Lecture 4 - Divide and Conquer

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Outline

- Divide-and-Conquer
- Merge Sort
- Matrix Multiplication
- Maximum Subarray Problem
- Binary Search
- Fast Powering
- Quick Sort

Divide-and-Conquer

Divide-and-Conquer

A military strategy that appears thousand of years ago

(Collins Dictionary) *A strategy by which someone remains in power by making sure that the people under their control quarrel among themselves and so cannot unite to achieve their aims and overthrow their leader*

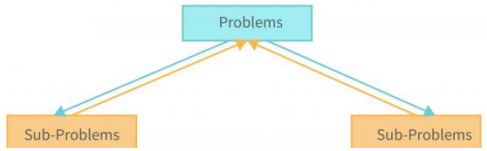
Example: Julius Caesar in the Gallic Wars (58–50 BCE).

- Expand Roman territory by subduing the various Gallic tribes, forming alliances with some while defeating others individually.

Divide-and-Conquer

A strategy to recursively solve the problem

- **Divide** the problem into subproblems : "Smaller" instances of the same problem (so can be further divided with the same strategy)
- **Conquer** the subproblems : (recursively) solve the small problems it can be divided into
- **Combine** the subproblems' solutions into original problem's solution



Not infinite division, since the scale is getting smaller

- **Recursive case**: instances to be solved recursively (non-leaf nodes)

  Base case: small enough instances that directly obtain the result (leaf nodes)

Recursively define the solution to a problem based on the subproblems

How to solve the subproblem?

Merge Sort

Sort with Divide-and-Conquer

Recall sort problem

Input: A sequence of n numbers $< a_1, a_2, a_3, \ldots, a_n >$

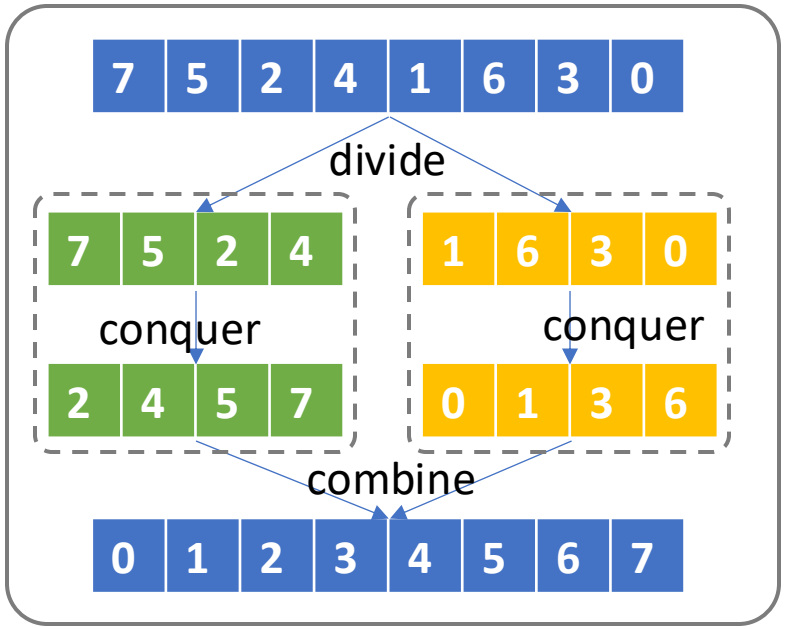
Output: A reordered sequence $< a_{k_1}, a_{k_2}, a_{k_3}, \ldots, a_{k_n} >$ of the input sequence such that $a_{k_1} \leq a_{k_2} \leq a_{k_3} \ldots \leq a_{k_n}$

- Insertion sort is slow: $O(n^2)$
- Think: divide-and-conquer for sorting? (Consider the constraints of divide-and-conquer)

Merge Sort

Consider the constraints of divide-and-conquer

- **Divide** the sequence into subsequences
- **Conquer** (Sort) each part
- **Merge** the different parts into a sorted sequence



sort($a_1, \dots\ a_n$)

if n==1: return

$S^1 = a_1, \dots, a_{n/2}$

$S^2 = a_{n/2+1}, \dots, a_n$

sort (S^1)

sort (S^2)

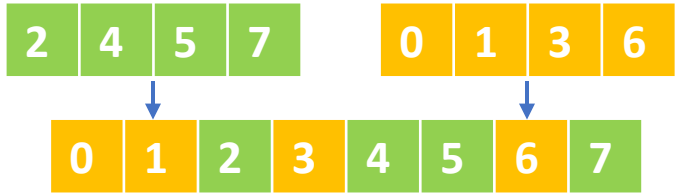
A = merge (S^1, S^2)

Prove the correctness for any n: mathematical induction

Divide is O(1), and the hierarchy is fixed decides the time cost

Merging

Given two sorted sequences $A =< a_1, a_2 \ldots, a_m >$ and $B =< a'_1, a'_2 \ldots, a'_{n-m} >$, produce a single sorted sequence as their combination



Idea: continuously pick the smallest element

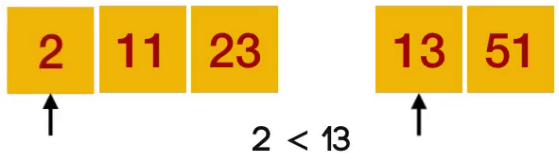
- Observation: the intrinsic order of both sequences remains consistent
- The smallest unpicked element is always at the front of either sequence A or B

Merging

Given two sorted sequences $A =< a_1, a_2 \ldots, a_n >$ and $B =< a'_1, a'_2 \ldots, a'_m >$, produce a single sorted sequence as their combination

1. Compare the leftmost unpicked element of A and B
2. Append the smaller one to the sorted array
3. Update the pointer to the unpicked element
4. Repeat until all elements are picked

```
01. merge(A, B):
02.     ret = []
03.     i, j = 1
04.     while i <=n or j<=m:
05.         if j == m+1 or A[i] <= B[j]:
06.             ret.append(A[i])
07.             i += 1
08.         else:
09.             ret.append(B[j])
10.             j += 1
```



Time complexity: $\Theta(n + m)$

- i and j traverse from beginning to the end of A and B

Auxiliary Space complexity: $\Theta(n + m)$

Merge Sort: Time complexity analysis

The time complexity for divide-and-conquer is not obvious

sort(A):	Time
if len(A) == 1:	
return	
sort$(A_1, \dots, A_{n/2})$	?
sort$(A_{n/2+1}, \dots, A_n)$	?
ret = merge$(A_1, \dots, A_{n/2}, A_{n/2} + 1, \dots, A_n)$	$\Theta(n)$
$A = ret$	$\Theta(n)$
return	

Recurrence

- An equation or inequality that describes a function in terms of its value on smaller inputs.
  - e.g., Fibonacci sequence f(n) = f(n-1) + f(n-2)
- Provide a natural way to characterize the running times of divide-and-conquer algorithms

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & n > 1 \text{ (recursive case)} \\ \Theta(1) & n = 1 \text{ (base case)} \end{cases}$$

Not enough, cannot be compared with other algorithms (better than insertion sort?)

