Master's Theorem, Quick Sort and Full History Recurrences

Subhabrata Samajder

IIIT, Delhi

Winter Semester,
15th March, 2023

Recurrences: Divide and Conquer

Divide and Conquer Relations: The Basic Idea

- The original problem is divided into smaller subproblems.

Divide and Conquer Relations: The Basic Idea

- The original problem is divided into smaller subproblems.

- Each subproblem is solved recursively.

Divide and Conquer Relations: The Basic Idea

- The original problem is divided into smaller subproblems.
- Each subproblem is solved recursively.
- A *combine* algorithm is used to solve the original problem.

Divide and Conquer Relations: Problem Statement

Assumptions:

- # Subproblems: a
- Size of Each Subproblem: $1/b$ of the original problem
- Combine Algorithm: Takes time cn^k

where a, b, c, and k are some constant.

Divide and Conquer Relations: Problem Statement

Assumptions:

- # Subproblems: a
- Size of Each Subproblem: $1/b$ of the original problem
- Combine Algorithm: Takes time cn^k

where a, b, c, and k are some constant.

Then,

$$T(n) = aT(n/b) + cn^k.$$

Divide and Conquer Relations: Problem Statement

Assumptions:

- # Subproblems: a
- Size of Each Subproblem: $1/b$ of the original problem
- Combine Algorithm: Takes time cn^k

where a, b, c, and k are some constant.

Then,

$$T(n) = aT(n/b) + cn^k.$$

For Simplicity: Further assume that $n = b^m$, so that n/b is always an integer (b is an integer greater than 1).

Divide and Conquer Relations (Cont.)

Expand:

$$T(n) \quad = \quad a\{aT(n/b^2) + c(n/b)^k\} + c(n)^k$$

Divide and Conquer Relations (Cont.)

Expand:

$$\begin{aligned} T(n) &= a\{aT(n/b^2) + c(n/b)^k\} + c(n)^k \\ &= a\{a\{aT(n/b^3) + c(n/b^2)^k\} + c(n/b)^k\} + cn^k \end{aligned}$$

Divide and Conquer Relations (Cont.)

Expand:

$$\begin{aligned}
T(n) &= a\{aT(n/b^2) + c(n/b)^k\} + c(n)^k \\
&= a\{a\{aT(n/b^3) + c(n/b^2)^k\} + c(n/b)^k\} + cn^k \\
&\vdots \\
&= a\{a\{a\{\cdots\{aT(n/b^m) + c(n/b^{m-1})^k\} + \cdots\} + cn^k,
\end{aligned}$$

where $n/b^m = 1$.

Divide and Conquer Relations (Cont.)

Expand:

$$\begin{aligned}
T(n) &= a\{aT(n/b^2) + c(n/b)^k\} + c(n)^k \\
&= a\{a\{aT(n/b^3) + c(n/b^2)^k\} + c(n/b)^k\} + cn^k \\
&\vdots \\
&= a\{a\{a\{\cdots\{aT(n/b^m) + c(n/b^{m-1})^k\} + \cdots\} + cn^k,
\end{aligned}$$

where $n/b^m = 1$.

Assume: $T(1) = c$.

Divide and Conquer Relations (Cont.)

Expand:

$$\begin{aligned}
T(n) &= a\{aT(n/b^2) + c(n/b)^k\} + c(n)^k \\
&= a\{a\{aT(n/b^3) + c(n/b^2)^k\} + c(n/b)^k\} + cn^k \\
&\vdots \\
&= a\{a\{a\{\cdots\{aT(n/b^m) + c(n/b^{m-1})^k\} + \cdots\} + cn^k,
\end{aligned}$$

where $n/b^m = 1$.

Assume: $T(1) = c$.

Remark: A different value would change the end result by only a constant.

Divide and Conquer Relations (Cont.)

$$\therefore \quad T(n) \quad = \quad ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk}$$

Divide and Conquer Relations (Cont.)

$$\begin{aligned}
\therefore\ T(n) &= ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk} \\
&= c\sum_{i=0}^{m} a^{m-i}b^{ik} = ca^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i,
\end{aligned}$$

Divide and Conquer Relations (Cont.)

$$\begin{aligned} \therefore\; T(n) &= ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk} \\ &= c\sum_{i=0}^{m} a^{m-i}b^{ik} = ca^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i, \end{aligned}$$

which is a simple geometric series.

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$**:**
  - The factor of the geometric series is less than 1.

Divide and Conquer Relations (Cont.)

The following cases may arise:

- **$a > b^k$:**
  - The factor of the geometric series is less than 1.
  - So the series converges to a constant as $m \to \infty$.

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$**:**
  - The factor of the geometric series is less than 1.
  - So the series converges to a constant as $m \to \infty$.
  - Therefore,

    $$T(n) = \mathcal{O}(a^m) = \mathcal{O}(a^{\log_b n}) = \mathcal{O}(n^{\log_b a}),$$

    as $m = \log_b n$.

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$:
- $a = b^k$:
    - The factor of the geometric series is equal to 1.

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$**:**
- $a = b^k$**:**
  - The factor of the geometric series is equal to 1.
  - Thus
    $$T(n) = \mathcal{O}(a^m m) = \mathcal{O}(n^k \log n),$$
    since, $a = b^k \;\Rightarrow\; \log_b a = k$ and $m = \log_b n$.

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$**:**
- $a = b^k$**:**
- $a < b^k$**:**
  - The factor of the geometric series is greater than 1.

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$:
- $a = b^k$:
- $a < b^k$:
  - The factor of the geometric series is greater than 1.
  - Let $F = b^k/a$ (F is a constant).

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$:
- $a = b^k$:
- $a < b^k$:
  - The factor of the geometric series is greater than 1.
  - Let $F = b^k/a$ (F is a constant).
  - First element of the series is a^m, therefore we obtain

$$T(n) = \frac{a^m(F^{m+1} - 1)}{F - 1}$$

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$**:**
- $a = b^k$**:**
- $a < b^k$**:**
  - The factor of the geometric series is greater than 1.
  - Let $F = b^k/a$ (F is a constant).
  - First element of the series is a^m, therefore we obtain

$$\begin{aligned} T(n) &= \frac{a^m(F^{m+1}-1)}{F-1} \\ &= \mathcal{O}(a^m F^m) = \mathcal{O}((b^k)^m) = \mathcal{O}((b^m)^k) \end{aligned}$$

Divide and Conquer Relations (Cont.)

