DISCRETE STRUCTURES
Lecture 7. Recursion

Bui Anh Tuan

Advanced Program in Computer Science

Fall, 2018

Content

Big-O notation

Example 1.1

Suppose we write a function, named *sumOfN*, to find the sum of first n positive integers and count the number of assignments used

1. $sum := 0$ <– nrAssigments =1
2. for j in [1..n] do $sum := sum + j$ <– nrAssigments = n
3. Totally, nrAssigments = 1+n.

We denote the number of assignments or number of steps, with respect to the size of the problem, as $T(n)$ then $T(n) = 1 + n$.

Behavior of $T(n)$ when n large?

Big-O notation

Example 1.1

Suppose we write a function, named *sumOfN*, to find the sum of first n positive integers and count the number of assignments used

1. $sum := 0$ <– nrAssigments =1
2. for j in [1..n] do $sum := sum + j$ <– nrAssigments = n
3. Totally, nrAssigments = 1+n.

We denote the number of assignments or number of steps, with respect to the size of the problem, as $T(n)$ then $T(n) = 1 + n$.

Behavior of $T(n)$ when n large? Big-O notation (O stands for order)

Big-O notation

Example 1.1

Suppose we write a function, named *sumOfN*, to find the sum of first n positive integers and count the number of assignments used

1. $sum := 0$ <– nrAssigments =1
2. for j in [1..n] do $sum := sum + j$ <– nrAssigments = n
3. Totally, nrAssigments = 1+n.

We denote the number of assignments or number of steps, with respect to the size of the problem, as $T(n)$ then $T(n) = 1 + n$.

Behavior of $T(n)$ when n large? Big-O notation (O stands for order)

$$T(n) = O(n)$$

.

Big-O notation

Example 1.2

Consider an algorithm with the number of steps with respect to the size n of the problem is $T(n) = 2n^2 - 4n + 1000$.

Observe:

- when n is small, 1000 makes differences.
- when n is large, 1000 and $4n$ don't approximately make any differences.
- when n is larger coefficient 2 does not contribute much.

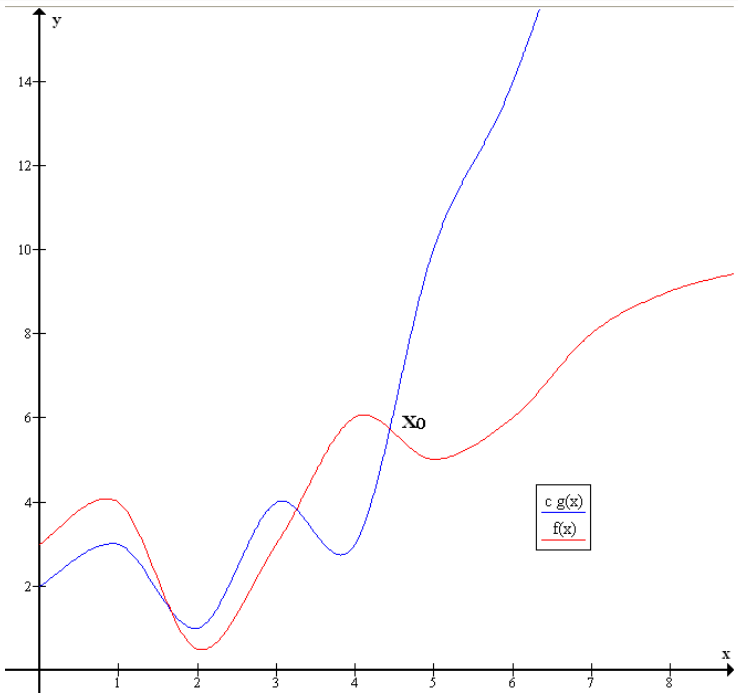
When n large enough, the behavior of $T(n) = 2n^2 - 4n + 1000$ is similar to n^2.

Thus $T(n) = O(n^2)$.