Practice: try to solve this recurrence

sort(A):	Time
if len(A) == 1:	
return	
sort$(A_1, \dots, A_{n/2})$	T(n/2)
sort$(A_{n/2+1}, \dots, A_n)$	T(n/2)
ret = merge$(A_1, \dots, A_{n/2}, A_{n/2+1}, \dots, A_n)$	$\Theta(n)$
$A = ret$	$\Theta(n)$
return	

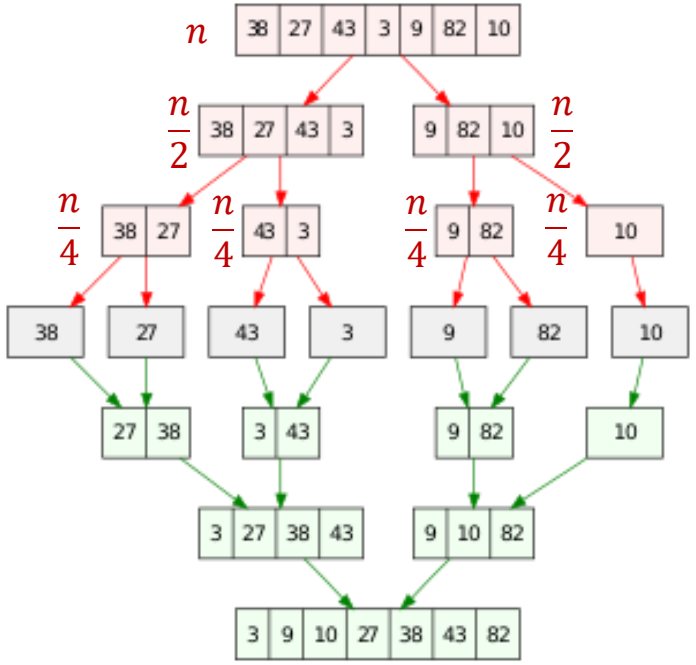
Solve the Recurrence: Direct Calculation

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn$$
$$= 4T\left(\frac{n}{4}\right) + 2cn$$
$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn$$
$$= 8T\left(\frac{n}{8}\right) + 3cn$$
$$= \cdots$$
$$= 2^m T\left(\frac{n}{2^m}\right) + mcn, m \in N^+$$

- When $m = \log_2 n, T(\frac{n}{2^m}) = \Theta(1)$
- $T(n) = nT(1) + nclog_2 n$
- $T(n) = \Theta(n \log n)$

Solve the Recurrence: Count Operations Holistically

- No matter how the arrays are divided, the total number of element iteration during merging procedure in each layer is n
- The number of layers in the tree: $\log_2 n$
- $T(n) = \Theta(n \log n)$



Solve the Recurrence

Direct expansion and holistic operation counting can be not rigorous and limited to specific types of problems

Generic methods to solve the recurrence:

- **Recursion-tree method**: converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion
- **Substitution method**: guess a bound and then use mathematical induction to prove our guess correct
- **Master theorem**: provides bounds for recurrences

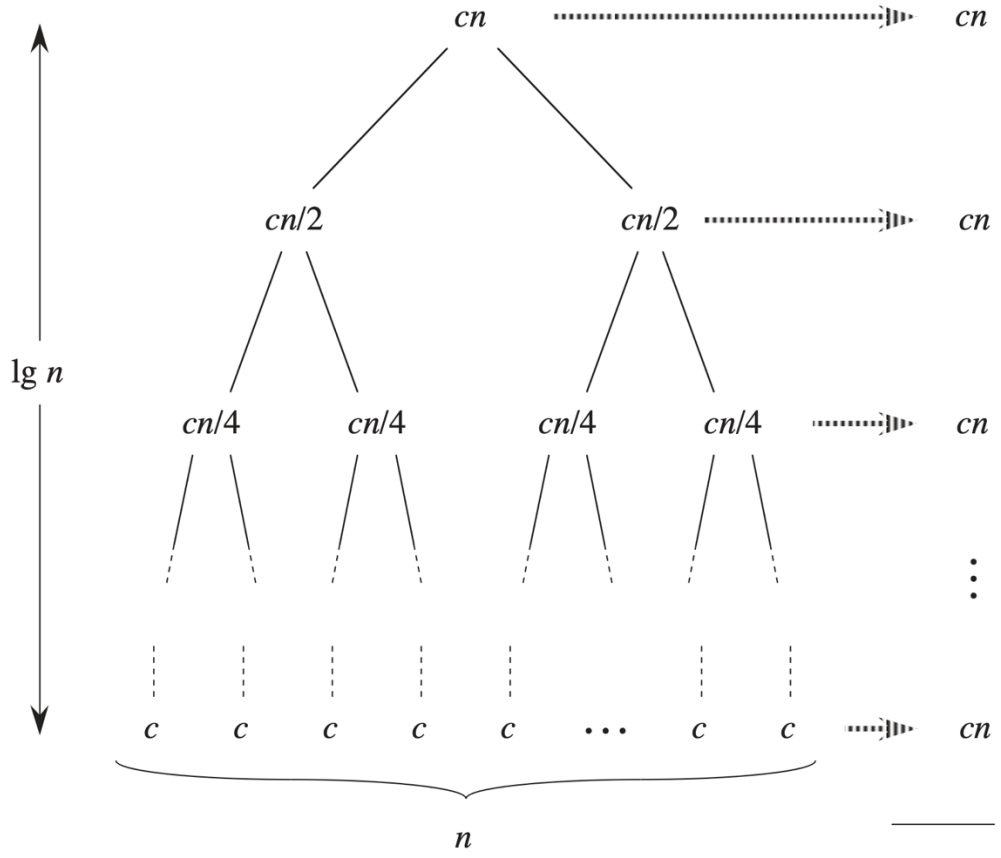
Independent of the problem, only care about the recurrence

Solve the Recurrence: Recursion Tree

Expand the recurrence in a tree structure for visualization

- Nodes represent the non-recursive costs
- Branches represent recursive calls
- Subtrees represent the cost of subproblems

Promote intuition on deriving the complexity

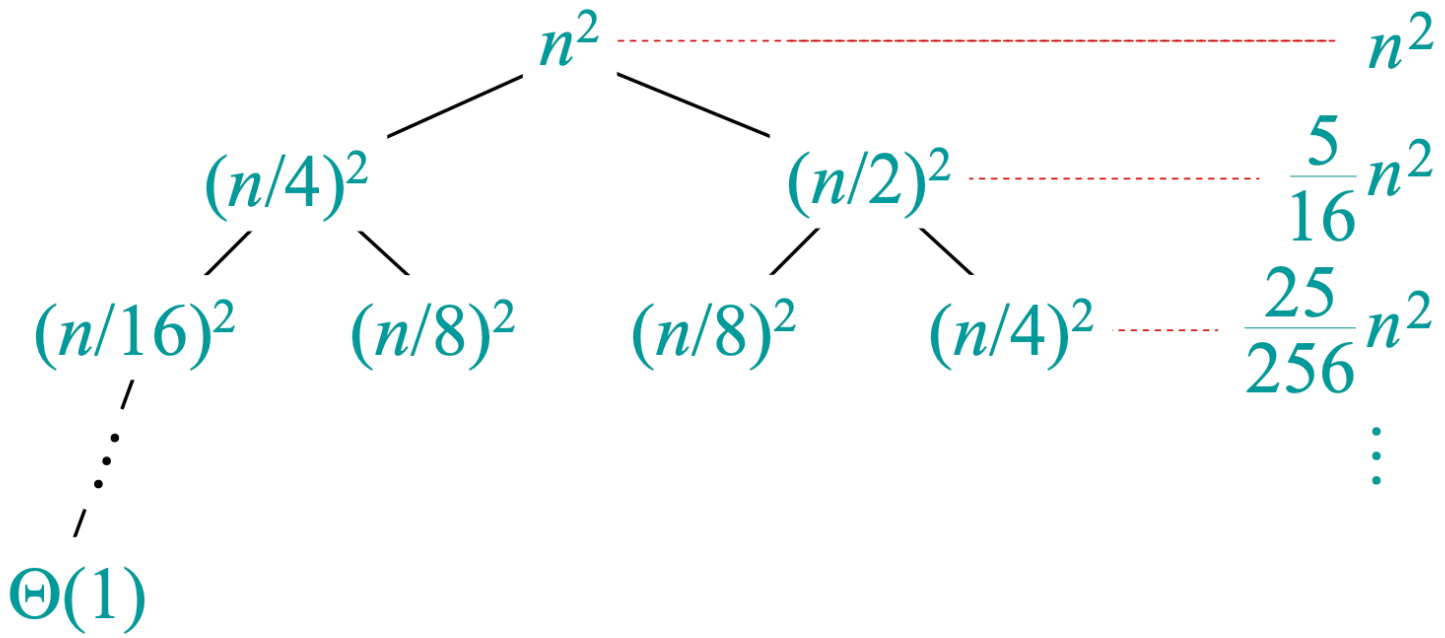


Solve the Recurrence: Recursion Tree

Practice: $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2$

Solve the Recurrence: Recursion Tree

Practice: $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2$



$$\mathrm{T}(n) = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right) = \Theta(n^2)$$