The following cases may arise:

- $a > b^k$:
- $a = b^k$:
- $a < b^k$:
  - The factor of the geometric series is greater than 1.
  - Let $F = b^k/a$ (F is a constant).
  - First element of the series is a^m, therefore we obtain

$$\begin{aligned} T(n) &= \frac{a^m(F^{m+1} - 1)}{F - 1} \\ &= \mathcal{O}(a^m F^m) = \mathcal{O}((b^k)^m) = \mathcal{O}((b^m)^k) \\ &= \mathcal{O}(n^k). \end{aligned}$$

Master's Theorem: A Simpler Version

Theorem

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$*, where* a *and* b *are integer constants,* $a \geq 1, b \geq 2$*, and* c *and* k *are positive constants, is*

$$T(n) = \begin{cases} \mathcal{O}(n^{\log_b a}) & \text{if } a > b^k \\ \mathcal{O}(n^k \log n) & \text{if } a = b^k \\ \mathcal{O}(n^k) & \text{if } a < b^k \end{cases}$$

Merge Sort: Cost Analysis

Cost Analysis

$$T(n) = 2T(\lceil n/2 \rceil) + \mathcal{O}(n) = \mathcal{O}(n \log n) \quad \text{[By Master's theorem].}$$

Cost Analysis

$$T(n) = 2T(\lceil n/2 \rceil) + \mathcal{O}(n) = \mathcal{O}(n \log n) \quad \text{[By Master's theorem].}$$

Note: The number of data movements is $\mathcal{O}(n \log n)$!!

Cost Analysis

$$T(n) = 2T(\lceil n/2 \rceil) + \mathcal{O}(n) = \mathcal{O}(n \log n) \quad \text{[By Master's theorem].}$$

Note: The number of data movements is $\mathcal{O}(n \log n)$!!

Drawbacks:

- Not as easy to implement.
- Additional storage required during each merge step.
- Thus, mergesort is not an in-place algorithm.
- This copying must be done every time two smaller sets are merged, making the procedure slower.

Cost Analysis

$$T(n) = 2T(\lceil n/2 \rceil) + \mathcal{O}(n) = \mathcal{O}(n \log n) \quad \text{[By Master's theorem].}$$

Note: The number of data movements is $\mathcal{O}(n \log n)$!!

Drawbacks:

- Not as easy to implement.
- Additional storage required during each merge step.
- Thus, mergesort is not an in-place algorithm.
- This copying must be done every time two smaller sets are merged, making the procedure slower.

Home Work: Write the algorithm for Mergesort and implement it in C.

Quicksort

Motivation

Recall:

- Mergesort needs extra storage.
- It is not possible to predict where each element will end up in the final order.

Motivation

Recall:

- Mergesort needs extra storage.
- It is not possible to predict where each element will end up in the final order.

Question: Can we somehow perform a different divide and conquer so that the position of the elements can be determined?

Motivation

Recall:

- Mergesort needs extra storage.
- It is not possible to predict where each element will end up in the final order.

Question: Can we somehow perform a different divide and conquer so that the position of the elements can be determined?

Basic Idea of Quicksort:

- Spend most of the effort in the divide step and
- very little in the conquer step!

The Divide and Combine Step

- **The Divide Step:**
  - Suppose that we know a number x such that *one-half* of the elements are $> x$ and the *other-half* of the elements are $\leq x$.
  - Compare all elements to x.
  - Partition the sequence into two parts according to the answer.
  - This partition requires $n - 1$ comparisons.
  - One part can occupy the first half of the array and the other the second half.
  - $\therefore$ can be done *in-place*.

The Divide and Combine Step

- **The Divide Step:**
  - Suppose that we know a number x such that *one-half* of the elements are $> x$ and the *other-half* of the elements are $\leq x$.
  - Compare all elements to x.
  - Partition the sequence into two parts according to the answer.
  - This partition requires $n - 1$ comparisons.
  - One part can occupy the first half of the array and the other the second half.
  - $\therefore$ can be done *in-place*.

- Then sort each subsequence recursively.

The Divide and Combine Step

- **The Divide Step:**
  - Suppose that we know a number x such that *one-half* of the elements are $> x$ and the *other-half* of the elements are $\leq x$.
  - Compare all elements to x.
  - Partition the sequence into two parts according to the answer.
  - This partition requires $n - 1$ comparisons.
  - One part can occupy the first half of the array and the other the second half.
  - $\therefore$ can be done *in-place*.

- Then sort each subsequence recursively.

- **The Combine Step:** Trivial!
  - The two parts already occupy the correct positions in the array.
  - Therefore, no additional space is required.

How To Find x?

- Till now, it was assumed that the value of x is known.
- However, x is usually unknown.

How To Find x?

- Till now, it was assumed that the value of x is known.
- However, x is usually unknown.
- **Note:** It is easy to see, that the same algorithm will work no matter which number is used for the *partition*.
- Call the number x as the *pivot*.

How To Find x?

- Till now, it was assumed that the value of x is known.

- However, x is usually unknown.

- **Note:** It is easy to see, that the same algorithm will work no matter which number is used for the *partition*.

- Call the number x as the *pivot*.

- Our purpose is to partition the array into two parts,
    - one with numbers $>$ than the pivot and
    - the other with numbers $\leq$ the pivot.

- This is achieved via the partitioning algorithm.

