การแบ่งแยกเพื่อเอาชนะ
Divide and Conquer

การอบรมคอมพิวเตอร์โอลิมปิก สอวน. ค่ายที่ 2 ปีการศึกษา 2563

อ.ดร.ฐาปนา บุญชู

สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี

มหาวิทยาลัยธรรมศาสตร์

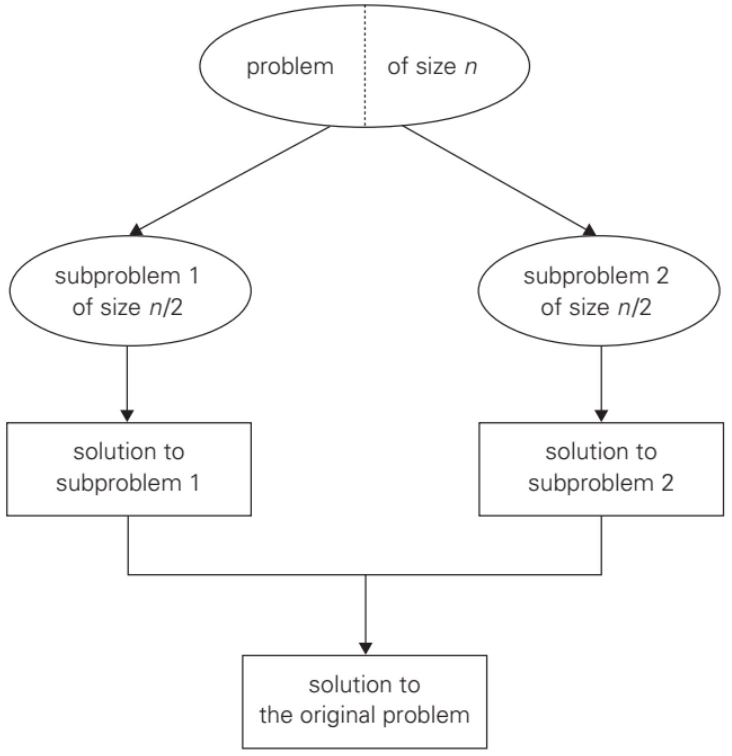
Divide and Conquer

- Divide-and-conquer is probably the best-known general algorithm design technique.
- Divide-and-conquer algorithms work according to the following general plan:

DIVIDE — **Divide** the problem into a number of subproblems that are smaller instances of the same problem.

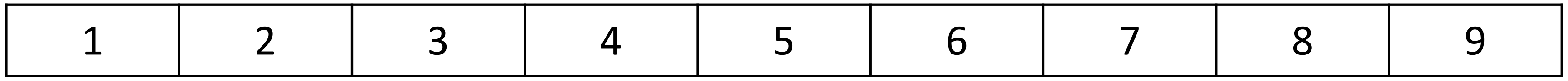
CONQUER — **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

COMBINE — **Combine** the solutions to the subproblems into the solution for the original problem.

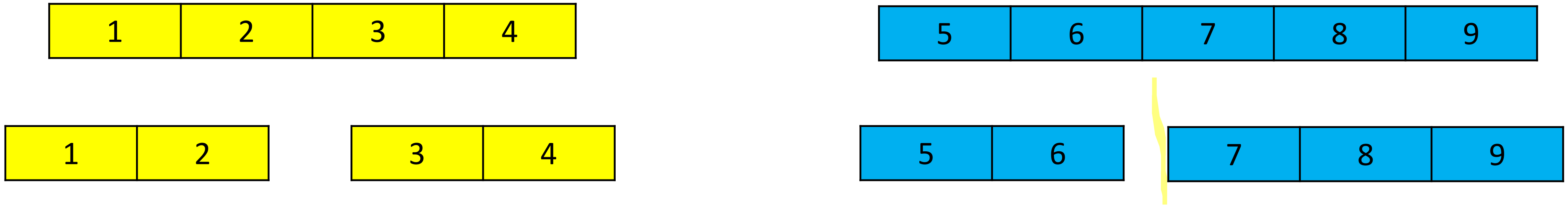

problem of size n
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
solution to subproblem 1
solution to subproblem 2
solution to the original problem

Divide and Conquer

- Let us consider the problem of computing the sum of n numbers.



If n > 1, we can divide the problem into two instances: first $\lfloor n/2 \rfloor$ numbers and the remaining $\lceil n/2 \rceil$ numbers



... Do it recursively...

Divide and Conquer

- In the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size $n/2$.
- More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. $(constants\ a >= 1, b > 0)$
- Assume that n is a power of b, i.e. $n = b^p$
- Thus, the running time $T(n)$ can be computed as:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \text{ ,} \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

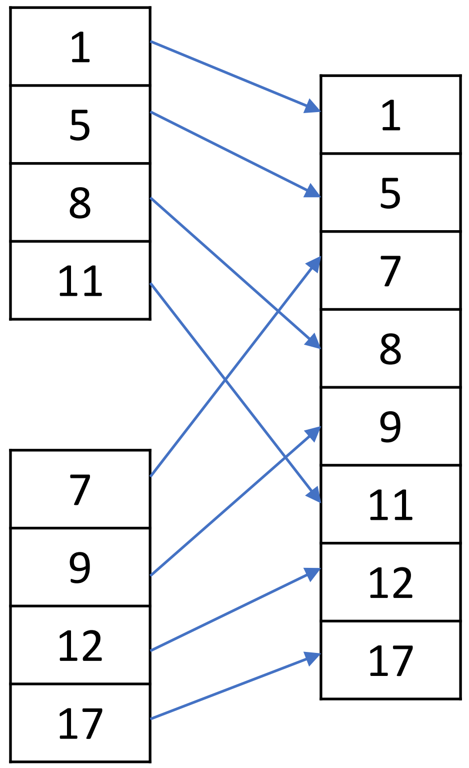
- If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem

Merge Sort

- We merge by calling an auxiliary procedure $\boldsymbol{MERGE(A, p, q, r)}$, where A is an array and p, q, and r are indices into the array such that $p, q < r$.
- The procedure assumes that the subarrays $A[p \dots q]$ and $A[q + 1 \dots r]$ are in sorted order.
- It merges them to form a single sorted subarray that replaces the current subarray $A[p \dots r]$.