Big-O notation

Definition 1.3

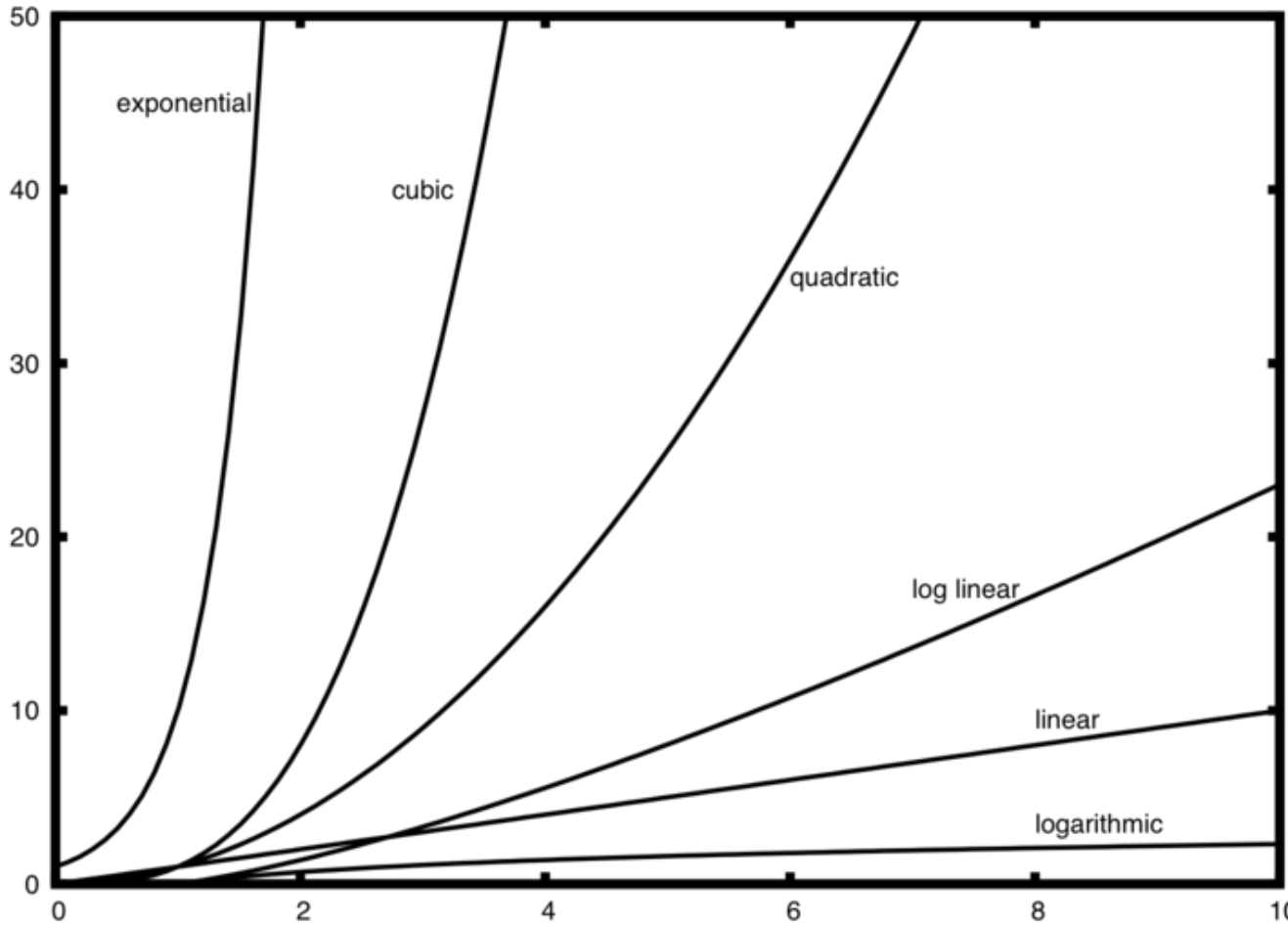
Let f be a real or complex valued function and g a real valued function, both defined on some unbounded subset of the real positive numbers, such that $g(x)$ is strictly positive for all large enough values of x. We say $f(x) = O(g(x))$ if and only if there exists a positive real number M and a real number x_0 such that $f(x) \leq Mg(x), \forall x \geq x_0$.