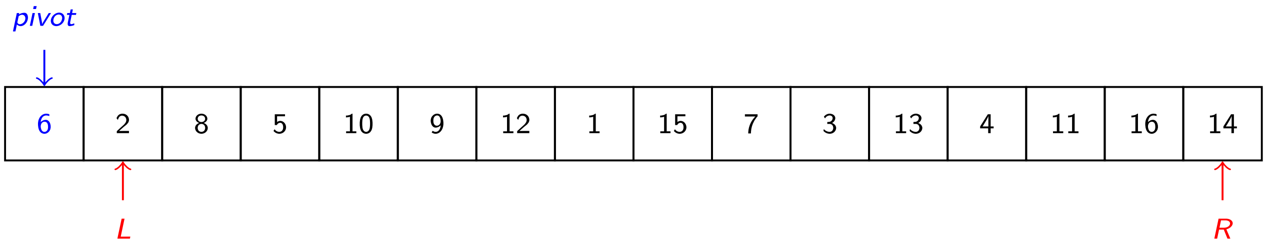
Quicksort: An Example

The Partitioning Phase:

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

Quicksort: An Example

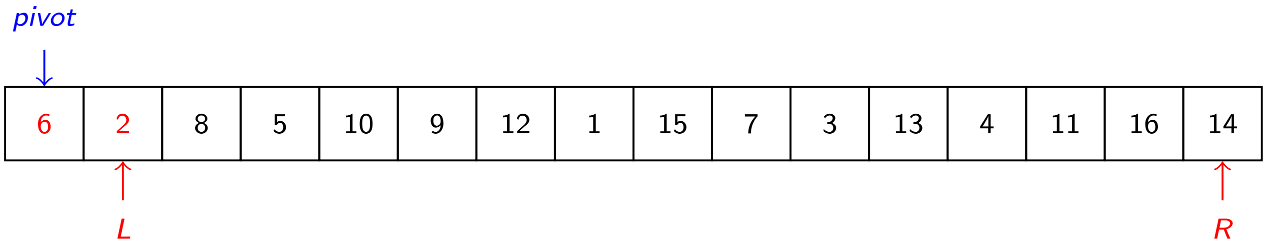
The Partitioning Phase:



6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

Quicksort: An Example

The Partitioning Phase:



pivot

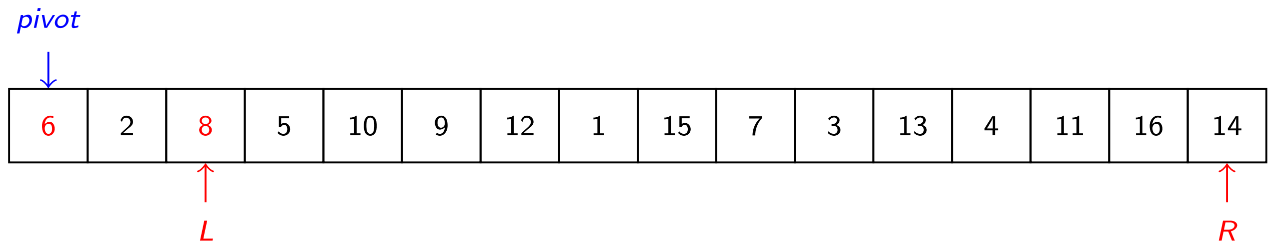
6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

L

R

Quicksort: An Example

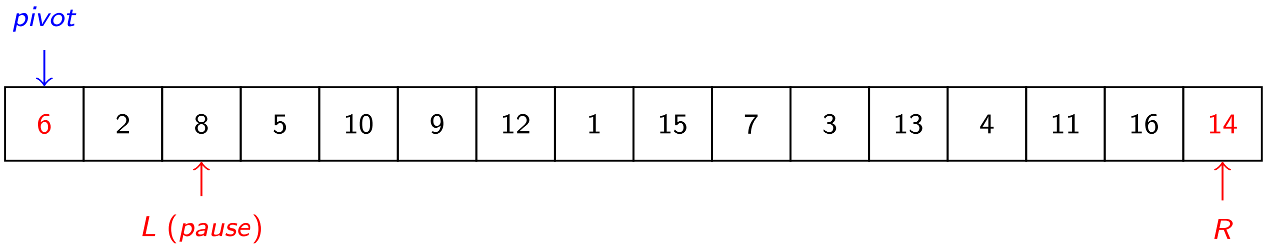
The Partitioning Phase:



6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

Quicksort: An Example

The Partitioning Phase:



pivot

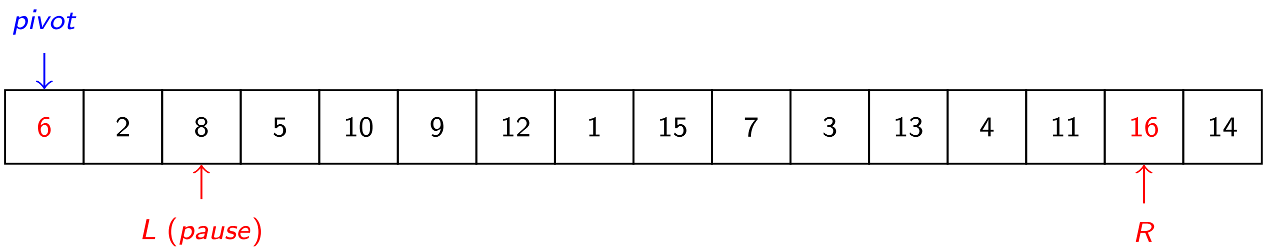
6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

L (pause)

R

Quicksort: An Example

The Partitioning Phase:



pivot

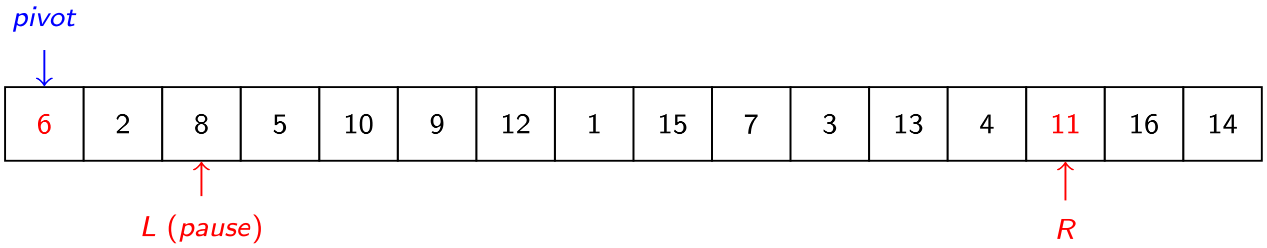
6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