Solve the Recurrence: Substitution

Guess a bound and then use mathematical induction to prove the guess correct

$$T(n) = 2T(n/2) + cn$$

Example: guess $\mathrm{T}(n) \leq cn\log_2 n + cn$ (prove O and Ω separately)

Proof by mathematical induction:

- Base case: for $n = 1, \mathrm{T}(1) = c$, the guess holds
- Induction: given $\mathrm{T}(k) \leq ck\log_2 k + ck$ for $k < n$, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \leq cn\log_2 \frac{n}{2} + 2cn = cn(\log_2 n - 1) + 2cn = cn\log_2 n + cn$$

- Conclusion: $T(n) = O(n\log n)$ if n is the power of 2

Also holds for n not a power of 2

- Represent the most recent power of 2 larger than n as m, obviously $m \leq 2n$ (consider the binary representation, left shift the highest bit)
- $\mathrm{T}(n) \leq \mathrm{T}(m) \leq c(m)\log(m) + c(m) \leq 2cn\log 2n + 2cn = 2cn\log n + 4cn$
- $T(n) = O(n\log n)$

Solve the Recurrence: Substitution

Practice: $T(n) = 4T(n/2) + n, T(1) = 1$, prove $T(n) = O(n^3)$

Guess $T(k) \leq ck^3$, with $c \geq 1$

Proof by mathematical induction:

- Base case: $T(1) = 1$ hods the assumption
- Prove $T(n) \leq cn^3$ when $n \geq 2$ and $T(k) \leq ck^3$ for all $k < n$ by induction:
  - $T(n) = 4T(\frac{n}{2}) + n \leq 4c\left(\frac{n}{2}\right)^3 + n = \frac{c}{2}n^3 + n = cn^3 - \left(\frac{c}{2}n^3 - n\right)$
  - Since $n \geq 2$, and $c \geq 1$, $\frac{2}{n^2} \leq \frac{1}{2} < 1 \leq c$
  - Therefore, $\left(\frac{c}{2}n^3 - n\right) \geq 0$, then we have $T(n) \leq cn^3$

Solve the Recurrence: Master Theorem

Division and combination

Directly solve the recursion formula formed $\mathrm{T(n)} = a\mathrm{T}(n/\mathrm{b}) + f(n)$

#Subproblems Subproblem size

Core idea: compare $f(n)$ with $n^{\log_b a}$

Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

- $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor)

Case 2: $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} lg\, n\right)$

- $f(n)$ and $n^{\log_b a}$ grow at similar rates

Case 3: $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$ and $f(n)$ satisfies the regularity condition that $af(n/b) \leq cf(n)$ for some constant c < 1 $\Rightarrow T(n) = \Theta(f(n))$

- $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

Solve the Recurrence: Master Theorem

Solve the recurrence for merge sort

$$\mathrm{T}(n) = 2\mathrm{T}(n/2) + \Theta(n)$$

- #Subproblems (a) = 2, divisor (b) = 2, division and combination $f(n) = \Theta(n)$
  - $n^{\log_b a} = n$ (same speed as f(n), check case 2!)
  - $f(n) = \Theta(n) = \Theta(n^{\log_b a})$, so case 2 holds
- Therefore, $\mathrm{T}(n) = \Theta(n\log n)$

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

Case 2: $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} lg\, n\right)$

Solve the Recurrence: Master Theorem

EX. $T(n) = 4T(n/2) + n$

EX. $T(n) = 4T(n/2) + n^2$

EX. $T(n) = 4T(n/2) + n^3$

Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

- $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor)

Case 2: $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} lg\, n\right)$

- $f(n)$ and $n^{\log_b a}$ grow at similar rates

Case 3: $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$ and $f(n)$ satisfies the regularity condition that $af(n/b) \leq cf(n)$ for some constant c < 1 $\Rightarrow T(n) = \Theta(f(n))$

- $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor)

Matrix Multiplication

Matrix Multiplication

Given two matrices $A = [a_{ij}] \in R^{N\times N}, B = [b_{ij}] \in R^{N\times N}$, get their matrix multiplication

$C = [c_{ij}] = \mathrm{A} \cdot \mathrm{B}$

$$\begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}, \qquad c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Solution 1: direct calculation

- Time complexity: $\Theta(n^3)$

Divide and conquer?

Supposing $n = 2^k$ to ease the problem

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

A Simple Divide-and-Conquer Algorithm

Key idea: Divide a $n \times n$ matrix into four $(n/2) \times (n/2)$ submatrices

Supposing $n = 2^k$ to ease the problem

Divide

- $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$
- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Conquer $A_{ij}B_{jk}$ (smaller instances of MM)

$T(n) = 8T(n/2) + 4n^2/4$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n × n matrix
 3  if n == 1
 4      c11 = a11 · b11
 5  else partition A, B, and C as in equations (4.9)
 6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

- 8 $(n/2) \times (n/2)$ matrix multiplications
- 4 $(n/2) \times (n/2)$ submatrices additions

A Simple Divide-and-Conquer Algorithm

Key idea: Divide a $n \times n$ matrix into four $(n/2) \times (n/2)$ submatrices

Complexity: $T(n) = 8T(n/2) + \Theta(n^2)$

- 8 $(n/2) \times (n/2)$ matrix multiplications
- 4 $(n/2) \times (n/2)$ submatrices additions
- $n^{\log_b a} = n^3$ faster than f(n), case 1!
- Case 1: when $\epsilon = 1$
  - $f(n) = \Theta(n^2) = O\left(n^{\log_b a - \varepsilon}\right)$
  - $\mathrm{T}(n) = \Theta(n^3)$ No faster!

