Algorithms: Divide-and-Conquer (Matrix multiplication)

Alessandro Chiesa, Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 5, 04.03.2025

RECALL LAST LECTURE

Solving Recurrences:

- Substitution method
- Recursion Trees
- (Master Method)

Maximum-Subarray Problem

Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$

Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$

$$cn^2$$

Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$

$$cn^2$$

$$cn^2/4 \qquad\qquad cn^2/4$$

Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$

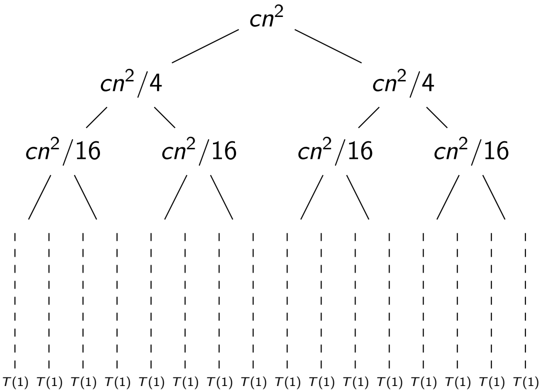
$$cn^2$$

$$cn^2/4 \qquad cn^2/4$$

$$cn^2/16 \qquad cn^2/16 \qquad cn^2/16 \qquad cn^2/16$$

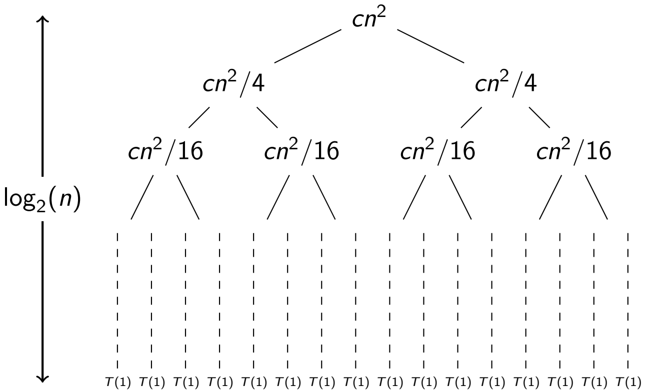
Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$



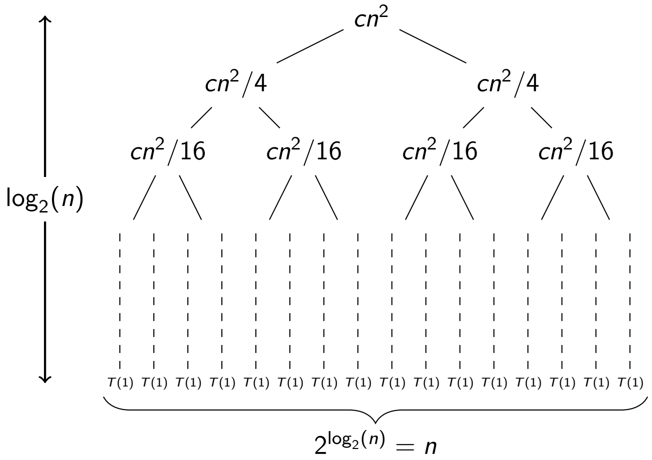
Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$



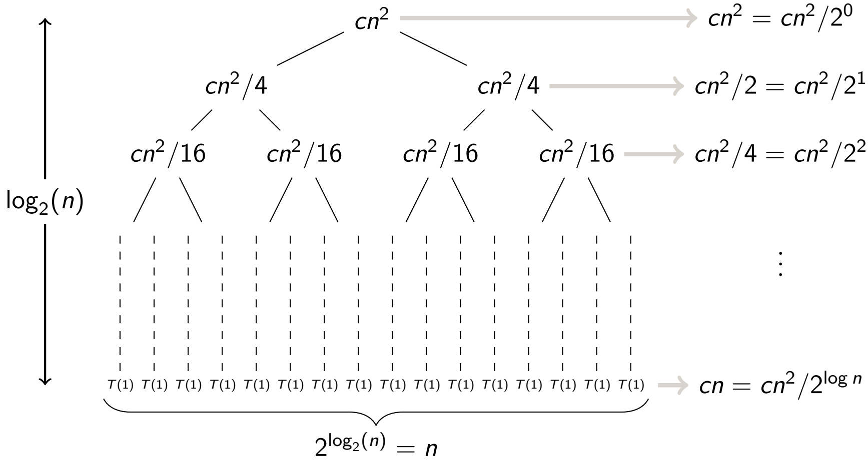
Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$



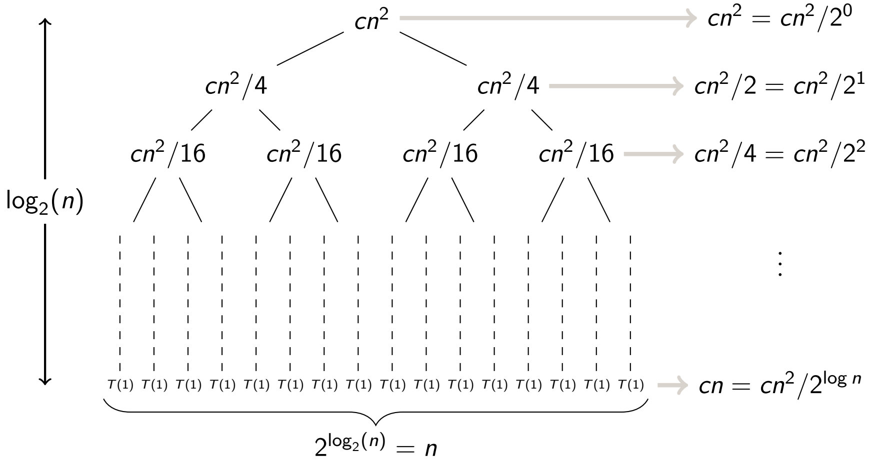
Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$



Recall Recursion trees

Example: $T(1) = c$ and $T(n) = 2T(n/2) + cn^2$



Qualified guess: $T(n) = cn^2 \sum_{i=0}^{\log n} \frac{1}{2^i} \leq cn^2 = \Theta(n^2)$

Recall Recursion trees

Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$

Recall Recursion trees

Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$

$$cn$$

Recall Recursion trees

Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$

$$cn$$

$$cn/3 \qquad\qquad c2n/3$$

Recall Recursion trees

Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$

$$cn$$

$$cn/3 \qquad\qquad c2n/3$$

$$cn/9 \qquad c2n/9 \qquad c2n/9 \qquad c4n/9$$

$$\vdots$$

Recall Recursion trees

Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$

$$cn \longrightarrow cn$$

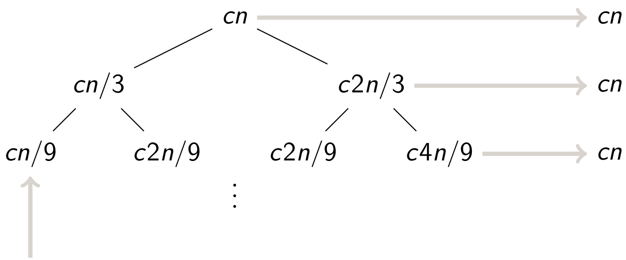
$$cn/3 \qquad c2n/3 \longrightarrow cn$$

$$cn/9 \qquad c2n/9 \qquad c2n/9 \qquad c4n/9 \longrightarrow cn$$

$$\vdots$$

Recall Recursion trees

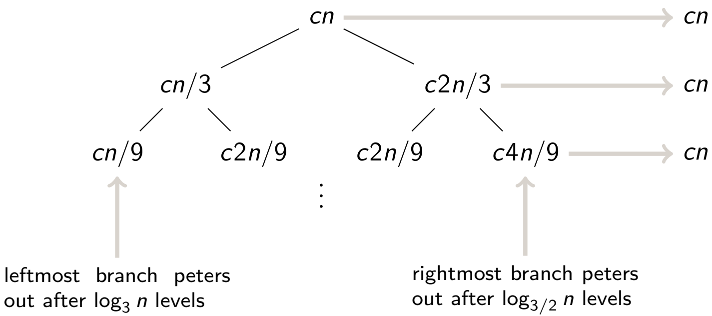
Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$



leftmost branch peters out after $\log_3 n$ levels

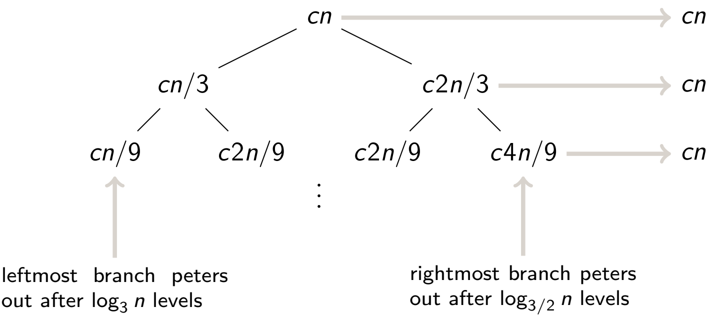
Recall Recursion trees

Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$



Recall Recursion trees

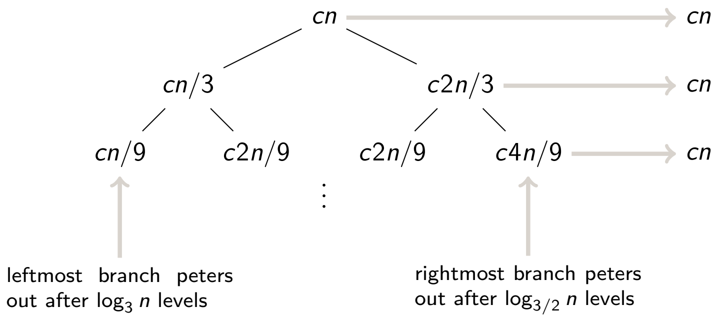
Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$



- There are $\log_3 n$ full levels and after $\log_{3/2} n$ levels the problem size is down to 1.

