January 5, 2022

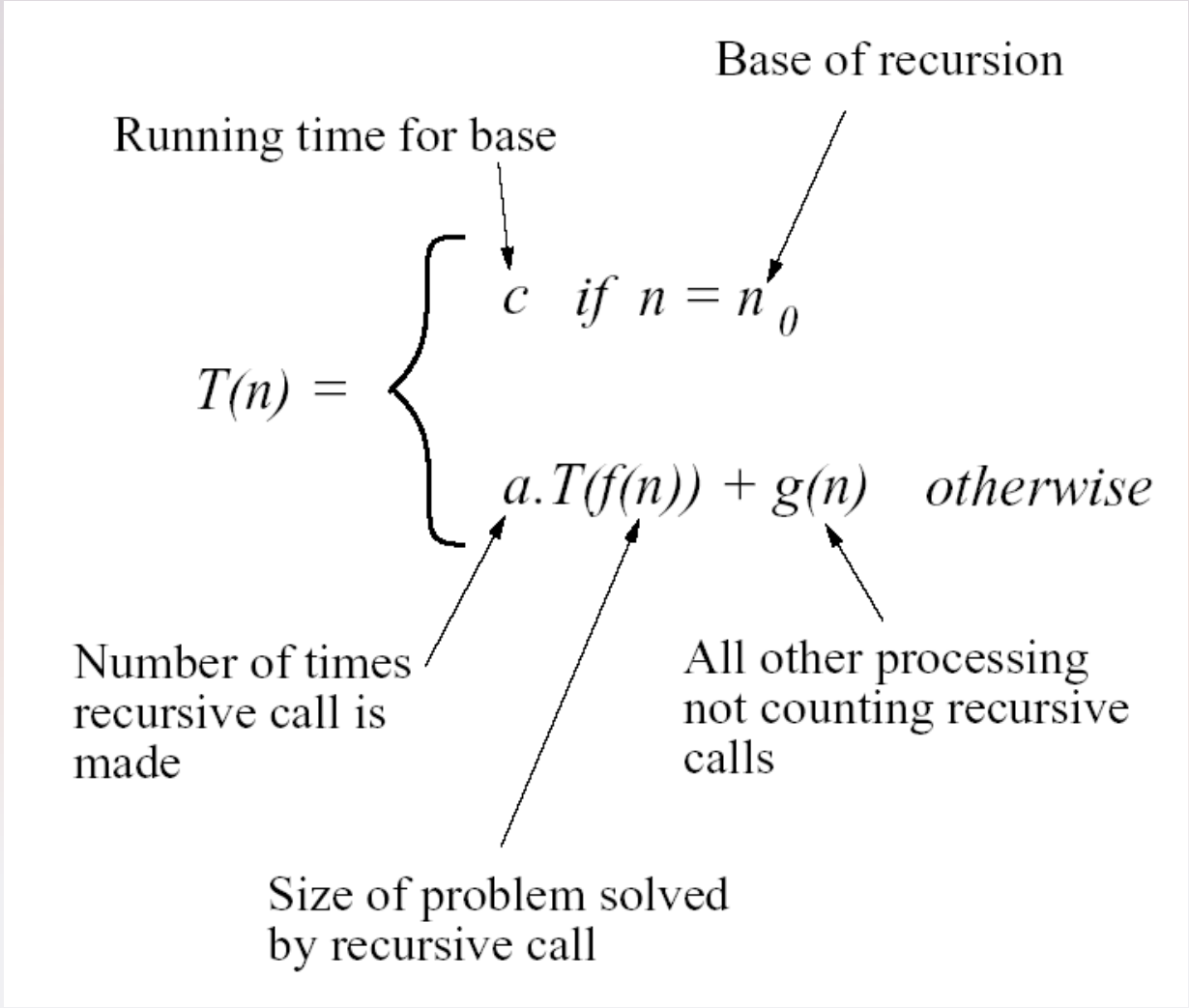
# Recurrence Relation Analysis Divide-and-Conquer

# Recurrence relation

- A *recurrence relation* is an equation that recursively defines a sequence: each term of the sequence is defined as a function of the preceding term(s)
- For instance

$$f(n) = \begin{cases} 2 & n=1 \\ f(n-1)+n & n>1 \end{cases}$$

# General form



# Definition of the Factorial function

$$F(n) = \begin{cases} 1 & n = 0 \\ nF(n-1) & n \geq 1 \end{cases}$$

## Recursive implementation

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

## Time complexity?

$$T(n) = \begin{cases} & n \leq \\ & n > \end{cases}$$

# Example

```
def bugs(n):
    if n <= 1:
        do_something()
    else:
        bugs(n-1)
        bugs(n-2)
        for i in range(n):
            do_something_else()
```

$$T(n) = \begin{cases} c_1 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + nc_2 & \text{otherwise} \end{cases}$$

# Example

```
def daffy(n):
    if n == 1 or n == 2:
        do_something()
    else:
        daffy(n-1)
        for i in range(n):
            do_something_else()
        daffy(n-1)
```

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \text{ or } n = 2 \\ 2T(n-1) + nc_2 & \text{otherwise} \end{cases}$$

# Example

```
def elmer(n):
    if n == 1:
        do_something()
    elif n == 2:
        do_something_else()
    else:
        for i in range(n):
            elmer(n-1)
            do_something_different()
```

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ c_2 & \text{if } n = 2 \\ n(T(n-1) + c_3) & \text{otherwise} \end{cases}$$

# Example

```
def yosemite(n):
    if n == 1:
        do_something()
    else:
        for i in range(1,n-1):
            yosemite(i)
        do_something_different()
```

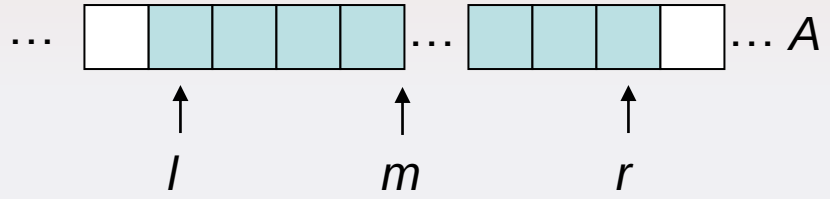
$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ \sum_{i=1}^{n-1} \left( T(i) + c_2 \right) & \text{otherwise} \end{cases}$$

# MergeSort

- MergeSort is a divide & conquer algorithm
  - *Divide*: divide an *n*-element sequence into two subsequences of approx *n*/2 elements
  - *Conquer*: sort the subsequences recursively
  - *Combine*: merge the two sorted subsequences to produce the final sorted sequence

# MergeSort

```
def mergesort(A):
    if len(A) < 2:
        return A
    else:
        m = len(A)/2
        l = mergesort(A[:m])
        r = mergesort(A[m:])
        return merge(l,r)
```



# Example

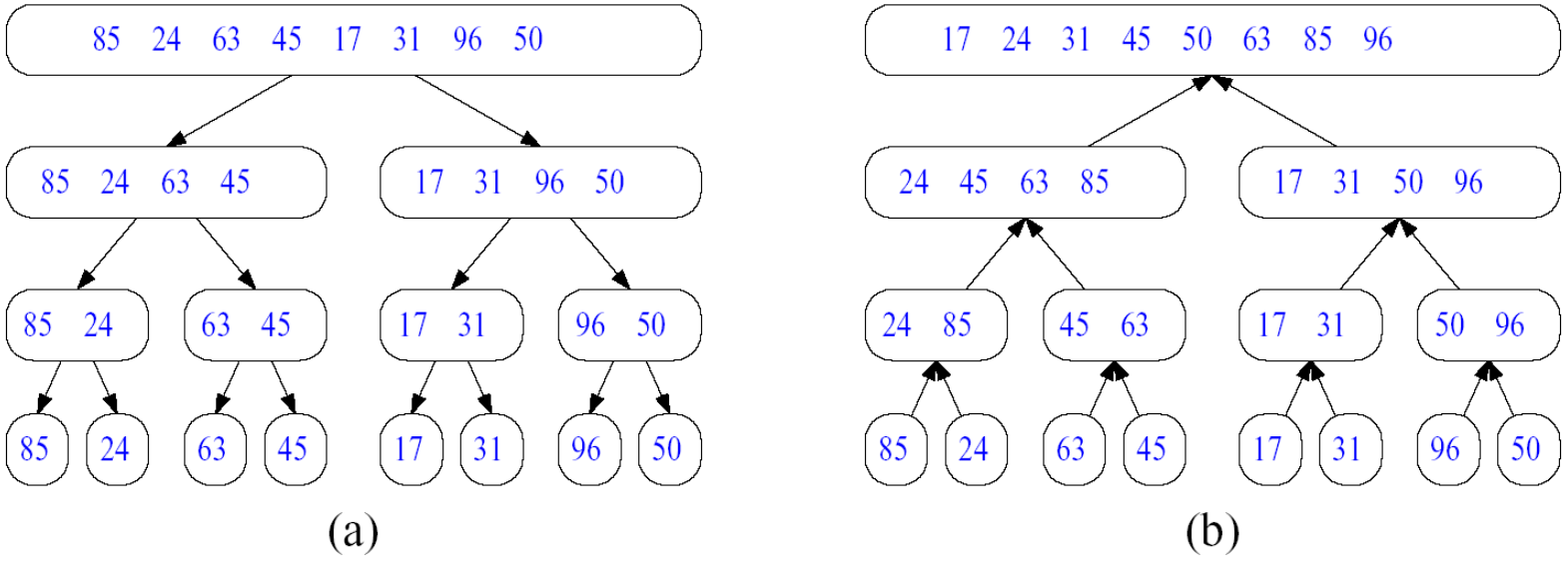



**Figure 4.2:** Merge-sort tree $T$ for an execution of the merge-sort algorithm on a sequence with 8 elements: (a) input sequences processed at each node of $T$; (b) output sequences generated at each node of $T$.