Big-O notation: common cases

Big-O	Name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	log linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

WIKI



Big-O Notation

Question

Compute

$$O(f(x)) + O(g(x))$$

Example 1.4

$$O(n^2) + O(n) = ?$$

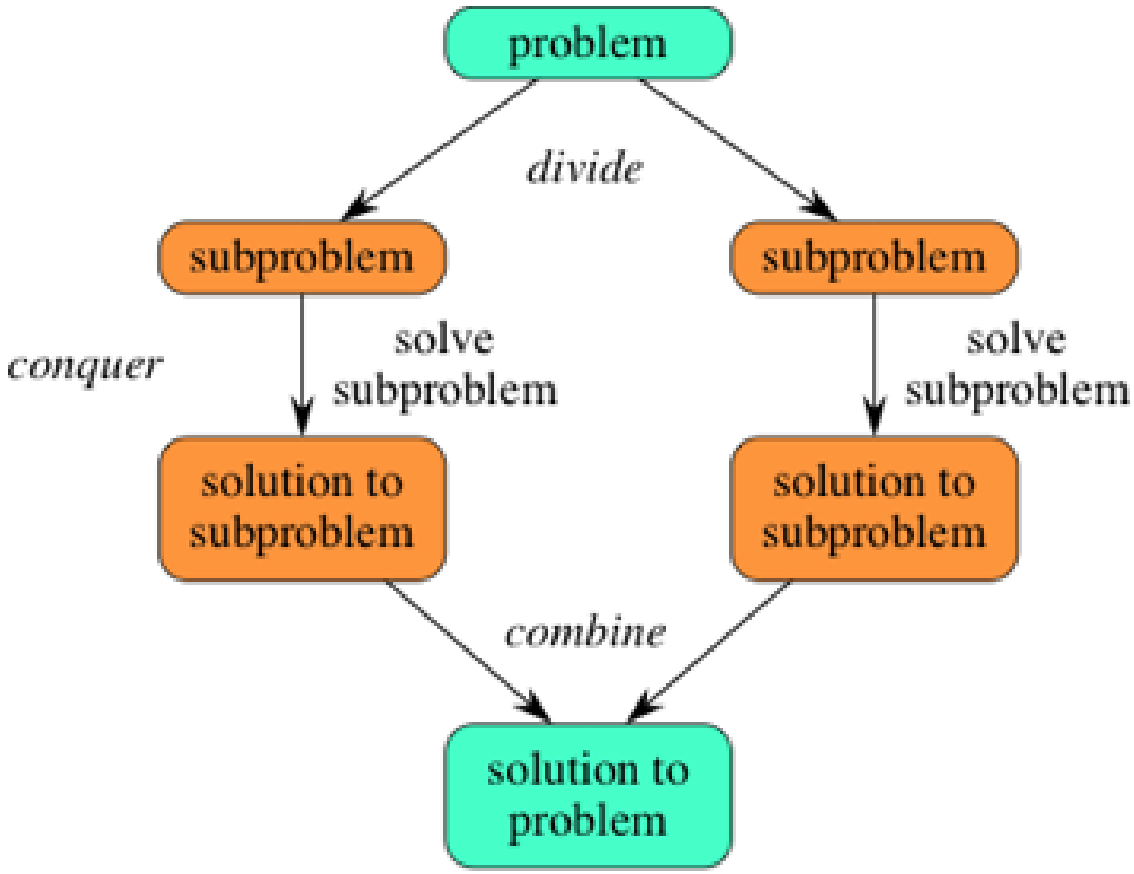
Divide and Conquer

Strategy

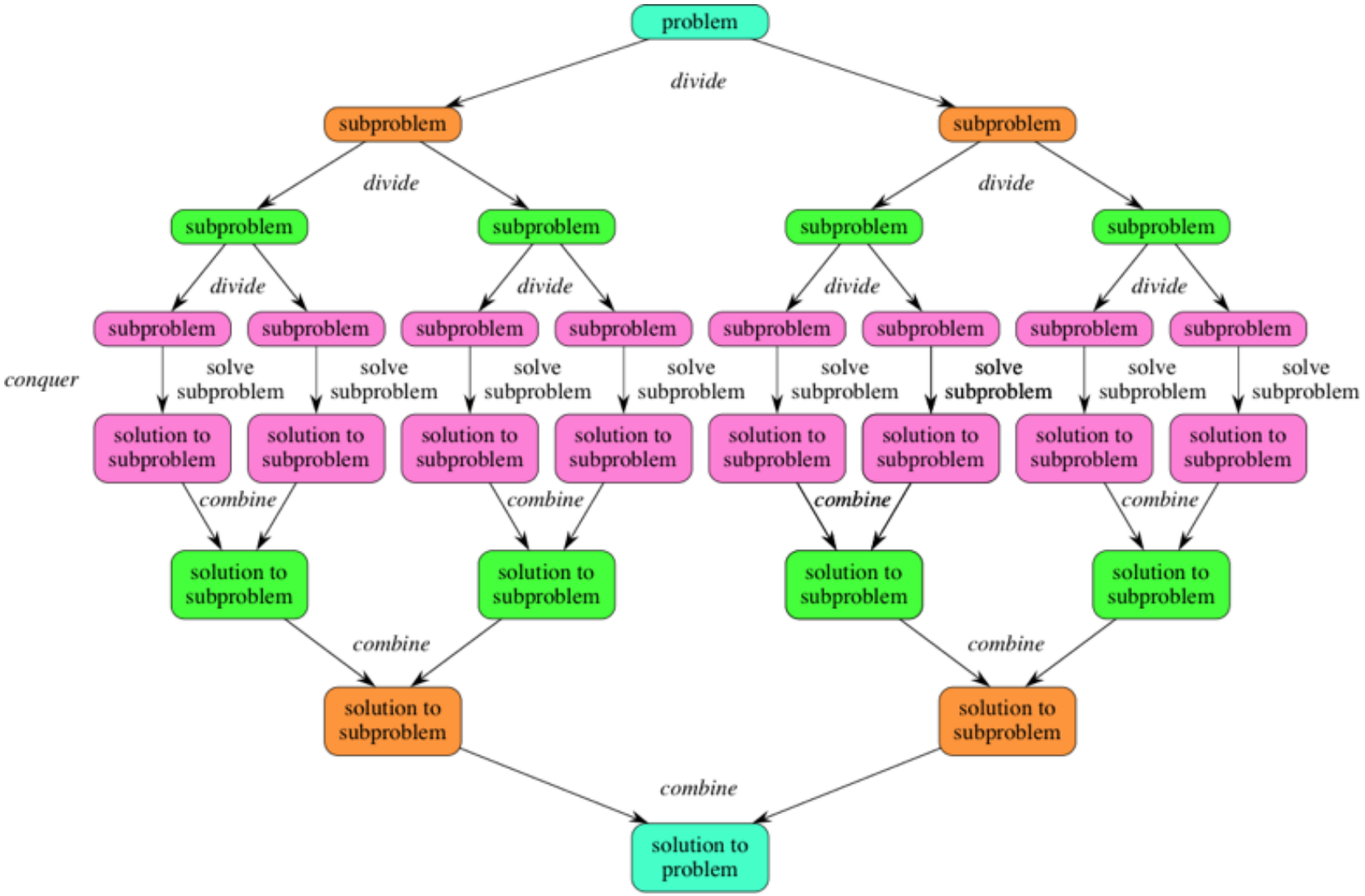
Divide-and-conquer, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem.