Recall Recursion trees

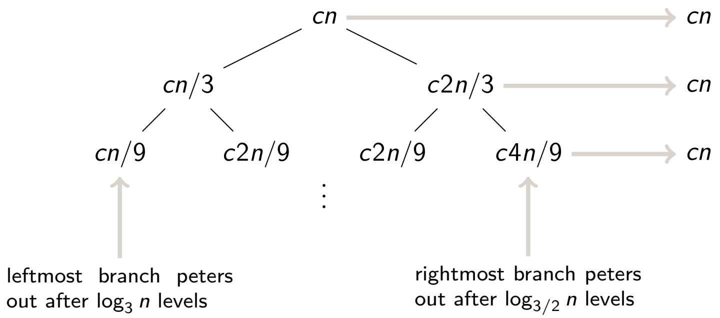
Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$



- There are $\log_3 n$ full levels and after $\log_{3/2} n$ levels the problem size is down to 1.
- Each level contributes $\approx cn$

Recall Recursion trees

Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$



- There are $\log_3 n$ full levels and after $\log_{3/2} n$ levels the problem size is down to 1.
- Each level contributes $\approx cn$

Qualified guess: exist positive constants a, b so that

$$a \cdot n \log_3(n) \leq T(n) \leq b \cdot n \log_{3/2} n \Rightarrow T(n) = \Theta(n \log n)$$

Master method

Used to black-box solve recurrences of the form $T(n) = aT(n/b) + f(n)$

Master method

Used to black-box solve recurrences of the form $T(n) = aT(n/b) + f(n)$

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $T(n)$ be defined on the nonnegative integers by the recurrence

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

Then, $T(n)$ has the following asymptotic bounds

Master method

Used to black-box solve recurrences of the form $T(n) = aT(n/b) + f(n)$

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $T(n)$ be defined on the nonnegative integers by the recurrence

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

Then, $T(n)$ has the following asymptotic bounds

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$*

Master method

Used to black-box solve recurrences of the form $T(n) = aT(n/b) + f(n)$

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $T(n)$ be defined on the nonnegative integers by the recurrence

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

Then, $T(n)$ has the following asymptotic bounds

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$*
- *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$*

Master method

Used to black-box solve recurrences of the form $T(n) = aT(n/b) + f(n)$

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $T(n)$ be defined on the nonnegative integers by the recurrence

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

Then, $T(n)$ has the following asymptotic bounds

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$*
- *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$*
- *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$*

Master method

Used to black-box solve recurrences of the form $T(n) = aT(n/b) + f(n)$

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $T(n)$ be defined on the nonnegative integers by the recurrence

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

Then, $T(n)$ has the following asymptotic bounds

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$*
- *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$*
- *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$*

Our favorite example: $T(1) = c$ and $T(n) = 2T(n/2) + cn$

- $f(n) = O(n)$ and $a = b = 2$ so $\log_b(a) = 1$ and $f(n) = \Theta(n^{\log_b(a)})$.

Master method

Used to black-box solve recurrences of the form $T(n) = aT(n/b) + f(n)$

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $T(n)$ be defined on the nonnegative integers by the recurrence

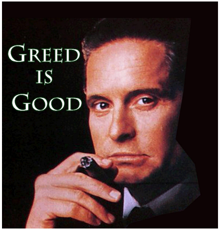
$$T(n) = aT(n/b) + f(n).$$

Then, $T(n)$ has the following asymptotic bounds

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$*
- *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$*
- *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$*

Our favorite example: $T(1) = c$ and $T(n) = 2T(n/2) + cn$

- $f(n) = O(n)$ and $a = b = 2$ so $\log_b(a) = 1$ and $f(n) = \Theta(n^{\log_b(a)})$.
- By Master theorem, we have $T(n) = \Theta(n \log n)$:)



MAXIMUM-SUBARRAY PROBLEM

Maximum-subarray problem

"If we let $A[i] =$ (price after day i) $-$ (price after day $i-1$) then if the maximum subarray is $A[i \ldots j]$ then we should have bought just before day i and sold just after day j."

Definition

INPUT: An array $A[1 \ldots n]$ of numbers

OUTPUT: Indices i and j such that $A[i \ldots j]$ has the greatest sum of any nonempty, contiguous subarray of A, along with the sum of the values in $A[i \ldots j]$

Maximum-subarray problem

"If we let $A[i] =$ (price after day i) $-$ (price after day $i-1$) then if the maximum subarray is $A[i \ldots j]$ then we should have bought just before day i and sold just after day j."

Definition

INPUT: An array $A[1 \ldots n]$ of numbers

OUTPUT: Indices i and j such that $A[i \ldots j]$ has the greatest sum of any nonempty, contiguous subarray of A, along with the sum of the values in $A[i \ldots j]$

Examples:

1	-4	3	-4

Maximum-subarray problem

"If we let $A[i] =$ (price after day i) $-$ (price after day $i-1$) then if the maximum subarray is $A[i \dots j]$ then we should have bought just before day i and sold just after day j."

Definition

INPUT: An array $A[1 \dots n]$ of numbers

OUTPUT: Indices i and j such that $A[i \dots j]$ has the greatest sum of any nonempty, contiguous subarray of A, along with the sum of the values in $A[i \dots j]$

Examples:

1	-4	3	-4

output is $i = j = 3$ and the sum 3

Maximum-subarray problem

"If we let $A[i] = (\text{price after day } i) - (\text{price after day } i-1)$ then if the maximum subarray is $A[i \dots j]$ then we should have bought just before day i and sold just after day j."

Definition

INPUT: An array $A[1 \dots n]$ of numbers

OUTPUT: Indices i and j such that $A[i \dots j]$ has the greatest sum of any nonempty, contiguous subarray of A, along with the sum of the values in $A[i \dots j]$

Examples:

1	-4	3	-4

output is $i = j = 3$ and the sum 3

-2	-4	3	-1	5	7	-7	-2	4	-3	2

Maximum-subarray problem

"If we let $A[i] = (\text{price after day } i) - (\text{price after day } i-1)$ then if the maximum subarray is $A[i \dots j]$ then we should have bought just before day i and sold just after day j."

Definition

INPUT: An array $A[1 \dots n]$ of numbers

OUTPUT: Indices i and j such that $A[i \dots j]$ has the greatest sum of any nonempty, contiguous subarray of A, along with the sum of the values in $A[i \dots j]$

Examples:

1	-4	3	-4

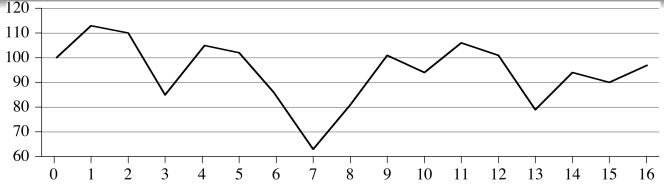
output is $i = j = 3$ and the sum 3

-2	-4	3	-1	5	7	-7	-2	4	-3	2

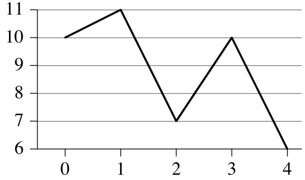
output is $i = 3$ and $j = 6$ and the sum 14