# Merge of MergeSort

```
def merge(l,r):
    result, i, j = [], 0, 0
    while i < len(l) and j < len(r):
        if l[i] <= r[j]:
            result.append(l[i])
            i += 1
        else:
            result.append(r[j])
            j += 1
    result += l[i:]
    result += r[j:]
    return result
```

# MergeSort Analysis

- Divide: Just computes the middle of the subsequence, thus takes constant time:

  $D(n) = \Theta(1)$

- Conquer: We solve 2 subproblems of size approximately n/2:

  $a = 2, \quad b = 2$

- Combine: Merge takes $\Theta(n)$:

  $C(n) = \Theta(n)$

- Noting that $\Theta(n) + \Theta(1)$ is still $\Theta(n)$, we get:

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

- Later we will see that:

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

# “Visual” Analysis

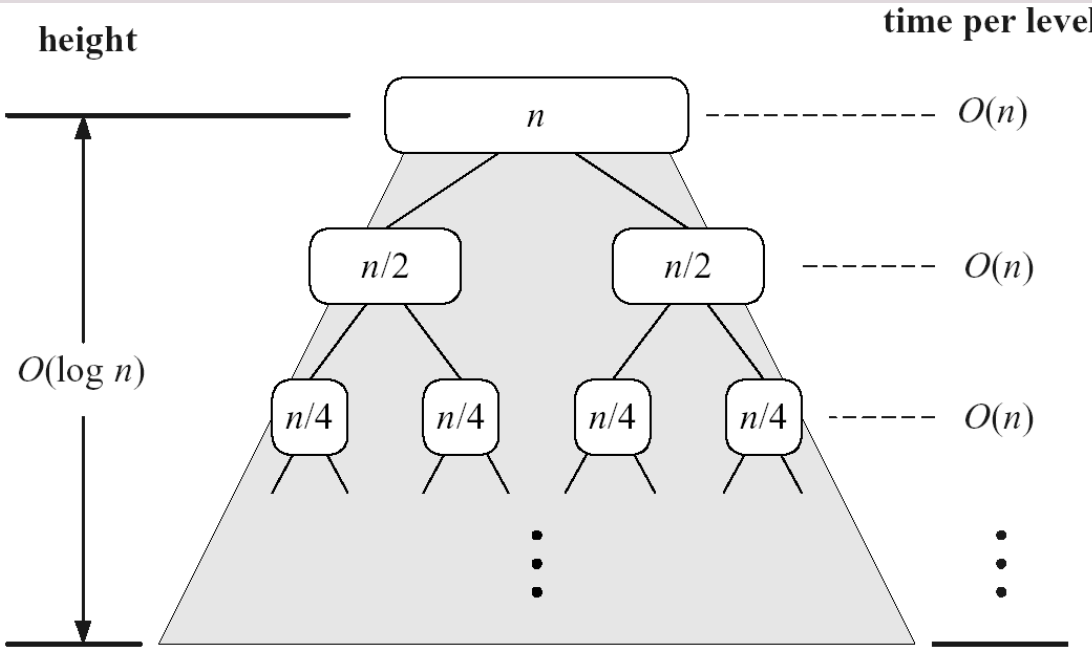



**Figure 4.4:** A visual analysis of the running time of merge-sort. Each node of the merge-sort tree is labeled with the size of its subproblem.

# Solving Recurrence Relation

# Methods

Two methods for solving recurrences

- Iterative substitution method
- Master method
- Recursive tree method

Will be on the quiz

# Iterative substitution

- Assume $n$ large enough
- Substitute $T$ on the right-hand side of the recurrence relation
- Iterate the substitution until we see a pattern which can be converted into a general closed-form formula

# MergeSort recurrence relation

$$T(N) = 2T\left(\frac{N}{2}\right) + N \qquad \text{for} \quad N \geq 2$$

$$T(1) = 1$$

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

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

$$= 4\left(2T\left(\frac{N}{8}\right) + \frac{N}{4}\right) + 2N$$

$$= 8T\left(\frac{N}{8}\right) + 3N$$

$T(1) = 1$

$$\ldots$$

$$= 2^i T\left(\frac{N}{2^i}\right) + iN$$

The expansion stops for $i = \log_2 N$, so that

$$T(N) = N + N \log_2 N$$

# Verify the correctness

- How to verify the solution is correct?
- Use proof by induction!
- Important: make sure the constant *c* works for both the base case and the induction step

# Proof by induction

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Fact: $T(n) \in O(n \log_2 n)$

Proof. Base case: $T(2) = 2T(1) + 2 = 4 \le c(2\log_2 2) = 2c.$

Hence, $c \ge 2$.

Induction hypothesis: $T(n/2) \le c\frac{n}{2}\log_2 \frac{n}{2}$

Induction: $T(n) = 2T(n/2) + n$

$$\le 2c\frac{n}{2}\log_2 \frac{n}{2} + n$$

$$= cn\log_2 \frac{n}{2} + n = cn\log_2 n - cn\log_2 2 + n$$

$$= cn\log_2 n + n(1 - c) \le cn\log_2 n \text{ when } c \ge 1$$

Choose $c = 2$.

The constant $c$ used in the induction and the base case has to be the *same* !

# Wrong proof by induction

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Fact (wrong): $T(n) \in O(n)$

Proof. Base case: $T(1) = 1 \le c1$, hence $c \ge 1$

Induction hypothesis: $T(n/2) \le c(n/2)$

Induction: $T(n) = 2T(n/2) + n$

$\le 2c(n/2) + n$

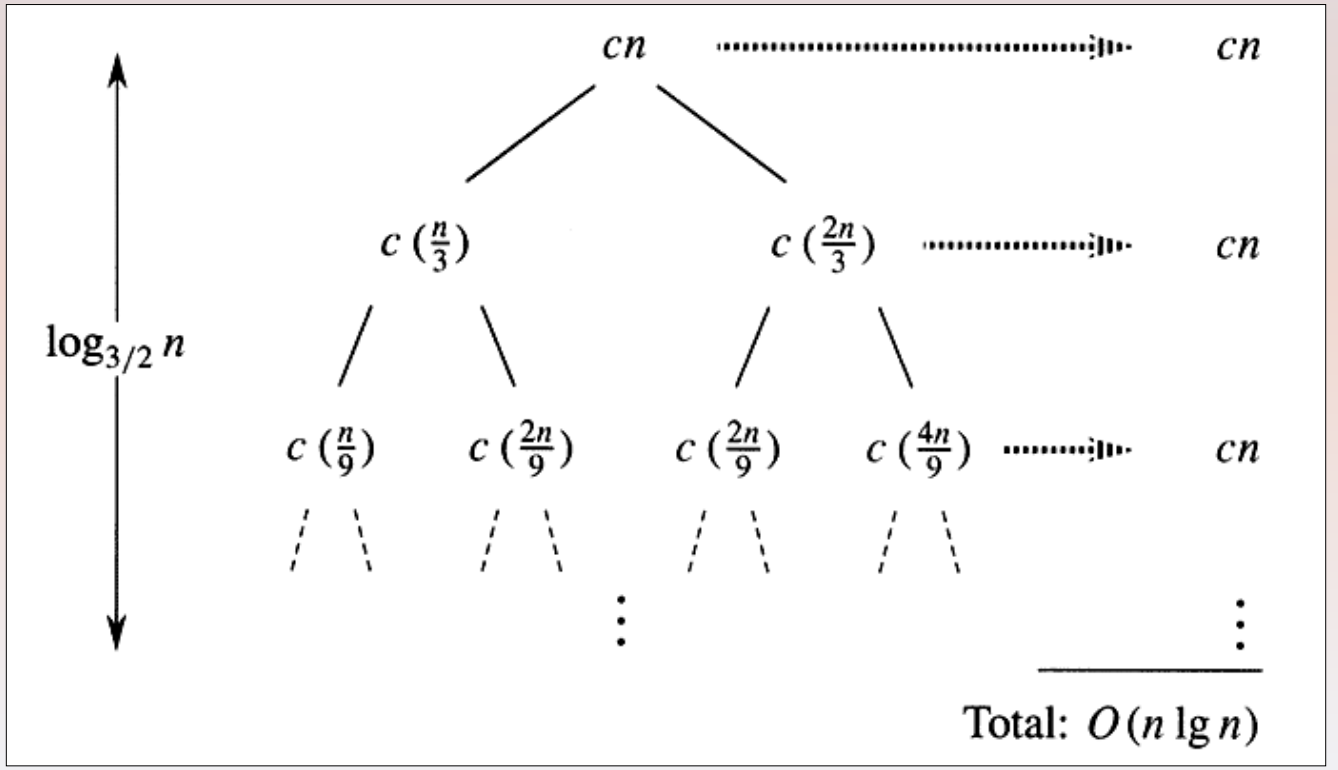
$= cn + n \in O(n)$

proof is WRONG, but where is the mistake?

# Recursion-Tree Method

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

estimate:



# Recursion-Tree Method

Recurrence:

$$T(n) = T(n/3) + T(2n/3) + cn$$

Guess:

$$T(n) = O(n \lg n)$$

Prove by induction:

$$T(n) \leq dn \lg n$$

for suitable $d > 0$

# Inductive Proof

Base case:

Inductive hypothesis:
For values of $n < k$ the inequality holds: $T(n) \leq dn \lg n$

Need to show that it also holds for $n = k$.