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```
1  n = A.rows
2  let C be a new n × n matrix
3  if n == 1
4      c11 = a11 · b11
5  else partition A, B, and C as in equations (4.9)
6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10 return C
```

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

Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

Strassen's Method

Performing only 7 recursive multiplications instead of 8

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$P_1 = A_{11} \cdot (B_{12} - B_{22})$$
$$P_2 = (A_{11} + A_{12}) \cdot B_{22}$$
$$P_3 = (A_{21} + A_{22}) \cdot B_{11}$$
$$P_4 = A_{22} \cdot (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$
$$P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$
$$P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$$

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

- 7 $(n/2) \times (n/2)$ matrix multiplications
- 18 $(n/2) \times (n/2)$ submatrices additions

Analysis of Strassen's algorithm

Performing only 7 recursive multiplications instead of 8

Complexity: $\mathrm{T}(n) = 7\mathrm{T}(n/2) + \Theta(n^2)$

- 7 $(n/2) \times (n/2)$ matrix multiplications
- 18 $(n/2) \times (n/2)$ submatrices additions
  - $n^{\log_b a} \approx n^{2.81}$, faster than f(n), case 1!
  - $f(n) = \Theta(n^2)$, e.g., when $\epsilon = 0.5, f(n) = \Theta(n^2) = \mathrm{O}\left(n^{\log_b a - \varepsilon}\right)$
  - $\mathrm{T}(n) = \Theta\left(n^{\log_2 7}\right)$

Strassen's algorithm beats ordinary algorithms in practice for $n \geq 32$

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

Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

Maximum Subarray Problem

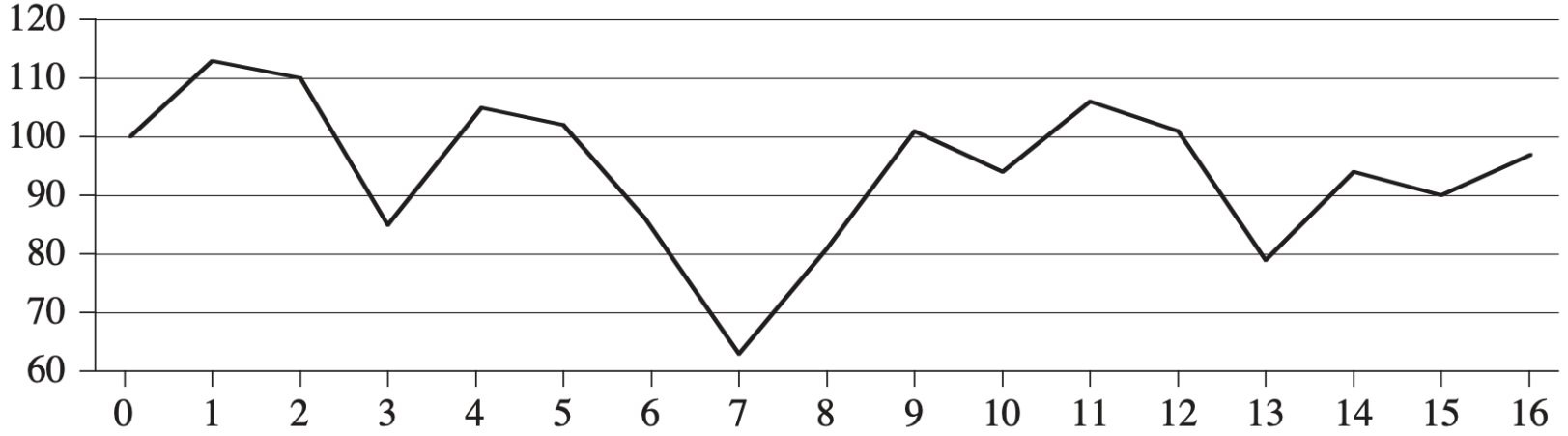
Stock Investment

Suppose that you invest in a corporation. Given the every-day price of the stock in the future, you can buy one unit of stock (only once) and sell it at a later date.

Goal: maximize the profit

Formulation: Given an array $a_1, a_2, \ldots, a_n$, calculate $\max_{1 \le i < j \le n} a_j - a_i$

- Brute Force: try every pair of $< i, j >$ $\Theta(n^2)$



Maximum Subarray Problem

Transform the problem: find a sequence of days over which the net change from the first day to the last is maximum

Formulation: Given an array $b_1, b_2, \ldots, b_{n-1}$, where $b_i = a_{i+1} - a_i$, find the nonempty, contiguous subarray of B whose values have the largest sum $\max_{1 \le i \le j \le n} \sum_{k=i}^{j} b_k$

Think: solve with divide-and-conquer

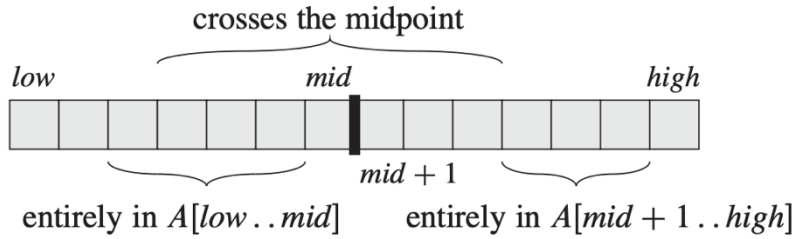
Maximum Subarray Problem

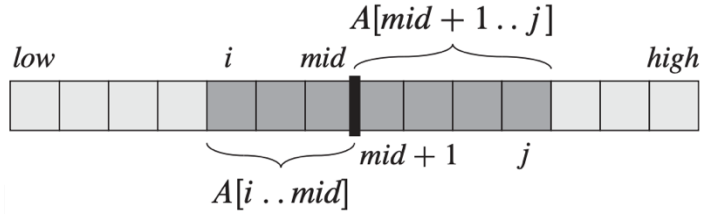
Formulation: Given an array $b_1, b_2, \ldots, b_{n-1}$, where $b_i = a_{i+1} - a_i$, find the nonempty, contiguous subarray of A whose values have the largest sum $\max_{1 \le i \le j \le n} \sum_{k=i}^{j} b_k$

Key idea: divide the array into subarrays $b_{:m}$ and $b_{m+1:}$ and search each for the maximum

Three possibilities:

- Entirely in the subarray $b_{:m}$, i.e., $1 \le i \le j \le m$
- Entirely in the subarray $b_{m+1:}$, i.e., $m + 1 \le i \le j \le n$
- Crossing the midpoint, i.e., $1 \le i \le m < j \le n$





Maximum Subarray Problem

Find crossing-maximum subarray $\Theta(n)$: the maximum subarray ending at m & the maximum subarray starting at $m + 1$

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

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

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n), \mathrm{T}(1) = \Theta(1)$ just like merge sort $\Theta(n\log n)$

Maximum Subarray Problem

A linear-time solution

Formulation: given $a_0, a_1, \ldots, a_{n-1}$, where $a_0 = 0, a_i = \sum_{k=1}^{i} b_k$, get $\max_{1 \le i < j \le n} (a_j - a_i)$

- Solution: $\max_{1 \le j \le n} (a_j - \min_{1 \le i < j} a_i)$
- Traverse the array from left to right (pointer j)
- Update the smallest value so far

Complexity: $\Theta(n)$

```
01. min_val = a[1]
02. ret = -inf
03. for i in range(2, n+1):
04.     ret = max(ret, a[i] - min_val)
05.     min_val = min(min_val, a[i])
```

Binary Search

Search In a Collection

Problem: search for a given key (k) in a sorted array $a_1, a_2, \ldots a_n$, with $a_1 \leq a_2 \leq \cdots \leq a_n$, return index if exists and -1 otherwise

Like searching in a static collection

0	1	2	3	4	5	6	7	8
2	5	10	12	15	20	25	31	40

Example:

- Search for 6: -1
- Search for 10: a[2]
- Search for 15: a[4]

- Simple approach: check the element in array one by one
  - Time complexity $\Theta(n)$, auxiliary space complexity $\Theta(1)$
- Data structure: create hash table
  - Time complexity $\Theta(n)$(insert each of them), auxiliary space complexity $\Theta(n)$

Time and space efficient solution? (with divide-and-conquer) —— key point: sorted

Search with Divide-and-Conquer

Problem: search for a given key (k) in a sorted array $a_1, a_2, \dots a_n$, with $a_1 \leq a_2 \leq \cdots \leq a_n$, return index if exists and -1 otherwise

- **Divide** the sequence into two parts $a_1, a_2, \dots a_{n/2}$ and $a_{n/2+1}, a_{n/2+2}, \dots a_n$
- **Conquer** (Search for the key) in each part

Recurrence $T(n) = 2T(n/2) + \Theta(1)$

- $n^{\log_b a} = n^1$ f(n) grows slower: case 1
- $T(n) = \Theta(n)$——No faster!

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

Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

Binary Search

Problem: search for a given key (k) in a sorted array $a_1, a_2, \ldots a_n$, with $a_1 \leq a_2 \leq \cdots \leq a_n$, return index if exists and -1 otherwise

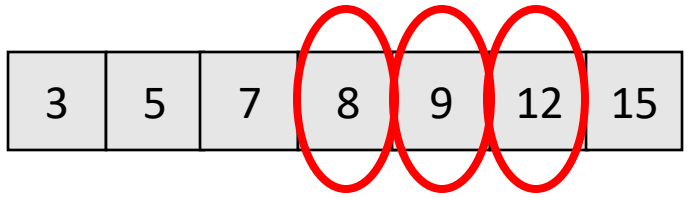
- **Divide** the sequence into two parts $a_1, a_2, \ldots a_{n/2}$ and $a_{n/2+1}, a_{n/2+2}, \ldots a_n$
- **Conquer** (Search for the key) in each part

Recall: looking up a word in a dictionary

Observation: we can know which part may contain the key by comparing $a_{n/2}$ with k

- $a_{n/2} = k$: return $n/2$
- $a_{n/2} > k$: cannot be in $a_{n/2+1}, a_{n/2+2}, \ldots a_n$
- $a_{n/2} < k$: cannot be in $a_1, a_2, \ldots a_{n/2}$

Example: find 9



Binary Search

Problem: search for a given key (k) in a sorted array $a_1, a_2, \dots a_n$, with $a_1 \leq a_2 \leq \cdots \leq a_n$, return index if exists and -1 otherwise

- **Divide** the sequence into two parts $a_1, a_2, \dots a_{n/2}$ and $a_{n/2+1}, a_{n/2+2}, \dots a_n$
- **Conquer** (Search for the key in) each part

Recall: looking up a word in a dictionary

Observation: we can know which part may contain the key by comparing $a_{n/2}$ with k

