Divide & Conquer (II)

Jordi Cortadella and Jordi Petit
Department of Computer Science

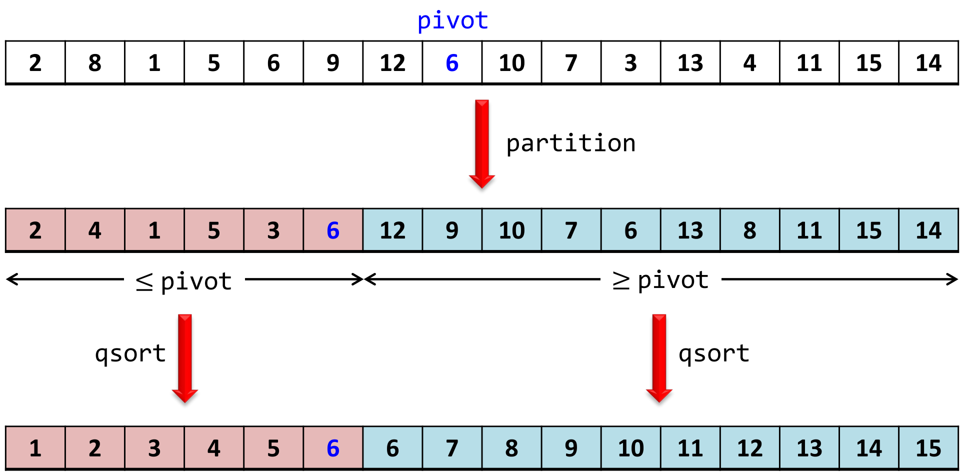
Examples

- Quick sort
- The selection problem
- The closest-points problem

Quick sort (Tony Hoare, 1959)

- Suppose that we know a number x such that one-half of the elements of a vector are greater than or equal to x and one-half of the elements are smaller than x.
  - Partition the vector into two equal parts ($n-1$ comparisons)
  - Sort each part recursively
- Problem: we do not know x.
- The algorithm also works no matter which x we pick for the partition. We call this number the **pivot**.
- **Observation:** the partition may be unbalanced.

Quick sort: example



The key step of quick sort is the partitioning algorithm.

Question: how to find a good pivot?

Quick sort partition: example

i   j

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

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

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

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

← ≤ 6 → ← ≥ 6 →

Quick sort: Hoare's partition

```
def partition(a: list[T], left: int, right: int) -> int:
    """a[left..right]: segment to be sorted.
       Output: The left part has elements ≤ than the pivot.
       The right part has elements ≥ than the pivot.
       Returns the index of the last element of the left part
    """
    pivot = a[left]
    i, j = left-1, right+1
    while True:
        while True:  # find a[i] ≥ pivot
            i += 1
            if a[i] >= pivot: break
        while True:  # find a[j] ≤ pivot
            j -= 1
            if a[j] <= pivot: break
        if i >= j:
            return j
        a[i], a[j] = a[j], a[i]  # swap a[i], a[j]
```

The first swap locates the sentinels of the two innermost loops.
No need to check for indices out-of-bounds.

Quick sort: algorithm

```
def quick_sort(a: list[T],
               left: int = 0, right: int = -1) -> None:
    """sorts a[left..right].
       If right < 0, it sorts the whole list.
       The initial call can be invoked as quick_sort(a)
    """

    if right < 0: # initial call (sort the whole list)
        right = len(a)-1

    if left < right:
        mid = partition(a, left, right)
        quick_sort(a, left, mid)
        quick_sort(a, mid+1, right)
```


Quick sort: hybrid approach

```
def quick_sort(a: list[T],
               left: int = 0, right: int = -1) -> None:
    """sorts a[left..right] partially, leaving unsorted
       chunks of size K (some break-even constant when
       compared to insertion sort).
       If right < 0, it sorts the whole list.
    """
    if right < 0: # initial call (sort the whole list)
        right = len(a)-1

    if left <= right - K: # K: size of the unsorted chunks
        mid = partition(a, left, right)
        quick_sort(a, left, mid)
        quick_sort(a, mid+1, right)

def sort(a: list[T]) -> None:
    """Sorts a"""
    quick_sort(a)
    insertion_sort(a)
```