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

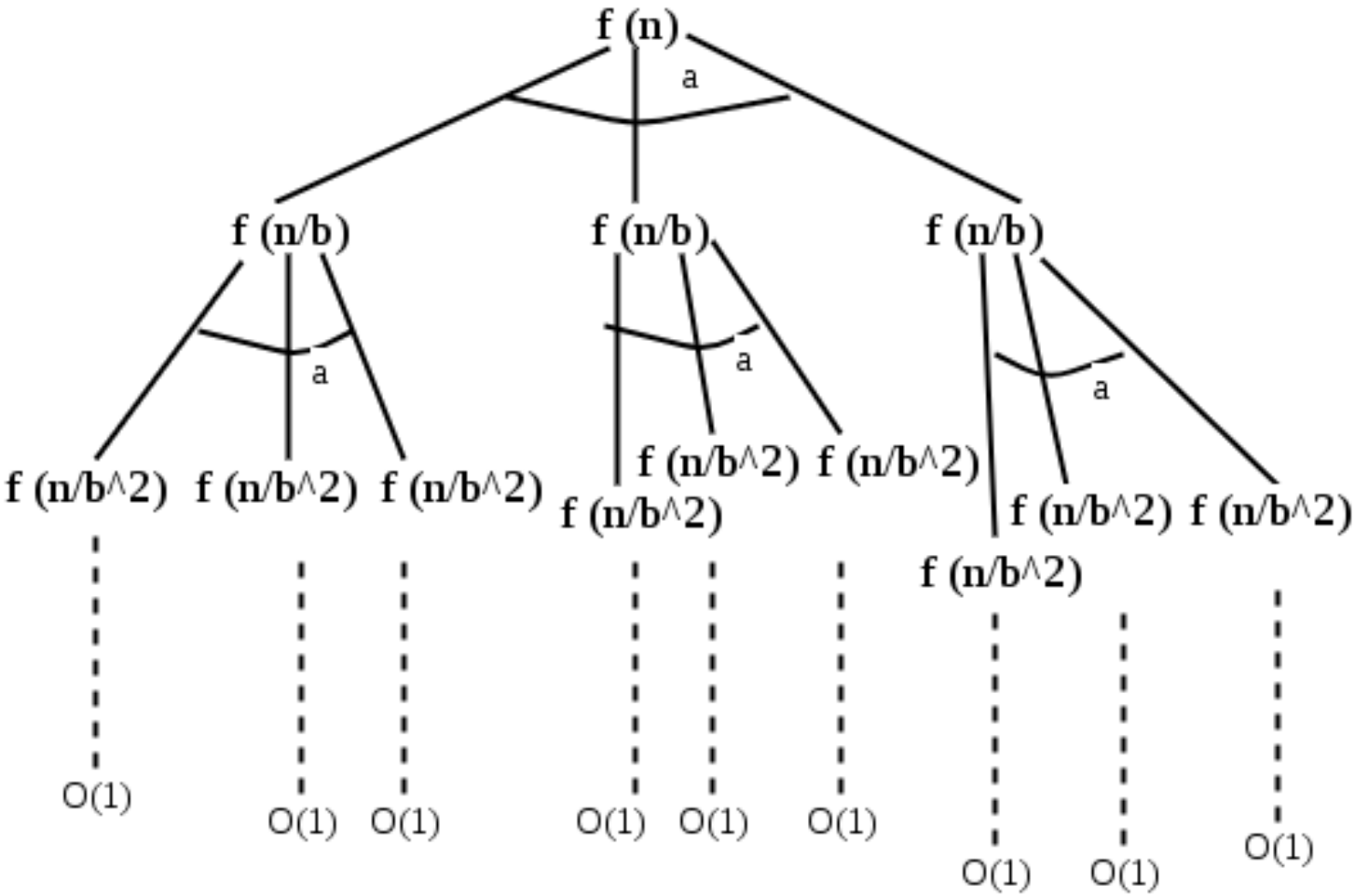
Divide and Conquer



Divide and Conquer



Divide and Conquer



Divide and Conquer

Recurrence

Let $T(n)$ be the total time spent for solving the given problem by divide and conquer strategy. Denote the *dividing and combing* time spent for the problem of size n is $f(n)$. If the sizes of each subproblem are the same in each stage, then $T(n)$ can be presented as a recurrence equation

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

The tree related to this equation is call recursion tree.

Divide and Conquer

Recurrence

Let $T(n)$ be the total time spent for solving the given problem by divide and conquer strategy. Denote the *dividing and combing* time spent for the problem of size n is $f(n)$. If the sizes of each subproblem are the same in each stage, then $T(n)$ can be presented as a recurrence equation

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

The tree related to this equation is call recursion tree.