Merge Procedure

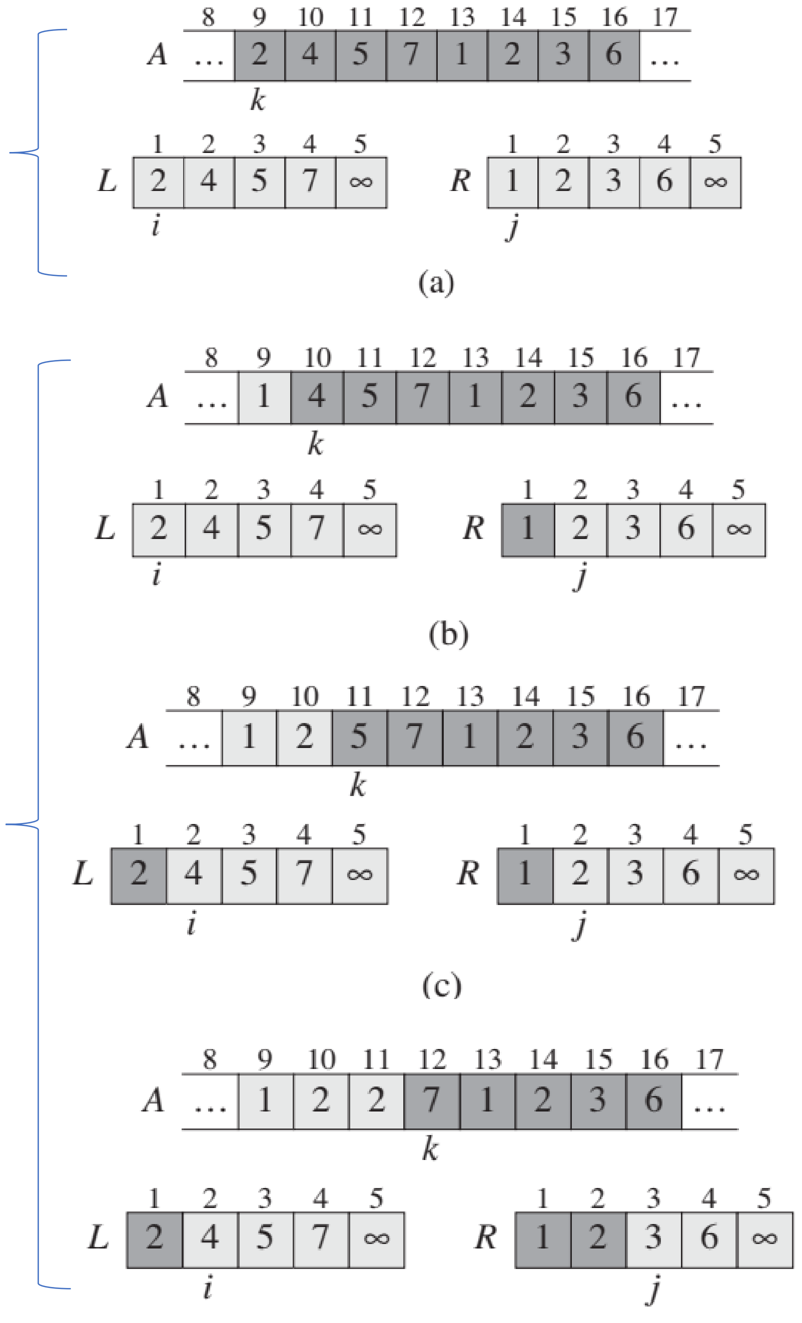
Both are in sorted order.



Merge Procedure

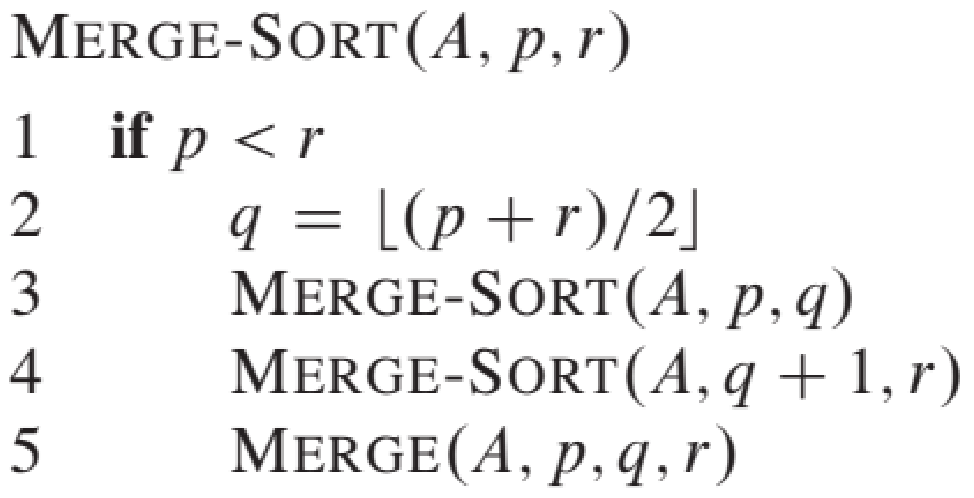
```
MERGE($A, p, q, r$)
1   $n_1 = q - p + 1$
2   $n_2 = r - q$
3   let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
4   for $i = 1$ to $n_1$
5       $L[i] = A[p + i - 1]$
6   for $j = 1$ to $n_2$
7       $R[j] = A[q + j]$
8   $L[n_1 + 1] = \infty$ sentinel card
9   $R[n_2 + 1] = \infty$ sentinel card
10  $i = 1$
11  $j = 1$
12  for $k = p$ to $r$
13      if $L[i] \le R[j]$
14          $A[k] = L[i]$
15          $i = i + 1$
16      else $A[k] = R[j]$
17          $j = j + 1$
```

It takes $\Theta(n_1 + n_2) = \Theta(n)$ time.



(a) (b) (c) (d)

