sorted array
  - Solving recurrence to obtain time complexity
    - Often arise from recursive algorithms

Part A:
Motivation of binary search in sorted array

- Recall the general Search problem
  - Given an arbitrary array of numbers A and a target key k, check whether A contains k or not.

- This general Search problem
  - has $\Omega(n)$ theoretical lower bound
  - and linear-Search procedure achieves an optimal running time.

- How can we do better?
  - Especially if we are going have many such search queries
  - What if there are special properties of input, say the input array is already sorted? We can do much better!

Search in Database

- Large data sets are often stored in databases

PID	FullName	Level
A1843	Wan Xuegang	SR
A8293	Deveron Greer	SR
A9821	Vinod Seth	FR
A8172	Aleix Bilbao	JR
A2882	Kayden Sutton	SO
A1829	Raghu Mahanta	FR
A9772	Cui Zemin	SR
⋮	⋮	⋮

Query:
What is the name of the student with PID A8172?

- Given the same database, one can make multiple queries
  - e.g, Search for a specific entity, find Maximum in salary, or Range queries

Preprocessing + Queries

- In general
  - We would like to organize data so that they can support many such queries efficiently.

  - Time taken to prepare / organize data into a form that is easier for later queries is call pre-processing time.
  - Time taken to answer queries is called query time.
  - Often it is worthwhile to spend pre-processing time if the query time is significantly reduced and there are many queries to be performed.

An example

- Suppose we have a database of size n, and we wish to perform m Search queries.

- Strategy 1: Brute force
  - Pre-process:
    - none, $O(1)$
  - Search:
    - Linear-Search: $\Theta(n)$
- Total time:
  - $O(1) + m \times \Theta(n) = \Theta(mn)$
- If $m = n$, total time is
  - $\Theta(n^2)$

- Strategy 2: Pre-sort
  - Pre-process:
    - Sort, $\Theta(n \log n)$
  - Search:
    - Binary-search: $\Theta(\log n)$
- Total time:
  - $\Theta(n \log n) + m \times \Theta(\log n) = \Theta\big((n + m) \log n\big)$
- If $m = n$, total time is
  - $\Theta(n \log n)$

An example

- Suppose we have a database of size n, and we wish to perform m Search queries.

- Strategy 1: Brute force
  - Pre-process:
    - none, $O(1)$
  - Search:
    - Linear-Search: $\Theta(n)$
- Total time:
  - $O(1) + m \times \Theta(n) = \Theta(mn)$
- If $m = n$, total t
  - $\Theta(n^2)$

- Strategy 2: Pre-sort
  - Pre-process:
    - Sort, $\Theta(n \log n)$
  - Search:
    - Binary-search: $\Theta(\log n)$
- Total time:
  - $\Theta(n \log n) + m \times \Theta(\log n) = \Theta\big((n+m)\log n\big)$
- me is

In general, Strategy 2 pays off if $m = \Omega(\log n)$

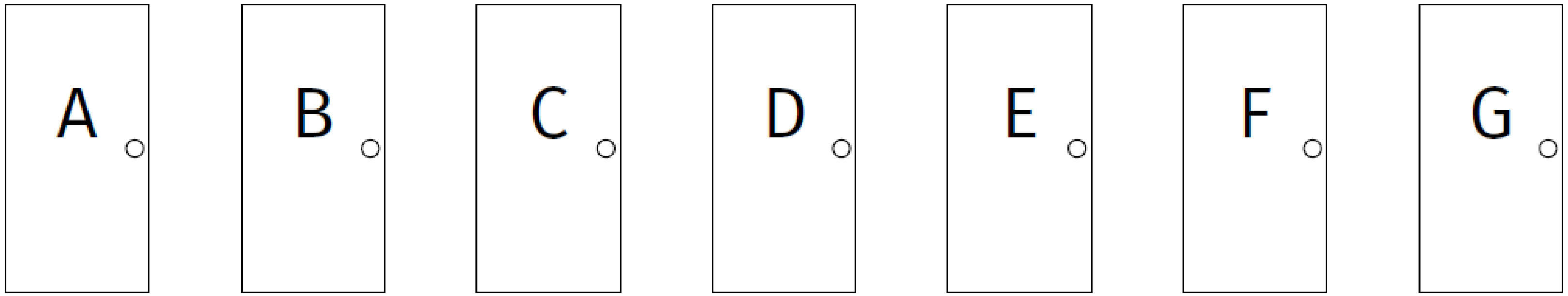
Preprocessing + Queries

- If there are many queries, then often, performing a preprocessing pays off if that makes answering the queries more efficient.