```
01. search(l, r, k):
02.     if l > r: return -1
03.     if l == r: return l if a[l] == k else -1
04.     m = (l + r) / 2
05.     if a[m] == k: return m
06.     if a[m] > k: return search(l, m - 1, k)
```

Recurrence: $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$

- $n^{\log_b a} = n^0 = 1$, same growth as f(n), case 2
- $T(n) = \Theta(\log n)$

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

Case 2: $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

Example: Missing Number

Given $n-1$ distinct natural numbers $a_1, a_2, \ldots, a_{n-1}$ from 0 to n - 1, one number between 0 and $n-1$ is missing. Without knowing the sequence, you can query (i, j) to learn the j-th lowest bit of a_i's binary representation

- Example: sequence is 3, 5, 2, 1, 0. When querying (1, 2), the 2-nd lowest bit of $a_1 = 5(101)$ is 0.

Problem: find out the missing number with the least number of queries

- Naïve solution: query all the bits for all the numbers $O(n\log n)$
- Faster solution: divide-and-conquer

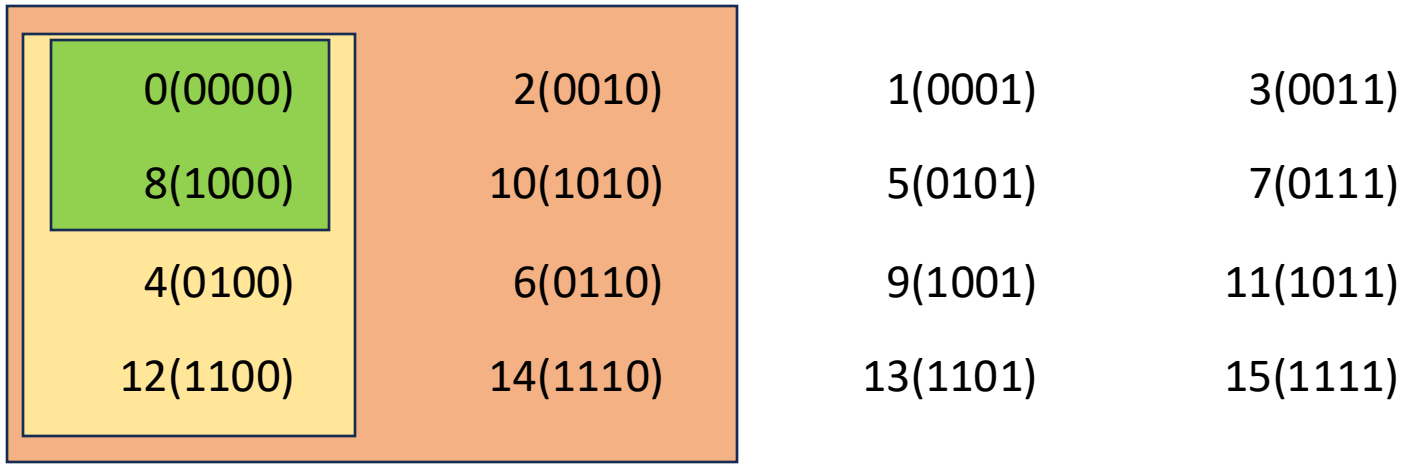
Example: Missing Number

Given $n-1$ distinct natural numbers $a_1, a_2, \ldots, a_{n-1}$ from 0 to n - 1, one number between 0 and $n-1$ is missing. Without knowing the sequence, you can query (i, j) to learn the j-th lowest bit of a_i's binary representation

Hint: check all the n numbers' 1st bit, then we know the missing number's 1st bit

- Example: n=16, we find 8 numbers with 1st bit as 1 and 7 numbers with 1st bit as 0, then the missing value must have the 1st bit as 0
- With n queries, we reduce half

Divide the numbers into different regions based on the lowest distinguishable bit



Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

- $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor)

Case 2: $f(n) = \Theta\left(n^{\log_b a}\right)$ for some constant $k \geq 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

- $f(n)$ and $n^{\log_b a}$ grow at similar rates

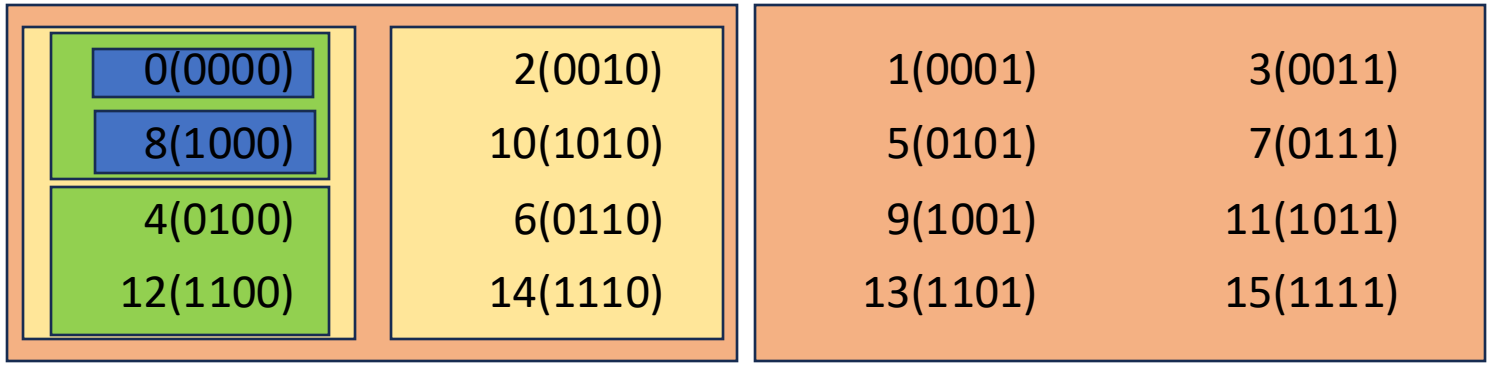
Case 3: $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$ and $f(n)$ satisfies the regularity condition that $af(n/b) \leq cf(n)$ for some constant $c < 1 \Rightarrow T(n) = \Theta(f(n))$

- $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor)

- **Conquer** (find the missing number in) the subregion

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

- Question: solve the recurrence?
- $\Theta(n)$



Fast Powering

Number Powering

Given a base a and a natural number exponent n, find the value of a^n

Naïve algorithm: Multiply the base by itself for the exponent number of times $\Theta(n)$

- What if n is large? (e.g., 1e18)