L (*pause*)

R

Quicksort: An Example

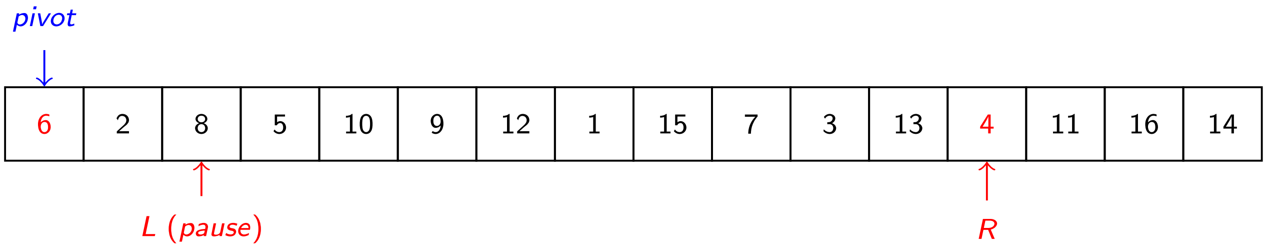
The Partitioning Phase:



6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

Quicksort: An Example

The Partitioning Phase:



pivot

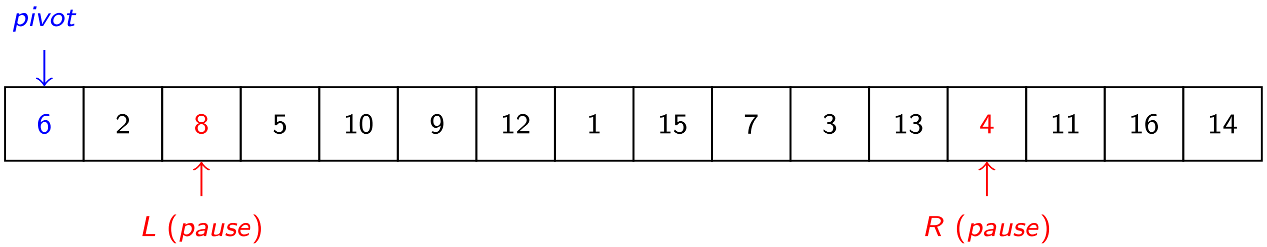
6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

L (*pause*)

R

Quicksort: An Example

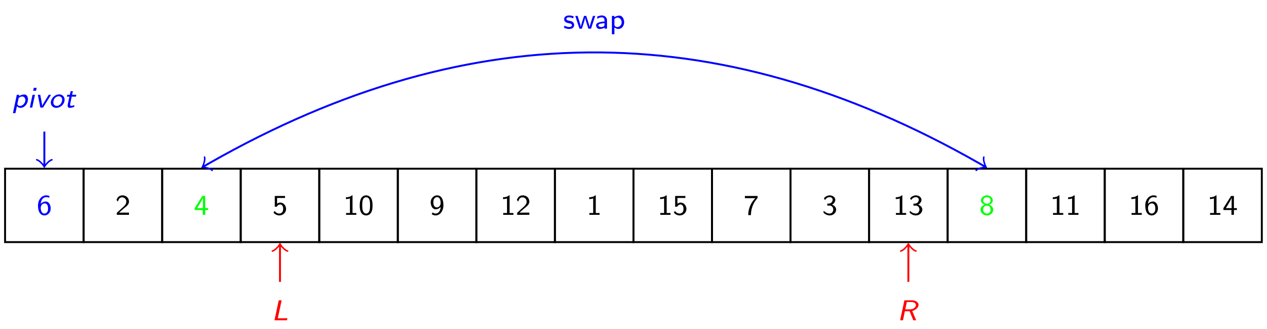
The Partitioning Phase:



6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

Quicksort: An Example

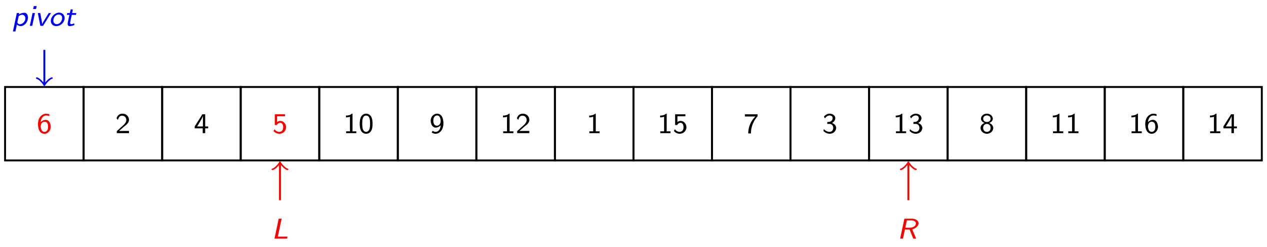
The Partitioning Phase:



6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14

Quicksort: An Example

The Partitioning Phase:



pivot

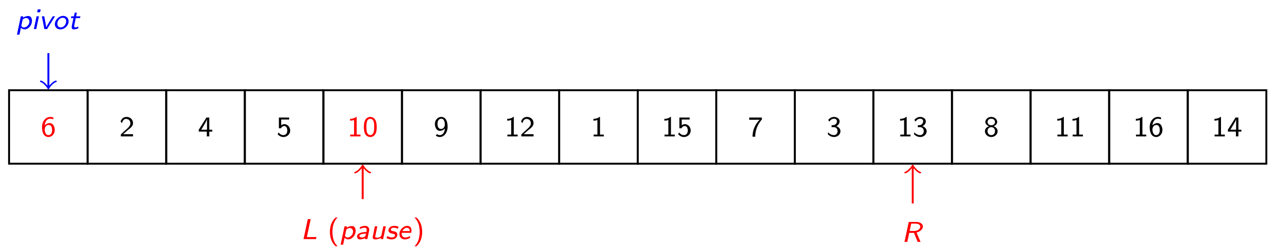
6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14