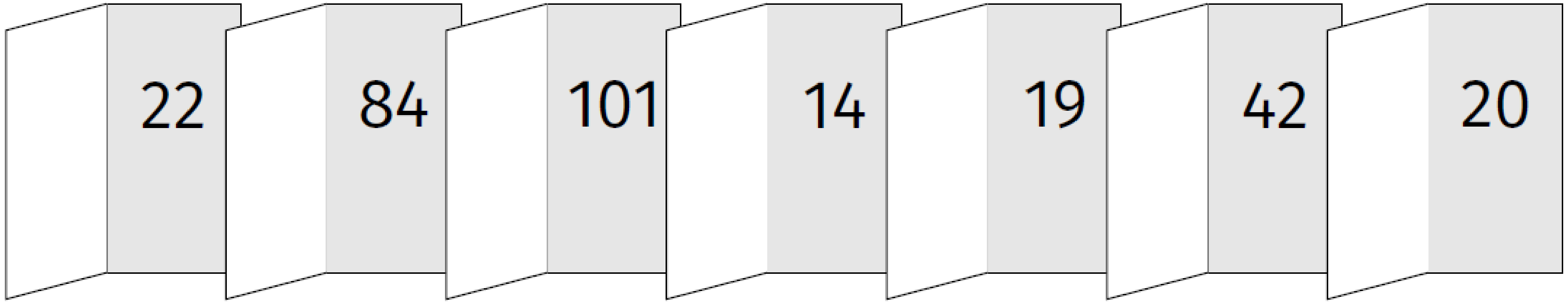
- Furthermore, often queries have to be done online, while preprocessing can be done offline
  - Hence sometimes, even if we don't have many queries, it still pays off to do preprocessing to support fast online queries for users.

Part B:
Binary search in sorted array

- Game show:
  - n doors in A,
    - opening i-th door is equivalent to access $A[i]$
  - you are supposed to guess behind which door is number 42
    - with every wrong guess, your reward money will be reduced.

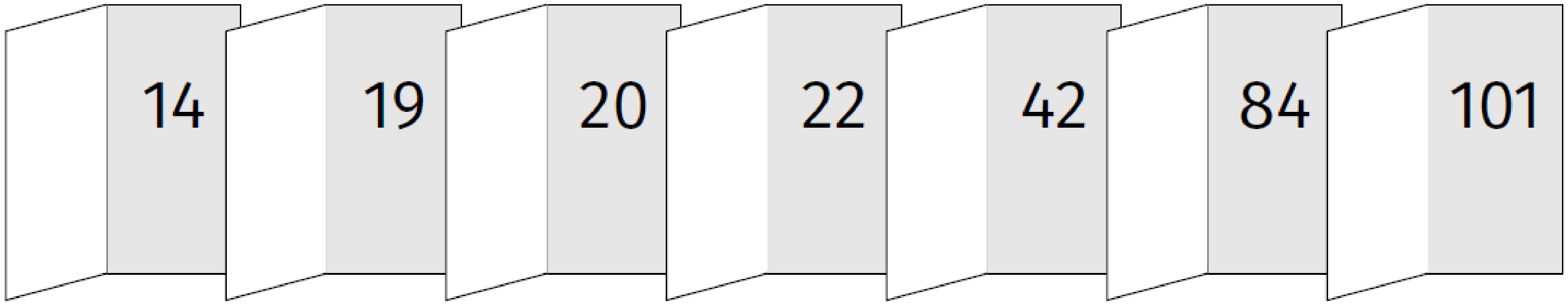


- If the numbers can be arbitrarily placed behind these doors
  - cannot do better than linear search
  - after opening each door, 42 can be in any of the remainder doors