```
01. ret = 1
02. for i in range(n):
03.     ret = ret * a
```

Solve number powering with divide-and-conquer

- **Divide:** Powers can be broken down into smaller powers

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & if\ n\ is\ even; \\ a^{(n-1)/2} \times a^{(n-1)/2} \times a & if\ n\ is\ odd. \end{cases}$$

- **Conquer** (calculate) each $a^{\left\lfloor \frac{n}{2} \right\rfloor}$

Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

- $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor)

Case 2: $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

- $f(n)$ and $n^{\log_b a}$ grow at similar rates

Case 3: $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$ and $f(n)$ satisfies the regularity condition that $af(n/b) \leq cf(n)$ for some constant $c < 1 \Rightarrow T(n) = \Theta(f(n))$

- $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor)

```
01. power(a, n):
02.     if n == 1:
03.         return a
04.     b = power(a, n//2)
05.     r = 1 if n%2==0 else a
06.     return b * b * r
```

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \quad \Rightarrow T(n) = \Theta(\log n)$$

Extension to Fibonacci Sequence

Problem: input an integer n, find the n^{th} Fibonacci number

- Fibonacci sequence

$$F_n = \begin{cases} 0 & if\ n = 0; \\ 1 & if\ n = 1; \\ F_{n-1} + F_{n-2} & if\ n \geq 2. \end{cases}$$

Solution 1: direct recursive calling

```
01. fib(n):
02.     if n<=1:
03.         return n
04.     else:
05.         return fib(n-1) + fib(n-2)
```

$$\mathrm{T}(n) = \mathrm{T}(n-1) + \mathrm{T}(n-2) + \Theta(1)$$

$$\mathrm{T}(1) = \Theta(1), \mathrm{T}(0) = \Theta(1)$$

Exactly the $(n+1)$-st Fibonacci number

Seems not good, 2T(n-2) <= T(n), exponential growth

Binet's formula: $F(n) = \frac{\phi^n - (-\phi^{-1})^n}{\sqrt{5}} = \Theta(\phi^n)$

ϕ: golden ratio

Extension to Fibonacci Sequence

Problem: input an integer n, find the n^{th} Fibonacci number

- Fibonacci sequence

$$F_n = \begin{cases} 0 & if\ n = 0; \\ 1 & if\ n = 1; \\ F_{n-1} + F_{n-2} & if\ n \geq 2. \end{cases}$$

Solution 2: non-recursive calculate one by one

```
01.  f = [0, 1]
02.  for u in range(2, n+1):
03.      f.append(f[-1] + f[-2])
```

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

Any faster solution?

Extension to Fibonacci Sequence

Problem: input an integer n, find the n^{th} Fibonacci number

Solution 3: Divide and Conquer (Fast powering)

- Fibonacci numbers can be expressed with matrix powering $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$
  - Base ($n = 1$): $\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$
  - Inductive step ($n \geq 2$):

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Extension to Fibonacci Sequence

Problem: input an integer n, find the n^{th} Fibonacci number

Solution 3: Divide and Conquer (Fast powering)

- Fibonacci numbers can be expressed with matrix powering $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$
- Compute F(n) by raising the matrix to the n^{th} power

```
01. power(A, n):
02.     if n == 1:
03.         return A
04.     B = power(A, n//2)
05.     r = 1 if n % 2 == 0 else A
06.     return B * B * r
```

```
01. f(n):
02.     A = matrix([[1, 1],
03.                 [1, 0]])
04.     F = power(A, n)
05.     return F[0, 1]
```

$\mathrm{T}(n) = \mathrm{T}(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\log n)$

Since the problem size is simply decreased, also called Decrease-and-Conquer

Quick Sort

Sort with Divide-and-Conquer 2

Input: A sequence of n numbers $< a_1, a_2, a_3, \ldots, a_n >$

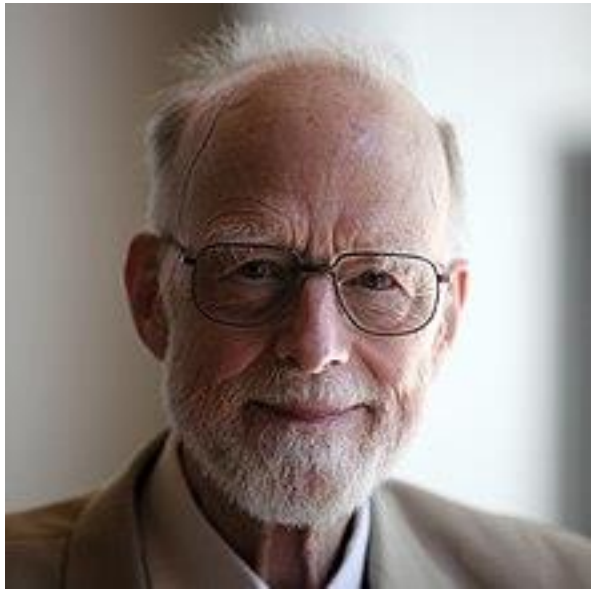
Output: A reordered sequence $< a_{k_1}, a_{k_2}, a_{k_3}, \ldots, a_{k_n} >$ of the input sequence such that $a_{k_1} \leq a_{k_2} \leq a_{k_3} \ldots \leq a_{k_n}$

- Insertion sort : time $\Theta(n^2)$, auxiliary space $\Theta(1)$
- Merge sort: time $\Theta(n\log n)$, auxiliary space $\Theta(n)$
- Both time and space efficient method?

Quick Sort

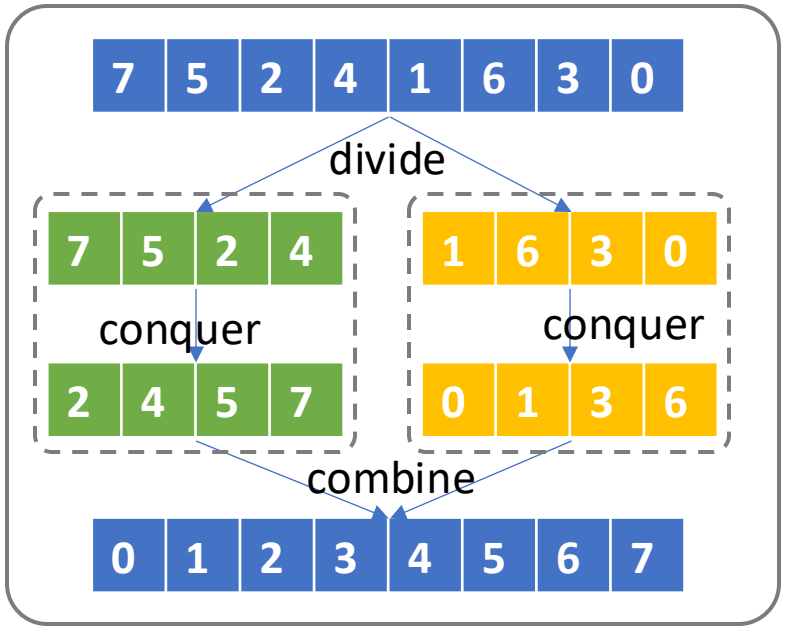
A most commonly used sorting algorithm

- Proposed by C.A.R. Hoare in 1962
- Another divide-and-conquer sorting algorithm
- Inplace sort (more space efficient than merge sort)

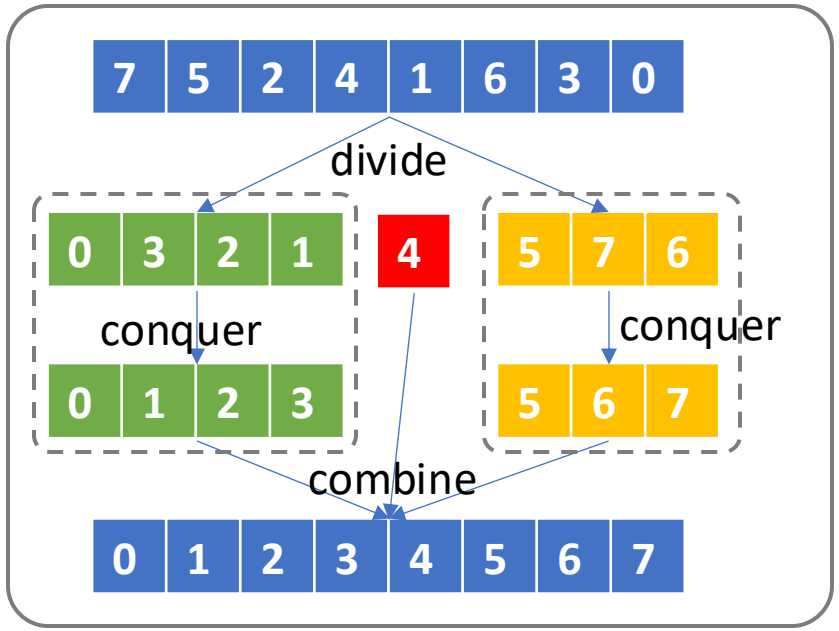


Quick Sort

- **Divide** the array according to value (not position) - around a pivot - elements ≤ pivot and elements ≥ pivot
- No need to consider the cross-array order. If both parts in order, the whole array in order
- **Conquer** (sort) both subarrays



Merge Sort



Quick Sort

Quick Sort

- **Divide** the array according to value (not position) - around a pivot - elements ≤ pivot and elements ≥ pivot
- No need to consider the cross-array order. If both parts in order, the whole array in order
- **Conquer** (sort) both subarrays

Auxiliary space complexity: inplace sort – should be $\Theta(1)$

Time complexity: $\mathrm{T}(n) = \mathrm{T}(a) + \mathrm{T}(n-a) + f(n)$

Merge sort: $a = n/2, f(n) = \Theta(n), \mathrm{T}(n) = \Theta(n\log n)$

- Can we also do linear partition?
- Can we make balanced partition?

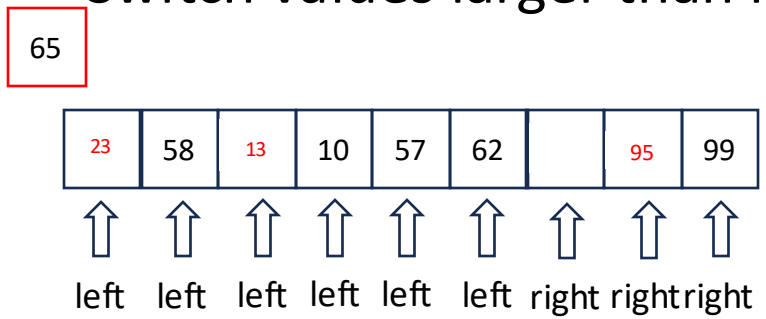
```
QUICKSORT(A, l, r)
1.    if l < r
2.         q = PARTITION(A, l, r)
3.         QUICKSORT(A, l, q-1)
4.         QUICKSORT(A, q+1, r)
```

Partition

$\Theta(n)$ time partition with $\Theta(1)$ auxiliary space

Implementation 1:

- Choose a pivot (usually the first element)
- Switch values smaller than it to its left
- Switch values larger than it to its right



Time complexity: move two pointers until they meet - $\Theta(n)$

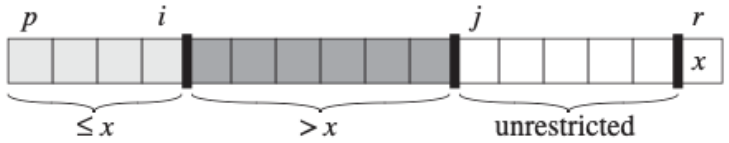
Space complexity: $\Theta(1)$ – inplace