L

R

Quicksort: An Example

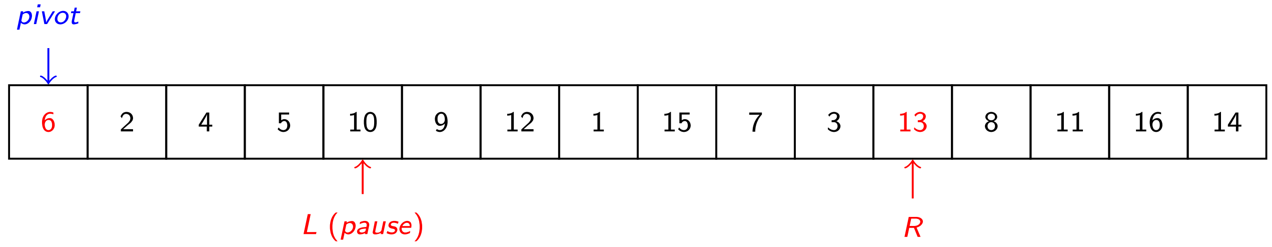
The Partitioning Phase:



6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14

Quicksort: An Example

The Partitioning Phase:



pivot

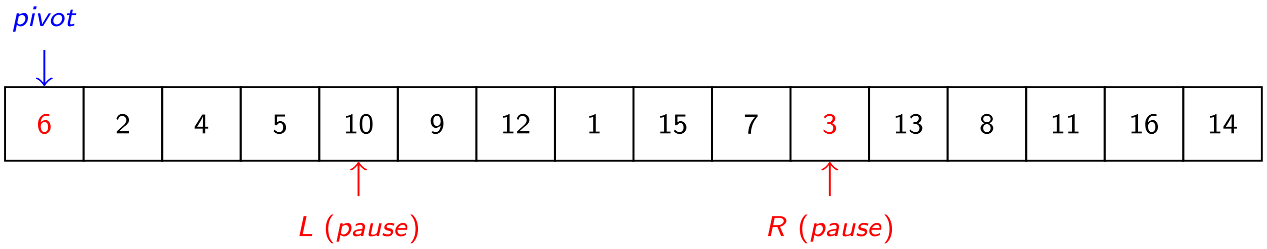
6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14

L (*pause*)

R

Quicksort: An Example

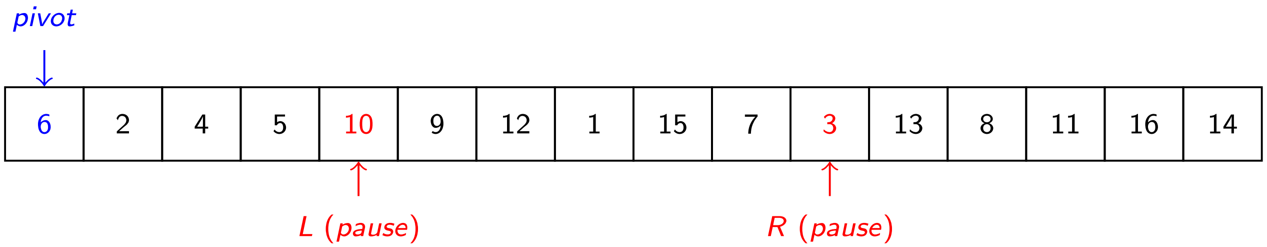
The Partitioning Phase:



6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14

Quicksort: An Example

The Partitioning Phase:



pivot

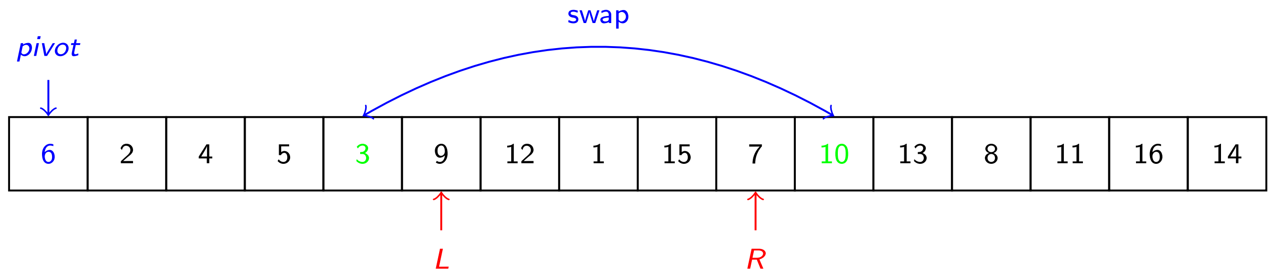
6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14

L (pause)

R (pause)

Quicksort: An Example

The Partitioning Phase:



swap

pivot

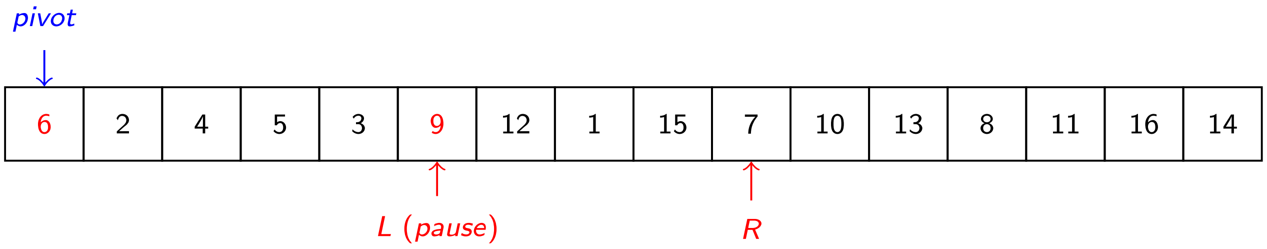
6	2	4	5	3	9	12	1	15	7	10	13	8	11	16	14

L

R

Quicksort: An Example

The Partitioning Phase:



pivot

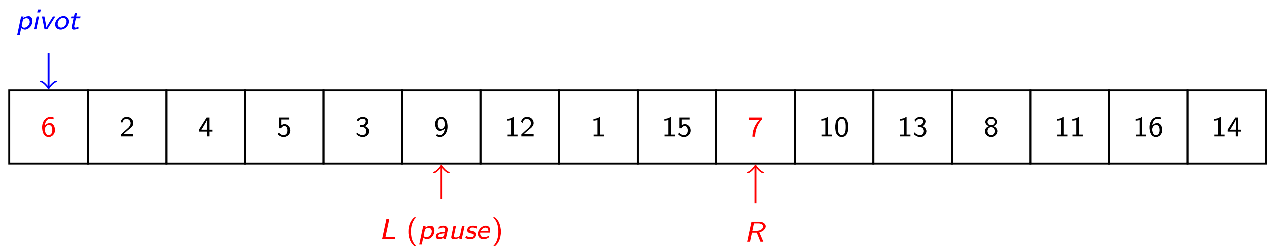
6	2	4	5	3	9	12	1	15	7	10	13	8	11	16	14

L (*pause*)

