Algorithms

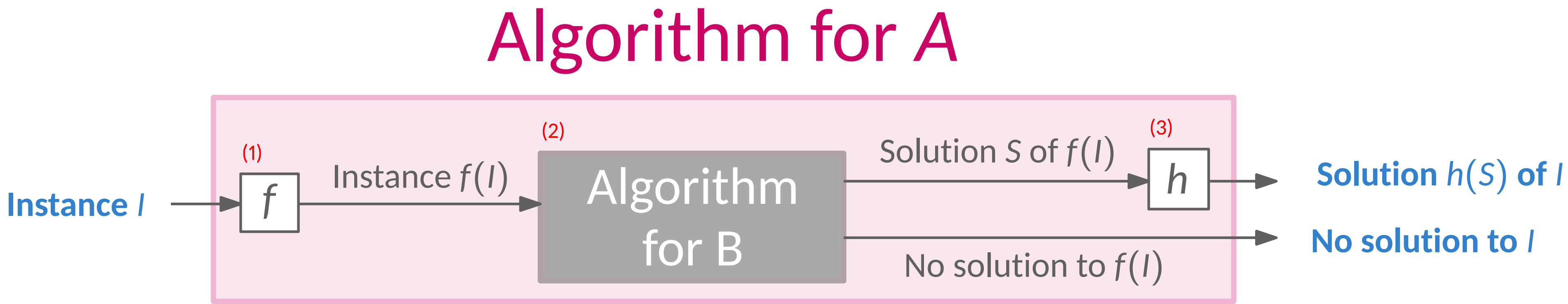
Recursion

Hee-Kap Ahn
Graduate School of Artificial Intelligence
Dept. Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)

Textbooks: Algorithms by Jeff Erickson /Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.

Reducing one problem A to another problem B.

- We know how to solve B efficiently, and use this to solve A.
- We write an algorithm for A using an algorithm for B as a subroutine.



Phase (1): Convert instance I to instance $f(I)$.

Phase (2): Execute the black box on $f(I)$.

Phase (3): Convert solution S to solution $h(S)$.

Reductions

Correctness. Assumption – The black box solves B correctly.

- The correctness of the algorithm for A is independent on the correctness of the algorithm for B.

We may not know exactly how the black boxes are implemented.

Running time. The running time T_B of the algorithm for B matters to the running time T_A of the algorithm for A.

$$T_A = T_B + \text{time for Phases (1) and (3).}$$

Recursion and Induction

Recursion. A kind of reduction. Simplify the original problem.

- If the given instance can be solved directly, solve it directly.
- Otherwise, reduce it to one or more **simpler instances of the same problem.**

Recursive reductions must lead to a **base case.**

Algorithm for computing n factorial:

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ f(n-1) \cdot n & \text{if } n > 1 \end{cases}$$

base case

The peasant multiplication algorithm:

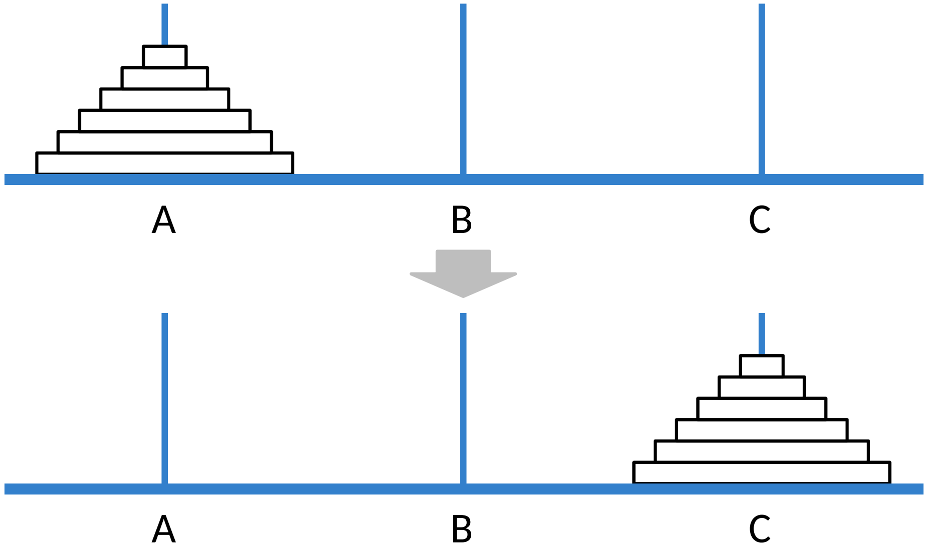
$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y+y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y+y) + y & \text{if } x \text{ is odd} \end{cases}$$

Let $x' = \lfloor x/2 \rfloor$ and $y' = y + y$. The given instance $x \cdot y$ is reduced to $x' \cdot y'$, a simpler instance because $x' < x$. If repeated, it decreases eventually to 0 (the base case).

Tower of Hanoi

Given a tower of n disks placed at peg A among three pegs (A, B, C),
Move the tower of n disks at peg A to peg C:

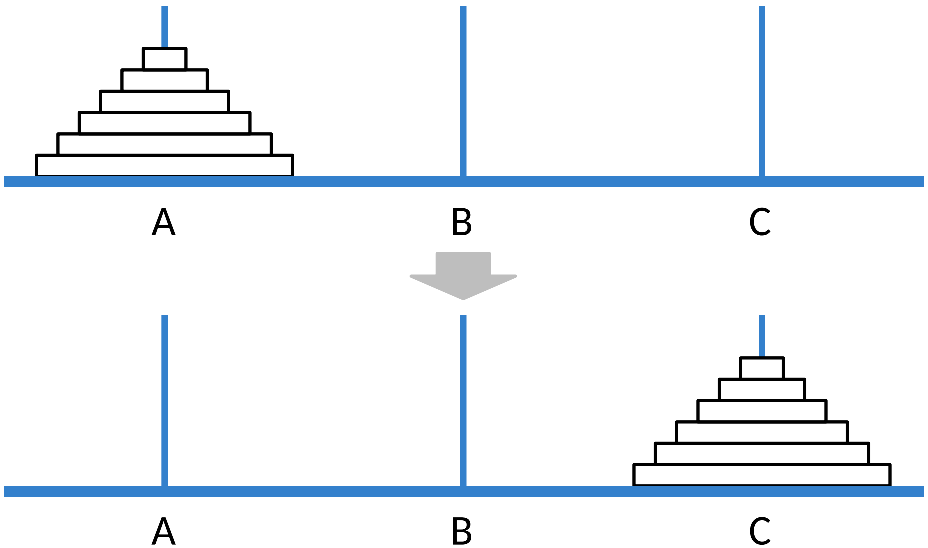
- Allowed to use a third spare peg as an occasional placeholder.
- Not allowed to place a disk on top of a smaller disk.

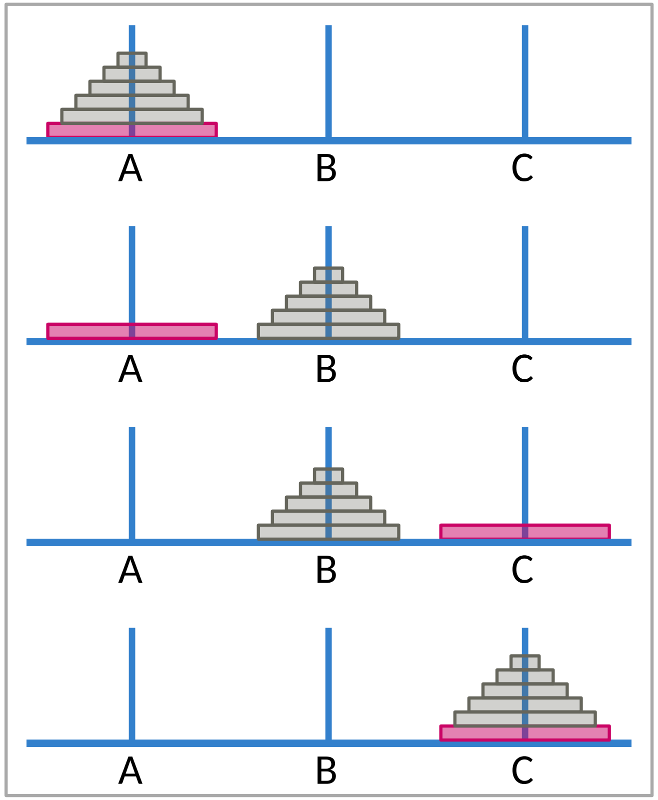


Tower of Hanoi

Given a tower of n disks placed at peg A among three pegs (A, B, C),
Move the tower of n disks at peg A to peg C:

- Allowed to use a third spare peg as an occasional placeholder.
- Not allowed to place a disk on top of a smaller disk.





Think recursively!

1. Move $n-1$ smaller disks to B. (**simpler instance**)
2. Move the largest disk to C. (**base case**)
3. Move $n-1$ smaller disks from B to C. (**simpler instance**)

Tower of Hanoi

1. Move $\underline{n-1 \text{ smaller disks}}$ to B. (**simpler instance**)
2. Move the largest disk to C. (**base case**)
3. Move $\underline{n-1 \text{ smaller disks}}$ from B to C. (**simpler instance**)

Reduction

n-disk Tower of Hanoi problem → $(n-1)$-disk Tower of Hanoi problem, $(n-1)$-disk Tower of Hanoi problem

Hanoi$(n, \text{A}, \text{C}, \text{B})$

if $n > 0$ **then**
  Hanoi$(n-1, \text{A}, \text{B}, \text{C})$
  move disk n from A to C
  Hanoi$(n-1, \text{B}, \text{C}, \text{A})$

Recurrence for the running time:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2T(n-1) + O(1) & \text{if } n \geq 1 \end{cases}$$

Thus, $T(n) = 2^n - 1$.

Find a key k in a sorted list $L[0 : n-1]$.