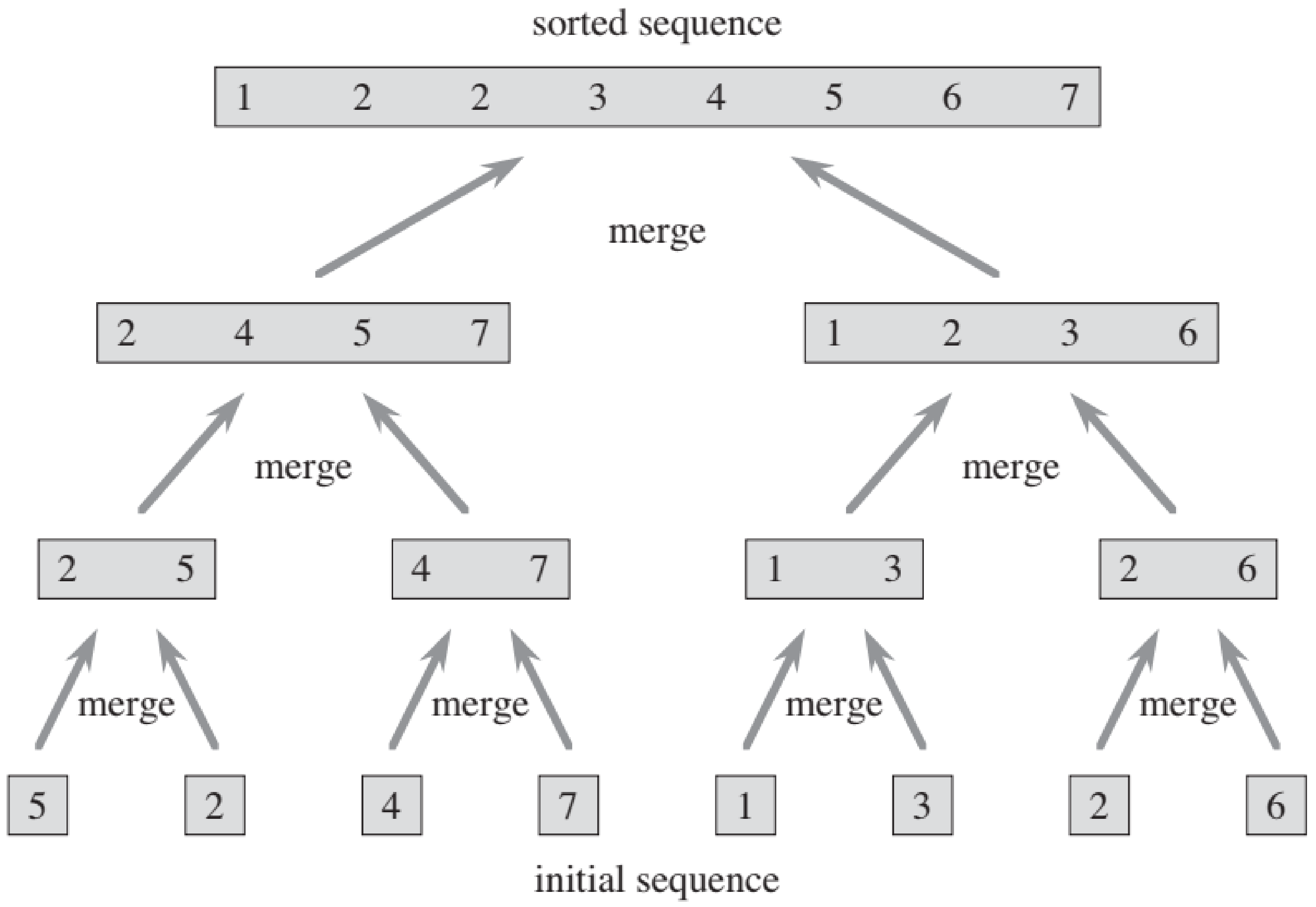
Merge Sort Procedure



```
MERGE-SORT(A, p, r)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

If $p < r$ the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p \dots r]$ into two subarrays: $A[p \dots q]$, containing $\left\lceil \frac{n}{2} \right\rceil$ elements, and $A[q + 1 \dots r]$, containing $\left\lfloor \frac{n}{2} \right\rfloor$elements.

Merge Sort Example



The operation of merge sort on the array $A = \langle 5,2,4,7,1,3,2,6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Merge Sort

Divide: The divide step just computes the middle of the subarray, which takes constant time. $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n = 2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$. $C(n) = \Theta(n)$.

- The recursive will stop when reaching the base case. Here, when the sequence to be sorted has length 1.
- The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step.

Analysis of Merge Sort

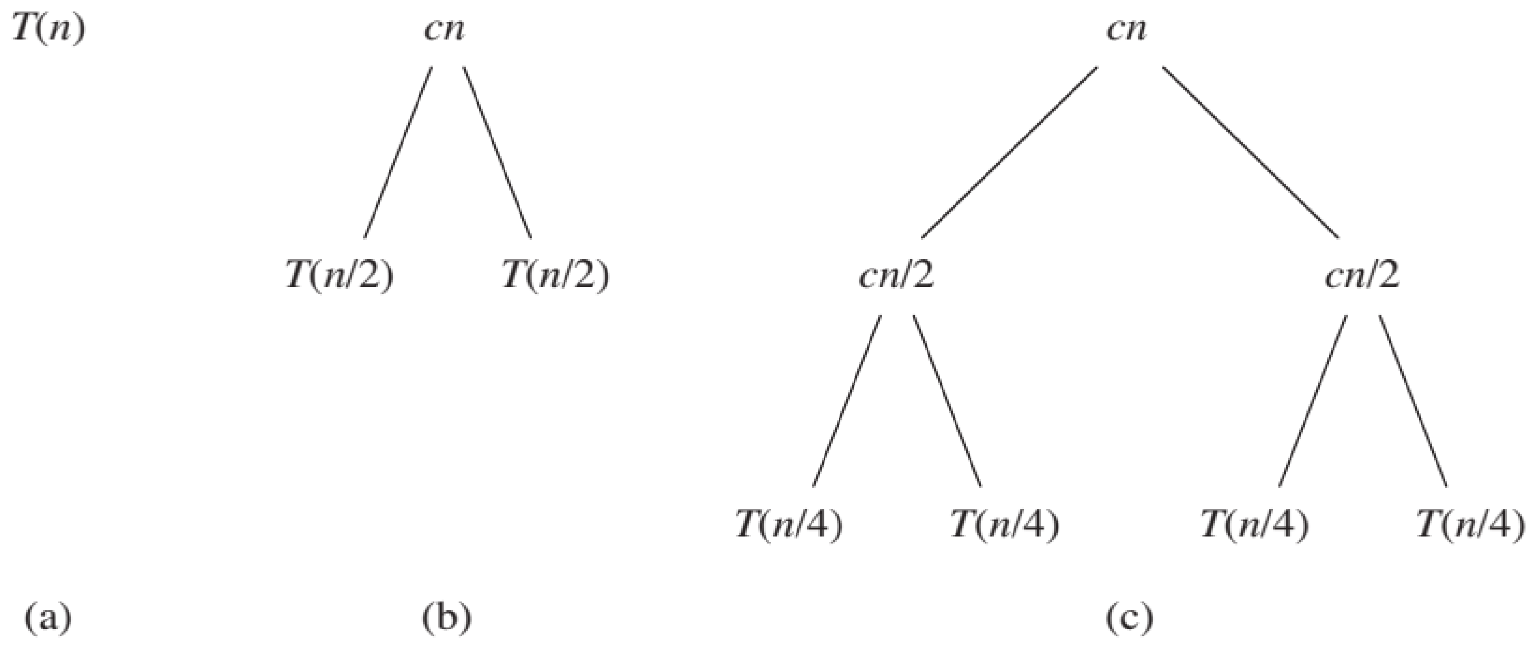
- let $T(n)$ be the running time on a problem of size n. If the problem size is small enough, say $n < c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$.

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n), & \text{if n} > 1 \\ \Theta(1), & \text{if } n = 1 \end{cases}$$

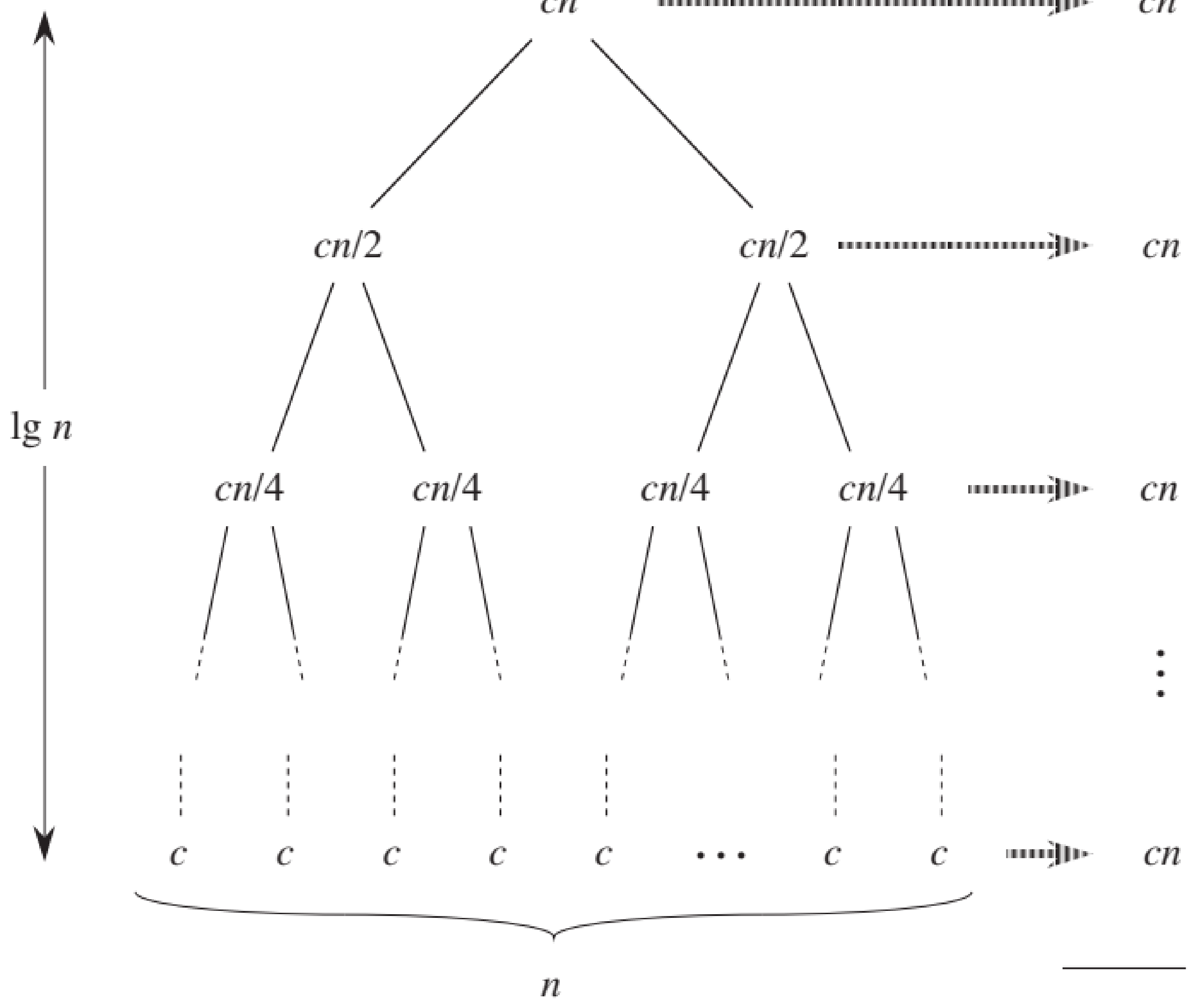
- Let $f(n) = D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$

Recurrence Tree

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + c\mathrm{n}, \text{if n} > 1 \\ c, \qquad\qquad\qquad \text{if } n = 1 \end{cases}$$

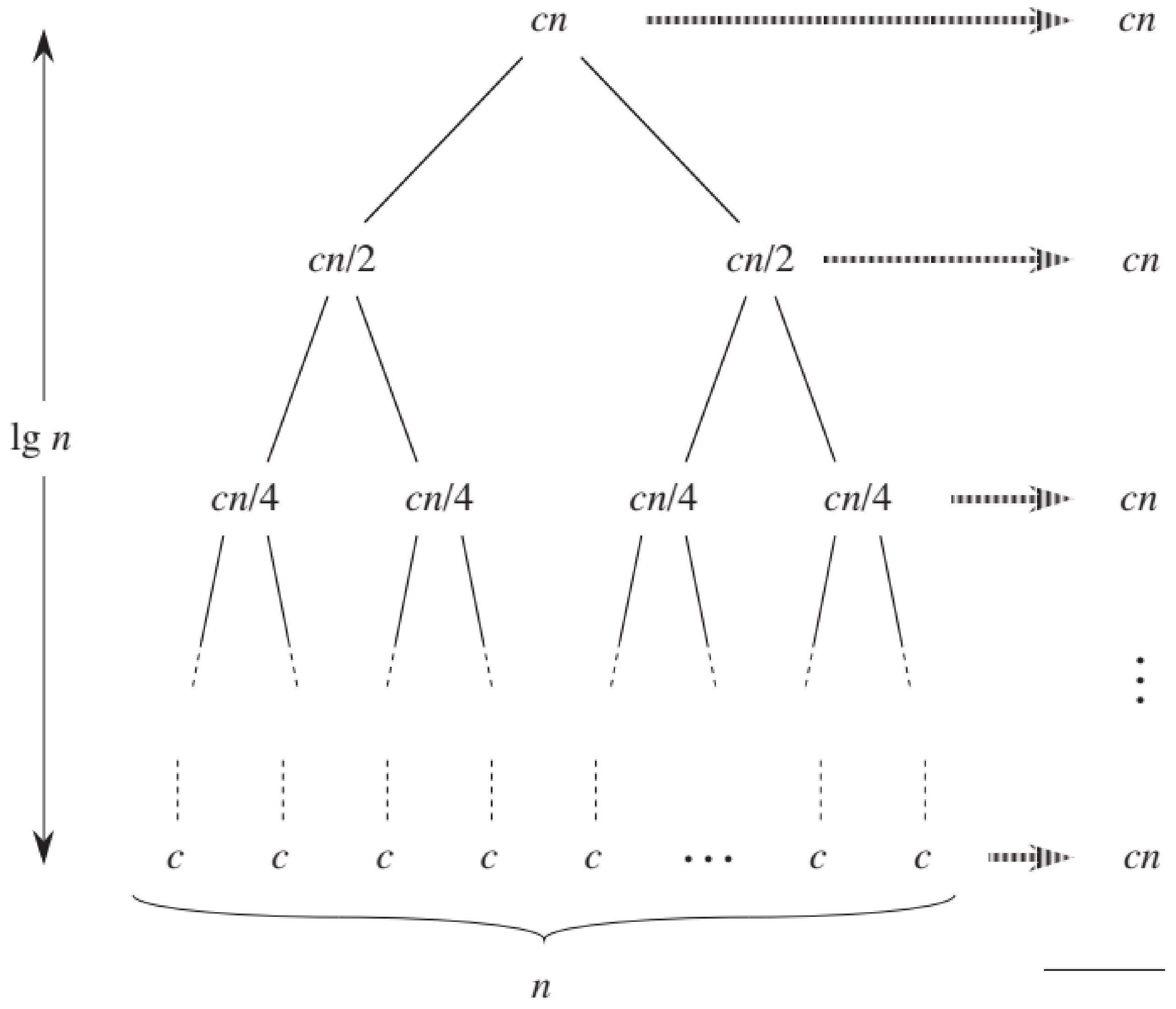


Recurrence Tree



Height (h)	#nodes at current level
1	1
2	2
3	4
4	8
.	.
$\log_2 n$	2^{h-1}

Recurrence Tree



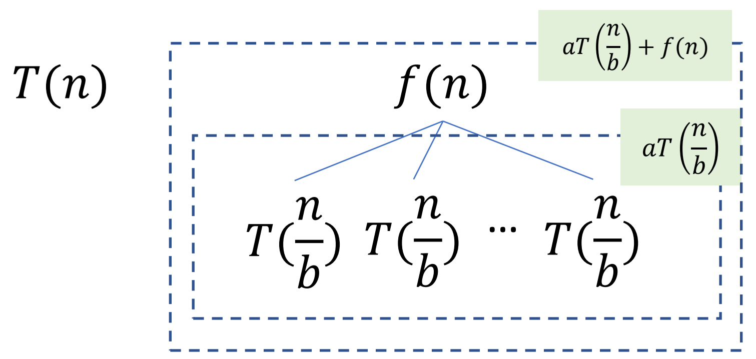
Height (h)	#nodes at current level
1	1
2	2
3	4
4	8
.	.
$\lg n + 1$	2^{h-1}

Proof
Base case: $n = 1,\ \lg 1 + 1 = 0 + 1 = 1$
Inductive: $\lg 2^{h-1} + 1 = (h-1)\lg 2 + 1 = h$

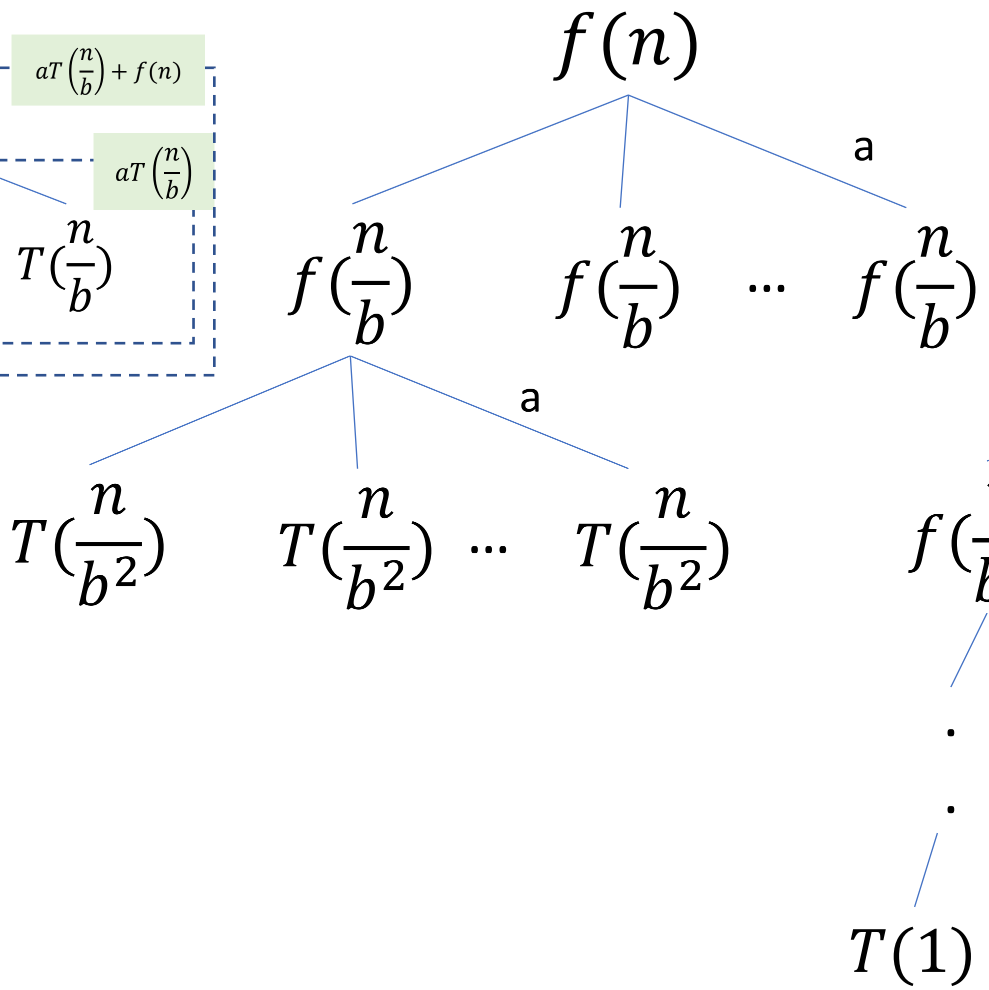
To compute the total cost represented by the recurrence, we simply add up the costs of all the levels.
The recursion tree has $\lg n + 1$ levels, each costing cn, for a total cost of $cn(\lg n + 1) = cn \lg n + cn$.
Giving us …$cn \lg n + cn \in \Theta(n \lg n)$

Recursion Tree

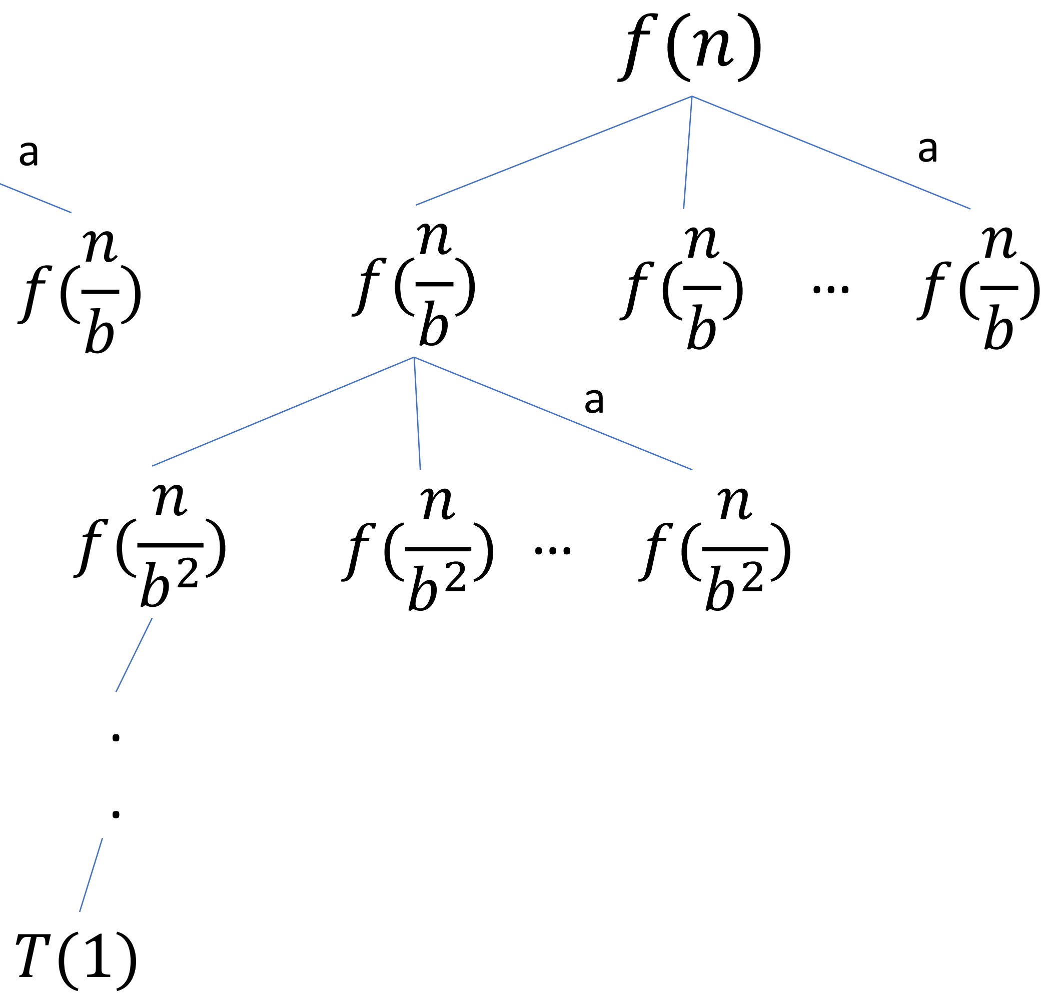
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$



$T(n)$
$aT\left(\frac{n}{b}\right) + f(n)$
$f(n)$
$aT\left(\frac{n}{b}\right)$
$T(\frac{n}{b})$ $T(\frac{n}{b})$ ⋯ $T(\frac{n}{b})$