Maximum-subarray problem

More examples



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

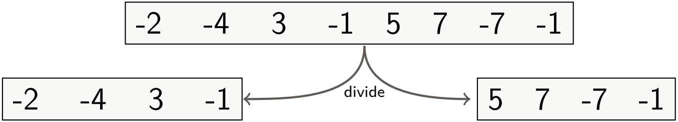


Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

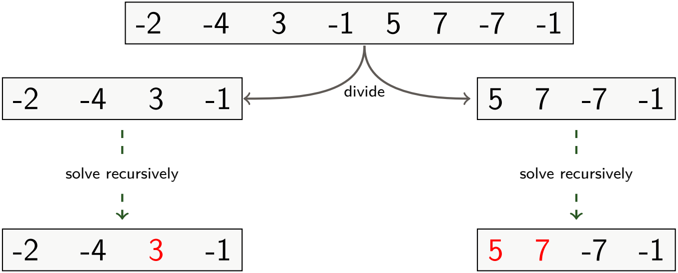
Divide-and-Conquer

-2	-4	3	-1	5	7	-7	-1

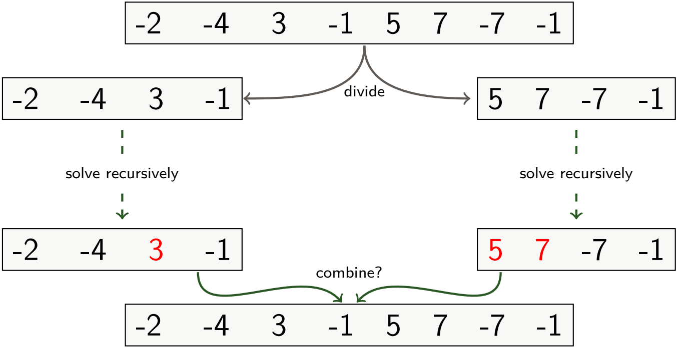
Divide-and-Conquer



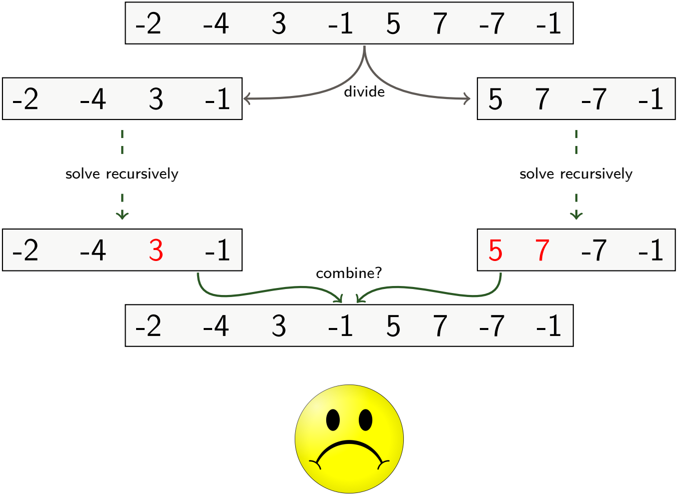
Divide-and-Conquer



Divide-and-Conquer



Divide-and-Conquer

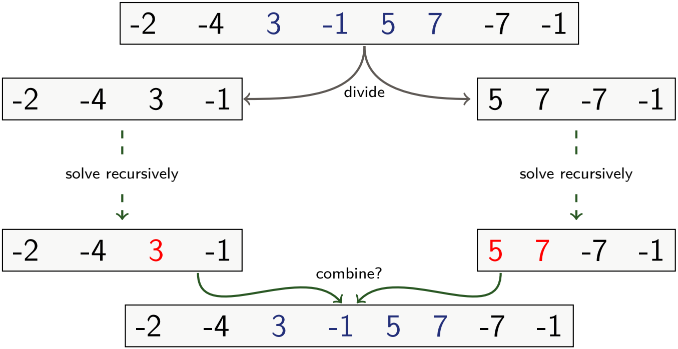


Solution

Also find the maximum subarray that crosses the midpoint!

Solution

Also find the maximum subarray that crosses the midpoint!



Divide-and-Conquer approach

Divide the subarray into two subarrays of as equal size as possible. Find the midpoint *mid* of the subarrays, and consider the subarrays $A[low \dots mid]$ and $A[mid + 1 \dots high]$.

Conquer by finding maximum subarrays of $A[low \dots mid]$ and $A[mid + 1 \dots high]$.

Combine by finding a maximum subarray that crosses the midpoint, and using the best solution out of the three

Divide-and-Conquer approach

Divide the subarray into two subarrays of as equal size as possible. Find the midpoint *mid* of the subarrays, and consider the subarrays $A[low \ldots mid]$ and $A[mid + 1 \ldots high]$.

Conquer by finding maximum subarrays of $A[low \ldots mid]$ and $A[mid + 1 \ldots high]$.

Combine by finding a maximum subarray that crosses the midpoint, and using the best solution out of the three

This strategy works because any subarray must either lie entirely on one side of the midpoint or cross the midpoint

Divide-and-Conquer approach

Divide the subarray into two subarrays of as equal size as possible. Find the midpoint *mid* of the subarrays, and consider the subarrays $A[low \ldots mid]$ and $A[mid + 1 \ldots high]$.

Conquer by finding maximum subarrays of $A[low \ldots mid]$ and $A[mid + 1 \ldots high]$.

Combine by finding a maximum subarray that crosses the midpoint, and using the best solution out of the three

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

Analysis

Assume that we can find max-crossing-subarray in time $\Theta(n)$

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

Analysis

Assume that we can find max-crossing-subarray in time $\Theta(n)$

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

Divide takes constant time,i.e., $\Theta(1)$

Analysis

Assume that we can find max-crossing-subarray in time $\Theta(n)$

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

Divide takes constant time,i.e., $\Theta(1)$

Conquer recursively solve two subproblems, each of size $n/2 \Rightarrow T(n/2)$

Analysis

Assume that we can find max-crossing-subarray in time $\Theta(n)$

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

Divide takes constant time,i.e., $\Theta(1)$

Conquer recursively solve two subproblems, each of size $n/2 \Rightarrow T(n/2)$

Merge time dominated by FIND-MAX-CROSSING-SUBARRAY $\Rightarrow \Theta(n)$

Analysis

Assume that we can find max-crossing-subarray in time $\Theta(n)$

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

Divide takes constant time,i.e., $\Theta(1)$

Conquer recursively solve two subproblems, each of size $n/2 \Rightarrow T(n/2)$

Merge time dominated by FIND-MAX-CROSSING-SUBARRAY $\Rightarrow \Theta(n)$

Recursion for the running time is

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

Analysis

Assume that we can find max-crossing-subarray in time $\Theta(n)$

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

Divide takes constant time,i.e., $\Theta(1)$

Conquer recursively solve two subproblems, each of size $n/2 \Rightarrow T(n/2)$

Merge time dominated by FIND-MAX-CROSSING-SUBARRAY $\Rightarrow \Theta(n)$

Recursion for the running time is

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

Hence, $T(n) = \Theta(n \log n)$

Finding maximum subarray crossing midpoint

Finding maximum subarray crossing midpoint

- Any subarray crossing the midpoint $A[mid]$ is made of two subarrays $A[i \ldots mid]$ and $A[mid+1, \ldots, j]$ where $low \leq i \leq mid$ and $mid < j \leq high$
- Find maximum subarrays of the form $A[i \ldots mid]$ and $A[mid+1 \ldots j]$ and then combine them.

-2	-4	3	-1	5	7	-7	-1

-2	-4	3	-1

5	7	-7	-1

-2	-4	3	-1	5	7	-7	-1

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i .. mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1 .. j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[i..mid].
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1..j].
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[i . . mid].
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1 . . j].
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i .. mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1 .. j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

low			mid	
-2	-4	3	-1	5

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i..mid].
  left-sum = -∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid+1..j].
  right-sum = -∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```


```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i..mid].
  left-sum = -∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid+1..j].
  right-sum = -∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i..mid].
  left-sum = -∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid+1..j].
  right-sum = -∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$