In particular, for $n = k/3$, and $n = 2k/3$, the inductive hypothesis should hold…

# Inductive Proof

That is

$$T(k/3) \leq d\ k/3 \lg k/3$$
$$T(2k/3) \leq d\ 2k/3 \lg 2k/3$$

The recurrence gives us:

$$T(k) = T(k/3) + T(2k/3) + ck$$

Substituting the inequalities above yields:

$$T(k) \leq [d\ (k/3) \lg (k/3)] + [d\ (2k/3) \lg (2k/3)] + ck$$

# Inductive Proof

$$T(k) \leq [d\,(k/3)\lg k - d\,(k/3)\lg 3] + [d\,(2k/3)\lg k - d\,(2k/3)\lg(3/2)] + ck$$

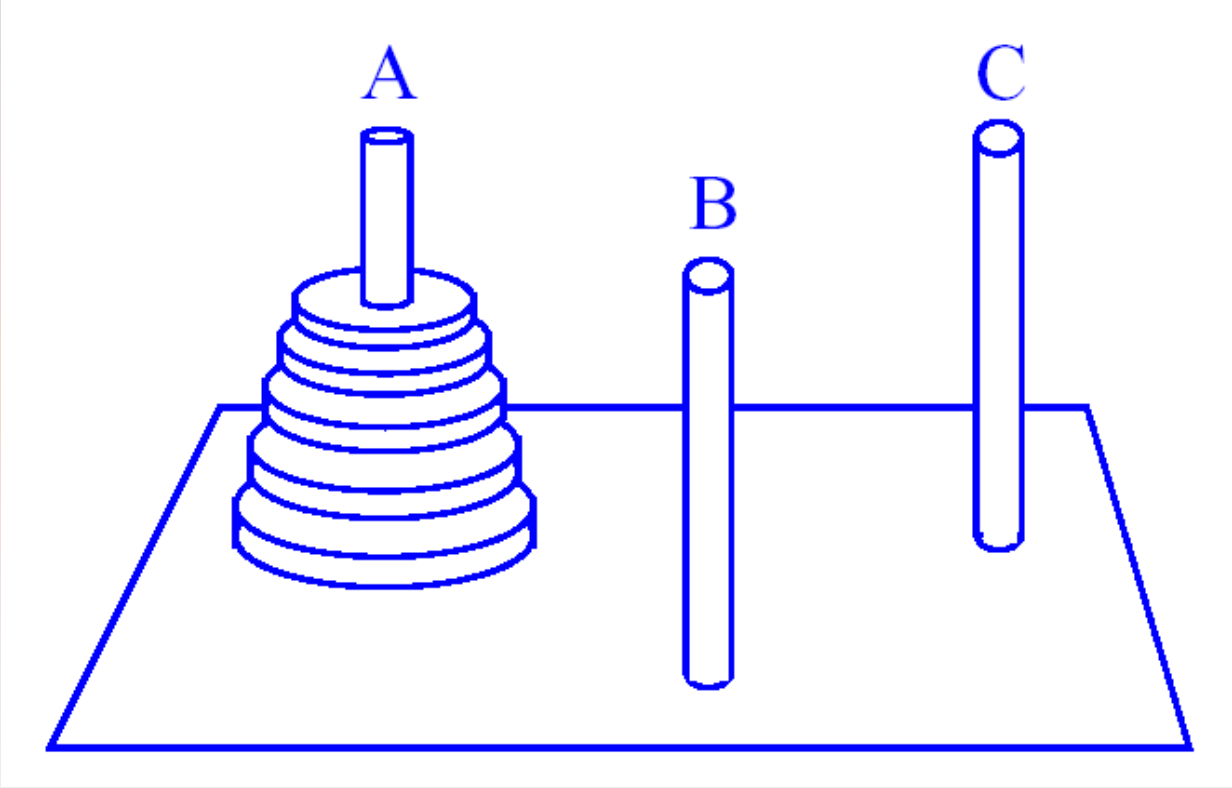
$$T(k) \leq dk\lg k - d[(k/3)\lg 3 + (2k/3)\lg(3/2)] + ck$$

$$T(k) \leq dk\lg k - dk[\lg 3 - 2/3] + ck$$

When $d \geq c / (\lg 3 - (2/3))$, we get

$$T(k) \leq dk\lg k$$

# Towers of Hanoi



# Towers of Hanoi

**Goal:** transfer all $N$ disks from peg A to peg C

**Rules:**

- move one disk at a time
- never place larger disk above smaller one

**Recursive solution:**

- transfer $N - 1$ disks from A to B
- move largest disk from A to C
- transfer $N - 1$ disks from B to C

**Total number of moves:**

- $T(N) = 2\ T(N - 1) + 1$

# Towers of Hanoi

```
def hanoi(n, a='A', b='B', c='C'):
    if n == 0:
        return
    hanoi(n-1, a, c, b)
    print a, '->', c
    hanoi(n-1, b, a, c)
```

# Towers of Hanoi: Recurrence Relation

Solve

$$T(N) = \begin{cases} 2T(N-1)+1 & N > 1 \\ 1 & N = 1 \end{cases}$$

# Towers of Hanoi: Unfolding the relation

$$
\begin{aligned}
T(N) &= 2\,(\,2\,T(N-2)+1\,)+1 = \\
&= 4\,T(N-2)+2+1= \\
&= 4\,(\,2\,T(N-3)+1\,)+2+1 = \\
&= 8\,T(N-3)+4+2+1 = \\
&\ldots \\
&= 2^{i}\,T(N-i)+2^{i-1}+2^{i-2}+\ldots+2^{1}+2^{0}
\end{aligned}
$$

the expansion stops when $i = N - 1$

$$T(N) = 2^{N-1}+2^{N-2}+2^{N-3}+\ldots+2^{1}+2^{0}$$

This is a ***geometric sum***, so that we have:

$$T(N) = 2^{N}-1 \in \Theta\left(2^{N}\right)$$

# Problem

Problem: Solve exactly (by iterative substitution)

$$T(n) = \begin{cases} 4 & n = 1 \\ 4T(n-1) + 3 & n > 1 \end{cases}$$

# Problem

Problem: Solve exactly (by iterative substitution)

$$T(n) = \begin{cases} 4 & n = 1 \\ 4T(n-1) + 3 & n > 1 \end{cases}$$

Solution: $T(n) = 4^n + 4^{n-1} - 1$

Proof?

# Another example

$$T(N) = 2T(\sqrt{N}) + 1 \qquad T(2) = 0$$

$$2T(N^{1/2}) + 1$$

$$2(2T(N^{1/4}) + 1) + 1$$

$$4T(N^{1/4}) + 1 + 2$$

$$8T(N^{1/8}) + 1 + 2 + 4$$

$$\ldots$$

# Another example

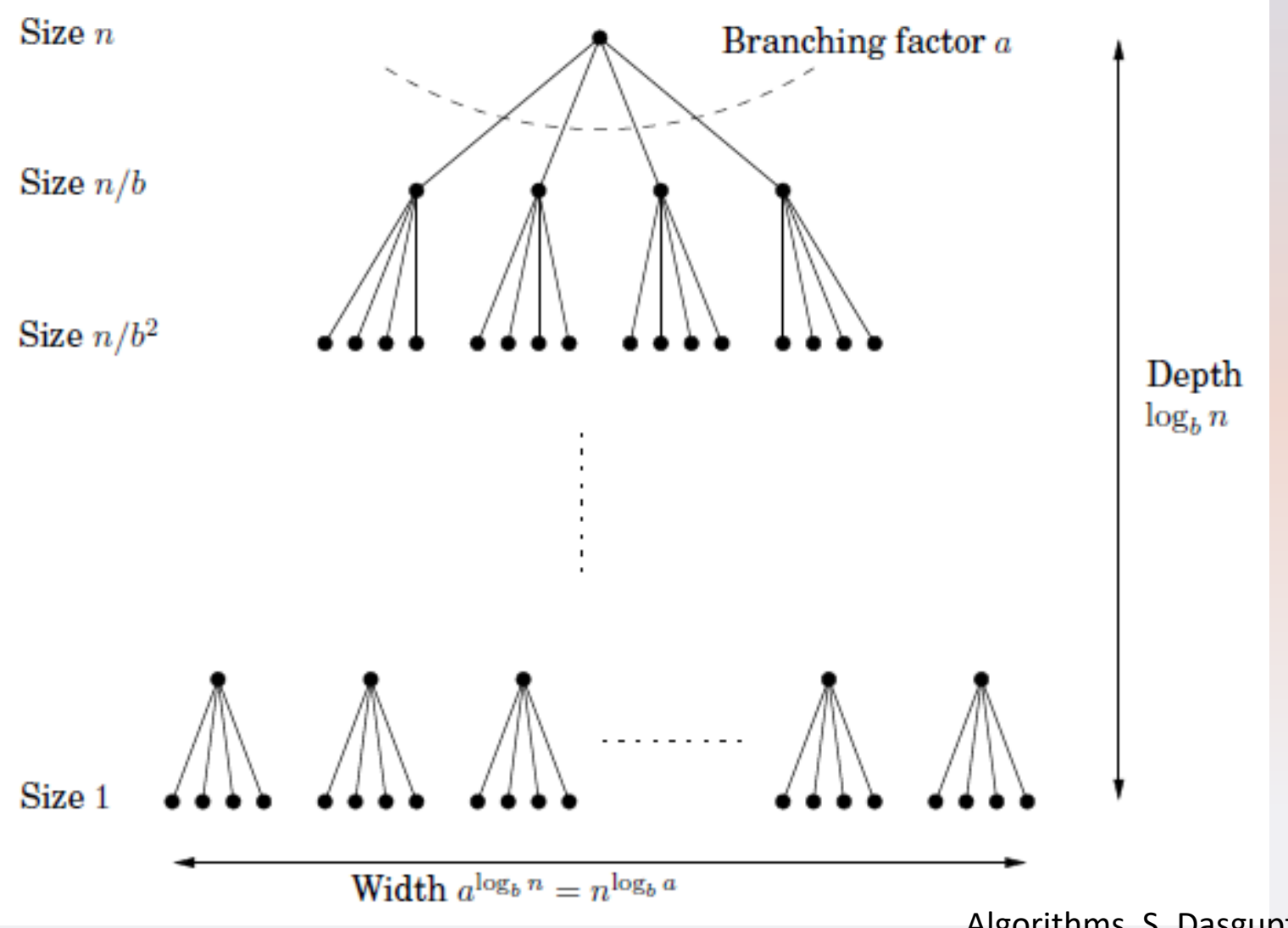
$$2^i T\left(N^{\frac{1}{2^i}}\right) + 2^0 + 2^1 + \ldots + 2^i - 1$$