R

Quicksort: An Example

The Partitioning Phase:



pivot

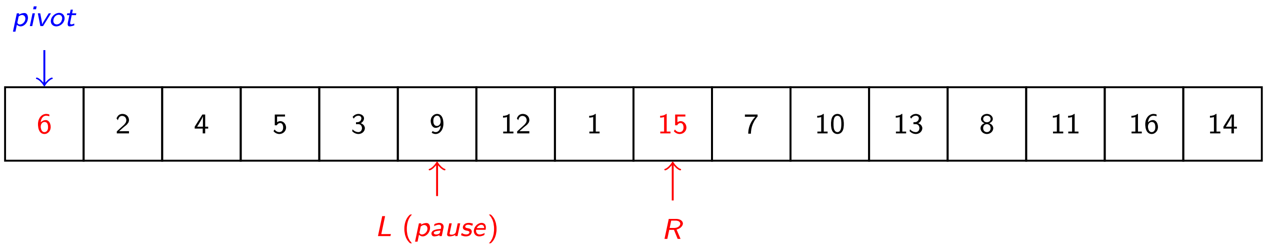
6	2	4	5	3	9	12	1	15	7	10	13	8	11	16	14

L (*pause*)

R

Quicksort: An Example

The Partitioning Phase:



pivot

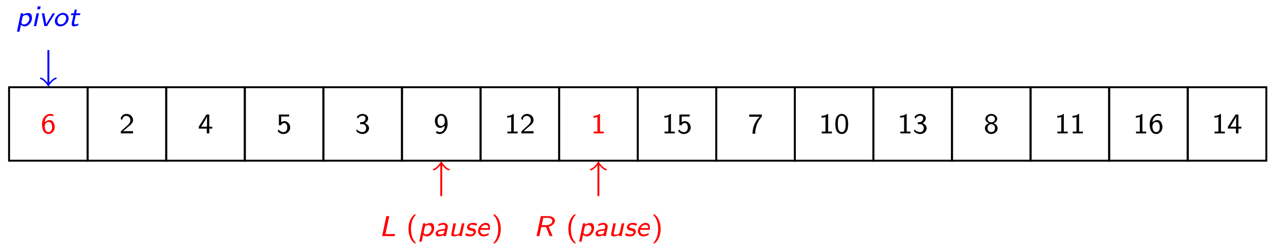
6	2	4	5	3	9	12	1	15	7	10	13	8	11	16	14

L (*pause*)

R

Quicksort: An Example

The Partitioning Phase:



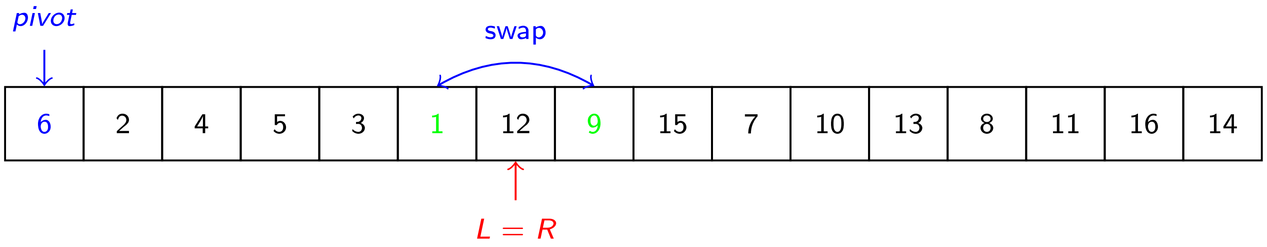
pivot

6	2	4	5	3	9	12	1	15	7	10	13	8	11	16	14

L (*pause*) R (*pause*)

Quicksort: An Example

The Partitioning Phase:



pivot

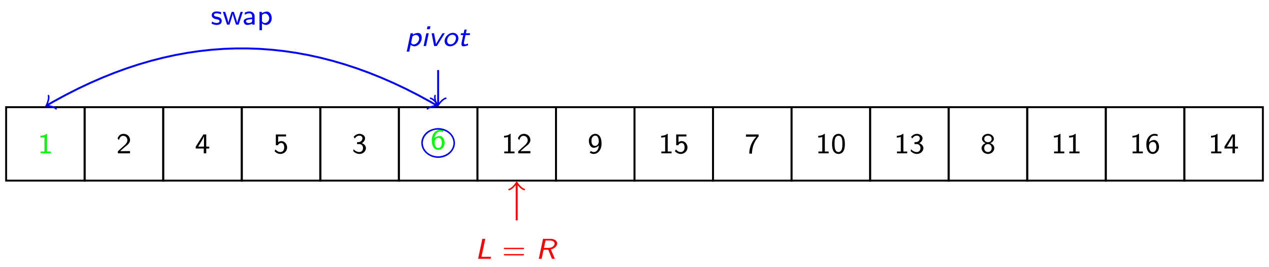
swap

6	2	4	5	3	1	12	9	15	7	10	13	8	11	16	14

$L = R$

Quicksort: An Example

The Partitioning Phase:



1	2	4	5	3	6	12	9	15	7	10	13	8	11	16	14

The Partition Algorithm

- Use two pointers to the array, L and R.
- Initially,
  - L points to the left side of the array and
  - R points to the right side of the array.
- The pointers “move” in opposite directions toward each other.
- Swap(x_L, x_R): If $x_L > pivot$ and $x_R \leq pivot$.