- If $j - i \leq 1$ for $L[i : j]$, solve it directly (**base case**).
- Otherwise,
  - compare k with $L[m]$ for $m = \lfloor (j-i)/2 \rfloor$, and
  - recurse either on $L[i : m-1]$ or on $L[m : j]$ (**simpler instance**).

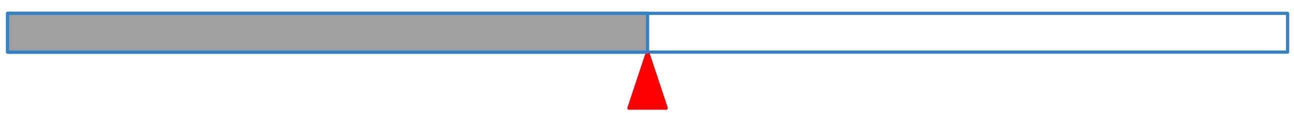
L

Binary Search

Find a key k in a sorted list $L[0 : n-1]$.

- If $j - i \leq 1$ for $L[i : j]$, solve it directly (**base case**).
- Otherwise,
  - compare k with $L[m]$ for $m = \lfloor (j-i)/2 \rfloor$, and
  - recurse either on $L[i : m-1]$ or on $L[m : j]$ (**simpler instance**).

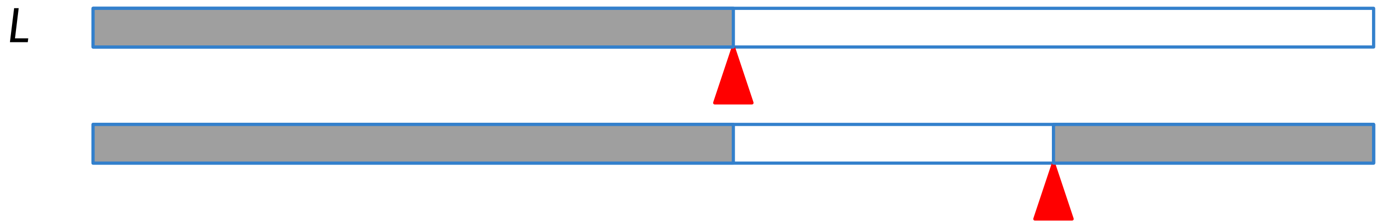
L



Binary Search

Find a key k in a sorted list $L[0 : n-1]$.

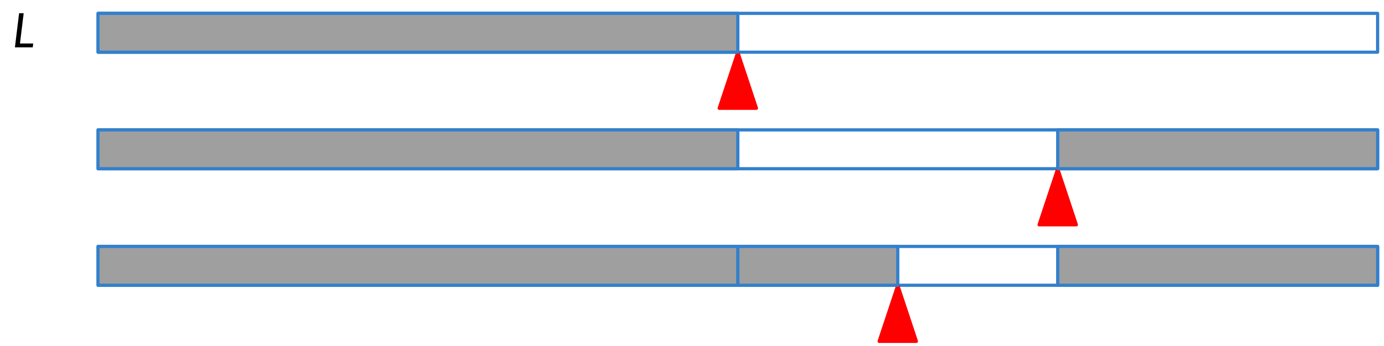
- If $j - i \leq 1$ for $L[i : j]$, solve it directly (**base case**).
- Otherwise,
  - compare k with $L[m]$ for $m = \lfloor (j-i)/2 \rfloor$, and
  - recurse either on $L[i : m-1]$ or on $L[m : j]$ (**simpler instance**).



Binary Search

Find a key k in a sorted list $L[0 : n-1]$.

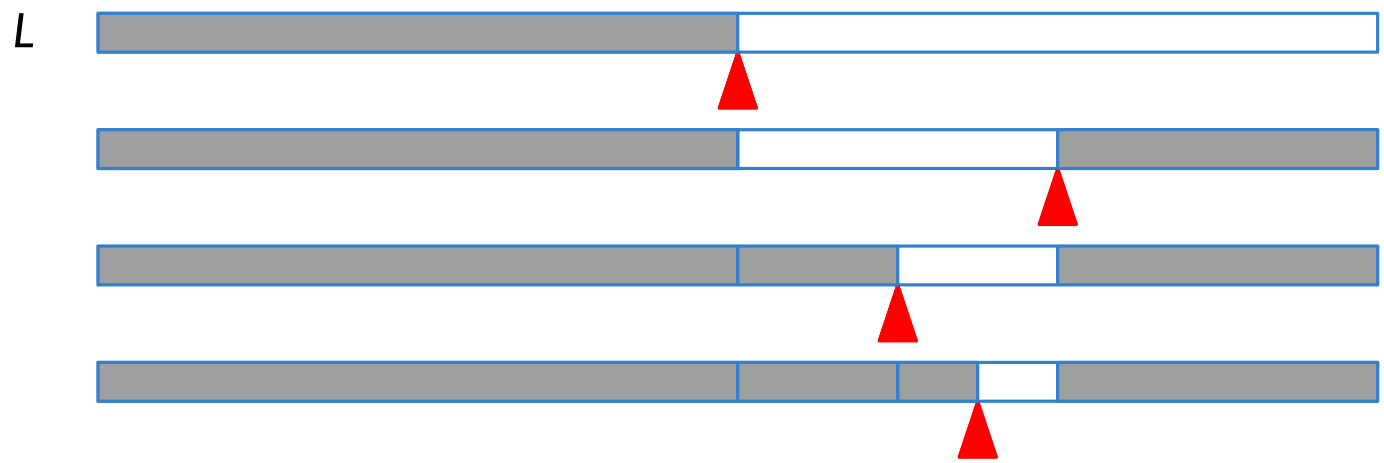
- If $j - i \leq 1$ for $L[i : j]$, solve it directly (**base case**).
- Otherwise,
  - compare k with $L[m]$ for $m = \lfloor (j-i)/2 \rfloor$, and
  - recurse either on $L[i : m-1]$ or on $L[m : j]$ (**simpler instance**).



Binary Search

Find a key k in a sorted list $L[0:n-1]$.

- If $j-i \leq 1$ for $L[i:j]$, solve it directly (**base case**).
- Otherwise,
  - compare k with $L[m]$ for $m = \lfloor (j-i)/2 \rfloor$, and
  - recurse either on $L[i:m-1]$ or on $L[m:j]$ (**simpler instance**).



Recurrence for the running time: $T(n) = T(\lceil n/2 \rceil) + O(1)$.

By induction, we can verify the running time of $T(n) = O(\log n)$.

Mergesort

Sort a list of numbers.

Developed by John von Neumann in 1945.

1. Split the list into two halves, (base case: $n \leq 1$)
2. Recursively mergesort each half, (**simpler instance**)
3. Merge them into a single sorted list.

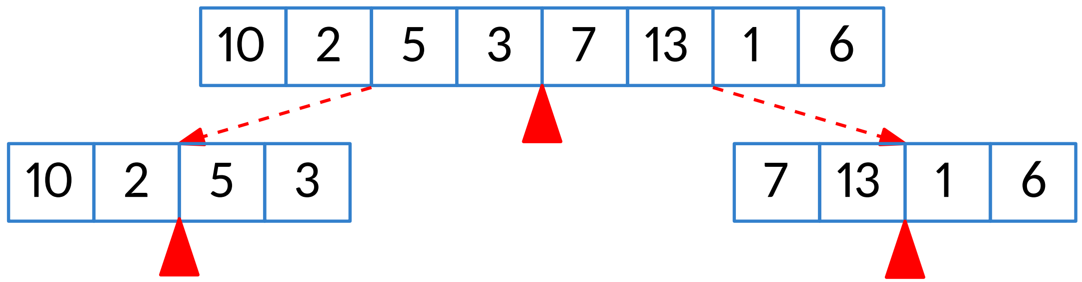
10	2	5	3	7	13	1	6

Mergesort

Sort a list of numbers.

Developed by John von Neumann in 1945.

1. Split the list into two halves, (base case: $n \leq 1$)
2. Recursively mergesort each half, (**simpler instance**)
3. Merge them into a single sorted list.

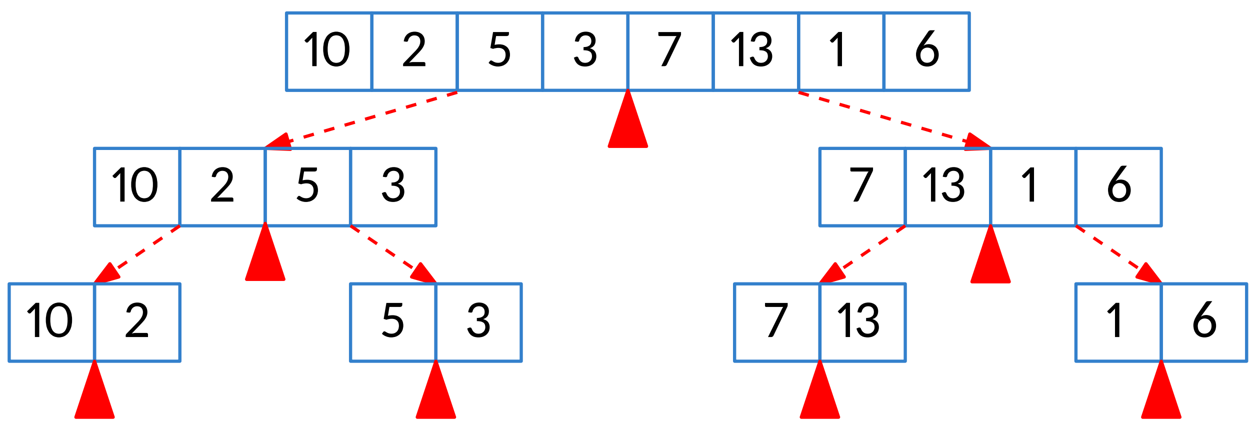


Mergesort

Sort a list of numbers.