The expansion stops for $N^{\frac{1}{2^i}} = 2$

i.e., $i = \log\log N$

$$T(N) = 2^0 + 2^1 + \ldots + 2^{\log\log N - 1} = \log N \text{-}1$$

# Divide-and-Conquer Master Theorem




Algorithms, S. Dasgupta

Let's solve the following recurrence in general:

$$T(n) = aT(n/b) + O(n^d)$$

*where a > 0, b > 1, T(1) = 1*

*Do repeated substitutions:*

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

$=$

Let's solve the following recurrence in general:

$$T(n) = aT(n/b) + O(n^d)$$

*where a > 0, b > 1, T(1) = 1*

*Do repeated substitutions:*

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

$= a[aT(n/b^2) + c(n/b)^d] + cn^d$

$= a^2T(n/b^2) + c\left(a/b^d\right)n^d + cn^d$

…

=

Let's solve the following recurrence in general:

$$T(n) = aT(n/b) + O(n^d)$$

*where a > 0, b > 1, T(1) = 1*

*Do repeated substitutions:*

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

$= a[aT(n/b^2) + c(n/b)^d] + cn^d$

$= a^2T(n/b^2) + c(a/b^d)n^d + cn^d$

…

$= a^jT(n/b^j) + cn^d\left[(a/b^d)^{j-1} + \cdots + (a/b^d)^2 + (a/b^d) + 1\right]$

…

=

Let's solve the following recurrence in general:

$$T(n) = aT(n/b) + O(n^d)$$

*where a > 0, b > 1, T(1) = 1*

*Do repeated substitutions:*

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

$= a[aT(n/b^2) + c(n/b)^d] + cn^d$

$= a^2T(n/b^2) + c(a/b^d)n^d + cn^d$

…

$= a^jT(n/b^j) + cn^d\left[(a/b^d)^{j-1} + \cdots + (a/b^d)^2 + (a/b^d) + 1\right]$

…

$= a^{\log_b n}T(1) + cn^d \cdot \sum_{i=0}^{\log_b n-1} (a/b^d)^i$

$= n^{\log_b a} + cn^d \cdot \sum_{i=0}^{\log_b n-1} (a/b^d)^i$

$$n^{\log_b a} + cn^d \cdot \sum_{i=0}^{\log_b n - 1} (a/b^d)^i$$

Case 1: $a < b^d$ ($d > \log_b a$). The second term is a geometric series with the ratio smaller than 1, so $\sum_{i=0}^{\log_b n-1}(a/b^d)^i = O(1)$. The first term is $n^{\log_b a}$ with $\log_b a < \log_b b^d = d$, so we get $T(n) = O(n^d)$.

Case 2: $a = b^d$. The first term is $n^d$. In the summation, we have $\log_b n$ terms and they are all equal a$/b^d = 1$, so the second term is $cn^d \log_b n$. Thus we get $T(n) = O(n^d \log n)$.

Case 3: $a > b^d (d < \log_b a)$. Summing the geometric series in the second term, we get

$$\sum_{i=0}^{\log_b n-1} (a/b^d)^i = \frac{(a/b^d)^{\log_b n} - 1}{(a/b^d) - 1} = \frac{b^d}{a - b^d}(a^{\log_b n}/b^{d \log_b n} - 1)$$
$$= \frac{b^d}{a - b^d}(n^{\log_b a}/n^d - 1)$$

So

$$T(n) = n^{\log_b a} + c\frac{b^d}{a-b^d}\left(n^{\log_b a} - n^d\right) = O(n^{\log_b a}).$$

# Divide-and-Conquer
# Master Theorem

Theorem **(Master Theorem):** If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0, b > 1$, and d $\geq 0$, then

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

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

# Master Theorem

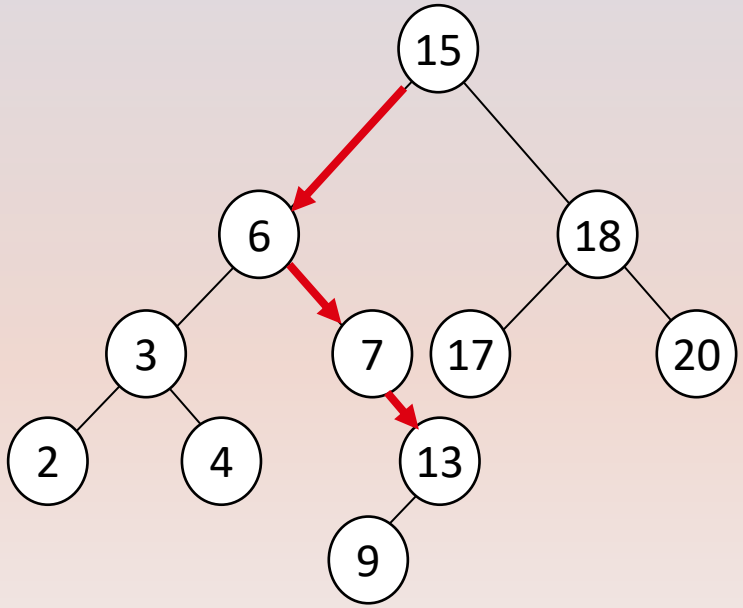
Given the recurrence

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is a asymptotically positive function,

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then
   $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then
   $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$,
   and if $af(n/b) \leq cf(n)$ for some constant $c < 1$
   and all sufficiently large $n$, then
   $T(n) = \Theta(f(n))$

# Master method: binary search




$$T(n) = T(n/2) + O(1)$$

a = 1; b = 2; d = 0;

$\log_b a = \log_2 1 = 0$; $\log_b a = d$;

$$T(n) = O(n^0 \log n) = O(\log n)$$

- Search for key 13:

  $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

# Divide-and-Conquer

(d) **Algorithm** PrintUs ($n$ : **integer**)

```
if n < 4
    print("U")
else
    PrintUs(⌈n/4⌉)
    PrintUs(⌊n/4⌋)
    for i ← 1 to 11 do print("U")
```

(d)
There are 2 recursive calls, each with parameter $\lceil n/4 \rceil$. Since we are looking for an asymptotic solution, we can ignore rounding. Then the number of letters printed can be expressed by the recurrence:

$$X(n) = 2X(n/4) + 11.$$

We apply the Master Theorem with $a = 2$, $b = 4$, $c = 11$, $d = 0$. Here, we have $a > b^d$, so the solution is $\Theta(n^{\log_4 2})$.

# Divide-and-Conquer

(e) **Algorithm** PRINTVS ($n$ : **integer**)

```
if n < 3
    print("V")
else
    for j ← 1 to 9 do PRINTVS(⌊n/3⌋)
    for i ← 1 to 2n^3 do print("V")
```

(e)
There are 9 recursive calls, each with parameter $\lfloor n/3 \rfloor$. Since we are looking for an asymptotic solution, we can ignore rounding. Then the number of letters printed can be expressed by the recurrence:

$$X(n) = 9X(n/3) + 2n^3.$$

We apply the Master Theorem with $a = 9$, $b = 3$, $c = 2$, $d = 3$. Here, we have $a < b^d$, so the solution is $\Theta(n^3)$.

# Linear-time selection

# Linear-time selection

- Problem: Select the *i*-th smallest element in an unsorted array of size *n*
  (assume distinct elements)
- Trivial solution: sort *A*, select *A[i]*
  time complexity is *O(n log n)*

- Can we do it in linear time? Yes, thanks to Blum, Floyd, Pratt, Rivest, and Tarjan

# Linear-time selection

Select (*A*, *start*, *end*, *i*) /* *i* is the *i*-th order statistic */

1. divide input array *A* into *[n/5]* groups of size 5
   (and one leftover group if *n % 5* is not 0)
2. find the median of each group of size 5 by sorting
   the groups of 5 and then picking the middle element
3. call Select recursively to find *x*, the median of the *[n/5]*
   medians
4. partition array around *x*, splitting it into two arrays
   *L* (elements smaller than *x*) and *R* (elements bigger than *x*)
5. $k \leftarrow |L|+1$
   **if** $(i = k)$ **then return** *x*
   **else if** $(i < k)$ **then** Select $(L, i)$
   **else** Select $(R, i-k)$

$\lceil r \rceil$ means the *ceiling* (rounding to the next integer) of real number *r*

# Python linear-time selection

```
def selection(a, rank):
    n = len(a)
    if n <= 5:
        return rank_by_sorting(a, rank)
    medians = [rank_by_sorting(a[i:i+5], 3)
               for i in range(0, n-4, 5)]
    median = selection(medians, (len(medians) + 1) // 2)
    L, R = [], []
    for x in a:
        if x < median:
            L += [x]
        else:
            R += [x]
    if rank <= len(L):
        return selection(L, rank)
    else:
        return selection(R, rank - len(L))
```

# Example

Let us run Select(A, 1, 28, 11), where

A={12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2, 19, 12, 5, 18, 20, 33, 16, 33, 21, 30, 3, 47}

Note that the elements in this example are not distinct.