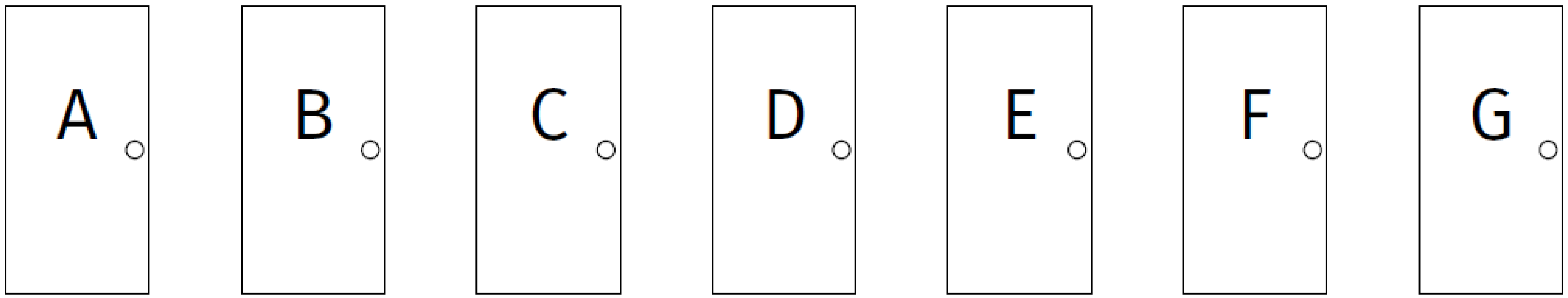
If the list of numbers are sorted

- Suppose that the input array A is sorted in *non-decreasing order*



If the list of numbers are sorted

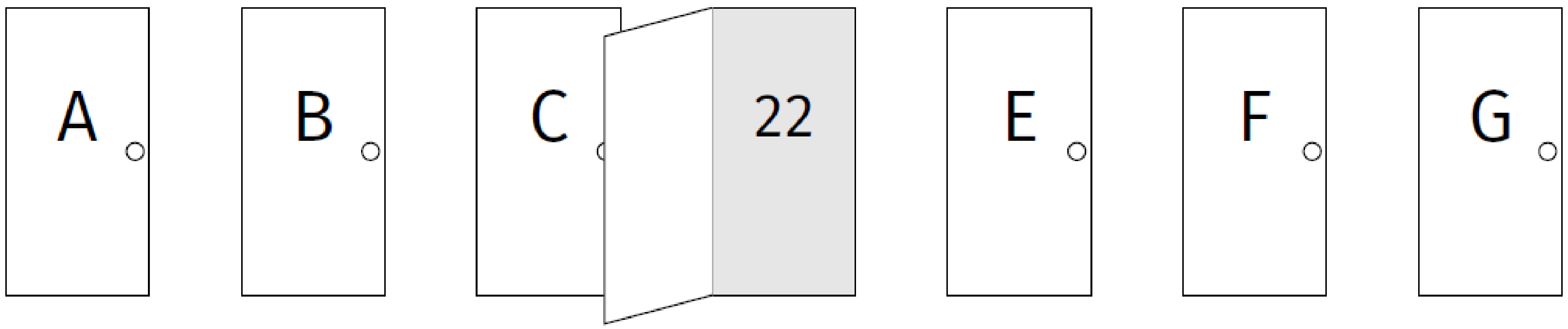
- Equivalently, the input array A is sorted in non-decreasing order



- Which door to open first?