Developed by John von Neumann in 1945.

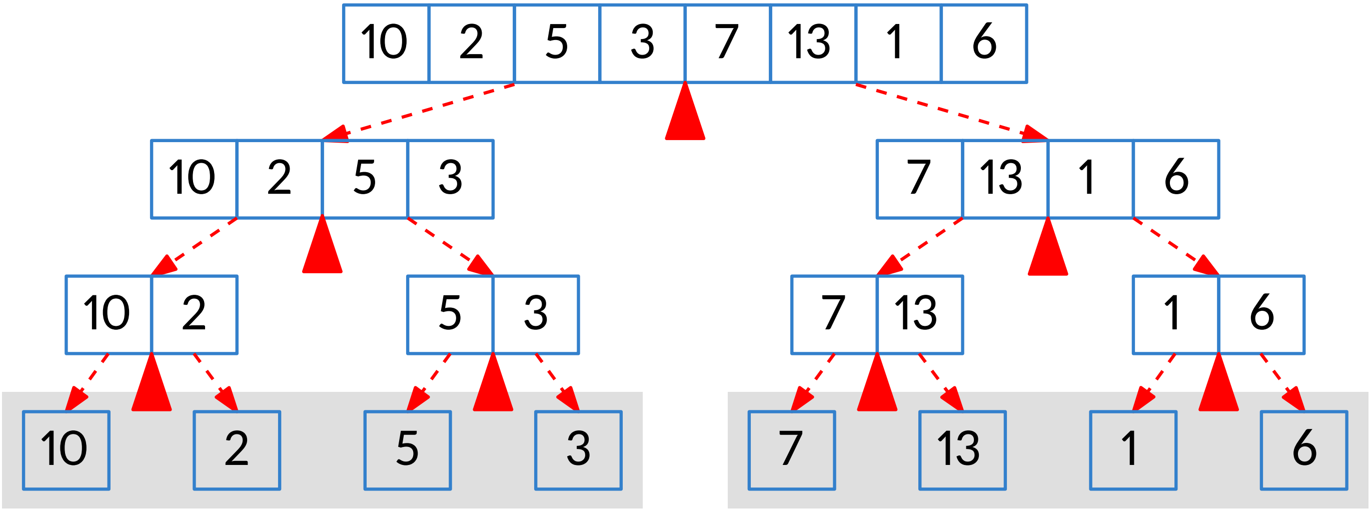
1. Split the list into two halves,
   (base case: $n \leq 1$)
2. Recursively mergesort each half, (**simpler instance**)
3. Merge them into a single sorted list.



Sort a list of numbers.

Developed by John von Neumann in 1945.

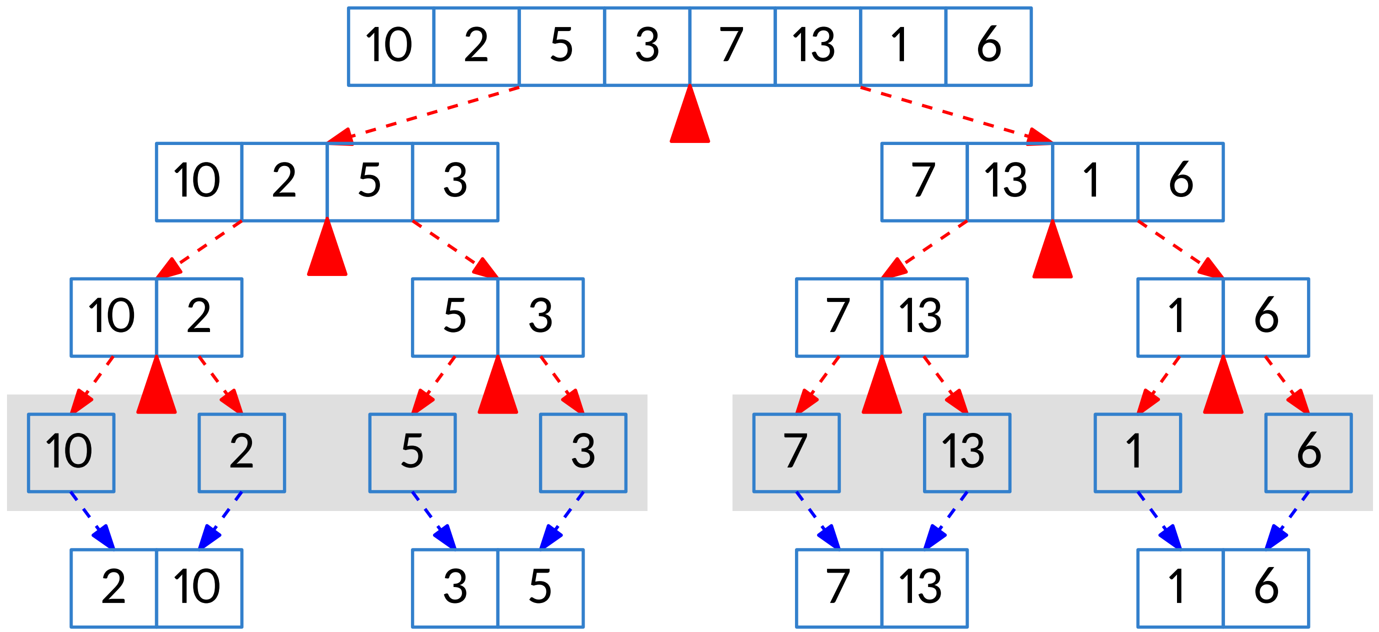
1. Split the list into two halves, (base case: $n \leq 1$)
2. Recursively mergesort each half, (**simpler instance**)
3. Merge them into a single sorted list.



Sort a list of numbers.

Developed by John von Neumann in 1945.

1. Split the list into two halves, (base case: $n \leq 1$)
2. Recursively mergesort each half, (**simpler instance**)
3. Merge them into a single sorted list.

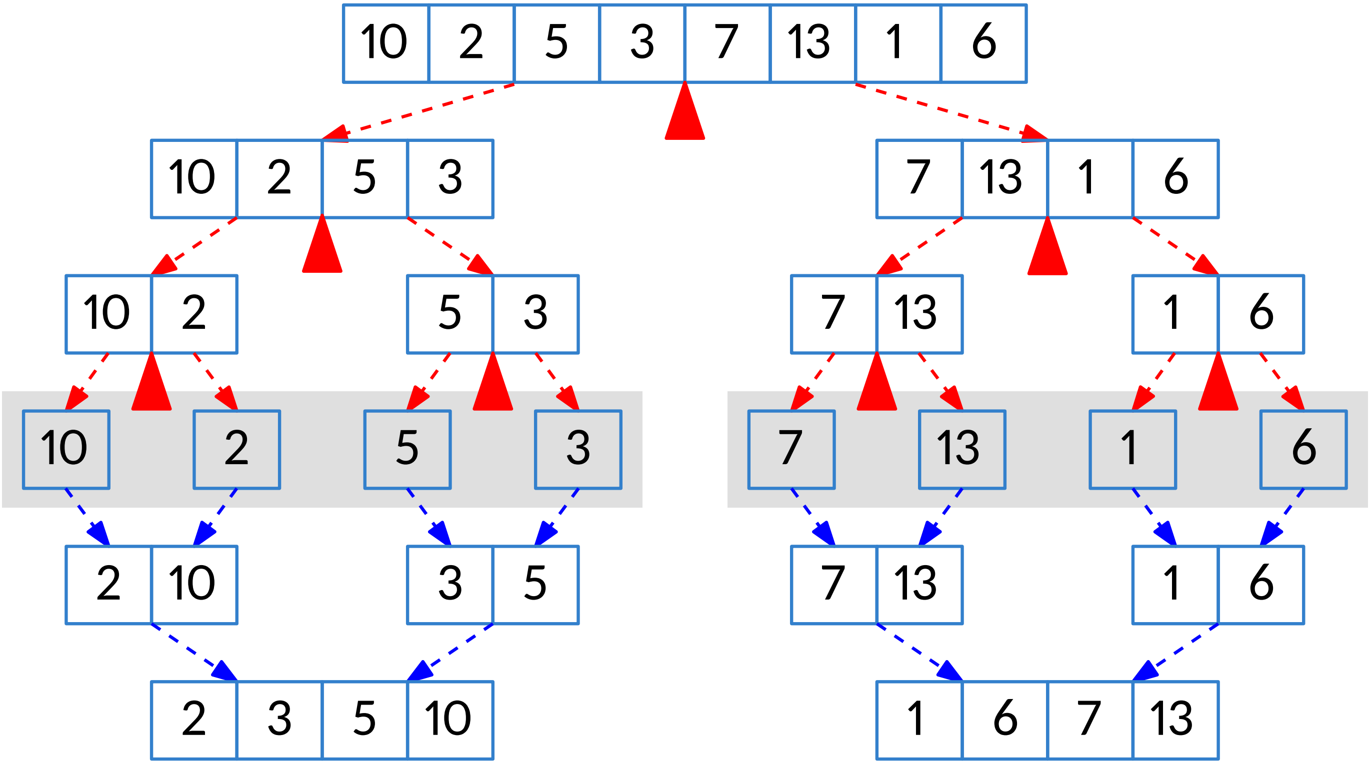


Sort a list of numbers.

1. Split the list into two halves,
2. Recursively mergesort each half, (**simpler instance**)
3. Merge them into a single sorted list.

Developed by John von Neumann in 1945.