# Example

First make groups of 5

| | | | | | |
|---|---|---|---|---|---|
| 12 | 4 | 43 | 2 | 20 | 30 |
| 34 | 17 | 82 | 19 | 33 | 3 |
| 0 | 32 | 25 | 12 | 16 | 47 |
| 3 | 3 | 27 | 5 | 33 | |
| 22 | 28 | 34 | 18 | 21 | |

# Example

Then find medians in each group

| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 25 | 2 | 16 | 3 |
| 3 | 4 | 27 | 5 | 20 | 30 |
| 12 | 17 | 34 | 12 | 21 | 47 |
| 22 | 28 | 43 | 18 | 33 | |
| 34 | 32 | 82 | 19 | 33 | |

# Example

Then find median of medians

| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 25 | 2 | 16 | 3 |
| 3 | 4 | 27 | 5 | 20 | 30 |
| 12 | 17 | 34 | 12 | 21 | 47 |
| 22 | 28 | 43 | 18 | 33 | |
| 34 | 32 | 82 | 19 | 33 | |

12, 12, 17, 21, 30, 34

# Example

Use 17 as the pivot value and partition original array

| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 25 | 2 | 20 | 3 |
| 3 | 4 | 27 | 5 | 16 | 30 |
| 12 | 17 | 34 | 12 | 21 | 47 |
| 22 | 28 | 43 | 18 | 33 | |
| 34 | 32 | 82 | 19 | 33 | |

12, 12, 17, 21, 30, 34

# Example

After partitioning

$L$ = {12, 0, 3, 4, 3, 2, 12, 5, 16, 3}

*L contains 10 elements smaller than 17*

{17} *this is the 11-th smallest*

$R$ = {34, 22, 32, 28, 43, 82, 25, 27, 34, 19, 18, 20, 33, 33, 21, 30, 47}

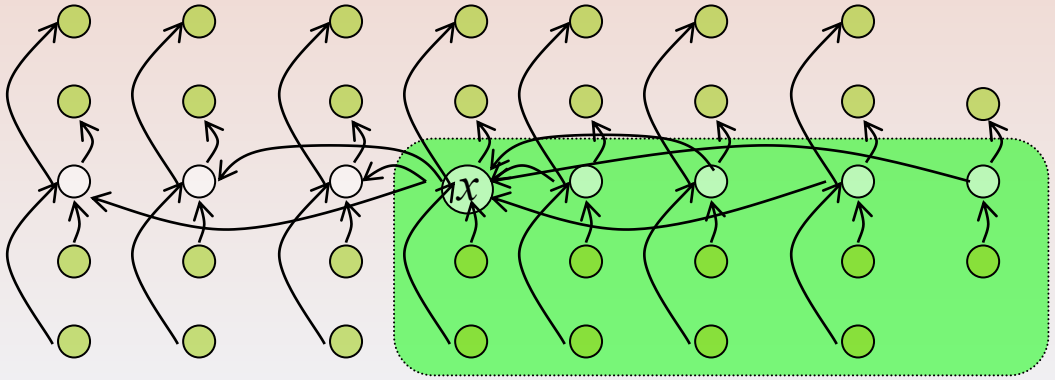
*R contains 17 elements bigger than 17*

# Linear-time selection

- Finding the median of medians guarantees that $x$ causes a "good split"
- At least a constant fraction of the $n$ elements $\leq x$ and a constant fraction $> x$
- Analysis: we need to find the worst case for the size of $L$ and $R$

# Linear-time selection: analysis

Observation: At least 1/2 of the medians found in step 2 are greater than the median of medians $x$. So at least half of the $\lceil n/5 \rceil$ groups contribute 3 elements that are bigger than $x$, except for the one group with less than 5 elements and the group with $x$ itself



# Linear-time selection: analysis

- Therefore there are
$$3([1/2\ [n/5]] - 2) \geq (3n/10) - 6$$
elements that are $> x$ (or $< x$)
- So worst-case split has at most $(7n/10) + 6$ elements in "big" section of the problem, that is:
$$max\{|L|,|R|\} < (7n/10) + 6$$

# Linear-time selection: analysis

Running Time:

1. $O(n)$ (break into groups of 5)
2. $O(n)$ (sorting 5 numbers and finding median is $O(1)$ time)
3. $T([n/5])$ (recursive call to find median of medians)
4. $O(n)$ (partition is linear time)
5. $T(7n/10 + 6)$ (maximum size of subproblem)

Recurrence relation

$$T(n) = T([n/5]) + T(7n/10 + 6) + O(n) \quad n > 80$$
$$= \Theta(1) \quad n \leq 80$$

# Linear-time selection: analysis

Fact: $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$ *is* $O(n)$

Proof:

Base case: easy (omitted).

$$
\begin{aligned}
T(n) &= T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) \\
&\le c\lceil n/5 \rceil + c(7n/10 + 6) + O(n) \\
&\le c((n/5) + 1) + 7cn/10 + 6c + O(n) \\
&= cn - [c(n/10 - 7) - dn] \\
&\le cn
\end{aligned}
$$

This step holds since $n \ge 80$ implies $(n/10 - 7)$ is positive.

Choosing $c$ big enough makes $c(n/10 - 7) - dn$ positive, so last line holds.

# Summary (1/3)

- Goal: analyze the worst-case time-complexity of iterative and recursive algorithms
- Tools:
  - Pseudo-code
  - Big-O, Big-Omega, Big-Theta notations
  - Recurrence relations
  - Discrete Math (summations, induction proofs, methods to solve recurrence relations)

# Summary (2/3)

- Pure iterative algorithm:
  - Analyze the loops
  - Determine how many times the inner core is repeated as a function of the input size
  - Determine the worst-case for the input
  - Write the number of repetitions as a function of the input size
  - Simplify the function using big-O or big-Theta notation (optional)

# Summary (3/3)

- Recursive + iterative algorithm:
  - Analyze the recursive calls and the loops
  - Determine how many recursive calls are made and the size of the arguments of the recursive calls
  - Determine how much extra processing (loops) is done
  - Determine the worst-case for the input
  - Derive a recurrence relation
  - Solve the recurrence relation
  - Simplify the solution using big-O, or big-Theta

# Integer multiplication (Karatsuba)

# Integer multiplication

- Given positive integers *y*, *z*, compute *x*=*y***z*
- A naïve multiplication algorithm is below

```
def naive_mul(y,z):
    x = 0
    while z > 0:
        if z % 2 == 1:
            x += y
        y *= 2
        z /= 2
    return x
```

Remark: these two operations can be implemented as O(1) shifts

# Integer multiplication

Addition takes $O(n)$ bit operations, where $n$ is the number of bits in $y$ and $z$. The naive multiplication algorithm takes $O(n)$ $n$-bit additions. Therefore, the naive multiplication algorithm takes $O(n^2)$ bit operations.

Can we multiply using fewer bit operations?

# Integer multiplication

Suppose $n$ is a power of 2. Divide $y$ and $z$ into two halves, each with $n/2$ bits.

| | | |
|---|---|---|
| $y$ | $a$ | $b$ |
| $z$ | $c$ | $d$ |

# Integer multiplication

Then

$$\begin{aligned} y &= a2^{n/2} + b \\ z &= c2^{n/2} + d \end{aligned}$$

and so

$$\begin{aligned} yz &= (a2^{n/2} + b)(c2^{n/2} + d) \\ &= ac2^n + (ad + bc)2^{n/2} + bd \end{aligned}$$

# Integer multiplication

This computes $yz$ with 4 multiplications of $n/2$ bit numbers, and some additions and shifts. Running time given by $T(1) = c$, $T(n) = 4T(n/2) + dn$, which has solution $O(n^2)$ by the General Theorem. No gain over naive algorithm!

**Example 5.7:** *Consider the recurrence*

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

*In this case,* $n^{\log_b a} = n^{\log_2 4} = n^2$. *Thus, we are in Case 1, for* $f(n)$ *is* $O(n^{2-\varepsilon})$ *for* $\varepsilon = 1$. *This means that* $T(n)$ *is* $\Theta(n^2)$ *by the master method.*

# Integer multiplication (Karatsuba algorithm)

- Consider the product
  $(a-b)(d-c) = (ad + bc) – (ac + bd)$
- It contains two of the products we need ($ad$ and $bc$):
- Then
  $yz = ac2^n + [(a-b)(d-c)+(ac+bd)]2^{n/2} + bd$

$$yz = ac2^n + (ad + bc)2^{n/2} + bd$$

- We need three multiplications of $n/2$ bits and $O(n)$ additional work

# Integer multiplication (Karatsuba algorithm)

Therefore,

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 3T(n/2) + dn & \text{otherwise} \end{cases}$$

where $c, d$ are constants.

Therefore, by our general theorem, the divide and conquer multiplication algorithm uses

$$T(n) = O(n^{\log 3}) = O(n^{1.59})$$

bit operations.

# Karatsuba algorithm

```
def multiply(y,z):
    l = max(len(y), len(z))
    if l == 1:
        return [y[0] * z[0]]
    y = [0 for i in range(len(y), l)] + y;
    z = [0 for i in range(len(z), l)] + z;
    m0 = (l + 1) / 2
    a = y[:m0]
    b = y[m0:]
    c = z[:m0]
    d = z[m0:]
```

Remark: pad y and z so that they have the same length

# Karatsuba algorithm (continued)

```
p0 = multiply(a, c)
p1 = multiply(add(a, b), add(c, d))
p2 = multiply(b, d)

z0 = p0
z1 = subtract(p1, add(p0, p2))
z2 = p2

z0prod = z0 + [0 for i in range(0, l)]
z1prod = z1 + [0 for i in range(0, l / 2)]

return add(add(z0prod, z1prod), z2)
```