```
// Find a maximum subarray of the form A[i .. mid].
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1 .. j].
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$

```
// Find a maximum subarray of the form A[i .. mid].
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1 .. j].
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i..mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1..j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i .. mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1 .. j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i .. mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1 .. j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i..mid].
  left-sum = -∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid+1..j].
  right-sum = -∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

Running time?

Space?

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i..mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1..j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

Crossing subarray

Running time? $\Theta(n)$

Space?

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i..mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1..j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1

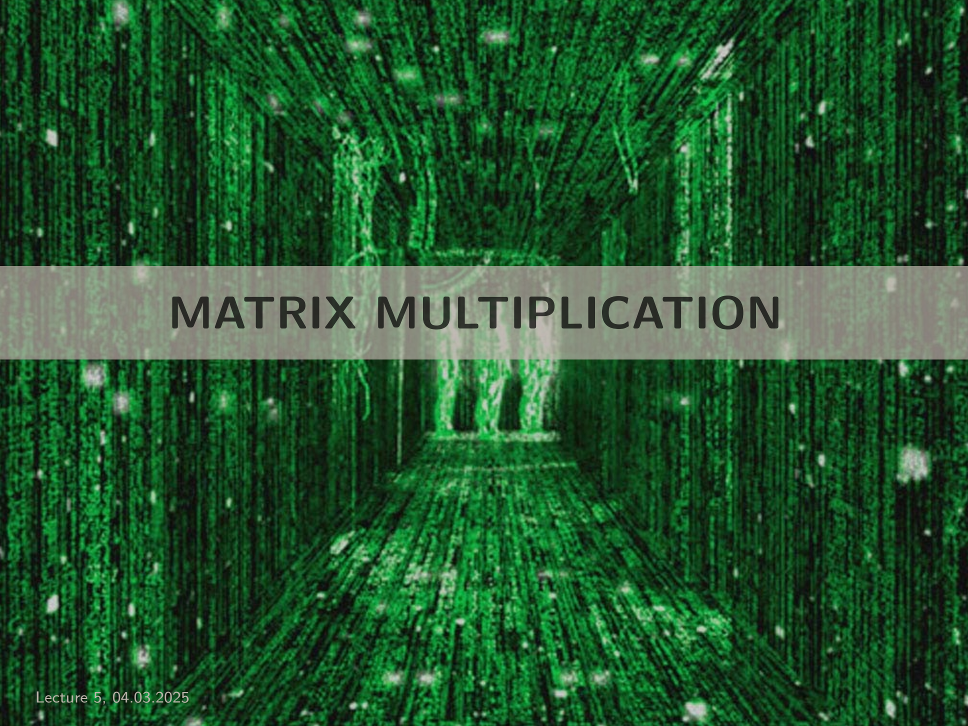
Crossing subarray

Running time? $\Theta(n)$

Space? $\Theta(n)$

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i .. mid].
  left-sum = -∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1 .. j].
  right-sum = -∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

low			mid				high
-2	-4	3	-1	5	7	-7	-1



MATRIX MULTIPLICATION

Lecture 5, 04.03.2025

Matrix Multiplication

Definition

Input: Two $n \times n$ (square) matrices, $A = (a_{ij})$ and $B = (b_{ij})$

Output: $n \times n$ matrix $C = (c_{ij})$, where $C = A \cdot B$

Matrix Multiplication

Definition

Input: Two $n \times n$ (square) matrices, $A = (a_{ij})$ and $B = (b_{ij})$

Output: $n \times n$ matrix $C = (c_{ij})$, where $C = A \cdot B$

Example ($n = 2$):

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

where

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21}, \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22}, \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21}, \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22}. \end{aligned}$$

Matrix Multiplication

Definition

Input: Two $n \times n$ (square) matrices, $A = (a_{ij})$ and $B = (b_{ij})$

Output: $n \times n$ matrix $C = (c_{ij})$, where $C = A \cdot B$

Example ($n = 2$):

$$\begin{pmatrix} ? & ? \\ ? & ? \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Matrix Multiplication

Definition

Input: Two $n \times n$ (square) matrices, $A = (a_{ij})$ and $B = (b_{ij})$

Output: $n \times n$ matrix $C = (c_{ij})$, where $C = A \cdot B$

Example ($n = 2$):

$$\begin{pmatrix} 5 & 3 \\ 0 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

How to multiply two matrices?

How to multiply two matrices?

$$\begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,n} \end{pmatrix}$$

$$c_{1,1} = a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} + \ldots, a_{1,n}b_{n,1} = \sum_{k=1}^{n} a_{1k}b_{k1}$$

How to multiply two matrices?

$$\begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,n} \end{pmatrix}$$

$$c_{2,1} = a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + a_{2,3}b_{3,1} + \ldots, a_{2,n}b_{n,1} = \sum_{k=1}^{n} a_{2k}b_{k1}$$

How to multiply two matrices?

$$\begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,n} \end{pmatrix}$$

$$c_{2,2} = a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + a_{2,3}b_{3,2} + \dots, a_{2,n}b_{n,2} = \sum_{k=1}^{n} a_{2k}b_{k2}$$

How to multiply two matrices?

$$\begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,n} \end{pmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Example:

$$\begin{pmatrix} ? & ? \\ ? & ? \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Example:

$$\begin{pmatrix} {\color{red}?} & ? \\ ? & ? \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Example:

$$\begin{pmatrix} 3 & ? \\ ? & ? \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Example:

$$\begin{pmatrix} 5 & ? \\ ? & ? \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Example:

$$\begin{pmatrix} 5 & \color{red}{?} \\ ? & ? \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Example:

$$\begin{pmatrix} 5 & -1 \\ ? & ? \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Example:

$$\begin{pmatrix} 5 & 3 \\ ? & ? \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Example:

$$\begin{pmatrix} 5 & 3 \\ 0 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Running time?

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Running time? $\Theta(n^3)$

Naive Algorithm

Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Running time? $\Theta(n^3)$ 

Space?

Naive Algorithm

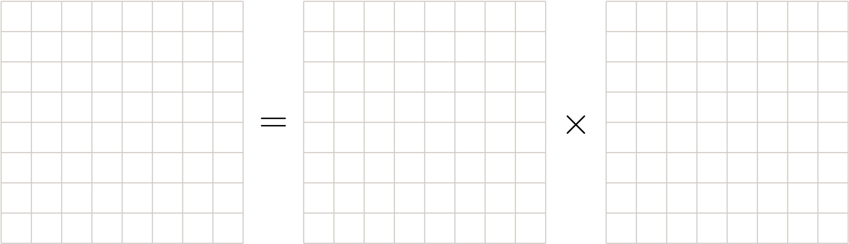
Well simply multiply the matrices...