Observation: during **insertion_sort**, elements will never be shifted by more than **K** locations

Quick sort: complexity analysis

- The partition algorithm is $O(n)$.
- Assume that the partition is balanced:

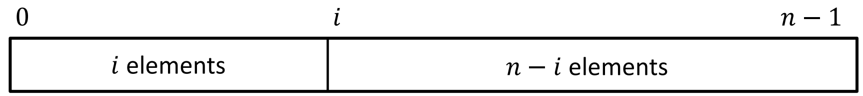
$$T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$$

- Worst case runtime: the pivot is always the smallest element in the vector → $O(n^2)$
- Selecting a good pivot is essential. There are different strategies, e.g.,
  - Take the median of the first, last and middle elements
  - Take the pivot at random

Quick sort: complexity analysis

- Let us assume that x_i is the i-th smallest element in the vector.
- Let us assume that each element has the same probability of being selected as pivot.
- The runtime if x_i is selected as pivot is:

$$T(n) = \underset{\text{partition}}{\underset{\uparrow}{n}} + \underset{\text{qsort}}{\underset{\uparrow}{T(i)}} + \underset{\text{qsort}}{\underset{\uparrow}{T(n-i)}}$$



Possible values for i: $1 \dots n-1$. Assumption: all values are equiprobable (random pivots)

Quick sort: complexity analysis

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

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

$$T(n) = n + \frac{2}{n-1}\sum_{i=1}^{n-1}T(i) = 2n \cdot H(n)$$

$H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$ is the Harmonic series, that has a simple approximation: $H(n) = \ln n + \gamma + O(1/n)$.
$\gamma = 0.577 \dots$ is Euler's constant. **[see the appendix]**

$$T(n) = 2n(\ln n + \gamma + O(1/n)) \in O(n \log n)$$

Quick sort: complexity analysis summary

- Runtime of quicksort:

$$T(n) = O(n^2)$$
$$T(n) = \Omega(n \log n)$$
$$T_{\text{avg}}(n) = O(n \log n)$$

- Be careful: some malicious patterns may increase the probability of the worst case runtime, e.g., when the vector is sorted or almost sorted.
- Possible solution: use random pivots.

The selection problem

- Given a collection of N elements, find the k-th smallest element.
- Options:
  - Sort a vector and select the k-th location: $O(N \log N)$
  - Read k elements into a vector and sort them. The remaining elements are processed one by one and placed in the correct location (similar to insertion sort). Only k elements are maintained in the vector. Complexity: $O(kN)$. Why?

Quick sort

```
def quick_sort(a: list[T],
               left: int = 0, right: int = -1) -> None:
    """sorts a[left..right].
       If right < 0, it sorts the whole list.
       The initial call can be invoked as quick_sort(a)
    """

    if right < 0: # initial call (sort the whole list)
        right = len(a)-1

    if left < right:
        mid = partition(a, left, right)
        quick_sort(a, left, mid)
        quick_sort(a, mid+1, right)
```


Quick select

```
def quick_select(a: list[T], k: int,
                 left: int = 0, right: int = -1) -> T:
    """Returns the element at location k assuming
       a[left..right] would be sorted.
       Pre: left ≤ k ≤ right.
       Post: the elements of a have changed their locations.
       The initial call can be invoked as quick_select(a, k)
    """

    if right < 0: # initial call (use the whole list)
        right = len(a)-1

    if left == right:
        return a[left]

    mid = partition(a, left, right)
    if k <= mid:
        return quick_select(a, k, left, mid)
    return quick_select(a, k, mid+1, right)
```


Quick Select: complexity

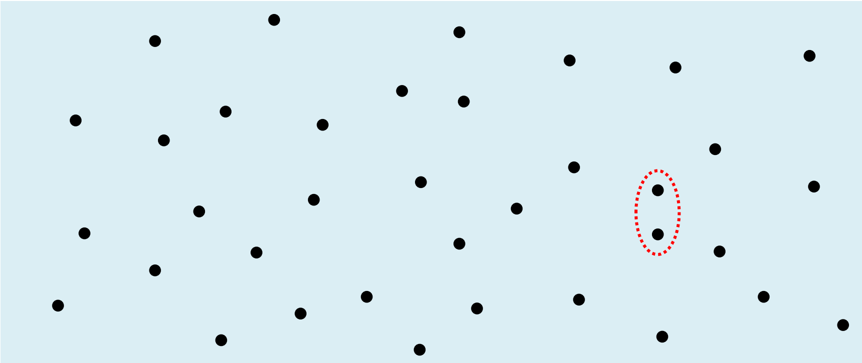
- Master theorem:

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

- Assume that the partition is balanced:
  - Quick sort: $T(n) = 2T(n/2) + O(n) = O(n \log n)$
  - Quick select: $T(n) = T(n/2) + O(n) = O(n)$
- The average linear time complexity can be achieved by choosing good pivots (similar strategy and complexity computation to quick_sort).

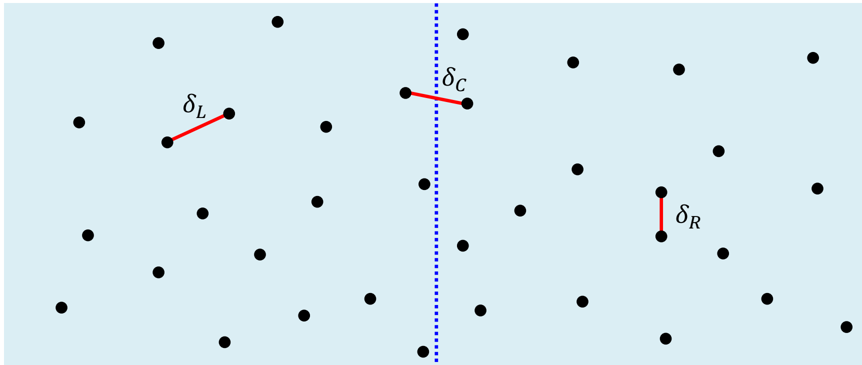
The Closest-Points problem

- **Input:** A list of n points in the plane
  $$\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$$
- **Output:** The pair of closest points
- **Simple approach:** check all pairs → $O(n^2)$
- We want an $O(n \log n)$ solution !



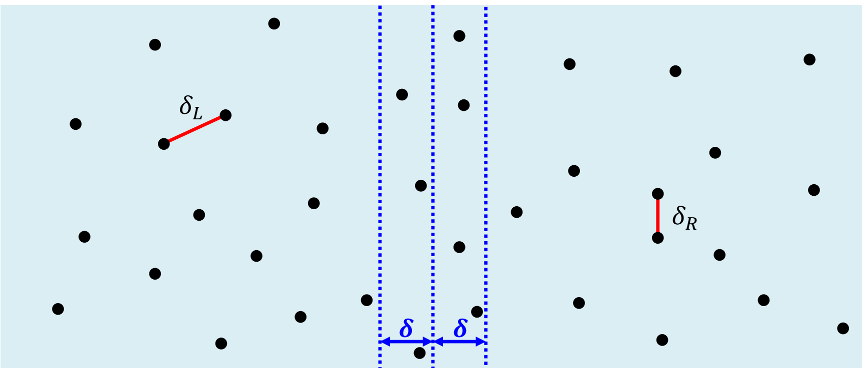
The Closest-Points problem

- We can assume that the points are sorted by the x-coordinate. Sorting the points is free from the complexity standpoint ($O(n \log n)$).
- Split the list into two halves. The closest points can be both at the left, both at the right or one at the left and the other at the right (center).
- The left and right pairs are easy to find (recursively). How about the pairs in the center?



The Closest-Points problem

- Let $\delta = min(\delta_L, \delta_R)$. We only need to compute δ_C if it improves δ.
- We can define a strip around de center with distance δ at the left and right. If δ_C improves δ, then the points must be within the strip.
- In the worst case, all points can still reside in the strip.
- But how many points do we really have to consider?



The Closest-Points problem

Let us take all points in the strip and sort them by the y-coordinate. We only need to consider pairs of points with distance smaller than δ.