$f(n)$
a
$f(\frac{n}{b})$ $f(\frac{n}{b})$ ⋯ $f(\frac{n}{b})$
a
$T(\frac{n}{b^2})$ $T(\frac{n}{b^2})$ ⋯ $T(\frac{n}{b^2})$



$f(n)$
a
$f(\frac{n}{b})$ $f(\frac{n}{b})$ ⋯ $f(\frac{n}{b})$
a
$f(\frac{n}{b^2})$ $f(\frac{n}{b^2})$ ⋯ $f(\frac{n}{b^2})$
$T(1)$

In-class exercise

- Solve the recurrence: $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$ using the recursion tree method.
- T(1) for base case.

Master Theorem

(Master Theorem) Let $a \geq 1$ and $b > 1$ be constants, let $\mathrm{f}(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where $\frac{n}{b}$ can be either $\lfloor n/b \rfloor$ $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

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

Using Master Theorem

Example 1: $T(n) = 9T\left(\frac{n}{3}\right) + n$

$a = 9, b = 3, f(n) = n,$

We have $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2).$ Since $f(n) = O\left(n^{\log_3 9-\epsilon}\right)$ and $\epsilon = 1$, we can apply case 1 to conclude that the solution is $T(n) = \Theta(n^2)$.

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

$a = 1, b = \frac{2}{3}, f(n) = 1$

We have $n^{\log_{\frac{2}{3}} 1} = n^0 = 1.$ Since $f(n) = \Theta(1)$, we apply case 2 to conclude that $T(n) = \Theta(\ln n)$.

Using Master Theorem

Example 3: $\mathrm{T(n)} = 3T\left(\frac{n}{4}\right) + n \lg n$

$a = 3, b = 4, f(n) = n \lg n$

We have $n^{\log_b a} = n^{\log_4 3} = O(n^{0.79})$. Since $f(n) = n \lg n = \Omega(n^{0.79+\epsilon})$ and $\epsilon = 0.2$, we can apply case 3 to conclude that the solution is $T(n) = \Theta(n \lg n)$.

Be careful!!, before concluding for case 3, we are required to show that the regularity condition holds for $f(n)$.

That is, for sufficiently large n, we have that $af\left(\frac{n}{b}\right) = 3f\left(\frac{n}{4}\right) = 3\frac{n}{4}\lg\frac{n}{4} \leq cn \lg n$, let $c = \frac{3}{4}$,

We can show that $3\frac{n}{4}\lg\frac{n}{4} \leq \frac{3}{4} n \lg n$. (Done)

Using Master Theorem

Example 4: $\mathrm{T(n)} = 2T\left(\frac{n}{2}\right) + n \lg n$

$a = 2, b = 2, f(n) = n \lg n$

We have $n^{\log_b a} = n^{\log_2 2} = \mathbf{n}$. Since $f(n) = n \lg n = \Omega(n)$, we may think that we apply case 3 to conclude that the solution is $T(n) = \Theta(n \lg n)$.

Be careful!!, before concluding for case 3, we are required to show that the regularity condition holds for $f(n)$.