Quicksort: An Example (Cont.)

Recursive Phase:

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

1	2	4	5	3	⑥	12	9	15	7	10	13	8	11	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	2	4	5	3	⑥	12	9	15	7	10	13	8	11	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	4	5	3	⑥	12	9	15	7	10	13	8	11	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	12	9	15	7	10	13	8	11	16	14

Note: When a single number appears between two pivots it is obviously in the right position.

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	8	9	11	7	10	⑫	13	15	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	7	⑧	11	9	10	⑫	13	15	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	7	⑧	11	9	10	⑫	13	15	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	7	⑧	10	9	⑪	⑫	13	15	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	7	⑧	9	⑩	⑪	⑫	13	15	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	7	⑧	9	⑩	⑪	⑫	⑬	15	16	14

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	7	⑧	9	⑩	⑪	⑫	⑬	14	⑮	16

Quicksort: An Example (Cont.)

Recursive Phase:

①	②	3	④	5	⑥	7	⑧	9	⑩	⑪	⑫	⑬	14	⑮	16

Home Work: Write the Quicksort algorithm and implement it in C.

Correctness

- Guaranteed by the following *loop invariant*:

  "At step k of the algorithm, $pivot \geq x_i$ for all i such that $i < L$, and $pivot < X_j$ for all j such that $j > R$" .

Correctness

- Guaranteed by the following *loop invariant*:

  “At step k of the algorithm, $pivot \geq x_i$ for all i such that $i < L$, and $pivot < X_j$ for all j such that $j > R$”.

  Home Work: Prove it using mathematical induction.

Correctness

- Guaranteed by the following *loop invariant*:

  “At step k of the algorithm, *pivot* $\geq x_i$ for all i such that $i < L$, and *pivot* $< X_j$ for all j such that $j > R$”.

  Home Work: Prove it using mathematical induction.

- **Termination:** When $L = R$.

Choosing a Good Pivot

- Divide-and-conquer algorithms work best when the parts have equal sizes.
- $\therefore$ the closer the pivot is to the middle, the faster the algorithm.
- It is possible to find the median of the sequence (using the *Median finding algorithm*), but it is not worth the effort.
- In fact, choosing a uniform random element suffices.
- If the sequence is in a *uniformly random order*, then we might as well choose the first element as the pivot.

Cost Analysis

- Running time: Depends on the input sequence and pivot.
- If the pivot always partitions the list into two equal parts, then

$$\begin{aligned} T(n) &= 2T(n/2) + \mathcal{O}(n), \quad T(2) = 1, \\ \Rightarrow \quad T(n) &= \mathcal{O}(n \log n). \end{aligned}$$

Cost Analysis

- Running time: Depends on the input sequence and pivot.
- If the pivot always partitions the list into two equal parts, then

$$\begin{aligned} T(n) &= 2T(n/2) + \mathcal{O}(n), \quad T(2) = 1, \\ \Rightarrow T(n) &= \mathcal{O}(n \log n). \end{aligned}$$

- But we can get $\mathcal{O}(n \log n)$ even under much *weaker conditions*!

Cost Analysis

- Running time: Depends on the input sequence and pivot.
- If the pivot always partitions the list into two equal parts, then

$$\begin{aligned} T(n) &= 2T(n/2) + \mathcal{O}(n), \quad T(2) = 1, \\ \Rightarrow T(n) &= \mathcal{O}(n \log n). \end{aligned}$$

- But we can get $\mathcal{O}(n \log n)$ even under much *weaker conditions*!
- However, if the pivot is very close to one side of the sequence, then the running time is much higher.

Cost Analysis

- Running time: Depends on the input sequence and pivot.
- If the pivot always partitions the list into two equal parts, then

$$\begin{aligned} T(n) &= 2T(n/2) + \mathcal{O}(n), \quad T(2) = 1, \\ \Rightarrow T(n) &= \mathcal{O}(n \log n). \end{aligned}$$

- But we can get $\mathcal{O}(n \log n)$ even under much *weaker conditions*!
- However, if the pivot is very close to one side of the sequence, then the running time is much higher.

  Example:

  - If the sequence is already sorted.
  - **Time Complexity:** $\mathcal{O}(n^2)$.

Cost Analysis (Cont.)

- ...
- The quadratic worst case for (almost) sorted sequences can be eliminated
  - by comparing the first, last, and middle elements,
  - and then taking their median (the second largest) as the pivot.

Cost Analysis (Cont.)

- . . .
- The quadratic worst case for (almost) sorted sequences can be eliminated
  - by comparing the first, last, and middle elements,
  - and then taking their median (the second largest) as the pivot.
- **Safer method:** Choose the pivot randomly from among the elements in the sequence.
- **Worst-case complexity:** $\mathcal{O}(n^2)$ (since there is still a chance that the pivot is the smallest element in the sequence.)

Cost Analysis (Cont.)

- . . .
- The quadratic worst case for (almost) sorted sequences can be eliminated
  - by comparing the first, last, and middle elements,
  - and then taking their median (the second largest) as the pivot.
- **Safer method:** Choose the pivot randomly from among the elements in the sequence.
- **Worst-case complexity:** $\mathcal{O}(n^2)$ (since there is still a chance that the pivot is the smallest element in the sequence.)
- However, the likelihood that this worst case occur is very small.

Average-case Complexity

Given a sequence $x_1, \ldots, x_n$, assume that each of the x_i has the same probability of being selected as the pivot.

Average-case Complexity

Given a sequence $x_1, \ldots, x_n$, assume that each of the x_i has the same probability of being selected as the pivot.

Running time when i^{th} smallest element is the pivot:

$$T(n) = \underbrace{n-1}_{\text{partitioning}} + T(i-1) + T(n-i).$$

Average-case Complexity

Given a sequence $x_1, \ldots, x_n$, assume that each of the x_i has the same probability of being selected as the pivot.