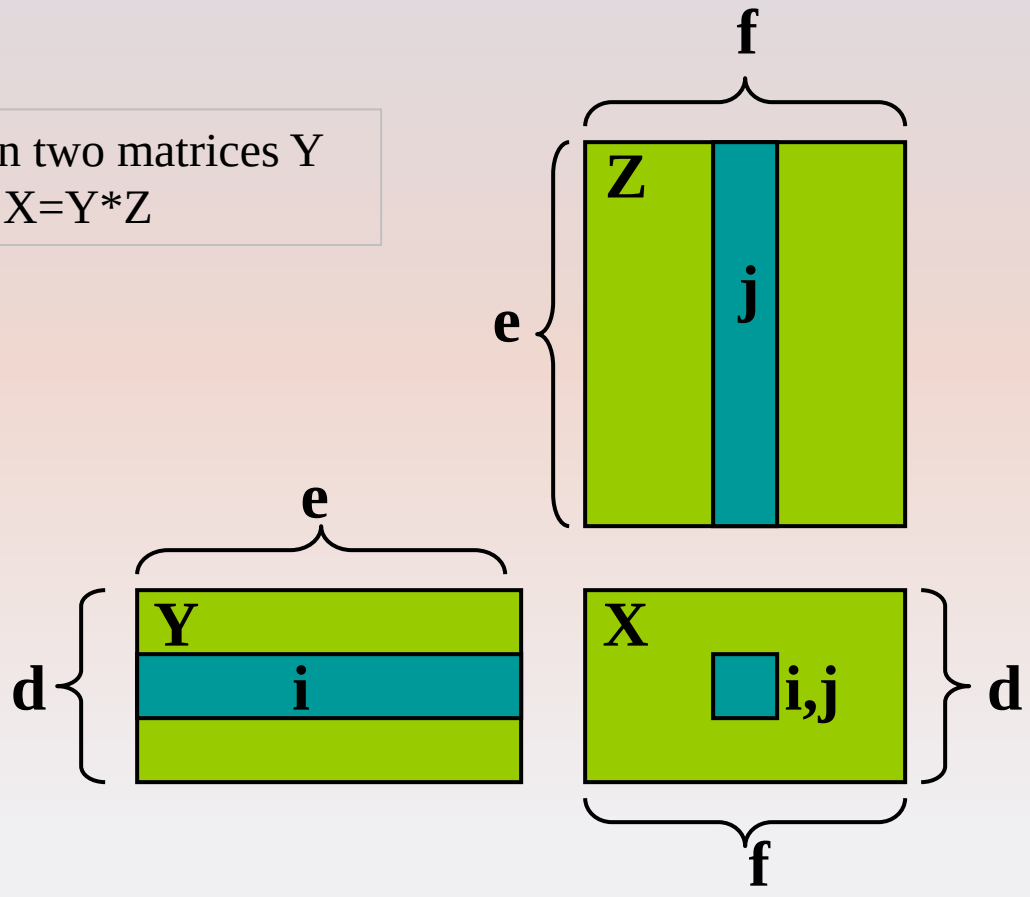
Remark: compute
z1 = p1 – p0 – p2

Remark: compute
z0 b^l + z1 b^(l/2) + z2

# Matrix multiplication (Strassen)

# Matrix multiplication

**Problem**: Given two matrices Y and Z compute X=Y*Z



# Matrix multiplication

```
def mult(Y,Z):
    X = zero(len(Y),len(Z[0]))

    for i in range(len(Y)):
        for j in range(len(Z[0])):
            for k in range(len(Z)):
                X[i][j] += Y[i][k]*Z[k][j]

    return X
```

Algorithm **`mult(Y,Z)`** is $O(n^3)$, can we do better?

# Matrix multiplication

Divide $X, Y, Z$ each into four $(n/2) \times (n/2)$ matrices.

$$X = \begin{bmatrix} I & J \\ K & L \end{bmatrix}$$

$$Y = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

$$Z = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

# Matrix multiplication

Then

$$
\begin{aligned}
I &= AE + BG \\
J &= AF + BH \\
K &= CE + DG \\
L &= CF + DH
\end{aligned}
$$

# Matrix multiplication

Let $T(n)$ be the time to multiply two $n \times n$ matrices.

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 8T(n/2) + dn^2 & \text{otherwise} \end{cases}$$

where $c, d$ are constants.

# Matrix multiplication

Therefore,

$$
\begin{aligned}
T(n) &= 8T(n/2) + dn^2 \\
&= 8(8T(n/4) + d(n/2)^2) + dn^2 \\
&= 8^2T(n/4) + 2dn^2 + dn^2 \\
&= 8^3T(n/8) + 4dn^2 + 2dn^2 + dn^2 \\
&= 8^iT(n/2^i) + dn^2 \sum_{j=0}^{i-1} 2^j \\
&= 8^{\log n}T(1) + dn^2 \sum_{j=0}^{\log n-1} 2^j \\
&= cn^3 + dn^2(n-1) \\
&= O(n^3)
\end{aligned}
$$

# Matrix multiplication

- The naïve Divide and Conquer algorithm is no better than the straightforward algorithm
- However, it gives us an insight on the next algorithm
- Strassen's algorithm uses only 7 multiplications instead of 8

# Strassen algorithm

Compute

$$
\begin{aligned}
M_1 &:= (A + C)(E + F) \\
M_2 &:= (B + D)(G + H) \\
M_3 &:= (A - D)(E + H) \\
M_4 &:= A(F - H) \\
M_5 &:= (C + D)E \\
M_6 &:= (A + B)H \\
M_7 &:= D(G - E)
\end{aligned}
$$

# Strassen algorithm

Then,

$$
\begin{aligned}
I &:= M_2 + M_3 - M_6 - M_7 \\
J &:= M_4 + M_6 \\
K &:= M_5 + M_7 \\
L &:= M_1 - M_3 - M_4 - M_5
\end{aligned}
$$

# Strassen algorithm

$$
\begin{aligned}
I &:= M_2 + M_3 - M_6 - M_7 \\
&= (B+D)(G+H) + (A-D)(E+H) \\
&\quad - (A+B)H - D(G-E) \\
&= (BG + BH + DG + DH) \\
&\quad + (AE + AH - DE - DH) \\
&\quad + (-AH - BH) + (-DG + DE) \\
&= BG + AE
\end{aligned}
$$

# Strassen algorithm

$$
\begin{aligned}
J &:= M_4 + M_6 \\
&= A(F - H) + (A + B)H \\
&= AF - AH + AH + BH \\
&= AF + BH
\end{aligned}
$$

# Strassen algorithm

$$
\begin{aligned}
K &:= M_5 + M_7 \\
&= (C + D)E + D(G - E) \\
&= CE + DE + DG - DE \\
&= CE + DG
\end{aligned}
$$

# Strassen algorithm

$$
\begin{aligned}
L &:= M_1 - M_3 - M_4 - M_5 \\
&= (A + C)(E + F) - (A - D)(E + H) \\
&\quad - A(F - H) - (C + D)E \\
&= AE + AF + CE + CF - AE - AH \\
&\quad + DE + DH - AF + AH - CE - DE \\
&= CF + DH
\end{aligned}
$$

```
def strassen(Y,Z):
    if len(Y) <= 2:
        return mult(Y,Z)
    else:
        A,B,C,D = partition(Y)
        E,F,G,H = partition(Z)
        M1 = strassen(add(A,C),add(E,F))
        M2 = strassen(add(B,D),add(G,H))
        M3 = strassen(sub(A,D),add(E,H))
        M4 = strassen(A,sub(F,H))
        M5 = strassen(add(C,D),E)
        M6 = strassen(add(A,B),H)
        M7 = strassen(D,sub(G,E))
        I = sub(sub(add(M2,M3),M6),M7)
        J = add(M4,M6)
        K = add(M5,M7)
        L = sub(sub(sub(M1,M3),M4),M5)
        return recompose(I,J,K,L)
```

# Analysis of Strassen algorithm

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 7T(n/2) + dn^2 & \text{otherwise} \end{cases}$$

where $c, d$ are constants.

# Analysis of Strassen algorithm

$$
\begin{aligned}
T(n) &= 7T(n/2) + dn^2 \\
&= 7(7T(n/4) + d(n/2)^2) + dn^2 \\
&= 7^2T(n/4) + 7dn^2/4 + dn^2 \\
&= 7^3T(n/8) + 7^2dn^2/4^2 + 7dn^2/4 + dn^2 \\
&= 7^iT(n/2^i) + dn^2 \sum_{j=0}^{i-1} (7/4)^j \\
&= 7^{\log n}T(1) + dn^2 \sum_{j=0}^{\log n-1} (7/4)^j \\
&= cn^{\log 7} + dn^2 \frac{(7/4)^{\log n} - 1}{7/4 - 1} \\
&= cn^{\log 7} + \frac{4}{3}dn^2(\frac{n^{\log 7}}{n^2} - 1) \\
&= O(n^{\log 7}) \\
&\approx O(n^{2.8})
\end{aligned}
$$

# Discussion

- There is a large constant hidden which makes Strassen impractical, unless the matrices are large ($n>45$) and dense
- For sparse matrices there are faster methods
- Strassen is not as *numerically stable* as the naïve
- Sub-matrices at each level consume space
- FYI: the current best algorithm for dense matrices runs in $O(n^{2.376})$
- Lower bound $\Omega(n^2)$ [for dense matrices]

# Closest Pair

# Closest Pair Problem

- Let $P_1=(x_1,y_1),\ldots,P_n=(x_n,y_n)$ be a set $S$ of $n$ points in the plane
- Problem: Find the two closest points in $S$
- Assumptions:
  - $n$ is a power of two
  - points are ordered by their $x$ coordinate (if not, we can sort them in $O(n \log n)$ time)