```
PARTITION(A, l, r)
1.x = A[l]
2.i = l, j = r
3.dir = 1
4.while i != j:
5.    if dir == 1:
6.        if A[j] ≤ x:
7.            A[i] = A[j]
8.            dir = 0
9.            i = i + 1
10.       else:
11.           j = j - 1
12.   if dir == 0:
13.       if A[i] >= x:
14.           A[j] = A[i]
15.           dir = 1
16.           j = j - 1
17.       else:
18.           i = i + 1
19.A[i] = x
20.return i
```

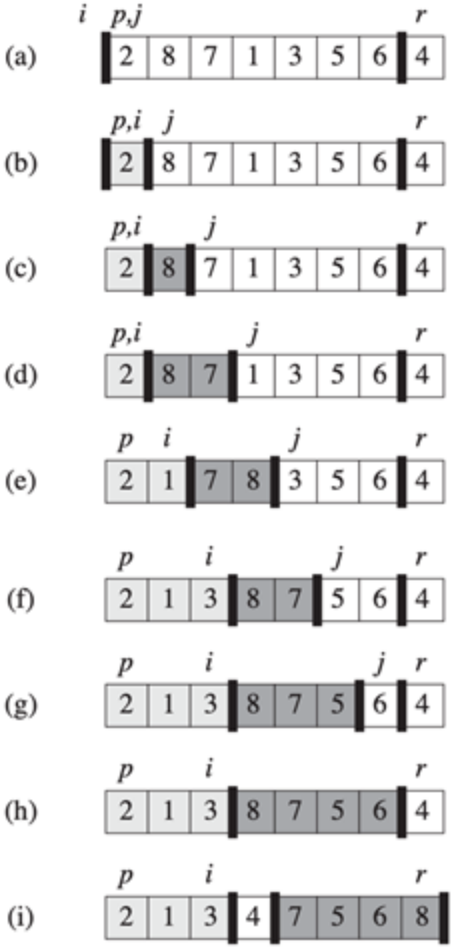
Partition

Another implementation

- Choose a pivot x (usually the first/last element, suppose the last)
- Update two (in place) arrays ($A_{\leq x}$) and ($A_{>x}$) when adding elements e in the array into them from left to right (like insertion sort)
  - $e \leq x$: add to $A_{\leq x}$
    - Switch with the first element of $A_{>x}$, update the cut point i and pointer j
  - $e > x$: add to $A_{>x}$
    - Appended to the next position of $A_{>x}$ (no action, just update the pointer j)



Partition




```
PARTITION(A, l, r)
1.x = A[r]
2.i = l - 1
3.for j = l to r – 1
4.     if A[j] ≤ x
5.          i = i + 1
6.          exchange A[i] with A[j]
7.exchange A[i+1] with A[r]
8.return i+1
```

Time complexity: move two pointers from left to right – $\Theta(n)$

Auxiliary Space complexity: $\Theta(1)$

Time Complexity Analysis

$$\mathrm{T}(n) = \mathrm{T}(a) + \mathrm{T}(n-a) + \Theta(n)$$ Depends on the partition

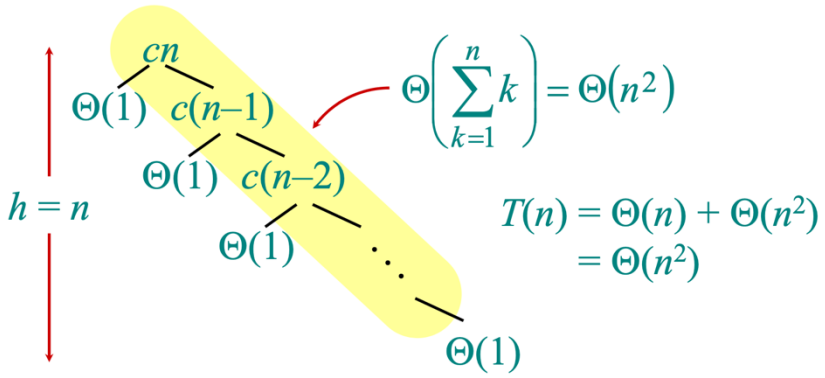
- Best case: PARTITION always splits the array evenly (a always equal to n/2)

$$\mathrm{T}(n) = 2\mathrm{T}(n/2) + \Theta(n) = \Theta(n \log n)$$ Same as merge sort

- Worst case: sorted or reversely sorted, the pivot is always the minimum or maximum (a always equal to 1)

$$\mathrm{T}(n) = \mathrm{T}(1) + \mathrm{T}(n-1) + \Theta(n) = \mathrm{T}(n-1) + \Theta(n) = \Theta(n^2)$$

Same as insertion sort



Complexity Analysis

The time cost is not determined: analyze average-case time complexity

Lemma The running time of QUICKSORT is $\Theta(X)$, where X is the number of comparisons over the entire execution

X is at least n

z_i: the i-th smallest element; X_{ij}: #comparison between element z_i and z_j

- Expectation of total number of comparisons $\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$

Key observation: elements are only compared to the pivot element

- Each pair is compared at most once:

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p(z_i \; compared \; with \; z_j)$$