If the list of numbers are sorted

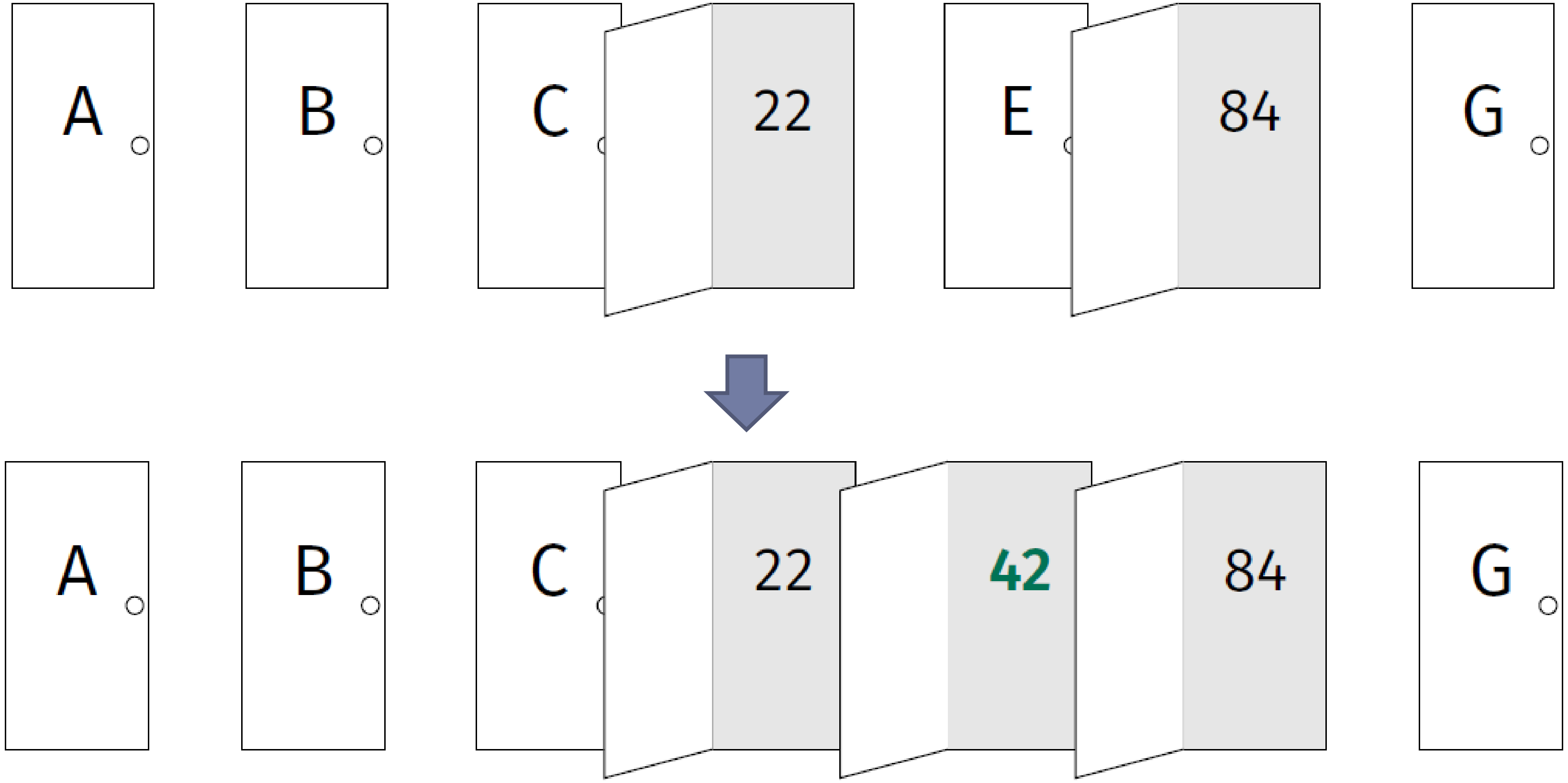
- Equivalently, the input array A is sorted in non-decreasing order



- Which door to open first?
  - Door D – the middle one

If the list of numbers are sorted

- Equivalently, the input array A is sorted in non-decreasing order



Strategy

- First pick the middle door
- Allows you to rule out half of the other doors.
- Pick door in the middle of what remains.
- Repeat, recursively.

How to convert this strategy to a piece of code (an algorithm)?

Search Problem in sorted array

- Input:
  - a sorted array A whose elements are in non-decreasing order as indices increase
  - a target key t
- Output:
  - return the index of A whose element equals to t, or None otherwise

Exercise:
Given a sorted array A, and two indices $b \leq d$, and we want to search t in $A[b:d]$.
Which element will you check first?