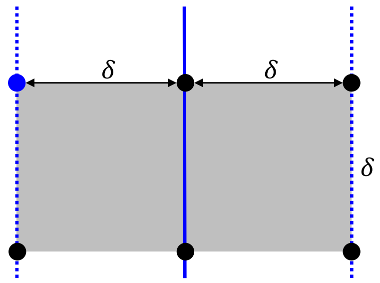
Once we find a pair (p_i, p_j) with y-coordinates that differ by more than δ, we can move to the next p_i.

```
for i in range(NumPointsInStrip):
  for j in range(i+1, NumPointsInStrip):

    if (pᵢ and pⱼ's y-coordinate differ by
        more than δ): break // Go to next pᵢ

    if dist(pᵢ,pⱼ) < δ:  δ = dist(pᵢ,pⱼ);
```

But, how many pairs (p_i, p_j) do we need to consider?

The Closest-Points problem

- For every point p_i at one side of the strip, we only need to consider points from p_{i+1}.



- The relevant points only reside in the $2\delta \times \delta$ rectangle below point p_i. There can only be 8 points at most in this rectangle (4 at the left and 4 at the right). Some points may have the same coordinates.

The Closest-Points problem: algorithm

- Sort the points according to their x-coordinates.
- Divide the set into two equal-sized parts.
- Compute the min distance at each part (recursively). Let δ be the minimal of the two minimal distances.
- Eliminate points that are farther than δ from the separation line.
- Sort the remaining points according to their y-coordinates.
- Scan the remaining points in the y order and compute the distances of each point to its 7 neighbors.

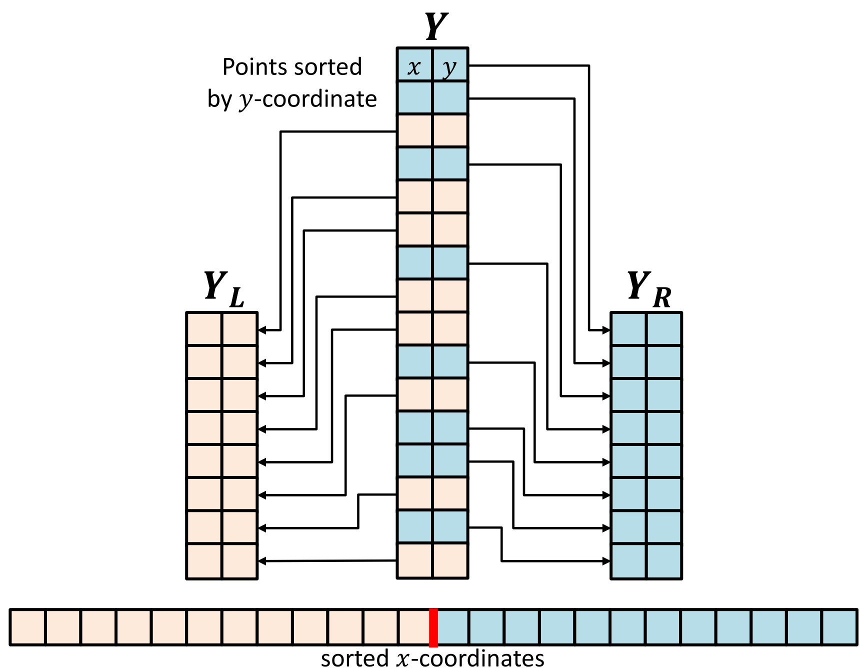
The Closest-Points problem: complexity

- Initial sort using x-coordinates: $O(n \log n)$. It comes for free.
- Divide and conquer:
  - Solve for each part recursively: $2T(n/2)$
  - Eliminate points farther than δ: $O(n)$
  - Sort remaining points using y-coordinates: $O(n \log n)$
  - Scan the remaining points in y order: $O(n)$

$$T(n) = 2T(n/2) + O(n) + O(n \log n) = O(n \log^2 n)$$

- Can we do it in $O(n \log n)$? Yes, we need to sort by y in a smart way.

Partitioning and sorting the points (visually)



The Closest-Points problem: complexity

- Let Y a vector with the points sorted by the y-coordinates. This can be done initially for free.