In this case, it fails to apply the case 3 because, f(n) is not polynomially larger than $n^{\log_b a}$. That is the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n \lg n}{n} = \lg n$ is asymptotically less than n^{ϵ} for any positive constant $\boldsymbol{\epsilon}$. (it falls into the gap between case 2 and 3)

1. If $f(n) = \Omega\left(n^{\log_b a+\epsilon}\right)$ for some constant $\epsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $\mathrm{c} < 1$ and all sufficiently large n, then $T(n) = \Theta\left(f(n)\right)$.

Using Master Theorem

Example 5: $\mathrm{T(n)} = 2T\left(\frac{n}{2}\right) + \Theta(n)$

$a = 2, b = 2, f(n) = \Theta(n)$

We have $n^{\log_b a} = n^{\log_2 2} = \mathrm{n}.$ Since $f(n) = \Theta(n)$, we can see that it is $\Theta(n) = \Theta(n)$, so case 2 can be applied here. So the solution will be $T(n) = \Theta(n \lg n)$

Example 6: $\mathrm{T(n)} = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$

$a = 8, b = 2, f(n) = \Theta(n)$

We have $n^{\log_b a} = n^{\log_2 8} = \mathrm{n}^3.$ Since $f(n) = \Theta(n^2)$, we can see that it is $\Theta(n^2) = \mathrm{O}(n^{3-\epsilon})$, choose $\epsilon = 1$, n^3 is polynomially learger than n^2, case 1 applies. So $T(n) \in \Theta(\mathrm{n}^3)$

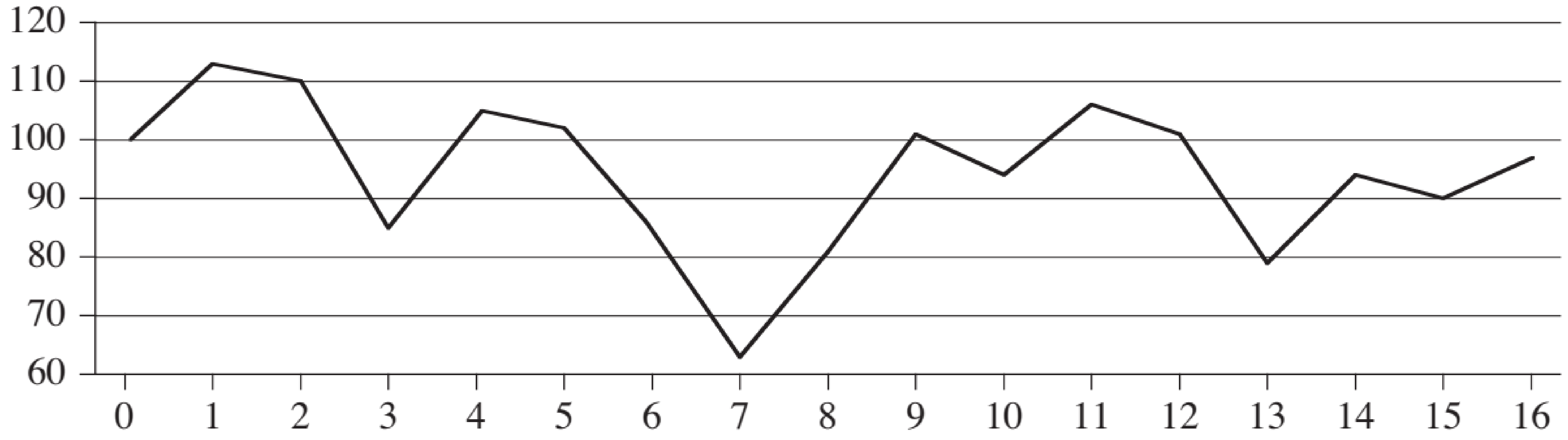
Substitution method

- In the substitution method, we guess a bound and then use mathematical induction to prove our guess correct.

The Maximum-subarray Problem

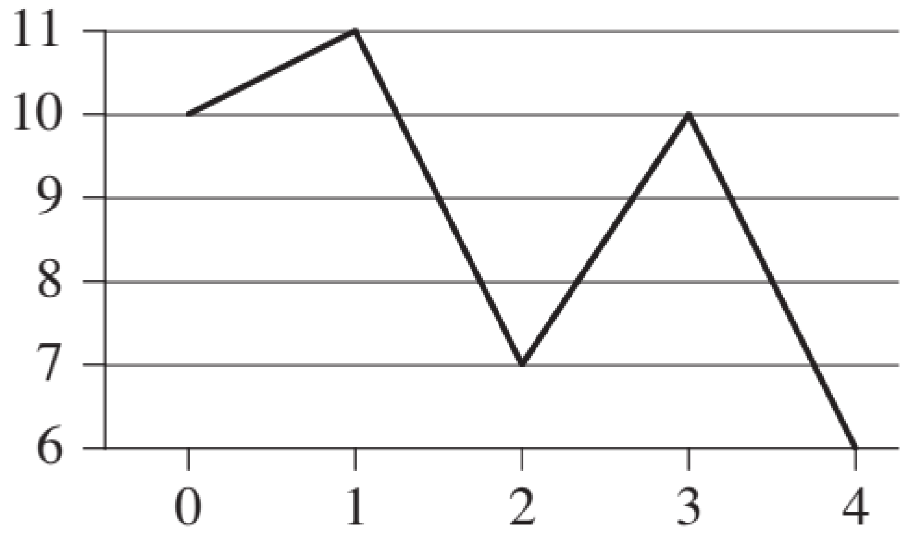
- Suppose that you have been offered the opportunity to invest in the Volatile Chemical Corporation.
  - The stock price of the Volatile Chemical Corporation is rather volatile.
  - You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day.
  - To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future.
  - **Your goal is to maximize your profit.**

The Maximum-subarray Problem



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

The Maximum-subarray Problem



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	–4	3	–4

An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of $3 per share would be earned by buying after day 2 and selling after day 3. The price of $7 after day 2 is not the lowest price overall, and the price of $10 after day 3 is not the highest price overall.

Come on… Think about brute force…

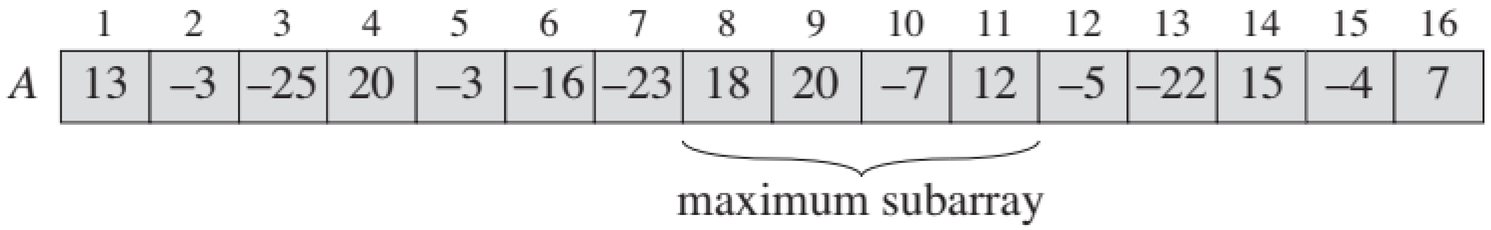
Brute force

- We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date.
- For n days, it has $\binom{n}{2}$ pairs of dates that you can buy the share.
- So $\binom{n}{2}$ is $\Theta(n^2)$