(base case: $n \leq 1$)



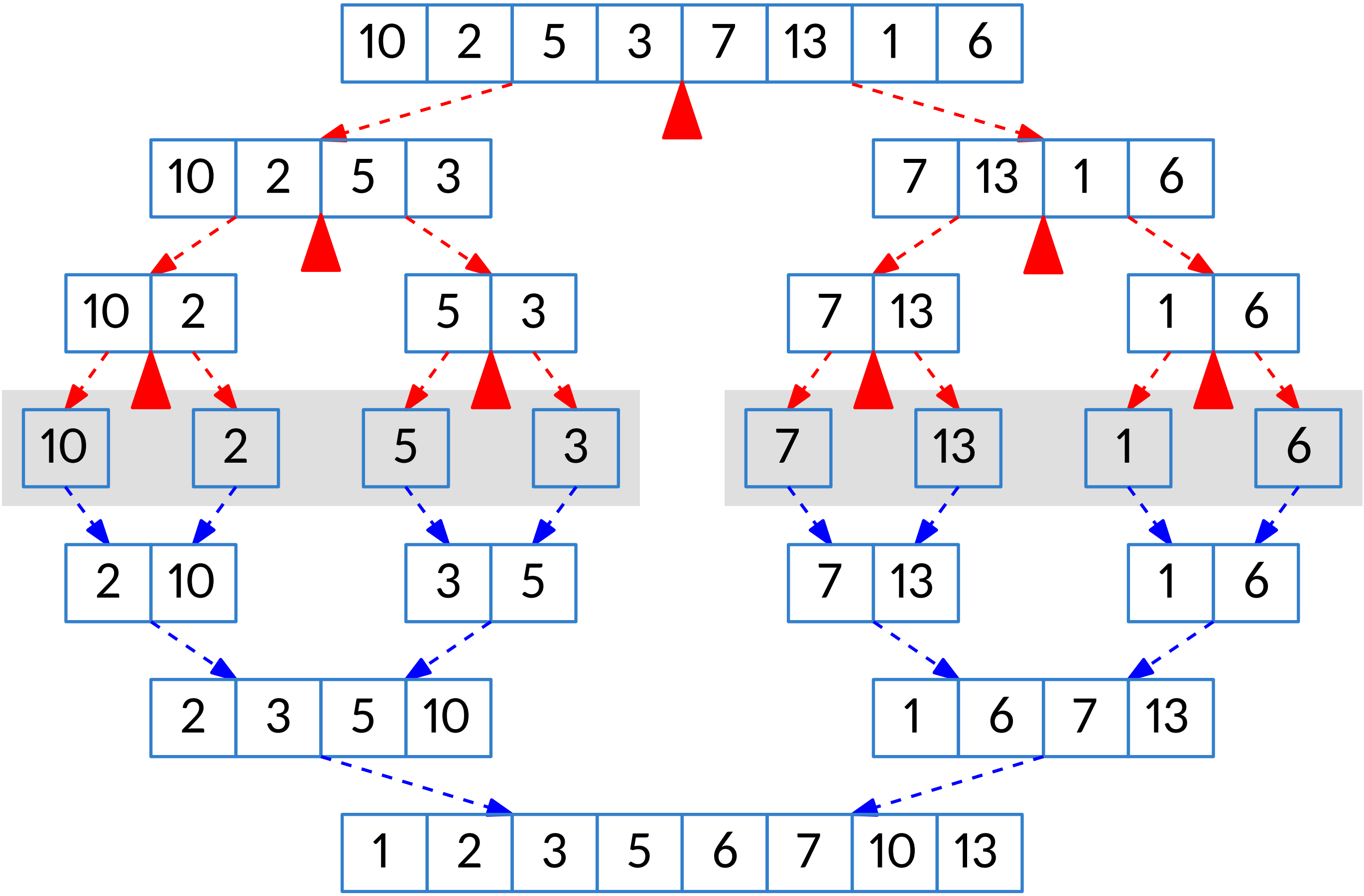
Sort a list of numbers.

Developed by John von Neumann in 1945.

1. Split the list into two halves,

(base case: $n \leq 1$)

2. Recursively mergesort each half, (**simpler instance**)
3. Merge them into a single sorted list.

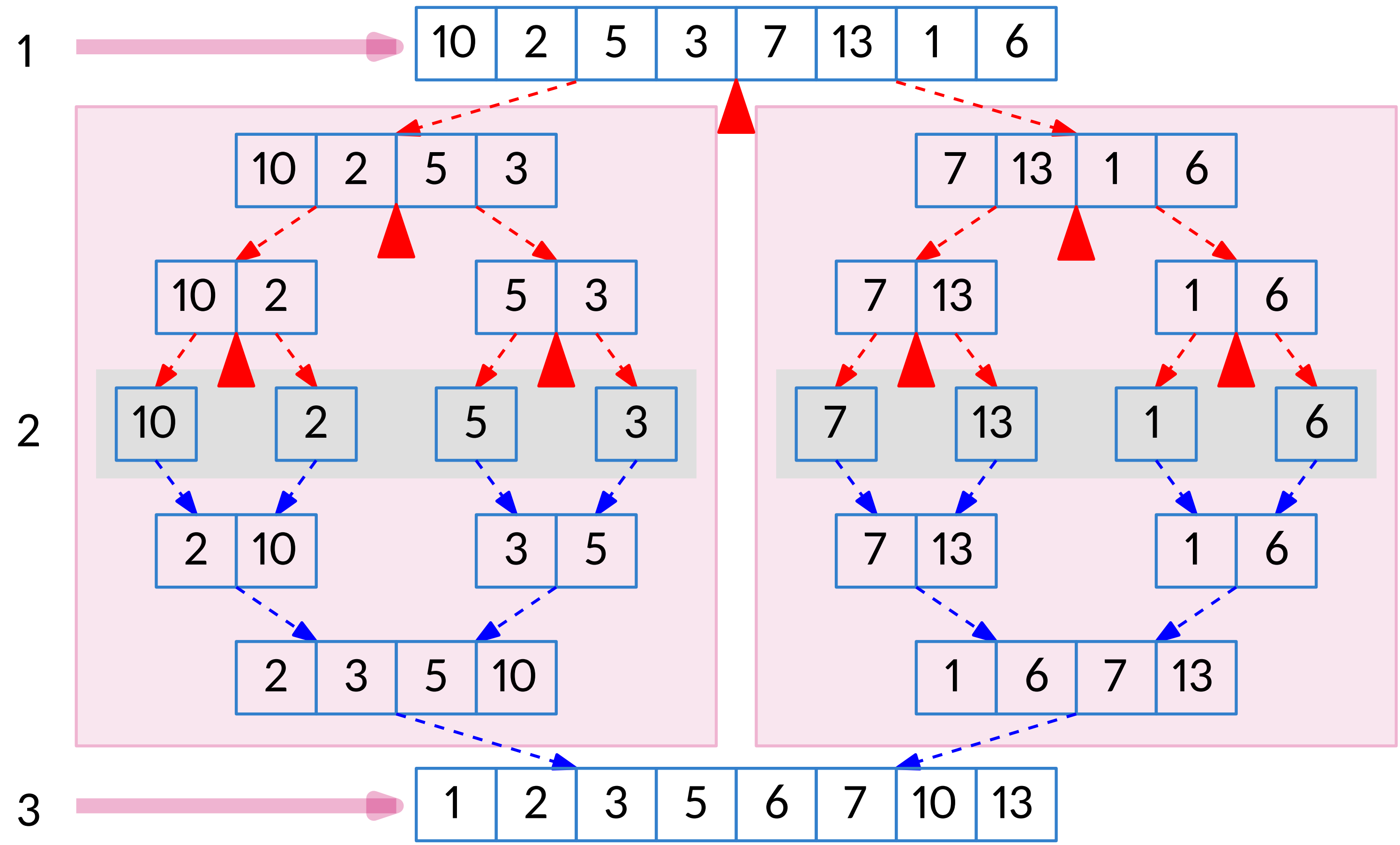


Mergesort

Sort a list of numbers.

Developed by John von Neumann in 1945.

1. Split the list into two halves, (base case: $n \leq 1$)
2. Recursively mergesort each half, (**simpler instance**)
3. Merge them into a single sorted list.



Mergesort

mergesort$(A[1:n])$
if $n > 1$ **then**
  $m \leftarrow \lfloor n/2 \rfloor$
  mergesort$(A[1:m])$
  mergesort$(A[m+1:n])$
  merge$(A[1:n], m)$

merge$(A[1:n], m)$
$i \leftarrow 1; j \leftarrow m+1$
for $k \leftarrow 1$ to n **do**
  if $j > n$ **then**
    $B[k] \leftarrow A[i]; i \leftarrow i+1$
  else if $i > m$ **then**
    $B[k] \leftarrow A[j]; j \leftarrow j+1$
  else if $A[i] < A[j]$ **then**
    $B[k] \leftarrow A[i]; i \leftarrow i+1$
  else
    $B[k] \leftarrow A[j]; j \leftarrow j+1$
for $k \leftarrow 1$ to n **do**
  $A[k] \leftarrow B[k]$

Correctness by induction.

merge merges $A[1:\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 : n]$ into a single sorted list $B[1:n]$ correctly, assuming the sublists are sorted.
Proof by induction on the number of merged elements.

mergesort correctly sorts any input lists.
(Proof by induction on n)
If $n \leq 1$, the algorithm does nothing.
Otherwise, by the induction hypothesis, the algorithm correctly sorts the two sublists of size $\leq \lceil n/2 \rceil$ (by two recursive calls to **mergesort**).
Then they are merged correctly into a single sorted list by **merge**.

Since **merge** can be done in linear time, the running time of **mergesort** is

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

Quicksort

Sort a list of numbers.

1. Choose a **pivot** element from the list.
2. Partition the list into three sublists:
   - sublist containing the elements smaller than the pivot,
   - sublist containing the pivot element,
   - sublist containing the elements larger than the pivot.
3. Recursively quicksort the first and last sublists (**simpler instances**).

Developed by Tony Hoare in 1959.