- Each time we partition the set of points by the x-coordinate, we also partition Y into two sorted vectors (using an "*unmerging*" procedure with linear complexity)

```
Y_L = Y_R = ∅   // Initial lists of points
for each p_i ∈ Y in ascending order of y:
    if p_i is at the left part: Y_L.push_back(p_i)
    else: Y_R.push_back(p_i)
```

- Now, sorting the points by the y-coordinate at each iteration can be done in linear time, and the problem can be solved in $O(n \log n)$

Subtract and Conquer

- Sometimes we may find recurrences with the following structure:

$$T(n) = a \cdot T(n-b) + O(n^c)$$

- Examples:

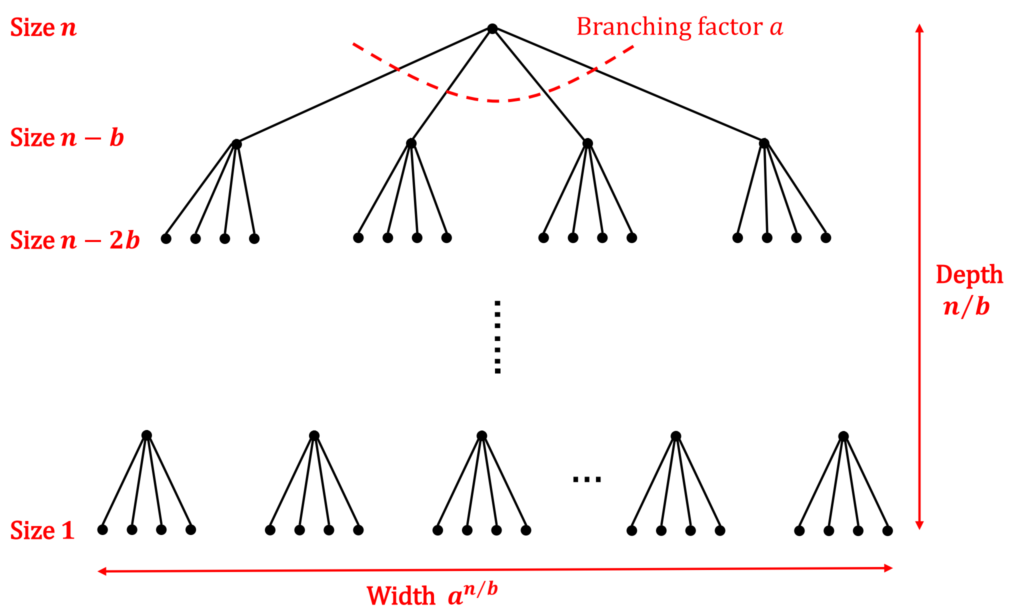
$$\text{Hanoi}(n) = 2 \cdot \text{Hanoi}(n-1) + O(1)$$

$$\text{Sort}(n) = \text{Sort}(n-1) + O(n)$$

- *Muster* theorem:

$$T(n) = \begin{cases} O(n^c) & \text{if } a < 1 \quad \text{(never occurs)} \\ O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

Muster theorem: recursion tree



Muster theorem: examples

- **Hanoi**: $T(n) = 2T(n-1) + O(1)$

  We have $a = 2$ and $c = 0$, thus $T(n) = O(2^n)$.

- **Selection sort** (recursive version):
  - Select the min element and move it to the first location
  - Sort the remaining elements

  $$T(n) = T(n-1) + O(n) \qquad (a = c = 1)$$

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

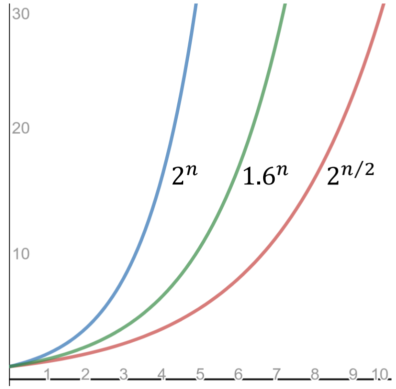
Muster theorem: examples

Fibonacci: $T(n) = T(n-1) + T(n-2) + O(1)$

We can compute bounds:

$$2T(n-2) + O(1) \le T(n) \le 2T(n-1) + O(1)$$

Thus, $O\left(2^{n/2}\right) \le T(n) \le O(2^n)$

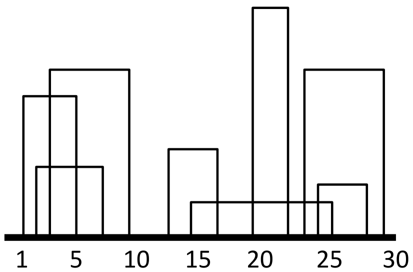


EXERCICES

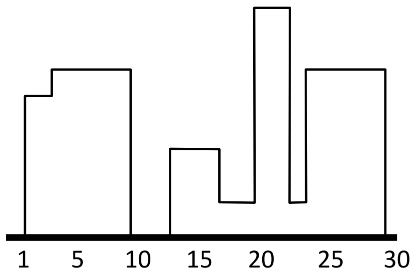
The skyline problem

Given the exact locations and shapes of several rectangular buildings in a city, draw the skyline (in two dimensions) of these buildings, eliminating hidden lines (source: Udi Manber, *Introduction to Algorithms*, Addison-Wesley, 1989).

Input:
(1,**11**,5) (2,**6**,7) (3,**13**,9) (12,**7**,16) (14,**3**,25) (19,**18**,22) (23,**13**,29) (24,**4**,28)



Output:
(1,**11**,3,**13**,9,**0**,12,**7**,16,**3**,19,**18**,22,**3**,23,**13**,29,**0**)
(numbers in boldface represent heights)



Describe (in natural language) two different algorithms to solve the skyline problem:

- By induction: assume that you know how to solve it for $n-1$ buildings.
- Using Divide&Conquer: solve the problem for $n/2$ buildings and combine.

Analyze the cost of each solution.

A, B or C?

Suppose you are choosing between the following three algorithms:

- Algorithm **A** solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm **B** solves problems of size n by recursively solving two subproblems of size $n-1$ and them combining the solutions in constant time.
- Algorithm **C** solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which one would you choose?

Source: Dasgupta, Papadimitriou and Vazirani, *Algorithms*, McGraw-Hill, 2008.

Crazy sorting

Let $T[i..j]$ be a vector with $n = j - i + 1$ elements. Consider the following sorting algorithm:

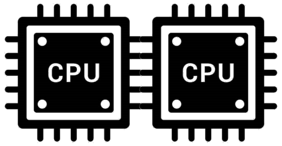
a) If $n \leq 2$ the vector is easily sorted (constant time).

b) If $n \geq 3$, divide the vector into three intervals $T[i..k-1]$, $T[k..l]$ and $T[l+1..j]$, where $k = i + \lfloor n/3 \rfloor$ and $l = j - \lfloor n/3 \rfloor$. The algorithm recursively sorts $T[i..l]$, then it sorts $T[k..j]$, and finally sorts $T[i..l]$.

- Prove the correctness of the algorithm.
- Analyze the asymptotic complexity of the algorithm (give a recurrence of the runtime and solve it).

VLSI chip testing

Professor Diogenes has n supposedly identical VLSI (Very-Large-Scale Integration) chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:



Chip A says	Chip B says	Conclusion
B is good	A is good	Both are good, or both are bad
B is good	A is bad	At least one is bad
B is bad	A is good	At least one is bad
B is bad	A is bad	At least one is bad

a. Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

b. Consider the problem of finding a single good chip among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

c. Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Source: Cormen, Leiserson and Rivest, Introduction to Algorithms, The MIT Press, 1989

Breaking into pieces

Let us assume that f is $\Theta(1)$ and g has a runtime proportional to the size of the vector it has to process, i.e., $\Theta(j-i+1)$. What is the asymptotic cost of A and B as a function of n? (n is the size of the vector).

If both functions do the same, which one would you choose?