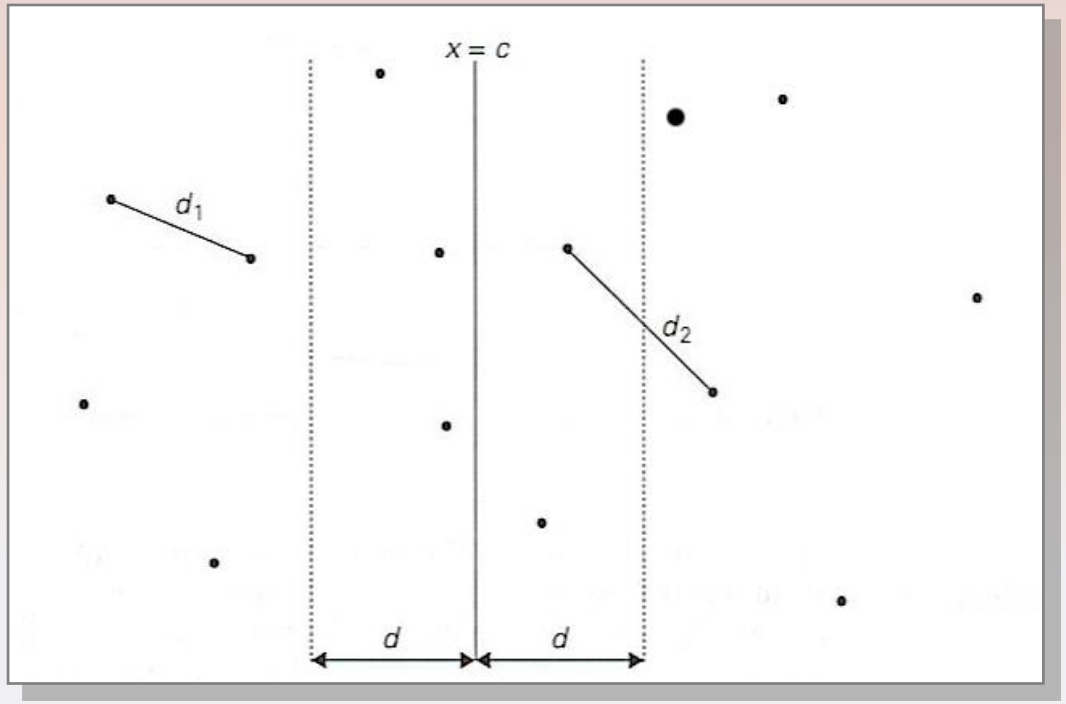
# Closest-Pair Problem: Brute-force

- Compute the distance between every pair of distinct points
- Return the indexes of the points for which the distance is the smallest

Time complexity?

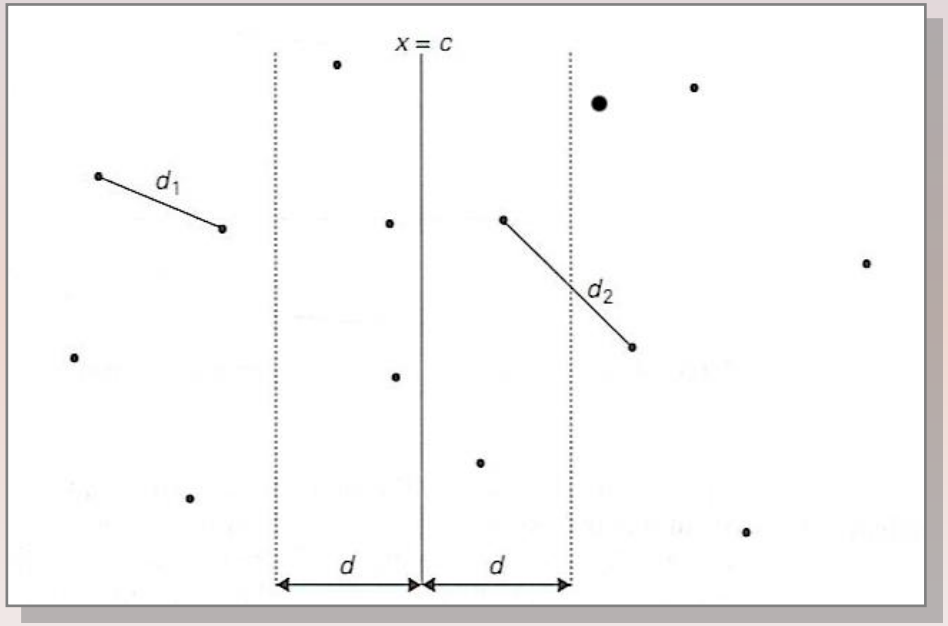
# Closest-Pair: Divide and Conquer

**Step 1.** Divide the points in $S$ into two subsets $S_1$ and $S_2$ by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line ($c$ is the median of the $x$ coord)



# Closest-Pair: Divide and Conquer

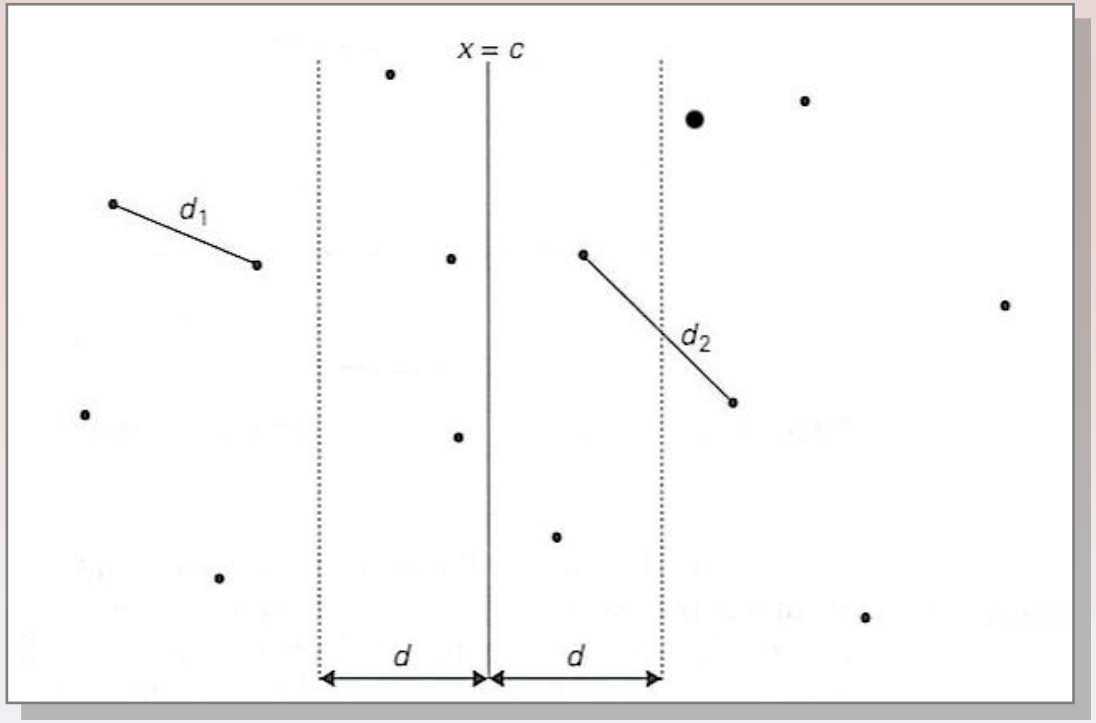
**Step 2.** Find recursively the closest pairs for the left and right subsets. Let $d_1$, $d_2$ be the distances of the two closest pairs.




Set $d = \min\{d_1, d_2\}$

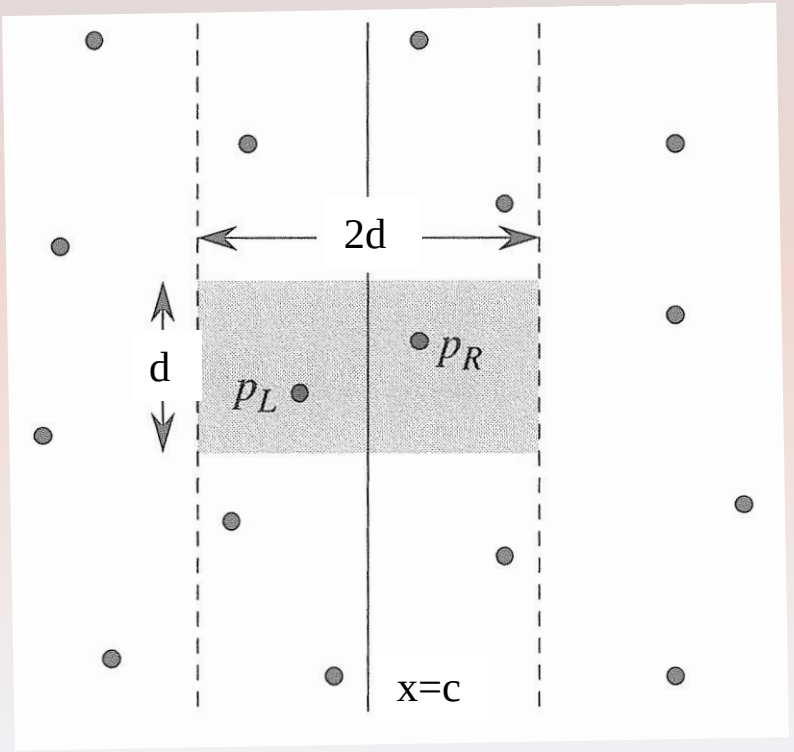
# Closest Pair: Divide and Conquer

**Step 3.** Consider the vertical strip $2d$-wide centered at $x=c$. Let $Y$ be the subset of points in this vertical strip of width $2d$