```
SQUARE-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          c_ij = 0
          for k = 1 to n
              c_ij = c_ij + a_ik · b_kj
  return C
```

Running time? $\Theta(n^3)$

Space? $\Theta(n^2)$

Smart Algorithm: Divide-and-Conquer



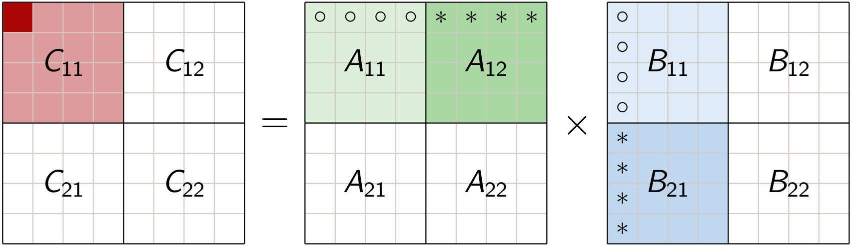
Smart Algorithm: Divide-and-Conquer

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

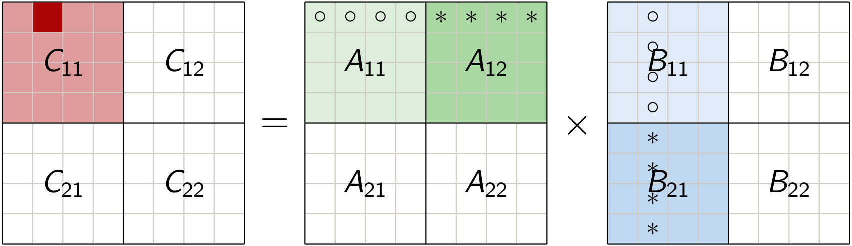
Smart Algorithm: Divide-and-Conquer

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

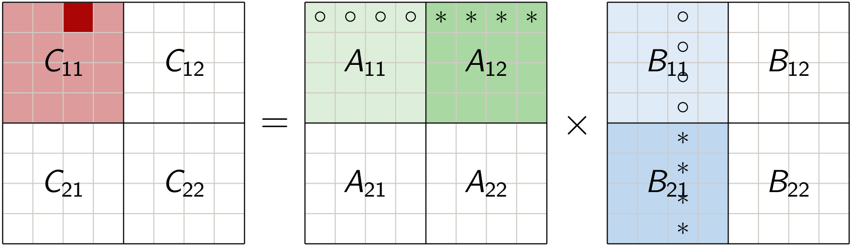
Smart Algorithm: Divide-and-Conquer



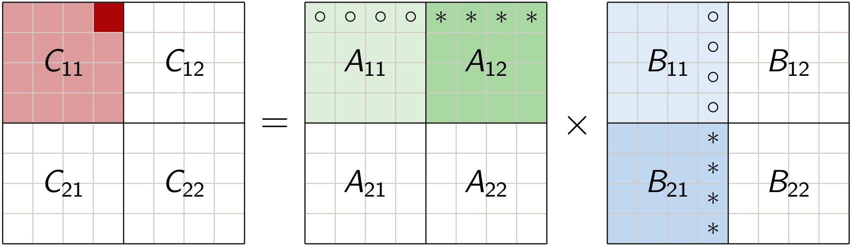
Smart Algorithm: Divide-and-Conquer



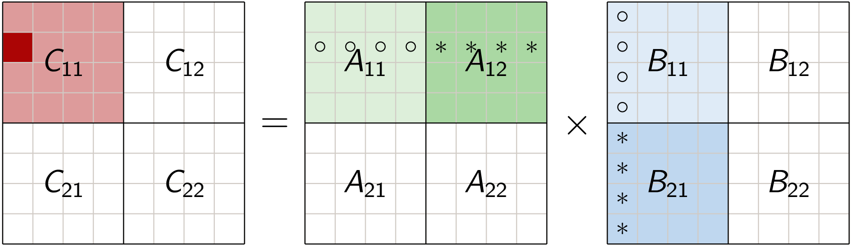
Smart Algorithm: Divide-and-Conquer



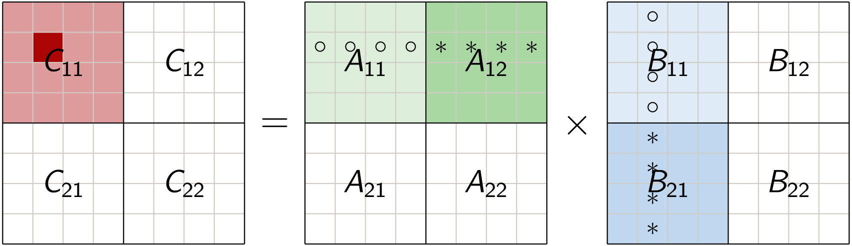
Smart Algorithm: Divide-and-Conquer



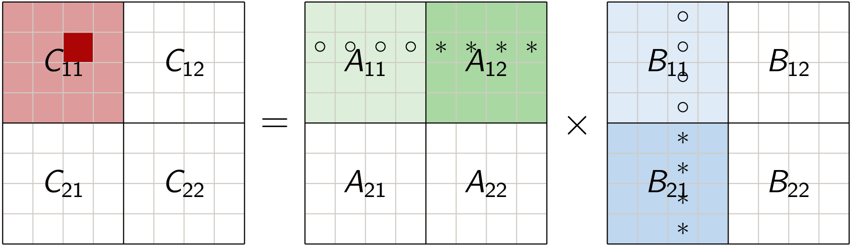
Smart Algorithm: Divide-and-Conquer



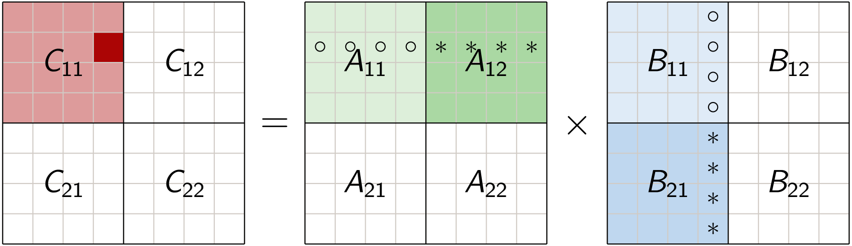
Smart Algorithm: Divide-and-Conquer



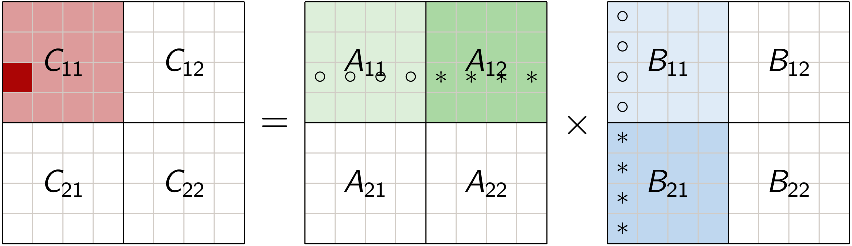
Smart Algorithm: Divide-and-Conquer



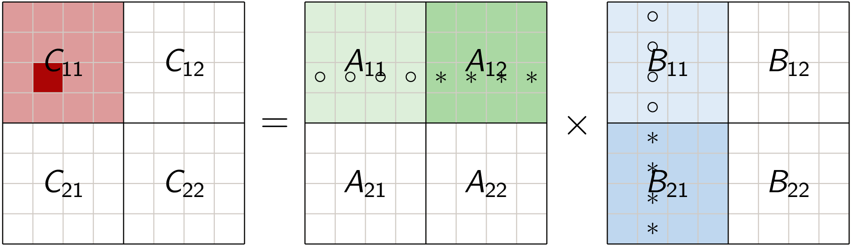
Smart Algorithm: Divide-and-Conquer



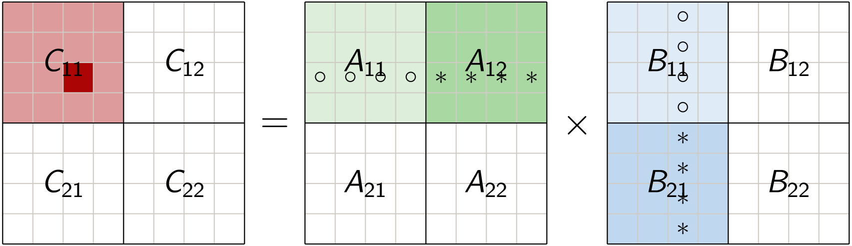
Smart Algorithm: Divide-and-Conquer



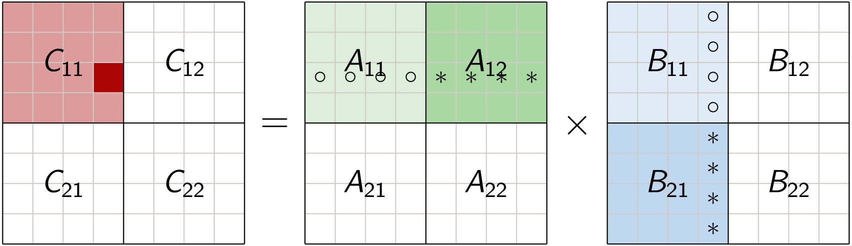
Smart Algorithm: Divide-and-Conquer



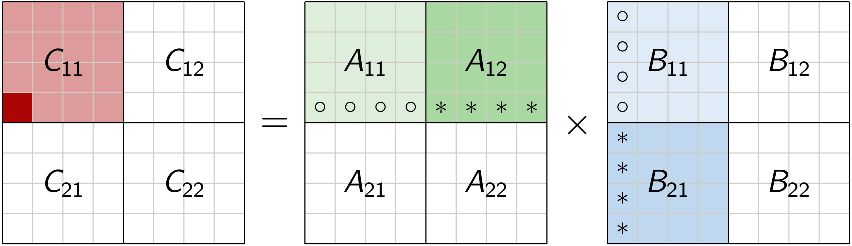
Smart Algorithm: Divide-and-Conquer



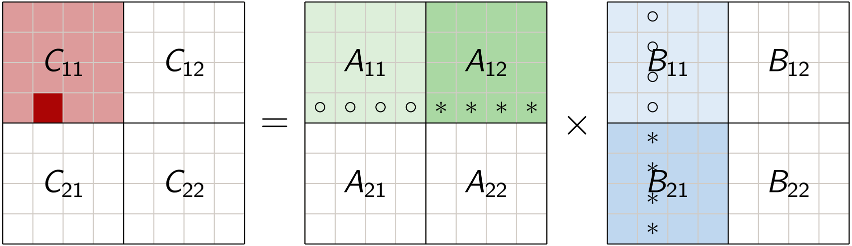
Smart Algorithm: Divide-and-Conquer



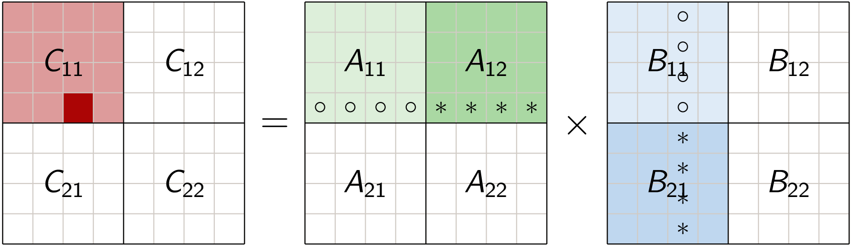
Smart Algorithm: Divide-and-Conquer



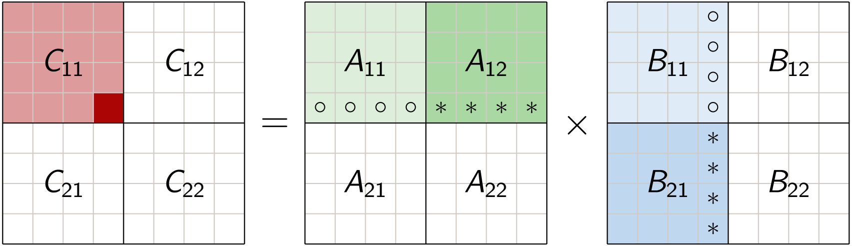
Smart Algorithm: Divide-and-Conquer



Smart Algorithm: Divide-and-Conquer



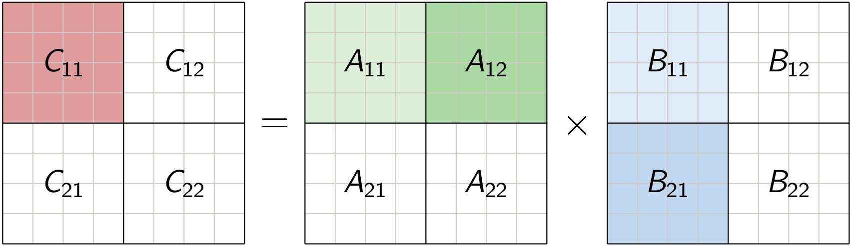
Smart Algorithm: Divide-and-Conquer



Smart Algorithm: Divide-and-Conquer

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

Smart Algorithm: Divide-and-Conquer



$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Divide-and-Conquer Algorithm

Divide each of A, B, C into four $n/2 \times n/2$ matrices: so that

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

Divide-and-Conquer Algorithm

Divide each of A, B, C into four $n/2 \times n/2$ matrices: so that

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

Conquer: Since

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

we recursively solve 8 matrix multiplications that each multiply two $n/2 \times n/2$ matrices.

Divide-and-Conquer Algorithm

Divide each of A, B, C into four $n/2 \times n/2$ matrices: so that

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

Conquer: Since

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

we recursively solve 8 matrix multiplications that each multiply two $n/2 \times n/2$ matrices.

Combine: Make the additions to get C

Pseudocode and Analysis

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