(base case: $n \leq 1$)

```
quicksort(A[1 : n])
if n ≤ 1 then
    return A
else
    Choose a pivot element A[p]
    r ← partition(A, p)
    quicksort(A[1 : r − 1])
    quicksort(A[r + 1 : n])
```

```
partition(A[1 : n], p)
swap A[p] ↔ A[n]
ℓ ← 0
for i ← 1 to n − 1 do
    if A[i] < A[n] then
        ℓ ← ℓ + 1
        swap A[ℓ] ↔ A[i]
swap A[n] ↔ A[ℓ + 1]
return ℓ + 1
```

Quicksort

Correctness by induction.

- **quicksort** correctly sorts, assuming **partition** works correct.

Analysis.

- **partition** runs in $O(n)$ time.
- Recurrence for the running time: $T(n) = T(r-1) + T(n-r) + O(n)$

r can be any value $1 \le r \le n$.

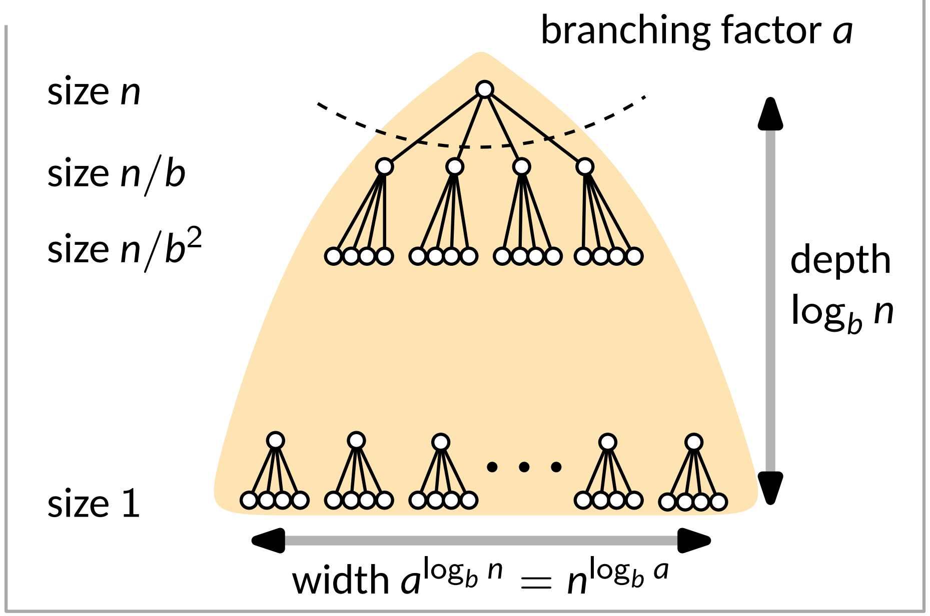
- **Best case.** the median element is always chosen for r:
  $T(n) \le 2T(n/2) + O(n) = O(n \log n)$.
- **Worst case.** $r = 1$ or $r = n$: $T(n) \le T(n-1) + O(n) = O(n^2)$.
  You may choose as r the median of three elements.
  But, in worst case, $T(n) \le T(1) + T(n-2) + O(n) = O(n^2)$.
- **Average case.** Usually r of rank between $n/10$ and $9n/10$.
  $T(n) = O(n \log n)$.

We will see later that the median element can be found in $O(n)$ time, though the process is complicated and the hidden constant in the running time is large.

$$T(n) = a \cdot T(\lceil n/b \rceil) + O(n^d)$$

Each problem of size n is divided into a subproblems of size n/b.

Recursion tree

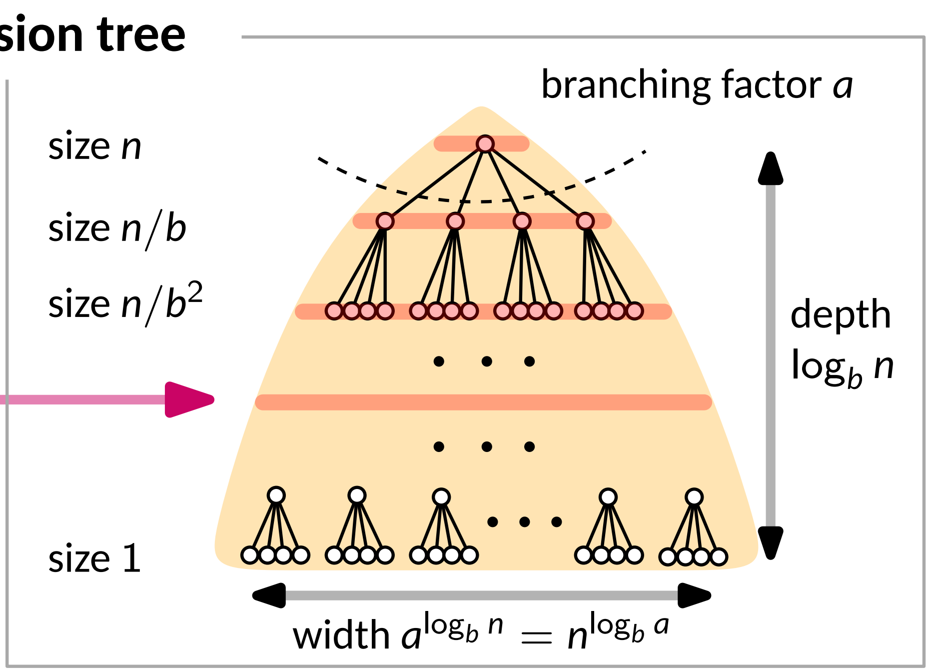


Recurrence Relations

$$T(n) = a \cdot T(\lceil n/b \rceil) + O(n^d)$$

Each problem of size n is divided into a subproblems of size n/b.

k-th level:
a^k subproblems,
each of size n/b^k.



- Base case: after $\log_b n$ levels, the problem size becomes 1.
- k-th level: a^k subproblems, each of size n/b^k.

$$a^k \cdot O((n/b^k)^d) = O(n^d)(a/b^d)^k.$$

- The total work done at all the levels:

$$\sum_{k=0}^{\log_b n} O(n^d)(a/b^d)^k.$$

Recurrence Relations

$$T(n) = a \cdot T(\lceil n/b \rceil) + O(n^d) = \boxed{\sum_{k=0}^{\log_b n} O(n^d) \cdot (a/b^d)^k}$$

(level-by-level series)

Master theorem:

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

As k goes from 0 to $\log_b n$, they form a geometric series with ratio a/b^d.

1. $a/b^d < 1$, that is, $d > \log_b a$.
   The series is decreasing, and its sum is just given by the first term, $O(n^d)$.
2. $a/b^d = 1$, that is, $d = \log_b a$.
   All $O(\log n)$ terms of the series are equal to $O(n^d)$.
3. $a/b^d > 1$, that is, $d < \log_b a$.
   The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$:
   $n^d(\frac{a}{b^d})^{\log_b n} = n^d(\frac{a^{\log_b n}}{(b^{\log_b n})^d}) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}$.

Recurrence Relations

$$T(n) = a \cdot T(\lceil n/b \rceil) + O(n^d) = \boxed{\sum_{k=0}^{\log_b n} O(n^d) \cdot (a/b^d)^k}$$

(level-by-level series)

Master theorem:

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

Binary search. $T(n) = T(n/2) + O(1)$. Since $a = 1, b = 2, d = 0, d = \log_b a$ and $T(n) = O(n^d \log n) = O(\log n)$.

Mergesort. $T(n) = 2T(n/2) + O(n)$. Since $a = 2, b = 2, d = 1, d = \log_b a$ and $T(n) = O(n^d \log n) = O(n \log n)$.

Quicksort. If the pivot always lands in the middle third of the sorted array,

$$T(n) \leq T(n/3) + T(2n/3) + O(n). \qquad T(n) = O(n \log n)$$

Try a few levels of the recursion tree!

- the sum of problem sizes on any level is at most n.
- the tree depth is $\log_{3/2} n = O(\log n)$.

Integer Multiplication

For two **complex numbers** $x = a + bi, y = c + di$,

$$xy = (a + bi)(c + di) = ac - bd + (bc + ad)i$$

Integer Multiplication

For two **complex numbers** $x = a + bi, y = c + di$,

$$xy = (a+bi)(c+di) = ac - bd + (bc + ad)i$$
$$= ac - bd + ((a+b)(c+d) - ac - bd)i$$

For two n**-bit numbers**? (n is a **large** number)

x	$n-1$		0
	x_L	x_R	

y	$n-1$		0
	y_L	y_R	

$$x = 2^{n/2} \cdot x_L + x_R \qquad y = 2^{n/2} \cdot y_L + y_R$$

$$\begin{aligned} xy &= (2^{n/2} \cdot x_L + x_R)(2^{n/2} \cdot y_L + y_R) \\ &= 2^n \cdot x_L y_L + 2^{n/2} \cdot (x_L y_R + x_R y_L) + x_R y_R. \end{aligned}$$

Requires 4 $n/2$-bit multiplications, handled by four recursive calls. Therefore

$$T(n) = 4 \cdot T(n/2) + O(n) = O(n^2).$$

But $x_L y_R + x_R y_L$ can be computed from $x_L y_L$ and $x_R y_R$ using one more recursive multiplication $(x_L + x_R)(y_L + y_R)$. (Karatsuba 1960)

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

Integer Multiplication

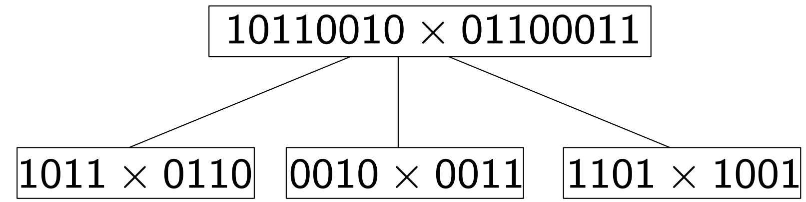
multiply(x, y)
Input: Two n-bit integers x and y
Output: Their product

if $n = 1$ **then**
    return xy
$x_L, x_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of x
$y_L, y_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of y
$P_1 =$**multiply**(x_L, y_L)
$P_2 =$**multiply**(x_R, y_R)
$P_3 =$**multiply**$(x_L + x_R, y_L + y_R)$
return $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$