Example 1.5

1. Binary Search: $T(n) = T(n/2) + O(1)$.

Divide and Conquer

Recurrence

Let $T(n)$ be the total time spent for solving the given problem by divide and conquer strategy. Denote the *dividing and combing* time spent for the problem of size n is $f(n)$. If the sizes of each subproblem are the same in each stage, then $T(n)$ can be presented as a recurrence equation

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

The tree related to this equation is call recursion tree.

Example 1.5

1. Binary Search: $T(n) = T(n/2) + O(1)$.
2. Merge Sort: $T(n) = 2T(n/2) + O(n)$.

Divide and Conquer

Some standard algorithms that are Divide and Conquer algorithms:

- Binary Search: searching algorithm
- Quicksort: sorting algorithm
- Merge Sort: sorting algorithm
- Closest Pair of Points: find the closest pair of points in a set of points in xy–plane
- Strassen's Algorithm: efficient algorithm to multiply two matrices
- ...

Merge Sort

Strategy

Consider the problem of sorting an array $[0..n-1]$. We use divide and conquer strategy. Suppose that at a certain stage, we are going to merge sort the subarray of index $[p..r]$:

1. Divide by finding the number q of the position midway between p and r. Do this step the same way we found the midpoint in binary search: add p and r, divide by 2, and round down.
2. Conquer by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray array $[p..q]$ and recursively sort the subarray array $[q+1..r]$.
3. Combine by merging the two sorted subarrays back into the single sorted subarray array $[p..r]$.