```
REC-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  if n == 1
      c11 = a11 · b11
  else partition A, B, and C into n/2 × n/2 submatrices
      C11 = REC-MAT-MULT(A11, B11, n/2) + REC-MAT-MULT(A12, B21, n/2)
      C12 = REC-MAT-MULT(A11, B12, n/2) + REC-MAT-MULT(A12, B22, n/2)
      C21 = REC-MAT-MULT(A21, B11, n/2) + REC-MAT-MULT(A22, B21, n/2)
      C22 = REC-MAT-MULT(A21, B12, n/2) + REC-MAT-MULT(A22, B22, n/2)
  return C
```

Pseudocode and Analysis

```
REC-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  if n == 1
      c11 = a11 · b11
  else partition A, B, and C into n/2 × n/2 submatrices
      C11 = REC-MAT-MULT(A11, B11, n/2) + REC-MAT-MULT(A12, B21, n/2)
      C12 = REC-MAT-MULT(A11, B12, n/2) + REC-MAT-MULT(A12, B22, n/2)
      C21 = REC-MAT-MULT(A21, B11, n/2) + REC-MAT-MULT(A22, B21, n/2)
      C22 = REC-MAT-MULT(A21, B12, n/2) + REC-MAT-MULT(A22, B22, n/2)
  return C
```

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

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$

Pseudocode and Analysis

```
REC-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  if n == 1
      c11 = a11 · b11
  else partition A, B, and C into n/2 × n/2 submatrices
      C11 = REC-MAT-MULT(A11, B11, n/2) + REC-MAT-MULT(A12, B21, n/2)
      C12 = REC-MAT-MULT(A11, B12, n/2) + REC-MAT-MULT(A12, B22, n/2)
      C21 = REC-MAT-MULT(A21, B11, n/2) + REC-MAT-MULT(A22, B21, n/2)
      C22 = REC-MAT-MULT(A21, B12, n/2) + REC-MAT-MULT(A22, B22, n/2)
  return C
```

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

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$

Recursive case: $n > 1$.

- Dividing takes $\Theta(1)$ time if careful and $\Theta(n^2)$ if simply copying

Pseudocode and Analysis

```
REC-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  if n == 1
      c11 = a11 · b11
  else partition A, B, and C into n/2 × n/2 submatrices
      C11 = REC-MAT-MULT(A11, B11, n/2) + REC-MAT-MULT(A12, B21, n/2)
      C12 = REC-MAT-MULT(A11, B12, n/2) + REC-MAT-MULT(A12, B22, n/2)
      C21 = REC-MAT-MULT(A21, B11, n/2) + REC-MAT-MULT(A22, B21, n/2)
      C22 = REC-MAT-MULT(A21, B12, n/2) + REC-MAT-MULT(A22, B22, n/2)
  return C
```

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

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$

Recursive case: $n > 1$.

- Dividing takes $\Theta(1)$ time if careful and $\Theta(n^2)$ if simply copying
- Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 8T(n/2)$

Pseudocode and Analysis

```
REC-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  if n == 1
      c11 = a11 · b11
  else partition A, B, and C into n/2 × n/2 submatrices
      C11 = REC-MAT-MULT(A11, B11, n/2) + REC-MAT-MULT(A12, B21, n/2)
      C12 = REC-MAT-MULT(A11, B12, n/2) + REC-MAT-MULT(A12, B22, n/2)
      C21 = REC-MAT-MULT(A21, B11, n/2) + REC-MAT-MULT(A22, B21, n/2)
      C22 = REC-MAT-MULT(A21, B12, n/2) + REC-MAT-MULT(A22, B22, n/2)
  return C
```

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

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$

Recursive case: $n > 1$.

- Dividing takes $\Theta(1)$ time if careful and $\Theta(n^2)$ if simply copying
- Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 8T(n/2)$
- Combining takes time $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices.

Pseudocode and Analysis

```
REC-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  if n == 1
      c11 = a11 · b11
  else partition A, B, and C into n/2 × n/2 submatrices
      C11 = REC-MAT-MULT(A11, B11, n/2) + REC-MAT-MULT(A12, B21, n/2)
      C12 = REC-MAT-MULT(A11, B12, n/2) + REC-MAT-MULT(A12, B22, n/2)
      C21 = REC-MAT-MULT(A21, B11, n/2) + REC-MAT-MULT(A22, B21, n/2)
      C22 = REC-MAT-MULT(A21, B12, n/2) + REC-MAT-MULT(A22, B22, n/2)
  return C
```

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

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$

Recursive case: $n > 1$.

- Dividing takes $\Theta(1)$ time if careful and $\Theta(n^2)$ if simply copying
- Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 8T(n/2)$
- Combining takes time $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices.

Recurrence is

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

Pseudocode and Analysis