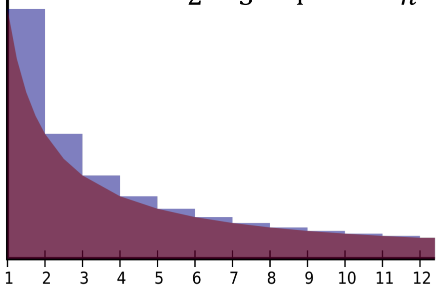
```
def a(v: list[float], i: int, j: int) -> float:
    if i < j:
        x = f(v, i, j)
        m = (i+j)//2
        return a(v, i, m-1) + a(v, m, j) + a(v, i+1, m) + x
    else:
        return v[i]


def b(v: list[float], i: int, j: int) -> float:
    if i < j:
        x = g(v, i, j)
        m1 = i + (j-i+1)//3
        m2 = i + (j-i+1)*2//3
        return b(v, i, m1-1) + b(v, m1, m2-1) + b(v, m2, j) + x
    else:
        return v[i]
```


APPENDIX

Logarithmic identities

$b^{\log_b a} = \log_b b^a = a$
$\log_b(xy) = \log_b x + \log_b y$
$\log_b \frac{x}{y} = \log_b x - \log_b y$
$\log_b x^c = c \log_b x$
$\log_b x = \frac{\log_c x}{\log_c b}$
$x^{\log_b y} = y^{\log_b x}$

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$



$$\gamma = \lim_{n\to\infty}\left(-\ln n + \sum_{k=1}^{n}\frac{1}{k}\right) \quad \Rightarrow \quad \sum_{k=1}^{n}\frac{1}{k} \in \Theta(\log n)$$

$\gamma = 0.5772 \ldots$ (Euler-Mascheroni constant) (Harmonic series)

Full-history recurrence relation

$$\boxed{T(n) = n + \frac{2}{n-1}\sum_{i=1}^{n-1} T(i)}$$

A recurrence that depends on all the previous values of the function.

$$(n-1)T(n) = n(n-1) + 2\sum_{i=1}^{n-1} T(i), \qquad nT(n+1) = (n+1)n + 2\sum_{i=1}^{n} T(i)$$

$$nT(n+1) - (n-1)T(n) = (n+1)n - n(n-1) + 2T(n) = 2n + 2T(n)$$

$$T(n+1) = \frac{n+1}{n}T(n) + 2 \qquad \Rightarrow \qquad T(n) = \frac{n}{n-1}T(n-1) + 2$$

$$T(n) = 2 + \frac{n}{n-1}\left(2 + \frac{n-1}{n-2}\left(2 + \frac{n-2}{n-3}\left(\cdots \frac{2}{1}\right)\right)\right)$$

$$T(n) = 2\left(1 + \frac{n}{n-1} + \frac{n}{n-1}\frac{n-1}{n-2} + \frac{n}{n-1}\frac{n-1}{n-2}\frac{n-2}{n-3} + \cdots + \frac{n}{n-1}\frac{n-1}{n-2}\frac{n-2}{n-3}\cdots\frac{2}{1}\right)$$

$$T(n) = 2\left(1 + \frac{n}{n-1} + \frac{n}{n-2} + \frac{n}{n-3} + \cdots + \frac{n}{1}\right) = 2n\left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + 1\right)$$

$$\boxed{T(n) = 2n \cdot H(n) \in \Theta(n \log n)}$$

Muster theorem: proof

- Expanding the recursion (assume that $f(n)$ is $\mathrm{O}(n^c)$)

$$\begin{aligned} T(n) &= aT(n-b) + f(n) \\ &= a\big(aT(n-2b) + f(n-b)\big) + f(n) \\ &= a^2T(n-2b) + af(n-b) + f(n) \\ &= a^3T(n-3b) + a^2f(n-2b) + af(n-b) + f(n) \end{aligned}$$

- Hence:

$$T(n) = \sum_{i=0}^{n/b} a^i \cdot f(n-ib)$$

- Since $f(n-ib)$ is in $\mathrm{O}\big((n-ib)^c\big)$, which is in $\mathrm{O}(n^c)$, then

$$T(n) = \mathrm{O}\left(n^c \sum_{i=0}^{n/b} a^i\right)$$

- The proof is completed by this property:

$$\sum_{i=0}^{n/b} a^i = \begin{cases} \mathrm{O}(1), & \text{if } a < 1 \\ \mathrm{O}(n), & \text{if } a = 1 \\ \mathrm{O}(a^{n/b}), & \text{if } a > 1 \end{cases}$$