Algorithm Visualization

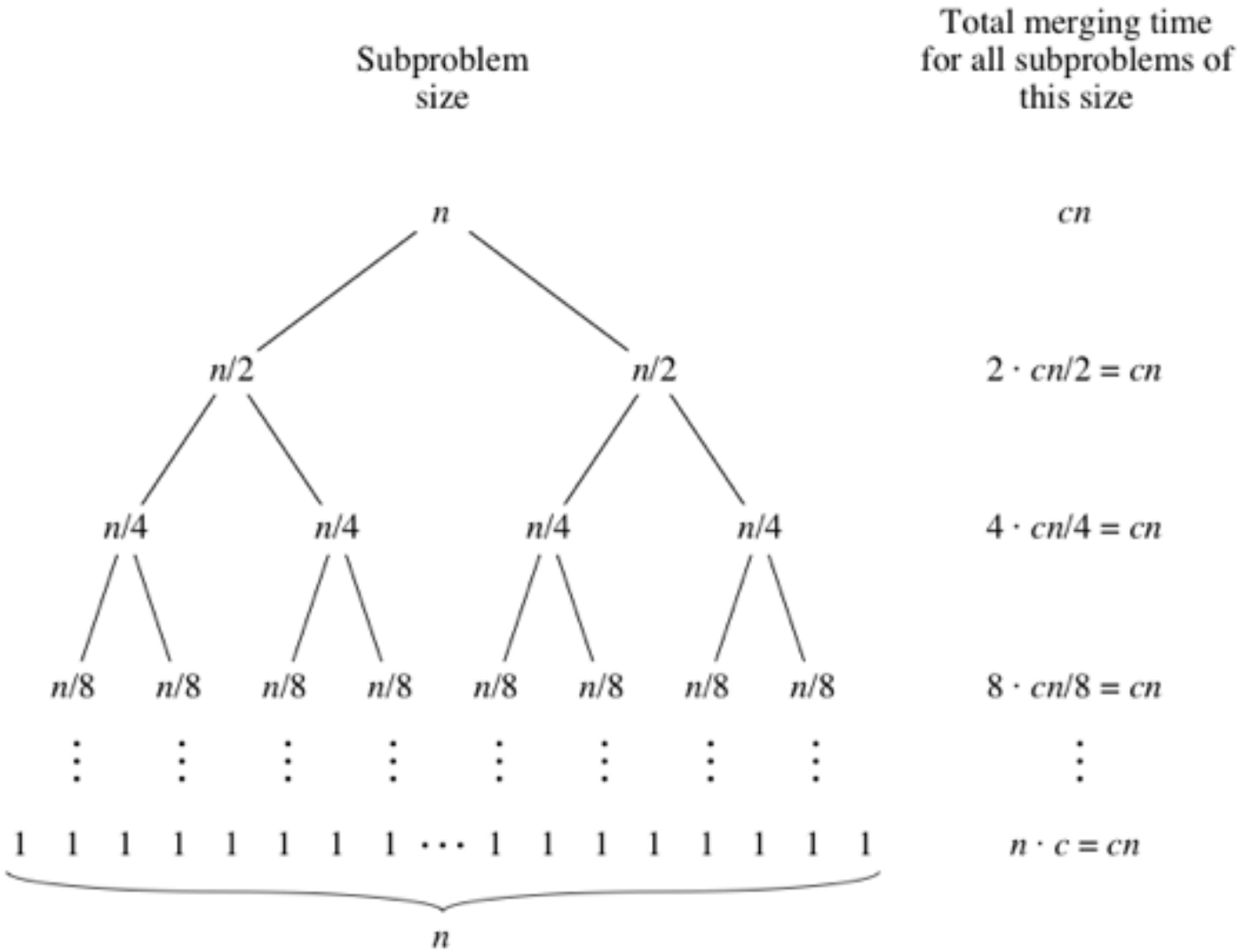
Merge Sort

Question

Complexity of Merge Sort Algorithm?

1. **Divide** takes constant time, regardless of the subarray size: $O(1)$.
2. **Conquer** where we recursively sort two subarrays of approximately $n/2$ elements each, takes some amount of time, but we'll account for that time when we consider the subproblems.
3. **Combine** merges a total of n elements, taking $O(n)$ time.

Merge Sort


Subproblem size
Total merging time for all subproblems of this size
n
cn
n/2
n/2
2 · cn/2 = cn
n/4
n/4
n/4
n/4
4 · cn/4 = cn
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
8 · cn/8 = cn
1 1 1 1 1 1 1 1 ··· 1 1 1 1 1 1 1 1
n · c = cn
n

Binary Search

Strategy

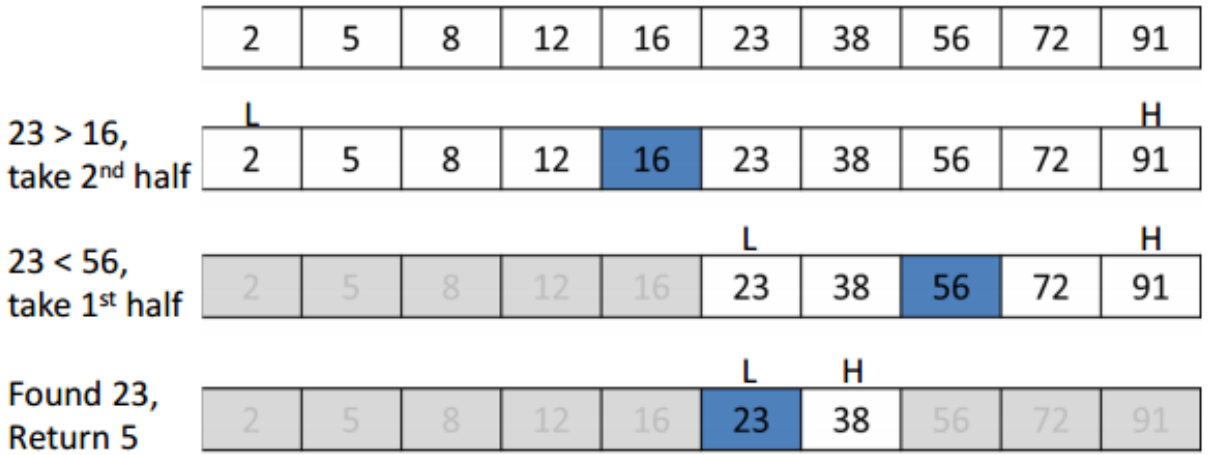
Search a **sorted array** by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Binary Search

Strategy

Search a **sorted array** by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

If searching for 23 in the 10-element array:



Binary Search

Question

Complexity of Binary Search Algorithm?