Is there any better way to solve this?

Transformation

- Instead of looking at the daily prices, let us instead consider the daily change in price, where the change on day i is the difference between the prices after day $i-1$ and after day i.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

- we now want to find the nonempty, contiguous subarray of A whose values have the largest sum. We call this contiguous subarray the **maximum subarray**.
  - From the above example, you would want to buy the stock just **before day 8** (that is, after day 7) and sell **it after day 11**, earning a profit of \$43 per share.
- Although computing the cost of one subarray might take time proportional to the length of the subarray, when computing all $\Theta(n^2)$ subarray sums, we can organize the computation so that each subarray sum takes $O(1)$ time, given the values of previously computed subarray sums, so that the brute-force solution takes $\Theta(n^2)$ time.

Non-negative array?

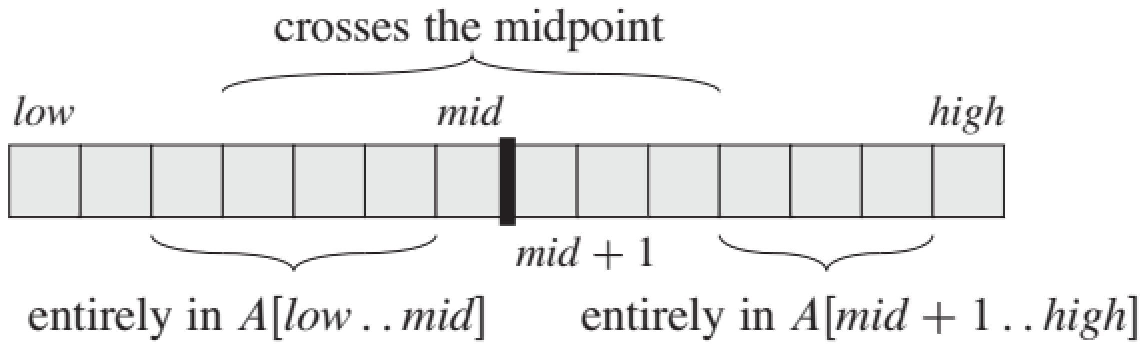
- What if all elements in the array are all non-negative numbers?

A solution using divide-and-conquer

- Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible.
- Suppose we want to find the maximum subarray $A = [low \ldots high]$,
- First, we find the midpoint, say mid, of the subarray, and consider the subarrays $A[low \ldots mid]$and $A[mid + 1 \ldots high]$.

A solution using divide-and-conquer

- Consider the following figure:

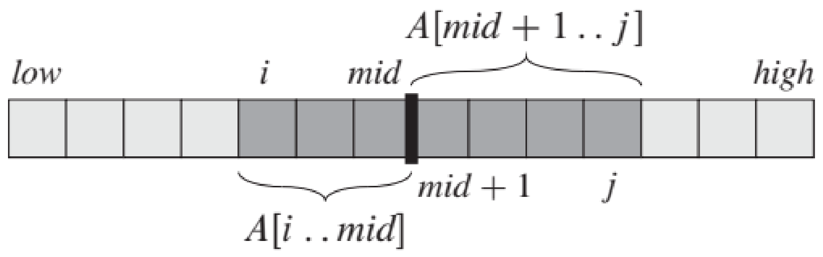


(a)

- Any contiguous subarray $A[i \dots j]$ of $A[low \dots high]$ must lie in exactly one of the following places:
  - entirely in the subarray $A[low \dots mid]$, so that $low \leq \ i \leq j \leq mid$,
  - entirely in the subarray $A[mid + 1 \dots high]$, so that $mid < \ i \leq j \leq high$,
  - crossing the midpoint, so that $low \leq i \leq mid < j \leq high$.

A solution using divide-and-conquer

- We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray $A[low \dots high]$.
- This problem is not a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint.



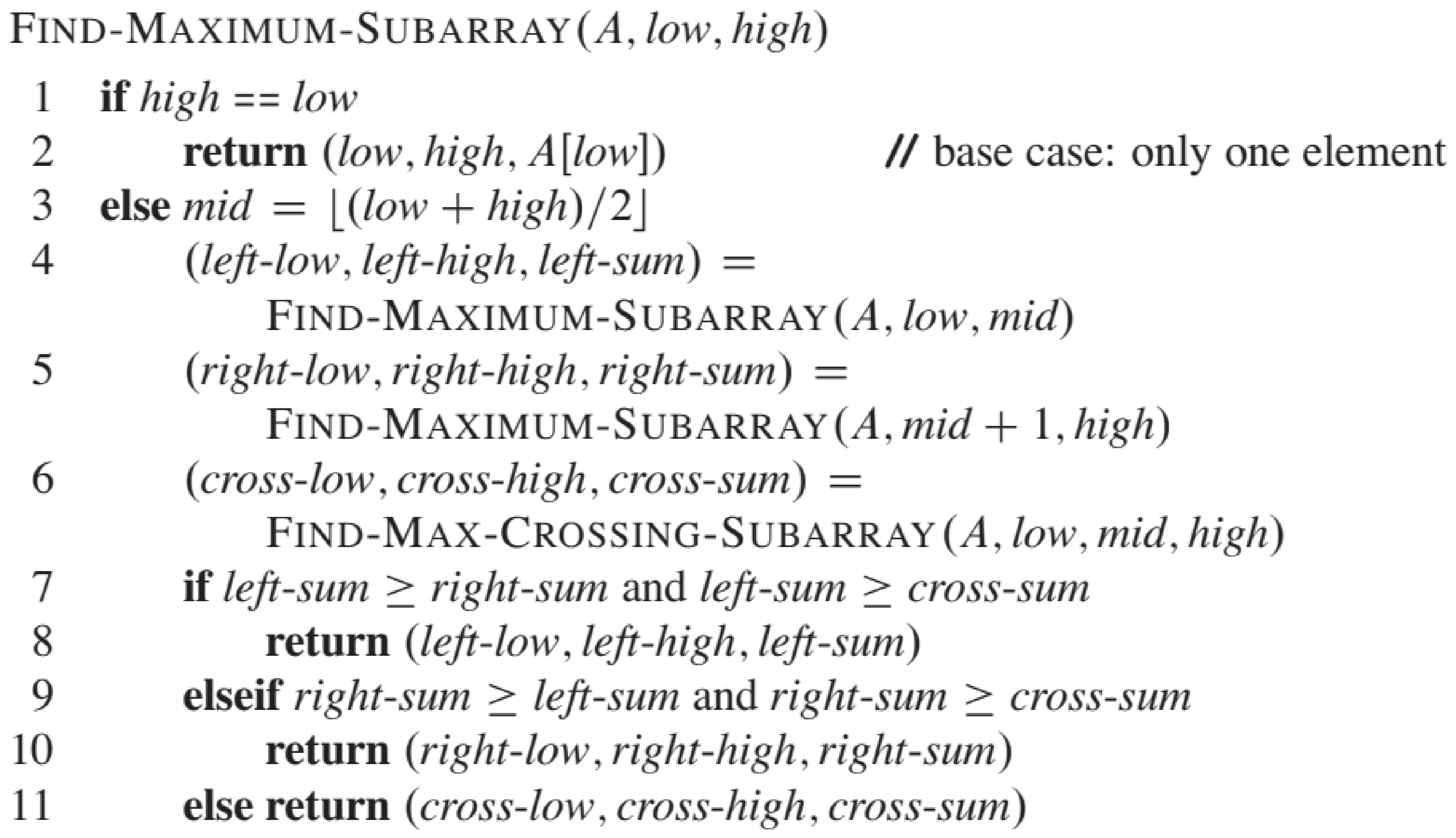
(b)

Any subarray crossing the midpoint is itself made of two subarrays $A[i \dots mid]$and $A[mid + 1 \dots j]$, where $low \le\ i \le\ mid$ and $mid\ \le\ j \le\ high$. Therefore, we just need to find maximum subarrays of the form $A[i \dots mid]$ and $A[mid + 1 \dots j]$, and then combine them.

A solution using divide-and-conquer