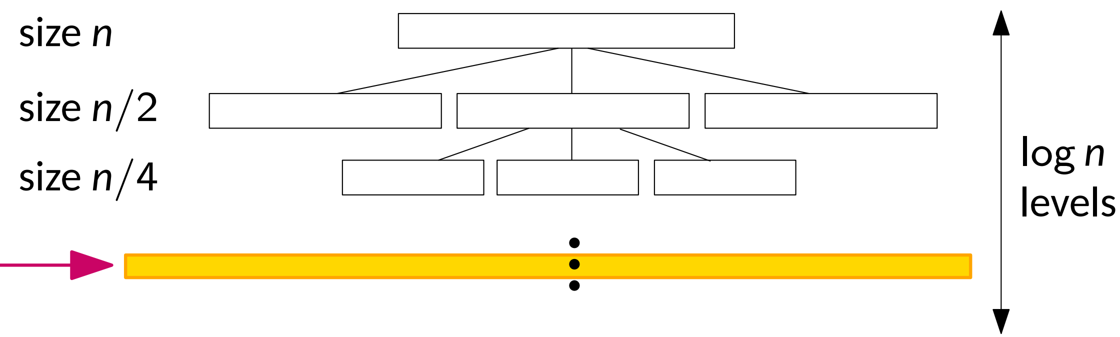
A constant factor improvement, occuring at every level of recursion. Therefore

$$T(n) = 3 \cdot T(n/2) + O(n) = O(?).$$

Integer Multiplication



k-th level:
3^k subproblems
each of size $n/2^k$.



$$
\begin{aligned}
\sum_{k=0}^{\log_2 n} 3^k \cdot O(n/2^k) &= \sum_{k=0}^{\log_2 n} (3/2)^k \cdot O(n) \\
&= O(n) + \cdots + O(3^{\log_2 n}) \\
&= O(n) + \cdots + O(n^{\log_2 3}) \\
&= O(n)\frac{(3/2)^{\log_2 n+1} - 1}{3/2 - 1} \\
&= O(n^{\log_2 3}).
\end{aligned}
$$

$$T(n) = 3 \cdot T(n/2) + O(n) = O(n^{\log_2 3}).$$

Integer Multiplication - FFT-based ones

Splitting two n-bit numbers into k pieces, each with n/k bits, and then compute the product of the numbers using only $2k-1$ recursive multiplications?

For any fixed k, Toom-Cook algorithm runs in $O(n^{1+1/(\lg k)})$ time.

Taking this divide-and-conquer strategy further $\rightarrow$ the **Fast Fourier transform**

by Gauss

The fast Fourier transform (FFT) is a very efficient algorithm for calculating the discrete Fourier Transform (DFT) of a sequence of numbers.

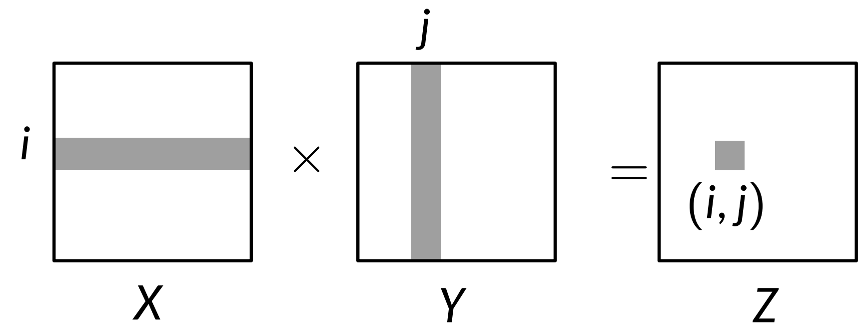
In 1971, $O(n \log n \log \log n)$-time algorithm by Schönhage and Strassen.
The first FFT-based integer multiplication algorithm. Fastest in practice.

In 2019, $O(n \log n)$-time algorithm by Harvey and van der Hoeven.

Matrix Multiplication

For two $n \times n$ matrices X and Y, $Z = XY$ has (i, j)-th entry

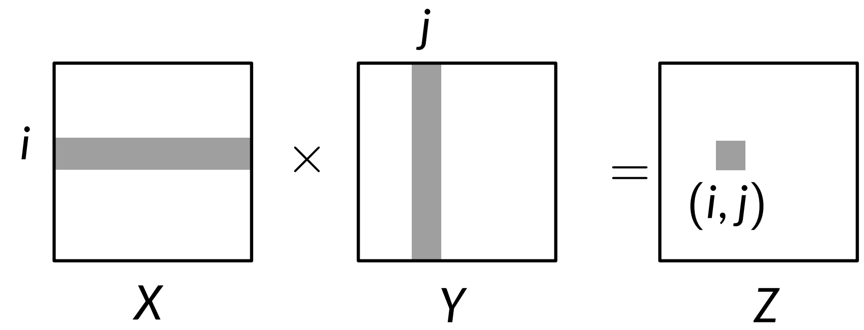
$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$



Matrix Multiplication

For two $n \times n$ matrices X and Y, $Z = XY$ has (i, j)-th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$



Brute force. n^2 entries, each taking $O(n)$ time. In total, $O(n^3)$ time.

Divide and Conquer!

$$X = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \qquad Y = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right]$$

$$XY = \left[\begin{array}{c|c} AE + BG & AF + BH \\ \hline CE + DG & CF + DH \end{array}\right]$$

There are 8 size-$n/2$ products and a few $O(n^2)$-time additions.

$$T(n) \leqslant 8 \cdot T(n/2) + O(n^2),$$

which is $O(n^3)$.

Divide and Conquer. by Volker Strassen.

$$XY = \left[\begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{array}\right] = \left[\begin{array}{c|c} AE + BG & AF + BH \\ \hline CE + DG & CF + DH \end{array}\right]$$

where

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

There are 7 size-$n/2$ products and eighteen $O(n^2)$-time additions.

$$T(n) \leqslant 7 \cdot T(n/2) + O(n^2).$$

By master theorem, $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$.

Linear-Time Selection

Given a list S of numbers and an integer k, find the k-th smallest element in S.

Reduction:

1. choose a pivot v from S,
2. split S into 3 sublists with respect to v, and
3. recurse on one of them.

$v = 5$

$$S = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle \qquad \begin{aligned} S_L &= \langle 2, 4, 1 \rangle \\ S_v &= \langle 5, 5 \rangle \\ S_R &= \langle 36, 21, 8, 13, 11, 20 \rangle \end{aligned}$$

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \leqslant |S_L| \\ v & \text{if } |S_L| < k \leqslant |S_L| + |S_v| \\ \text{selection}(S_R, k - (|S_L| + |S_v|)) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

If the smallest element is chosen for v repeatedly, $T(n) = O(n^2)$ as

$$T(n) = T(n-1) + O(n).$$

To avoid this worst-case behavior, magically choose a **good** pivot **quickly**!

Linear-Time Selection

Choose v **randomly**!

$$\text{If we are } = \begin{cases} \text{lucky } (S_L \text{ and } S_R \text{ are balanced in their sizes}) & \rightarrow \Theta(n) \text{ time,} \\ \text{unlucky (Otherwise)} & \rightarrow \Theta(n^2) \text{ time.} \end{cases}$$

S_L and S_R are **balanced** if the have size at most $3/4$ of that of S.

50% chance of being good!

Average #. coin flips to get a head: $A = (1-p)(1+A) + p \cdot 1$, giving $A = 1/p$.

New algorithm: repeat choosing v randomly until it is good, and then recurse.

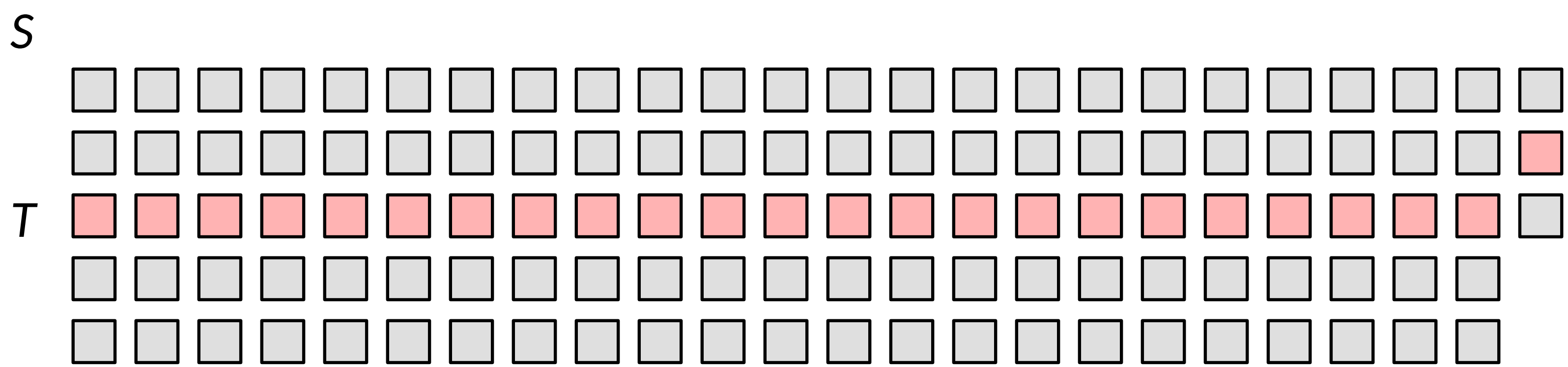
For each chosen v, it spends $O(n)$ time to check if v is good or not. The expected running time $T(n) \leqslant T(3n/4) + O(c \cdot n)$.

c: average number of consecutive splits (or choices of v) to find a good split.

Clearly $c = 2$. So we can conclude that $T(n) = O(n)$.