Binary Search Algorithm

```
import math
def binary_search(A, t, start, stop):
"""
Assumes A is sorted. Searches A[start:stop] for t.
"""
    if stop - start <= 0:           return None
    if stop - start == 1:
            if A[start] == t:       return start
            else return None
    middle = math.floor((start + stop)/2)
    if A[middle] == t :             return middle
    elif A[middle] > t :
            return binary_search(A, t, start, middle)
    else:
            return binary_search(A, t, middle+1, stop)
```

Example

t = 21

-10	-6	-3	1	2	5	12	21	33	35	42
0	1	2	3	4	5	6	7	8	9	10

- What is the first call?
  - binary_search$(A, t, 0, 11)$
    - (or in general, binary_search$(A, t, 0, n)$)

Part C:
Correctness of binary-search

Correctness

- How do we convince ourselves that such a recursive algorithm is correct?
- Often by inductive thinking (bottom-up):
  - (1) Make sure algorithm works in the base case.
  - (2) Check that all recursive calls are on smaller problems, and that it terminates
  - (3) Assuming that the recursive calls work, does the whole algorithm work?

(1) Base case

- What is the Base case for binary_search$(A, t, start, stop)$?
  - Base cases are when there are no more recursive calls

- Base case is $stop - start \leq 1$
  - If stop $- start \leq 0$, the algorithm returns None.
  - If $stop - start = 1$,
    - this means there is only one element $A[start]$ to check
    - the algorithm returns this element if it is t, otherwise None.

- So the base case is correct.

(2) Recursive steps -- termination

- Does the procedure terminate?
  - Or could it get into infinite loop?

- Yes, as each time, the size of the subproblem we consider is strictly smaller

(3) Recursive steps -- correctness

- In a specific call of binary_search$(A, t, start, stop)$
  - assume that all recursive calls return correct answers
  - then does the algorithm binary_search$(A, t, start, stop)$ return correct answer?

- Yes.

- Then by Inductive argument, the entire algorithm is correct.
  - We will not get into formal argument here, but it can be made precise and formal.

Intuitively, why it works:

- Show that it works for size 1 (base case).
- $\Longrightarrow$ will work for size 2 (inductive step).
- $\Longrightarrow$ will work for sizes 3, 4 (inductive step).
- $\Longrightarrow$ will work for sizes 5, 6, 7, 8 (inductive step) ..

Exercise

Does this code work? Why or why not?

```python
import math
def summation(numbers):
    n = len(numbers)
    if n == 0:
        return 0
    middle = math.floor(n / 2)
    return (
        summation(numbers[:middle])
        +
        summation(numbers[middle:])
    )
```

Part D:
Time complexity analysis: Recurrence relations

Best Case?

```
def binary_search(A, t, start, stop):
"""
Assumes A is sorted. Searches A[start:stop] for t.
"""
    if stop - start <= 0:          return None
    if stop - start == 1:
        if A[start] == t:          return start
        else return None
    middle = math.floor((start + stop)/2)
    if A[middle] == t :            return middle
    elif A[middle] > t :
        return binary_search(A, t, start, middle)
    else:
        return binary_search(A, t, middle+1, stop)
```

Worst Case?

```
def binary_search(A, t, start, stop):
"""
Assumes A is sorted. Searches A[start:stop] for t.
"""
    if stop - start <= 0:          return None
    if stop - start == 1:
        if A[start] == t:          return start
        else return None
    middle = math.floor((start + stop)/2)
    if A[middle] == t :            return middle
    elif A[middle] > t :
        return binary_search(A, t, start, middle)
    else:
        return binary_search(A, t, middle+1, stop)
```

- Let $T(n)$ be the worst-case time complexity of binary_search(A, t, start, stop) for a range of size n

- We have the following recurrence relation
  - $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + c, & n > 1 \\ \Theta(1), & n \leq 1 \end{cases}$

- Note the recurrence relation does not give yet an explicit time complexity. We have to solve it to obtain a non-recursive formula for $T(n)$.

Solving Recurrence

- One way is via the following strategy:
  - 1. "Unroll" several times to find a pattern.
  - 2. Write general formula for kth unroll.
  - 3. Solve for # of unrolls needed to reach base case.
  - 4. Plug this number into general formula.