```
FIND-MAX-CROSSING-SUBARRAY($A$, $low$, $mid$, $high$)
left-sum $= -\infty$
sum $= 0$
**for** $i = mid$ **downto** $low$
    sum $=$ sum $+ A[i]$
    **if** sum $>$ left-sum
        left-sum $=$ sum
        max-left $= i$
right-sum $= -\infty$
sum $= 0$
**for** $j = mid + 1$ **to** $high$
    sum $=$ sum $+ A[j]$
    **if** sum $>$ right-sum
        right-sum $=$ sum
        max-right $= j$
**return** (max-left, max-right, left-sum $+$ right-sum)
```

A solution using divide-and-conquer



```
FIND-MAXIMUM-SUBARRAY($A, low, high$)
if $high == low$
    return ($low, high, A[low]$)                // base case: only one element
else $mid = \lfloor (low + high)/2 \rfloor$
    ($left\text{-}low, left\text{-}high, left\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A, low, mid$)
    ($right\text{-}low, right\text{-}high, right\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A, mid + 1, high$)
    ($cross\text{-}low, cross\text{-}high, cross\text{-}sum$) =
        FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)
    if $left\text{-}sum \geq right\text{-}sum$ and $left\text{-}sum \geq cross\text{-}sum$
        return ($left\text{-}low, left\text{-}high, left\text{-}sum$)
    elseif $right\text{-}sum \geq left\text{-}sum$ and $right\text{-}sum \geq cross\text{-}sum$
        return ($right\text{-}low, right\text{-}high, right\text{-}sum$)
    else return ($cross\text{-}low, cross\text{-}high, cross\text{-}sum$)
```

Efficiency Analysis

- Left as an exercise …

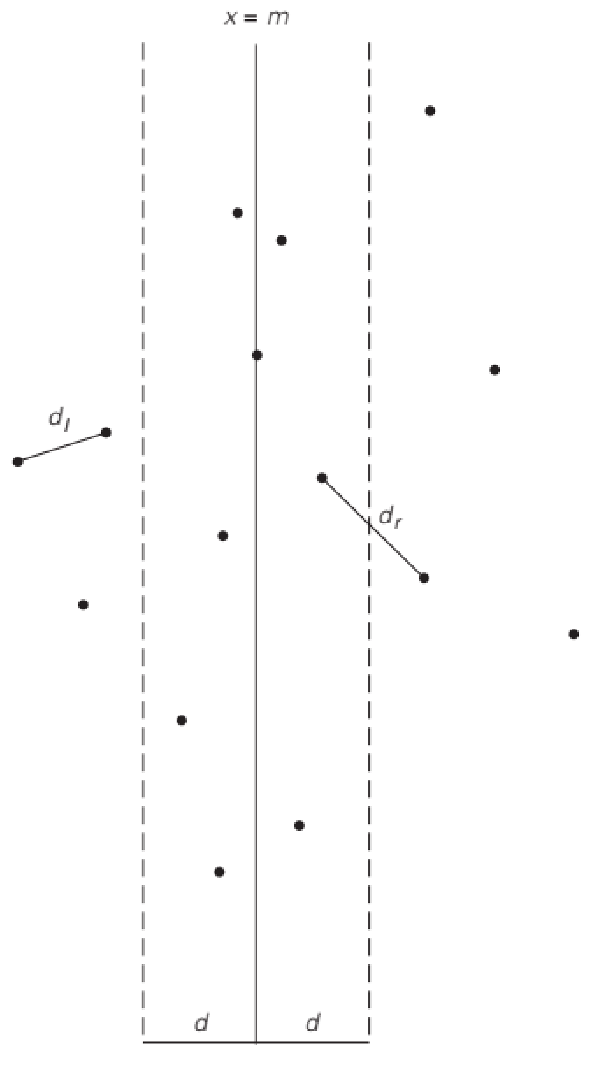
The Closest-Pair Problem

Closest-Pair Problem

Given a set (P) of n points, the closest-pair problem calls for finding the two closest points.

- Here, we assume that that points are sorted in nondecreasing order of their x coordinate.
- And, we also have another list of points Q that is also sorted in nondecreasing order of their y coordinate
- If $2 \leq n \leq 3$, the problem can be solved by the obvious brute-force algorithm.
- If $n > 3$, we can divide the points into two subsets P_l and P_r of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points, respectively, by drawing a vertical line through the median m of their x coordinates so that $\lceil n/2 \rceil$ points lie to the left of or on the line itself, and $\lfloor n/2 \rfloor$ points lie to the right of or on the line.

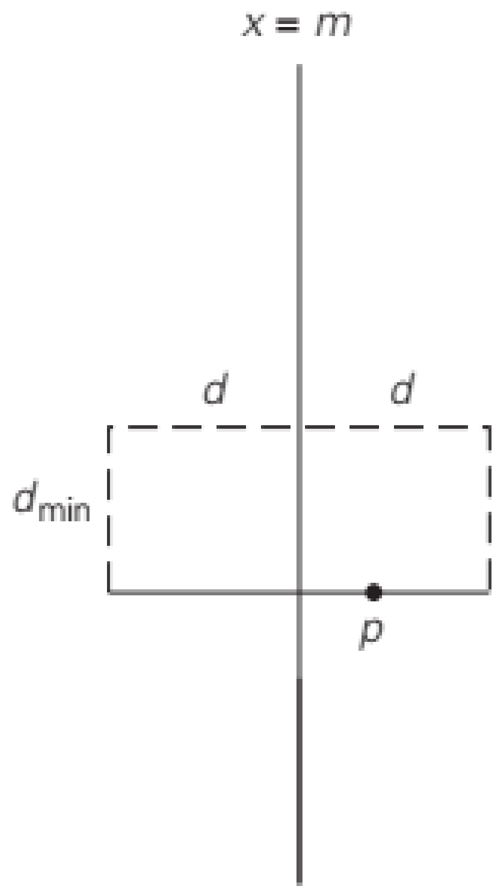
The Closest-Pair Problem



(a)

- **Then we can solve the closest-pair problem recursively for subsets $\boldsymbol{P_l}$ and $\boldsymbol{P_r}$.**
- Let d_l and d_r be the smallest distances between
- pairs of points in P_l and P_r, respectively, and let $d = \min\{d_l\ d_r\}$.
- Let S be the list of points inside the strip of width $2d$ around the separating line, obtained from Q and hence ordered in nondecreasing order of their y coordinate.

The Closest-Pair Problem



(b)

- Let $p(x, y)$ be a point on this list.
- For a point $p'(x', y')$ to have a chance to be closer to p than d_{min}, the point must follow p on list S and the difference between their y coordinates must be less than d_{min}.
- **It is easy to prove that the total number of such points in the rectangle, including $\boldsymbol{p}$, does not exceed eight**.
- Thus, the algorithm can consider no more than five next points following p on the list S, before moving up to the next point.