```
PARTITION(A, l, r)
1.x = A[r]
2.i = l - 1
3.for j = l to r - 1
4.    if A[j] ≤ x
5.        i = i + 1
6.        exchange A[i] with A[j]
7.exchange A[i+1] with A[r]
8.return i+1
```

Complexity Analysis

Consider $p(z_i\ compared\ with\ z_j)$:

- Once a pivot $z_i < x < z_j$ is selected, z_i and z_j are separated in different branches, not compared anymore

- $\Rightarrow z_i$ and z_j are compared iff the first pivot picked between z_i and z_j is either z_i or z_j
- $\Rightarrow p(z_i\ compared\ with\ z_j) = p(z_i/z_j\ is\ picked\ the\ first\ between\ z_i\ and\ z_j)$
- $\Rightarrow$ When the input permutates randomly, elements have equal chance to be pivot
- $\Rightarrow p(z_i\ compared\ with\ z_j) = p(z_i/z_j\ is\ the\ first\ picked\ between\ z_i and\ z_j) = \frac{2}{j-i+1}$

Therefore, $\mathrm{E}[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n}\frac{2}{k}$

$= \sum_{i=1}^{n-1}\Theta(\log n) = \Theta\,(n\log n)$

Randomized Version of Quicksort

Important restriction: elements have equal chance to be pivot

Pick the last element: distribution of the pivot depends on the input permutation

The input permutation does not necessarily follow uniform distribution

Randomly choose the pivot: make the pivot follow uniform distribution

```
QUICKSORT(A, l, r)
1.    if l < r
2.        i = RANDOM(l, r)
3.        exchange A[i] and A[r]
4.        q = PARTITION(A, l, r)
5.        QUICKSORT(A, l, q-1)
6.        QUICKSORT(A, q+1, r)
```

$$p(pick\ z_i) = \sum_{k=1}^{N} \frac{1}{N}\ p(z_i\ placed\ in\ k) = \frac{1}{N}$$

Quick sort is a randomized algorithm!

Find the k-th smallest number

Input: A unordered sequence of n numbers $a_1, a_2, \ldots, a_n$

Output: the k-th smallest number in $a_1, a_2, \ldots, a_n$

- Sort and get $a[k]$: $\Theta(n\log n)$
- Any faster solution?

Find the k-th smallest number

Input: A sequence of n numbers $a_1, a_2, a_3, \ldots, a_n$

Output: the k-th smallest number in $a_1, a_2, a_3, \ldots, a_n$

Key idea: partition function + binary search

- After the partition, we obtain two sequences $S^1 \leq$ pivot and $S^2 >$ pivot
- k-th largest locates either in S^1 or S^2
  - $k \leq m_1$: find the k-th smallest in S^1
  - $k > m_1$: find the $(k - m_1)$-th smallest in S^2


```
Find(A, l, r, k)
1.    if l == r: return l
2.    if l < r
3.        i = RANDOM(l, r)
4.        exchange A[i] and A[r]
5.        q = PARTITION(A, l, r)
6.        if k == q: return q
7.        if k < q:
8.            return Find(A, l, q-1, k)
9.        else:
10.           return Find(A, q+1, r, k-q)
```

Find the k-th smallest number

Worst case: $\mathrm{T}(n) = \mathrm{T}(a) + \Theta(n)$, when $a = n - 1$, $\mathrm{T}(n) = \Theta(n^2)$

Average case (supposing no duplicated elements): The pivot is random between z_1 and z_n

- Let $X_i = \mathrm{I}\{A_{<q}\ contains\ i\ elements\}$, $E[X_i] = p(X_i) = 1/n$ (pick z_{i+1} as the z_q)
- To get upper-bound time, assume recurse in the larger array between $A_{<q}$ and $A_{>q}$, $\mathrm{T}(n) \leq \sum_{i=1}^{n} X_i \mathrm{T}(\max(i-1, n-i)) + \Theta(n)$
- $\mathrm{E}[\mathrm{T}(n)] \leq \sum_{i=1}^{n} E[X_i \mathrm{T}(\max(i-1, n-i))] + \Theta(n)$

$$= \sum_{i=1}^{n} E[X_i] E[\mathrm{T}(\max(i-1, n-i))] + \Theta(n)$$

$$= \frac{1}{n} \sum_{i=1}^{n} E[\mathrm{T}(\max(i-1, n-i))] + \Theta(n)$$

$$= \frac{1}{n} \sum_{i=1}^{\lceil n/2 \rceil} E[\mathrm{T}(n-i)] + \sum_{i=\lceil n/2 \rceil + 1}^{n} E[\mathrm{T}(i-1)] + \Theta(n) \leq \frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} E[\mathrm{T}(i)] + \Theta(n)$$

X_i and $T(\max(i-1, n-i))$ are independent:
When i is fixed, $\max(i-1, n-i)$ is fixed
$T(\max(i-1, n-i))$ depends on A and k
X_i depends on the pivot q
q is dependent of A, k

Find the k-th smallest number

$$\mathrm{E}[\mathrm{T}(n)] \leq \frac{2}{n}\sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} E[\mathrm{T}(i)] + O(n) \leq \frac{2}{n}\sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} E[\mathrm{T}(i)] + an$$

Prove $\mathrm{E}[\mathrm{T}(n)] = O(n)$ with substitution:

Assume $\mathrm{E}[\mathrm{T}(n)] \leq cn$ and $\mathrm{E}[\mathrm{T}(n)] = \Theta(1)$ for some small n (we can induce $n < \frac{2c}{c-4a}$)

- $\mathrm{E}[\mathrm{T}(n)] \leq \frac{2}{n}\sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} ci + an = \frac{2c}{n}(\sum_{i=1}^{n-1} i - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor - 1} i) + an = \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{\left(\lfloor \frac{n}{2} \rfloor - 1\right)\lfloor \frac{n}{2} \rfloor}{2}\right) + an \leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{\left(\frac{n}{2}-2\right)\left(\frac{n}{2}-1\right)}{2}\right) + an = c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \leq \frac{3cn}{4} + \frac{c}{2} + an = cn - (\frac{cn}{4} - \frac{c}{2} - an)$
- When $n \geq \frac{2c}{c-4a}$, $\mathrm{E}[\mathrm{T}(n)] \leq cn$ (supposing $\mathrm{T}(n) = \Theta(1)$ when $n < \frac{2c}{c-4a}$)
- Therefore, $\mathrm{E}[\mathrm{T}(n)] = \mathrm{O}(n)$

Summary

- Divide-and-Conquer
- Merge Sort
- Matrix Multiplication
- Maximum Subarray Problem
- Binary Search
- Fast Powering
- Quick Sort

Thank you!

AIAA 5037 Advanced Algorithms and Data Structures