```
REC-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  if n == 1
      c11 = a11 · b11
  else partition A, B, and C into n/2 × n/2 submatrices
      C11 = REC-MAT-MULT(A11, B11, n/2) + REC-MAT-MULT(A12, B21, n/2)
      C12 = REC-MAT-MULT(A11, B12, n/2) + REC-MAT-MULT(A12, B22, n/2)
      C21 = REC-MAT-MULT(A21, B11, n/2) + REC-MAT-MULT(A22, B21, n/2)
      C22 = REC-MAT-MULT(A21, B12, n/2) + REC-MAT-MULT(A22, B22, n/2)
  return C
```

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

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$

Recursive case: $n > 1$.

- Dividing takes $\Theta(1)$ time if careful and $\Theta(n^2)$ if simply copying
- Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 8T(n/2)$
- Combining takes time $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices.

Recurrence is

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

Master method $\Rightarrow T(n) = \Theta(n^3)$

Volker Strassen

STRASSEN'S ALGORITHM FOR MATRIX MULTIPLICATION

The Idea

Make less recursive calls

- Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8

The Idea

Make less recursive calls

- Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8
- Will cost several additions of $n/2 \times n/2$ matrices, but just a constant more

$\Rightarrow$ can still absorb the constant factor for matrix additions into the $\Theta(n^2)$ term

The Idea

Make less recursive calls

- Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8
- Will cost several additions of $n/2 \times n/2$ matrices, but just a constant more

$\Rightarrow$ can still absorb the constant factor for matrix additions into the $\Theta(n^2)$ term

To obtain the recurrence

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

The Idea

Make less recursive calls

- Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8
- Will cost several additions of $n/2 \times n/2$ matrices, but just a constant more

$\Rightarrow$ can still absorb the constant factor for matrix additions into the $\Theta(n^2)$ term

To obtain the recurrence

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

Master method $\Rightarrow$ $T(n) = \Theta(n^{\log_2 7})$

Strassen's method

Divide each of A, B, C into four $n/2 \times n/2$ matrices: so that

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

Strassen's method

Divide each of A, B, C into four $n/2 \times n/2$ matrices: so that

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

Conquer: Calculate recursively 7 matrix multiplications, each of two $n/2 \times n/2$ matrices:

$$\begin{aligned} M_1 &:= (A_{11} + A_{22})(B_{11} + B_{22}) & M_5 &:= (A_{11} + A_{12})B_{22} \\ M_2 &:= (A_{21} + A_{22})B_{11} & M_6 &:= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_3 &:= A_{11}(B_{12} - B_{22}) & M_7 &:= (A_{12} - A_{22})(B_{21} + B_{22}) \\ M_4 &:= A_{22}(B_{21} - B_{11}) \end{aligned}$$

Strassen's method

Divide each of A, B, C into four $n/2 \times n/2$ matrices: so that

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

Conquer: Calculate recursively 7 matrix multiplications, each of two $n/2 \times n/2$ matrices:

$$\begin{aligned} M_1 &:= (A_{11} + A_{22})(B_{11} + B_{22}) & M_5 &:= (A_{11} + A_{12})B_{22} \\ M_2 &:= (A_{21} + A_{22})B_{11} & M_6 &:= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_3 &:= A_{11}(B_{12} - B_{22}) & M_7 &:= (A_{12} - A_{22})(B_{21} + B_{22}) \\ M_4 &:= A_{22}(B_{21} - B_{11}) \end{aligned}$$

Combine: Let

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 & C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 & C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

Analysis of Strassen's Method

Base case: $n = 1 \Rightarrow$ it takes time $\Theta(1)$

Recursive case: $n > 1$

- Dividing takes time $\Theta(n^2)$

Analysis of Strassen's Method

Base case: $n = 1 \Rightarrow$ it takes time $\Theta(1)$

Recursive case: $n > 1$

- Dividing takes time $\Theta(n^2)$
- Conquering makes 7 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 7T(n/2)$

Analysis of Strassen's Method

Base case: $n = 1 \Rightarrow$ it takes time $\Theta(1)$

Recursive case: $n > 1$

- Dividing takes time $\Theta(n^2)$
- Conquering makes 7 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 7T(n/2)$
- Combining takes time $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices.

Analysis of Strassen's Method

Base case: $n = 1 \Rightarrow$ it takes time $\Theta(1)$

Recursive case: $n > 1$

- Dividing takes time $\Theta(n^2)$
- Conquering makes 7 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 7T(n/2)$
- Combining takes time $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices.

Recurrence is

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

Analysis of Strassen's Method

Base case: $n = 1 \Rightarrow$ it takes time $\Theta(1)$