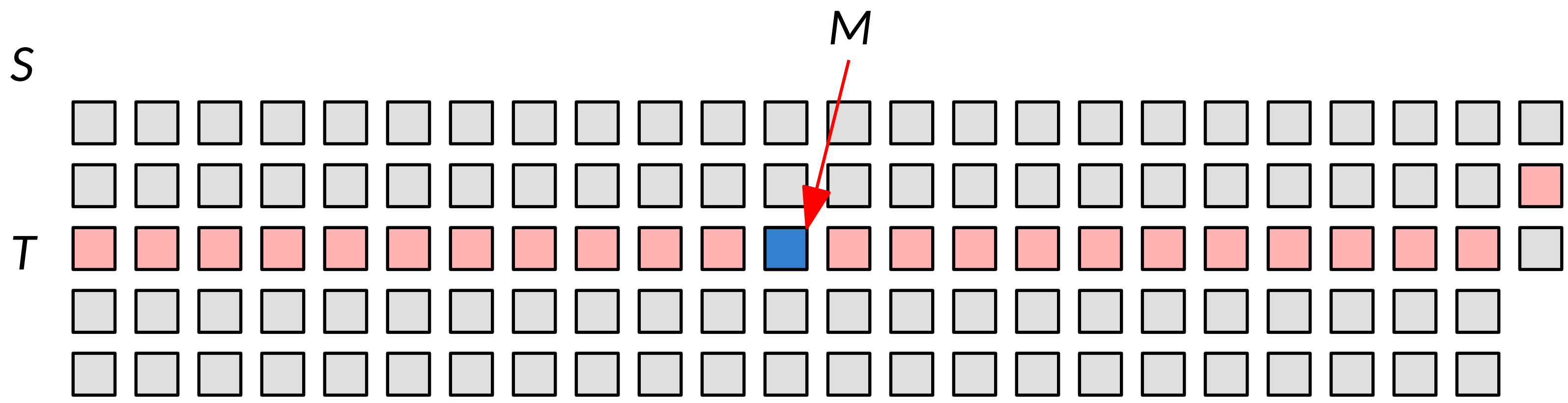
Given a list S of numbers and an integer k, find the k-th smallest element in S.

1. Divide the input array S into $\lceil n/5 \rceil$ blocks, each containing 5 elements, except possibly the last one.
2. Find the median of each block and collect them into a new array T.



Given a list S of numbers and an integer k, find the k-th smallest element in S.

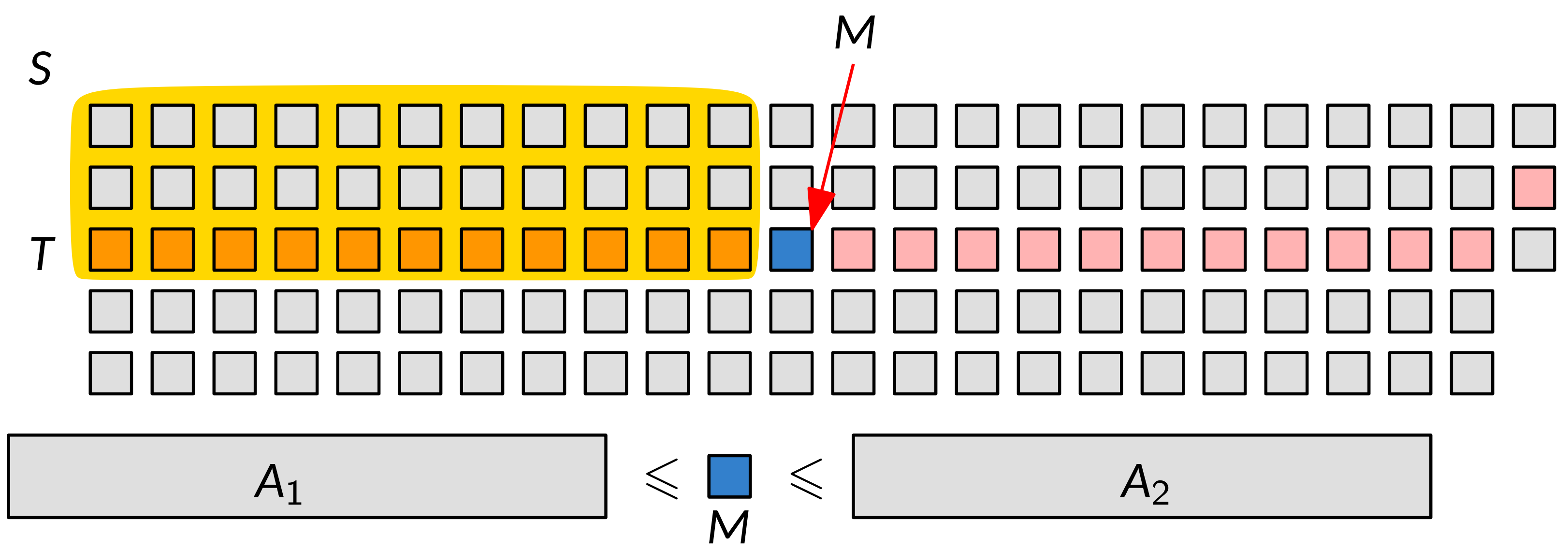
1. Divide the input array S into $\lceil n/5 \rceil$ blocks, each containing 5 elements, except possibly the last one.
2. Find the median of each block and collect them into a new array T.
3. Recurse on T and find the median M of T.



Linear-Time Selection

Given a list S of numbers and an integer k, find the k-th smallest element in S.

1. Divide the input array S into $\lceil n/5 \rceil$ blocks, each containing 5 elements, except possibly the last one.
2. Find the median of each block and collect them into a new array T.
3. Recurse on T and find the median M of T.
4. Partition S into two subarrays using M.
5. If M is the kth smallest element of S, we are done.
   Otherwise, we recursively search one of the two subarrays.



Linear-Time Selection

Given a list S of numbers and an integer k, find the k-th smallest element in S.

The key insight is that

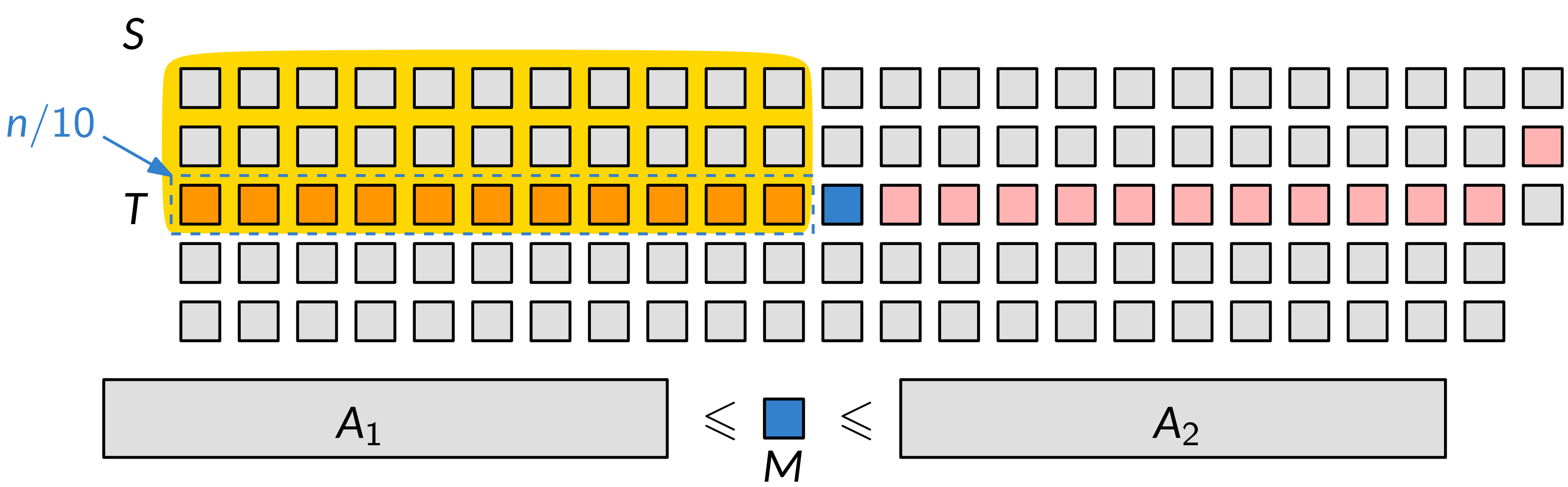
(a) M is larger than $\lceil \lceil n/5 \rceil /2 \rceil - 1 \approx n/10$ medians.

(b) Each such median is larger than **two other elements** in its block.

$\Longrightarrow$ M is larger than at least $3n/10$ elements in S.

(c) Similarly, M is smaller than at least $3n/10$ elements in S.

(d) The recursive call is on an array of size $7n/10$.



Given a list S of numbers and an integer k, find the k-th smallest element in S.

The key insight is that

(a) M is larger than $\lceil \lceil n/5 \rceil /2 \rceil - 1 \approx n/10$ medians.