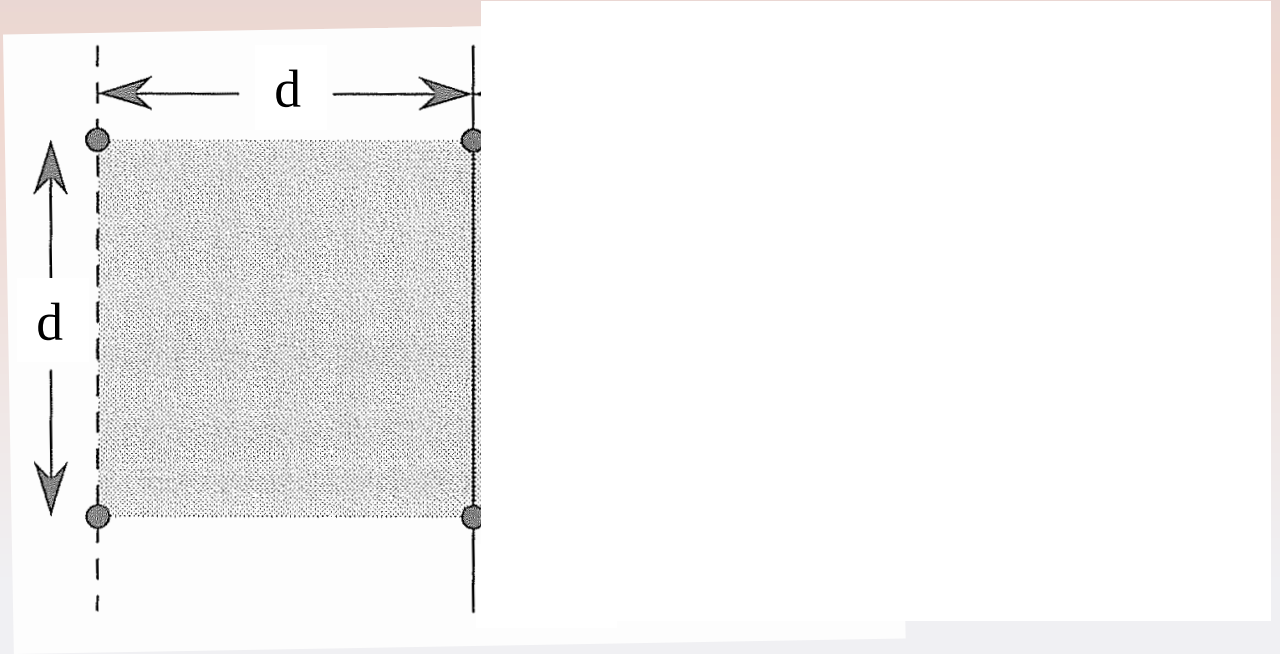
# Closest Pair: Divide and Conquer

- **Observation 1:** if a pair of points $p_L$,$p_R$ has distance less than $d$, both points of the pair **must** be within $Y$



# Closest Pair: Divide and Conquer

**Observation 2:** Since all the points within $S_1$ are at least $d$ units apart, at most 4 points can reside within the $d$ x $d$ square

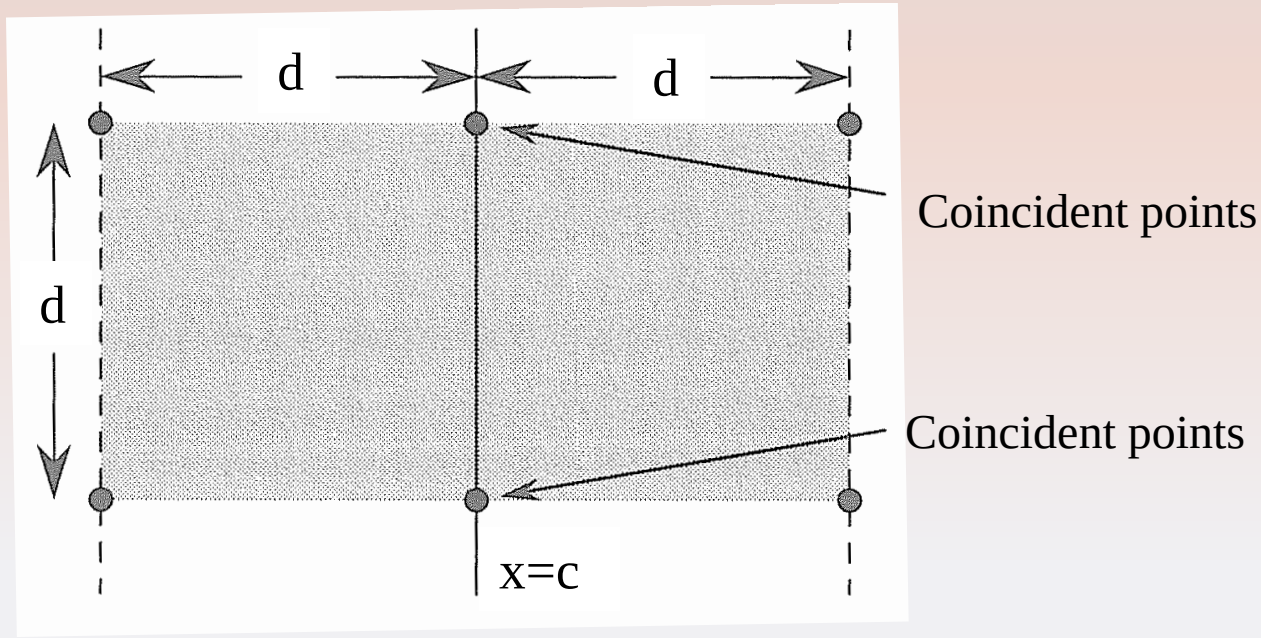


# Closest Pair: Divide and Conquer

**Proof:** Let's suppose (for sake of contradiction) that five or more points are found in a square of size *d* x *d*. Divide the square into four smaller squares of size *d/2* x *d/2*. At least one pair of points must fall within the same smaller square: these two points will be at a distance *d*/sqrt(2) < *d*, which leads to a contradiction.

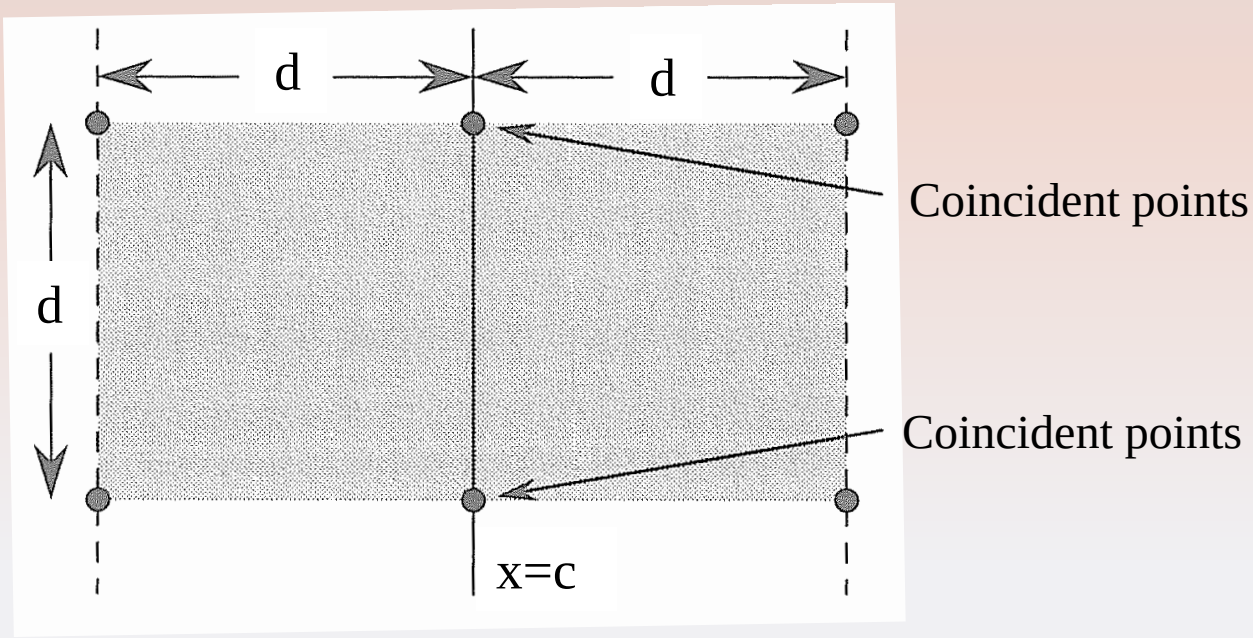
# Closest Pair: Divide and Conquer

**Consequence:** At most 8 points can reside within the $d$ x $2d$ rectangle, because on each side all points are at least $d$ unit apart



# Closest Pair: Divide and Conquer

**Step 4.** For each point *p* in *Y*, try to find points in *Y* that are within *d* units of *p*. Only *7* points in *Y* that follow *p* need to be considered



# Closest pair in Python

```
def closestPair(xP, yP):
    n = len(xP)
    if n <= 3:
        return bruteForceClosestPair(xP)
    Xl = xP[:n/2]
    Xr = xP[n/2:]
    Yl, Yr = [], []
    median = Xl[-1].x
    for p in yP:
        if p.x <= median:
            Yl.append(p)
        else:
            Yr.append(p)
```

Remark: **xP** and **yP** is the same of input points (x,y), but **xP** is sorted by x and **yP** is sorted by y

Remark: **Xl** is the first half of the points sorted by x, and **Xr** is the second half

Remark: **Yl** contains the points (sorted by y) which have a x coordinate smaller than the median

```
dl, pairl = closestPair(Xl, Yl)
 dr, pairr = closestPair(Xr, Yr)
 dm, pairm = (dl, pairl) if dl < dr else (dr, pairr)

 st = [p for p in yP if abs(p.x - median) < dm]
 n_st = len(st)
 closest = (dm, pairm)
 if n_st > 1:
     for i in range(n_st-1):
         for j in range(i+1,min(i+8, n_st)):
             if d(st[i],st[j]) < closest[0]:
                closest = (d(st[i],st[j]),(st[i],st[j]))
 return closest
```

Remark: variable **st** contains the points in the strip [median-dm, median+dm] sorted by y

Remark: **d**(x,y) returns the distance between x and y

# Analysis of the Closest-Pair Algorithm

- We can keep the points in *Y* stored in increasing order of their *y* coordinates, which is maintained by merging during the execution of step 4
- We can process the points in *Y* sequentially in linear time
- Running time is described by $T(n)=2T(n/2)+O(n)$
- By the Master Theorem, $T(n)$ is $O(n \log n)$

These slides were shared with me
by Dr. Stefano Lonardi
and modified with his permission.