Running time when i^{th} smallest element is the pivot:

$$T(n) = \underbrace{n-1}_{\text{partitioning}} + T(i-1) + T(n-i).$$

The average-case time complexity is then given by

$$\begin{aligned} T(n) &= n - 1 + \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)) \\ &= n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \quad \text{[Full history recurrence]} \end{aligned}$$

Full History Recurrences

Full History Recurrence

Definition

A full-history recurrence relation is one that depends on all the previous values of the function, not just on a few of them.

A Simplest Full-history Recurrence Relation

Consider

$$T(n) = c + \sum_{i=1}^{n-1} T(i),$$

where c is a constant and $T(1)$ is given.

A Simplest Full-history Recurrence Relation

Consider

$$T(n) = c + \sum_{i=1}^{n-1} T(i),$$

where c is a constant and $T(1)$ is given.

Solution:

- We use a method that cancels most of the intermediate terms.

A Simplest Full-history Recurrence Relation

Consider

$$T(n) = c + \sum_{i=1}^{n-1} T(i),$$

where c is a constant and $T(1)$ is given.

Solution:

- We use a method that cancels most of the intermediate terms.
- Sometimes called *elimination of history*.

A Simplest Full-history Recurrence Relation

Consider

$$T(n) = c + \sum_{i=1}^{n-1} T(i),$$

where c is a constant and $T(1)$ is given.

Solution:

- We use a method that cancels most of the intermediate terms.
- Sometimes called *elimination of history*.
- Compare $T(n+1)$ with $T(n)$ and subtract to get

$$T(n{+}1) - T(n) = T(n) \;\Rightarrow\; T(n{+}1) = 2T(n) \;\Rightarrow\; T(n{+}1) = T(1)2^n.$$

A Simplest Full-history Recurrence Relation

Consider

$$T(n) = c + \sum_{i=1}^{n-1} T(i),$$

where c is a constant and $T(1)$ is given.

Solution:

- We use a method that cancels most of the intermediate terms.
- Sometimes called *elimination of history*.
- Compare $T(n+1)$ with $T(n)$ and subtract to get

  $$T(n+1) - T(n) = T(n) \Rightarrow T(n+1) = 2T(n) \Rightarrow T(n+1) = T(1)2^n.$$

- **Note:**
  - The claim is true for $T(1)$, $\because$ $T(1) = T(0+1) = T(1)2^0 = T(1)$.

A Simplest Full-history Recurrence Relation

Consider

$$T(n) = c + \sum_{i=1}^{n-1} T(i),$$

where c is a constant and $T(1)$ is given.

Solution:

- We use a method that cancels most of the intermediate terms.
- Sometimes called *elimination of history*.
- Compare $T(n+1)$ with $T(n)$ and subtract to get

  $$T(n{+}1){-}T(n) = T(n) \;\Rightarrow\; T(n{+}1) = 2T(n) \;\Rightarrow\; T(n{+}1) = T(1)2^n.$$

- **Note:**
  - The claim is true for $T(1)$, $\because$ $T(1) = T(0+1) = T(1)2^0 = T(1)$.
  - **Induction step:** If the claim is true for $T(n)$, then

    $$T(n+1) = 2T(n) = T(1)2^n.$$

A Simplest Full-history Recurrence Relation (Cont.)

Correct?

A Simplest Full-history Recurrence Relation (Cont.)

Correct? No!!

A Simplest Full-history Recurrence Relation (Cont.)

Correct? No!!

Example: Set $T(1) = 1$ and $c = 5$, s.t., $T(2) = 6 \neq 2T(1) = 2$.

A Simplest Full-history Recurrence Relation (Cont.)

Correct? No!!

Example: Set $T(1) = 1$ and $c = 5$, s.t., $T(2) = 6 \neq 2T(1) = 2$.

Reason:

- This is an example of carelessly going through an induction proof *ignoring the base case*.

A Simplest Full-history Recurrence Relation (Cont.)

Correct? No!!

Example: Set $T(1) = 1$ and $c = 5$, s.t., $T(2) = 6 \neq 2T(1) = 2$.

Reason:

- This is an induction proof done carelessly, *ignoring the base case*.
- **Note:** The proof does not work for $T(2)$, since $T(2) - T(1) = c$ may not be equal to $T(1)$.

A Simplest Full-history Recurrence Relation (Cont.)

Correct? No!!

Example: Set $T(1) = 1$ and $c = 5$, s.t., $T(2) = 6 \neq 2T(1) = 2$.

Reason:

- This is an example of carelessly going through an induction proof *ignoring the base case*.
- **Note:** The proof does not work for $T(2)$, since $T(2) - T(1) = c$ may not be equal to $T(1)$.
- **Warning:** *One should be very suspicious when a parameter (c in this case) that appears in the expression does not appear in the final solution.*

A Simplest Full-history Recurrence Relation (Cont.)

Correct? No!!

Example: Set $T(1) = 1$ and $c = 5$, s.t., $T(2) = 6 \neq 2T(1) = 2$.

Reason:

- This is an example of carelessly going through an induction proof *ignoring the base case*.
- **Note:** The proof does not work for $T(2)$, since $T(2) - T(1) = c$ may not be equal to $T(1)$.
- **Warning:** *One should be very suspicious when a parameter (c in this case) that appears in the expression does not appear in the final solution.*
- **Correct Solution:** Note that $T(2) = T(1) + c$ (by definition), and that the proof above is correct for all $n \geq 2$.

A Simplest Full-history Recurrence Relation (Cont.)

Correct? No!!

Example: Set $T(1) = 1$ and $c = 5$, s.t., $T(2) = 6 \neq 2T(1) = 2$.

Reason:

- This is an example of carelessly going through an induction proof *ignoring the base case*.
- **Note:** The proof does not work for $T(2)$, since $T(2) - T(1) = c$ may not be equal to $T(1)$.
- **Warning:** *One should be very suspicious when a parameter (c in this case) that appears in the expression does not appear in the final solution.*
- **Correct Solution:** Note that $T(2) = T(1) + c$ (by definition), and that the proof above is correct for all $n \geq 2$.
- $\therefore\ T(n+1) = (T(1) + c)2^{n-1}$.

Books Consulted

1. Chapter 6, Section 4.3 & 4.4 of *Introduction to Algorithms: A Creative Approach* by Udi Manber.

Thank You for your kind attention!