```
ALGORITHM  EfficientClosestPair(P, Q)
    //Solves the closest-pair problem by divide-and-conquer
    //Input: An array P of $n \geq 2$ points in the Cartesian plane sorted in
    //       nondecreasing order of their x coordinates and an array Q of the
    //       same points sorted in nondecreasing order of the y coordinates
    //Output: Euclidean distance between the closest pair of points
    if $n \leq 3$
        return the minimal distance found by the brute-force algorithm
    else
        copy the first $\lceil n/2 \rceil$ points of P to array $P_l$
        copy the same $\lceil n/2 \rceil$ points from Q to array $Q_l$
        copy the remaining $\lfloor n/2 \rfloor$ points of P to array $P_r$
        copy the same $\lfloor n/2 \rfloor$ points from Q to array $Q_r$
        $d_l \leftarrow EfficientClosestPair(P_l, Q_l)$
        $d_r \leftarrow EfficientClosestPair(P_r, Q_r)$
        $d \leftarrow \min\{d_l, d_r\}$
        $m \leftarrow P[\lceil n/2 \rceil - 1].x$
        copy all the points of Q for which $|x - m| < d$ into array $S[0..num - 1]$
        $dminsq \leftarrow d^2$
        for $i \leftarrow 0$ to $num - 2$ do
            $k \leftarrow i + 1$
            while $k \leq num - 1$ and $(S[k].y - S[i].y)^2 < dminsq$
                $dminsq \leftarrow \min((S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2, dminsq)$
                $k \leftarrow k + 1$
    return $sqrt(dminsq)$
```

Efficiency Analysis…

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.