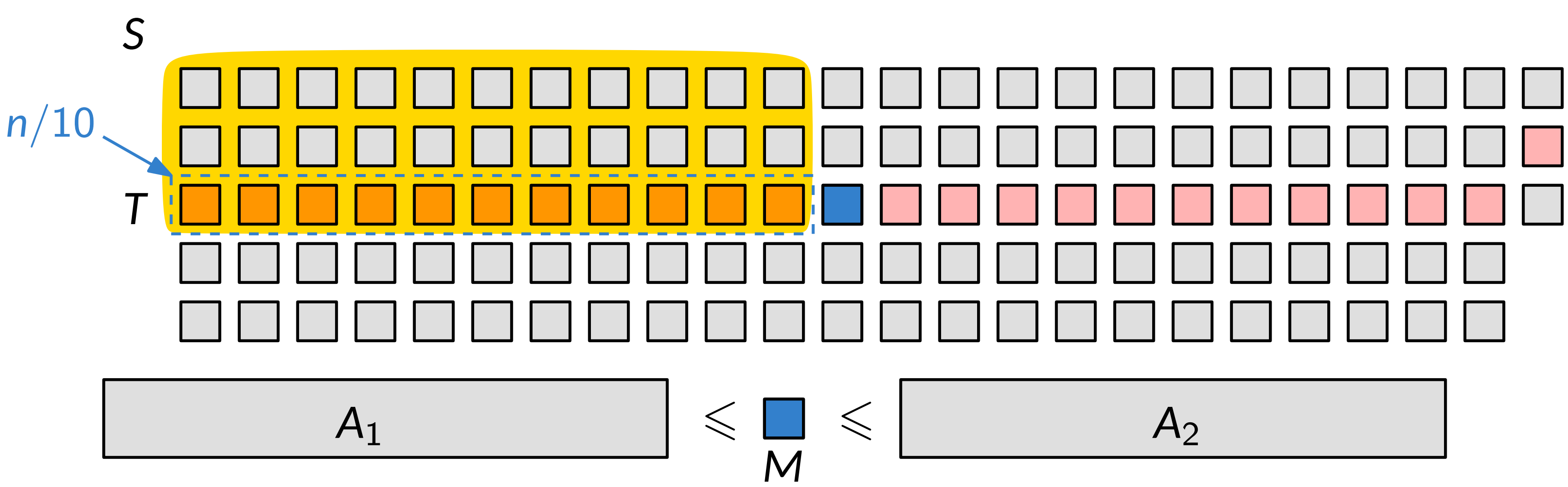
(b) Each such median is larger than **two other elements** in its block.

$\Longrightarrow$ M is larger than at least $3n/10$ elements in S.

(c) Similarly, M is smaller than at least $3n/10$ elements in S.

(d) The recursive call is on an array of size $7n/10$.

$$T(n) \leqslant O(n) + T(n/5) + T(7n/10)$$



Linear-Time Selection

Given a list S of numbers and an integer k, find the k-th smallest element in S.

$$
\begin{aligned}
T(n) &\leqslant O(n) + T(n/5) + T(7n/10) \\
&\leq O(n) \quad + O(n/5) + T(n/25) + T(7n/50) \\
&\qquad\qquad + O(7n/10) + T(7n/50) + T(49n/100)
\end{aligned}
$$

Linear-Time Selection

Given a list S of numbers and an integer k, find the k-th smallest element in S.

$$
\begin{aligned}
T(n) &\leqslant O(n) + T(n/5) + T(7n/10) \\
&\leq O(n) \quad + O(n/5) + T(n/25) + T(7n/50) \\
&\qquad\qquad + O(7n/10) + T(7n/50) + T(49n/100)
\end{aligned}
$$

$O(9n/10)$ $\quad$ $O(81n/100)$ $\quad \bullet \quad \bullet \quad \bullet$

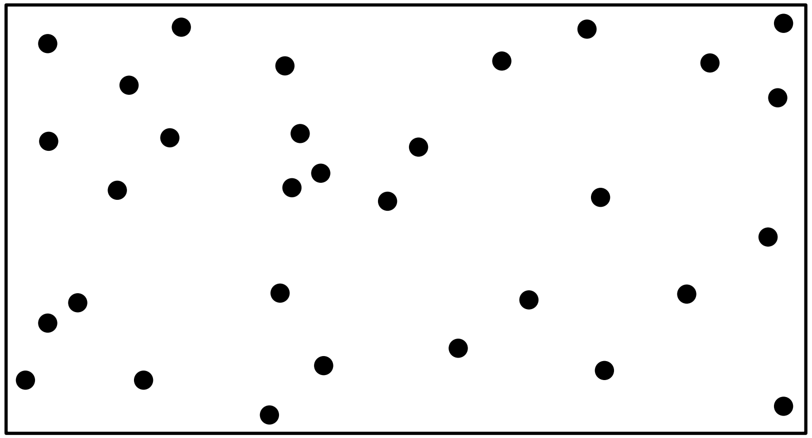
$$T(n) \leqslant O(n) + T(n/5) + T(7n/10) = O(n)$$

Why 5 for the block size?
5 is the smallest odd block size achieving exponential decay in the analysis.
If 3 is used, $T(n) \leq T(n/3) + T(2n/3) + O(n)$, implying $T(n) \leq O(n \log n)$.

Closest Pair of Points

Given n points in the plane, find a pair with smallest Euclidean distance.

Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

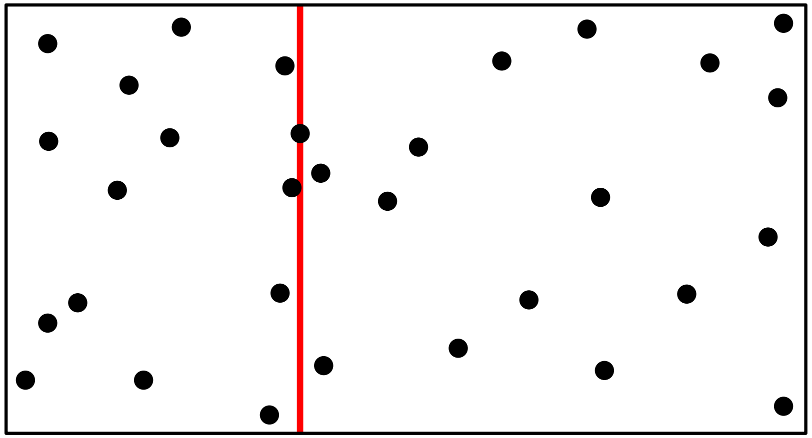


Closest Pair of Points

Given n points in the plane, find a pair with smallest Euclidean distance.

Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

Divide and Conquer. Divide the point set P into two equal-sized subsets P^L and P^R along x-median x_{mid} and solve the problem recursively.

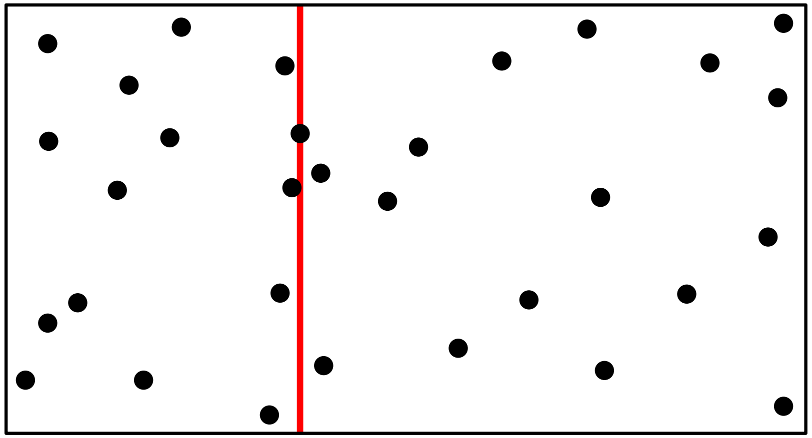


Closest Pair of Points

Given n points in the plane, find a pair with smallest Euclidean distance.

Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

Divide and Conquer. Divide the point set P into two equal-sized subsets P^L and P^R along x-median x_{mid} and solve the problem recursively.



Preprocessing. Sort the points in P by x-coordinates (sorted list P_x), and by y-coordinates (sorted list P_y).

In the recursion,

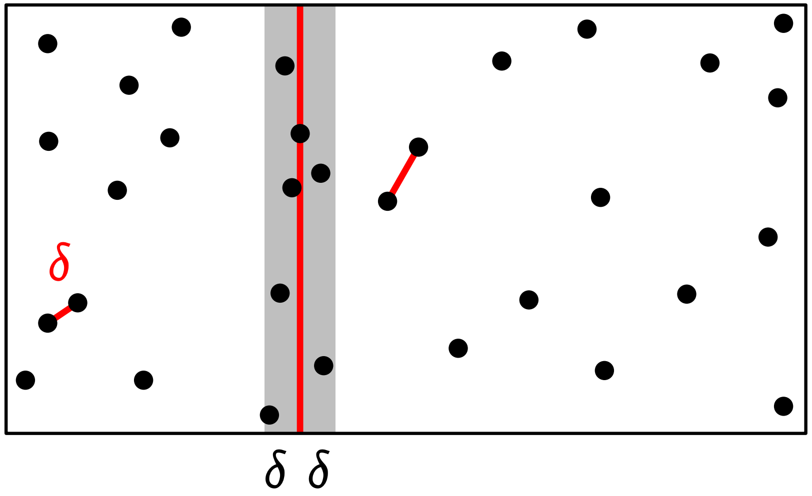
- Two sorted lists P_x^L and P_x^R can be computed in $O(1)$ time from P_x.
- P_y^L and P_y^R can be constructed in $O(n)$ time by scanning P_y once.

Closest Pair of Points

Given n points in the plane, find a pair with smallest Euclidean distance.

Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

Divide and Conquer. Divide the point set P into two equal-sized subsets P^L and P^R along x-median x_{mid} and solve the problem recursively.



Preprocessing. Sort the points in P by x-coordinates (sorted list P_x), and by y-coordinates (sorted list P_y).

In the recursion,

- Two sorted lists P^L_x and P^R_x can be computed in $O(1)$ time from P_x.
- P^L_y and P^R_y can be constructed in $O(n)$ time by scanning P_y once.

The minimum δ of the two smallest distances, returned from each subproblem.

Closest Pair of Points

The sorted list of points in the left (and right) gray strip by y-coordinates can be obtained from P_y^L (and P_y^R) in $O(n)$ time.

Consider a grid over the gray strips with cell dimension $\delta/2 \times \delta/2$.

Among the points in the gray strip,
let s_i denote the point with the i-th smallest y-coordinate.

Claim. If $|i - j| \geqslant 12$, then s_i and s_j are at least δ apart.
- No two points lie in same $\delta/2 \times \delta/2$ box.
- Two points at least 2 rows apart are at least $2(\delta/2)$ apart.

$T(n) \leqslant 2T(n/2) + O(n)$, implying $T(n) = O(n \log n)$.