Recurrence for binary-search

- $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + c, & n > 1 \\ \Theta(1), & \boxed{n \leq 1} \end{cases}$

Termination condition. In fact, for recurrence relations for time complexity $T(n)$, we can always assume that $T(c') = \Theta(1)$ when c' is a constant, e.g, $c' = 1, 2, 3, 4$.

- So the key relation is
  - $T(n) = T\left(\frac{n}{2}\right) + c$
  - Very often, for simplicity, we only write this as the recurrence relation, with the understanding that $T(c') = \Theta(1)$ for constant c'.
- Another simplification:
  - We assume n is power of 2, so we don't have to worry about $\frac{n}{2}$ is not integer at any moment.

Step (1): unroll several times to find a pattern

- $T(n) = T\left(\frac{n}{2}\right) + c$
-

Step (2): find general formula for kth unroll

- $T(n) = T\left(\frac{n}{2}\right) + c$

$$= T\left(\frac{n}{4}\right) + 2c$$

$$= T\left(\frac{n}{8}\right) + 3c \ldots$$

- on kth unroll:

$$= T\left(\frac{n}{2^k}\right) + c \cdot k$$

Step (3): # of unrolls needed to reach base case

- $T(n) = T\left(\frac{n}{2}\right) + c$

  $= T\left(\frac{n}{4}\right) + 2c$

  $= T\left(\frac{n}{8}\right) + 3c$ ...

- on kth unroll:

  $= T\left(\frac{n}{2^k}\right) + c \cdot k$

- The unrolling terminates when reaching $T(1)$,
  - i.e, when $\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$

Step (4): Plug into general formula

- $T(n) = T\left(\frac{n}{2}\right) + c$

  $= T\left(\frac{n}{4}\right) + 2c$

  $= T\left(\frac{n}{8}\right) + 3c \ \ldots$
- on kth unroll:

  $= T\left(\frac{n}{2^k}\right) + c \cdot k$
- The unrolling terminates when reaching $T(1)$,
  - i.e, when $\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$
- $T(n) = T\left(\frac{n}{2^k}\right) + c \cdot k = T(1) + c \log_2 n = \Theta(\log n)$

How fast is this compared to linear-search?

- Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- Goal: find a particular grain.
- Assume one basic operation takes 1 nanosecond.
- Linear search: 317 years.
- Binary search: ≈ 60 nanoseconds.

Remarks

- Note that binary-search requires a sorted input array!
  - So needs a preprocessing
- Worthwhile when there are many queries
  - Or when we need to perform quick online queries

- In practice,
  - Databases are often indexed (sorted) by certain keys
  - Most commonly, B-Trees are used for organizing databases.

Theoretical lower bounds

- It turns out that a theoretical lower bound for searching in a sorted list is $\Omega(\log n)$

- Hence the binary search procedure we just talked about has optimal worst case time complexity

Part E:
More Recurrence examples

Examples

- $T(n) = T\left(\frac{n}{3}\right) + c$

- $T(n) = T\left(\frac{n}{2}\right) + n$

Examples/3

- $T(n) = T(n-1) + c$

- $T(n) = T(n-3) + c$

  $= T(n - 2 * 3) + 2c = T(n - 3 * 3) + 3c = T(n - 4 * 3) + 4c$

  $= \cdots = T(n - k * 3) + kc$

  The process terminates when $n - 3k = 1 \Rightarrow k = \frac{n-1}{3}$

  It then follows that $T(n) = T(n - 3k) + kc = T(1) + c\ * \frac{n-1}{3} = \Theta(1) + \Theta(n) = \Theta(n)$

Examples

- $T(n) = T(n-1) + cn$

  $= T(n-2) + c(n-1) + cn = T(n-3) + c(n-2) + c(n-1) + cn$

  $= \cdots = T(n-k) + c(n-k+1) + \cdots + c(n-2) + c(n-1) + cn$
- This process stops when $n - k = 1 \Rightarrow k = n - 1$.
  - In this case, $n - k + 1 = 2$.
  - We then have that

  $T(n) = T(1) + c * 2 + c * 3 + \cdots c * (n-1) + cn$

  $= T(1) + c\sum_{i=2}^{n} i = \Theta(1) + \Theta(n^2) = \Theta(n^2)$

FIN