Binary Search

Question

Complexity of Binary Search Algorithm?

1. **Divide**

Binary Search

Question

Complexity of Binary Search Algorithm?

1. **Divide** takes $O(1)$ time;
2. **Conquer**

Binary Search

Question

Complexity of Binary Search Algorithm?

1. **Divide** takes $O(1)$ time;
2. **Conquer** is accounted for subproblems
3. **Combine**

Binary Search

Question

Complexity of Binary Search Algorithm?

1. **Divide** takes $O(1)$ time;
2. **Conquer** is accounted for subproblems
3. **Combine** takes $O(1)$ time.

Binary Search

Question

Complexity of Binary Search Algorithm?

1. **Divide** takes $O(1)$ time;
2. **Conquer** is accounted for subproblems
3. **Combine** takes $O(1)$ time.

Answer

Complexity of Binary Search Algorithm is $O(\log n)$.

Master Theorem

Theorem 1.6 (Master Theorem)

Master Theorem works for the following type of recurrences:

$$T(n) = aT(n/b) + f(n) \text{ where } a \geq 1 \text{ and } b > 1$$

There are three cases:

1. *If* $f(n) = O(n^c)$ *where* $c < \log_b a$ *then* $T(n) = O(n^{\log_b a})$;
2. *If* $f(n) = O(n^c)$ *where* $c = \log_b a$ *then* $T(n) = O(n^c \log n)$;
3. *If* $f(n) = O(n^c)$ *where* $c > \log_b a$ *then* $T(n) = O(f(n))$;

Example 1.7

1. Binary Search: $T(n) = T(n/2) + O(1)$

Master Theorem

Theorem 1.6 (Master Theorem)

Master Theorem works for the following type of recurrences:

$$T(n) = aT(n/b) + f(n) \text{ where } a \geq 1 \text{ and } b > 1$$

There are three cases:

1. *If* $f(n) = O(n^c)$ *where* $c < \log_b a$ *then* $T(n) = O(n^{\log_b a})$;
2. *If* $f(n) = O(n^c)$ *where* $c = \log_b a$ *then* $T(n) = O(n^c \log n)$;
3. *If* $f(n) = O(n^c)$ *where* $c > \log_b a$ *then* $T(n) = O(f(n))$;

Example 1.7

1. Binary Search: $T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$.

Master Theorem

Theorem 1.6 (Master Theorem)

Master Theorem works for the following type of recurrences:

$$T(n) = aT(n/b) + f(n) \text{ where } a \geq 1 \text{ and } b > 1$$

There are three cases:

1. *If* $f(n) = O(n^c)$ *where* $c < \log_b a$ *then* $T(n) = O(n^{\log_b a})$;
2. *If* $f(n) = O(n^c)$ *where* $c = \log_b a$ *then* $T(n) = O(n^c \log n)$;
3. *If* $f(n) = O(n^c)$ *where* $c > \log_b a$ *then* $T(n) = O(f(n))$;

Example 1.7

1. Binary Search: $T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$.
2. Merge Sort: $T(n) = 2T(n/2) + O(n)$

Master Theorem

Theorem 1.6 (Master Theorem)

Master Theorem works for the following type of recurrences:

$$T(n) = aT(n/b) + f(n) \text{ where } a \geq 1 \text{ and } b > 1$$

There are three cases:

1. *If* $f(n) = O(n^c)$ *where* $c < \log_b a$ *then* $T(n) = O(n^{\log_b a})$;
2. *If* $f(n) = O(n^c)$ *where* $c = \log_b a$ *then* $T(n) = O(n^c \log n)$;
3. *If* $f(n) = O(n^c)$ *where* $c > \log_b a$ *then* $T(n) = O(f(n))$;

Example 1.7

1. Binary Search: $T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$.
2. Merge Sort: $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$

Analysis of an Algorithm

To examine the efficiency of an algorithm, we need to consider:

1. Best case,
2. Worse case,
3. Average case.

Example 1.8

Complexity of quicksort:

- Best case: $O(n \log n)$,
- Worse case: $O(n^2)$,
- Average case: $O(n \log n)$.