Recursive case: $n > 1$

- Dividing takes time $\Theta(n^2)$
- Conquering makes 7 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 7T(n/2)$
- Combining takes time $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices.

Recurrence is

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

Master method $\Rightarrow T(n) = \Theta(n^{\log_2 7})$

Notes about Strassen's method

- First to beat $\Theta(n^3)$ time
- Faster known today, method by Coppersmith and Winograd runs in time $O(n^{2.376})$ recently improved by Vassilevska Williams to $O(n^{2.3727})$.
- Big open problem how to multiply matrices in best way
- Naive method better for small instances because of hidden constants

Summary

- Divide-and-conquer simple but powerful algorithmic paradigm

Summary

- Divide-and-conquer simple but powerful algorithmic paradigm
- Merge-sort and maximum subarray both run in time $\Theta(n \log n)$

Summary

- Divide-and-conquer simple but powerful algorithmic paradigm
- Merge-sort and maximum subarray both run in time $\Theta(n \log n)$
- Strassen's algorithm for matrix multiplication in time $\Theta(n^{\log_2 7})$ where $\log_2 7 \approx 2.8$.

HEAPS AND HEAPSORT

Heapsort

Algorithm	worst-case running time	in-place
Insertion Sort	$\Theta(n^2)$	YES
Merge Sort	$\Theta(n \log n)$	NO

Heapsort:

Heapsort

Algorithm	worst-case running time	in-place
Insertion Sort	$\Theta(n^2)$	YES
Merge Sort	$\Theta(n \log n)$	NO

Heapsort:

- $O(n \lg n)$ worst case – like merge sort

Heapsort

Algorithm	worst-case running time	in-place
Insertion Sort	$\Theta(n^2)$	YES
Merge Sort	$\Theta(n \log n)$	NO

Heapsort:

- $O(n \lg n)$ worst case – like merge sort
- Sorts in place – like insertion sort

Heapsort

Algorithm	worst-case running time	in-place
Insertion Sort	$\Theta(n^2)$	YES
Merge Sort	$\Theta(n \log n)$	NO

Heapsort:

- $O(n \lg n)$ worst case – like merge sort
- Sorts in place – like insertion sort
- Combines the best of both algorithms

Heapsort

Algorithm	worst-case running time	in-place
Insertion Sort	$\Theta(n^2)$	YES
Merge Sort	$\Theta(n \log n)$	NO

Heapsort:

- $O(n \lg n)$ worst case – like merge sort
- Sorts in place – like insertion sort
- Combines the best of both algorithms

Uses a cool datastructure: heaps

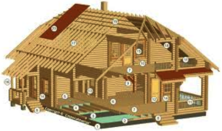
Data Structures = “Building Blocks”

Data Structures = “Building Blocks”

Algorithm

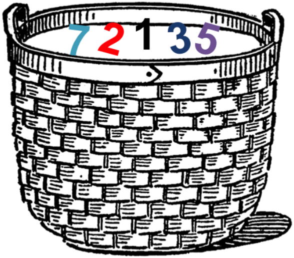
Data Structures = “Building Blocks”

Algorithm



Algorithm

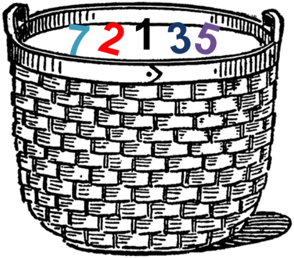
Data structures = dynamic sets of items



Data structure containing numbers

Data structures = dynamic sets of items

What kind of operations do we want to do?

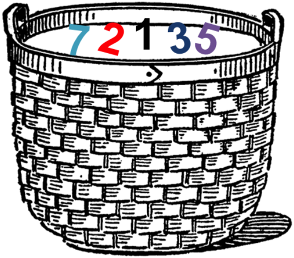


Data structure containing numbers

Data structures = dynamic sets of items

What kind of operations do we want to do?

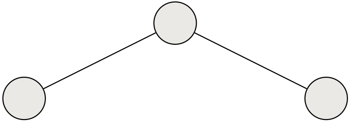
- Modifying operations: insertion, deletion, ...
- Query operations: search, maximum, minimum, ...



Data structure containing numbers

(Binary) heap data structure

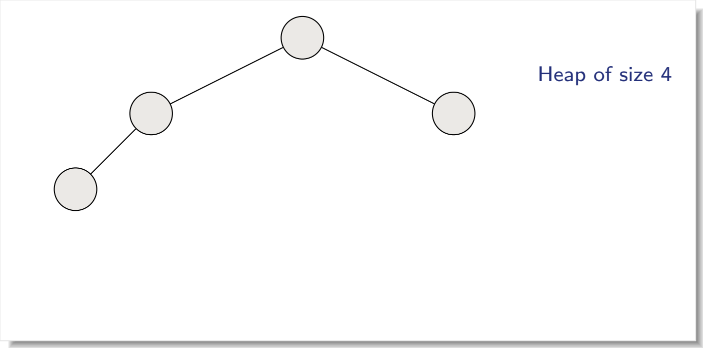
Heap A (not garbage-collected storage) is a nearly complete binary tree



Heap of size 3

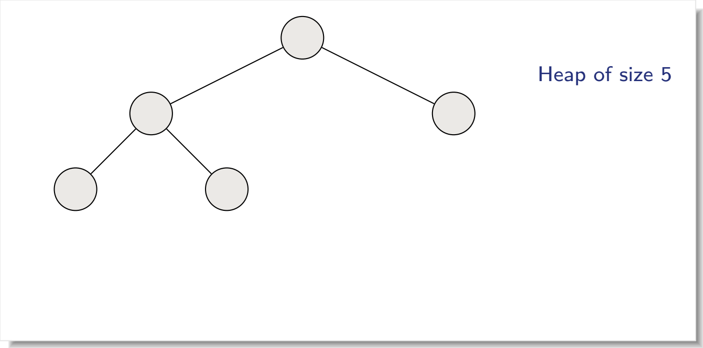
(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree



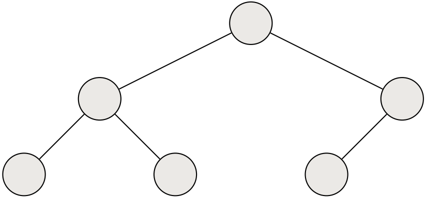
(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree



(Binary) heap data structure

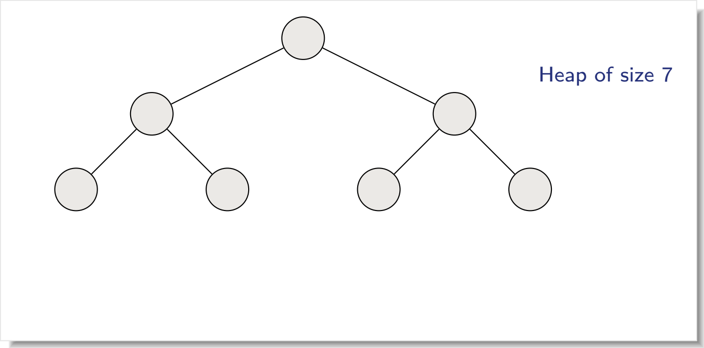
Heap A (not garbage-collected storage) is a nearly complete binary tree



Heap of size 6

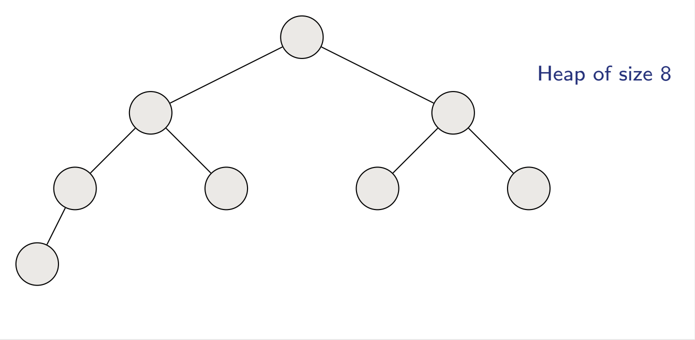
(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree



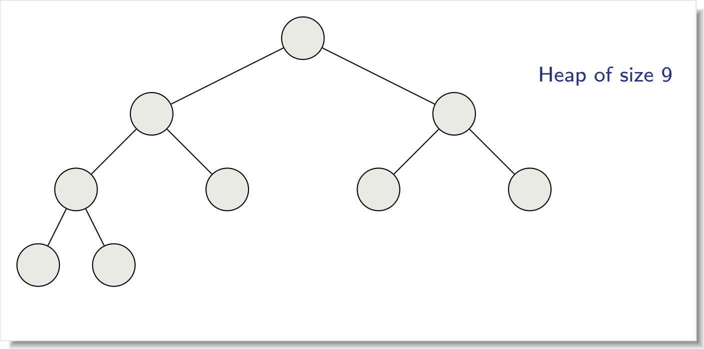
(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree



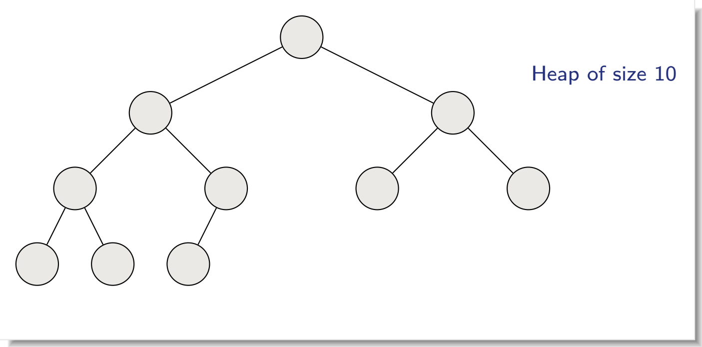
(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree



(Binary) heap data structure

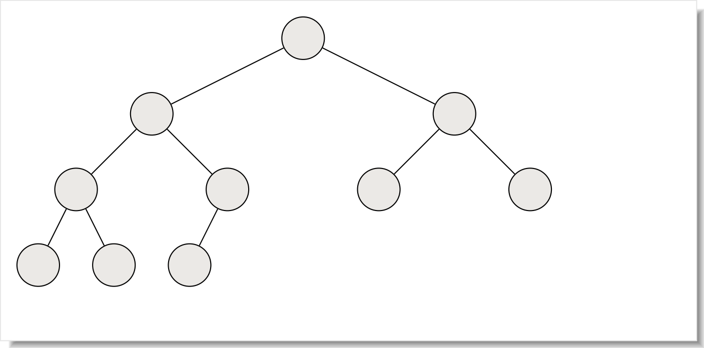
Heap A (not garbage-collected storage) is a nearly complete binary tree



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

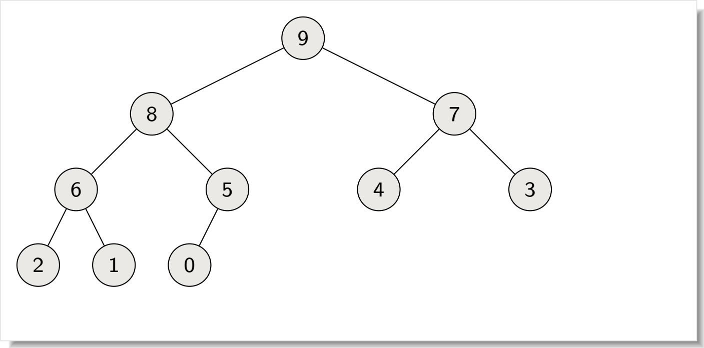
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

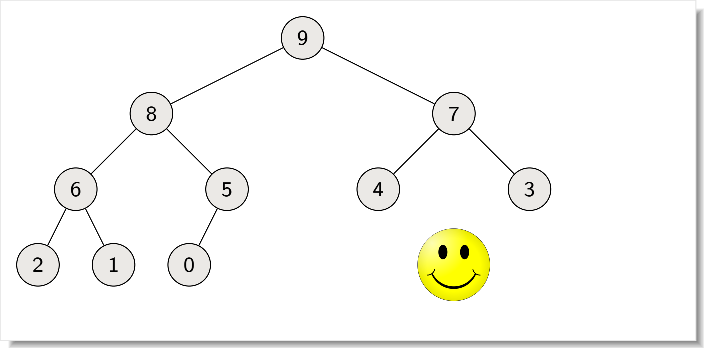
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

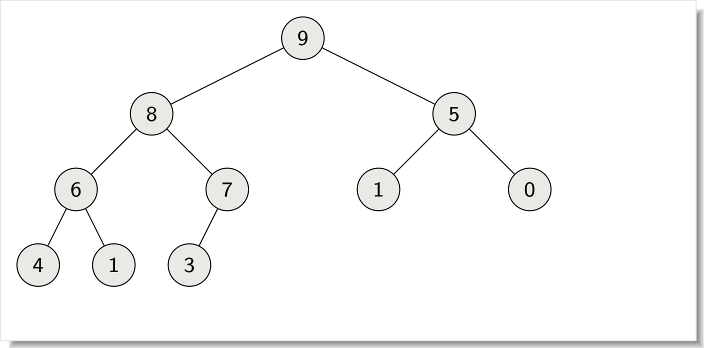
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

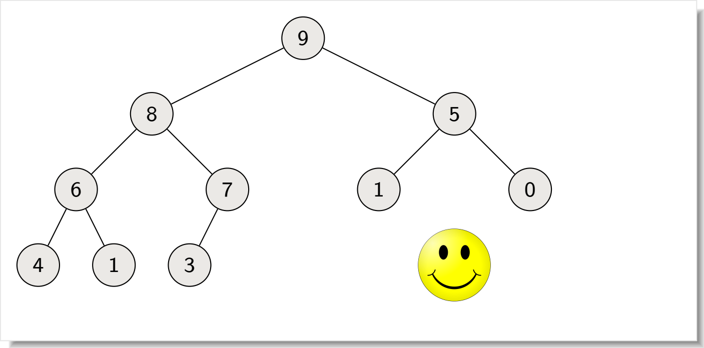
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

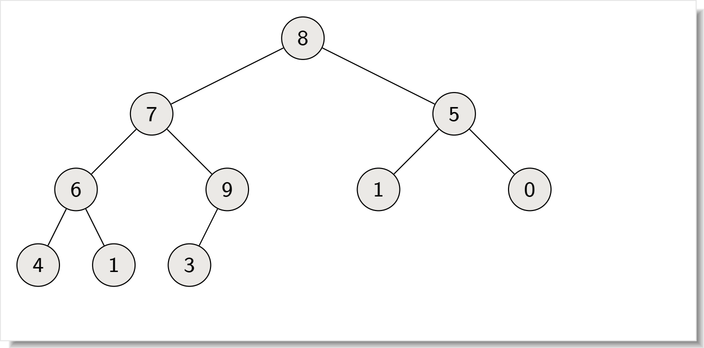
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

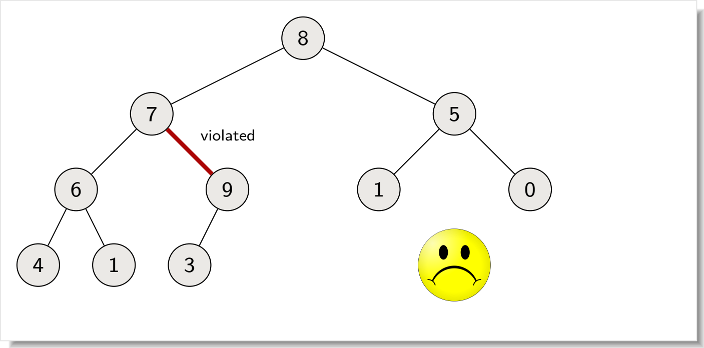
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

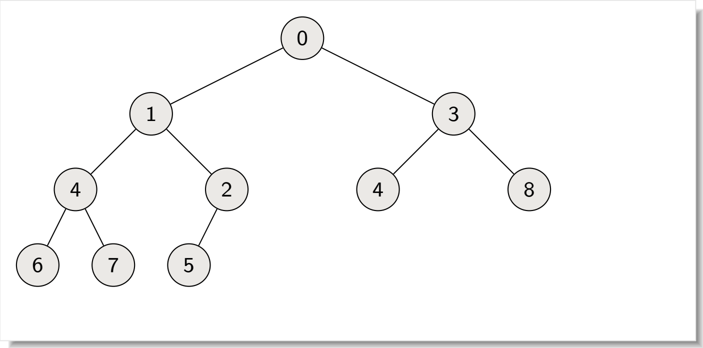
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

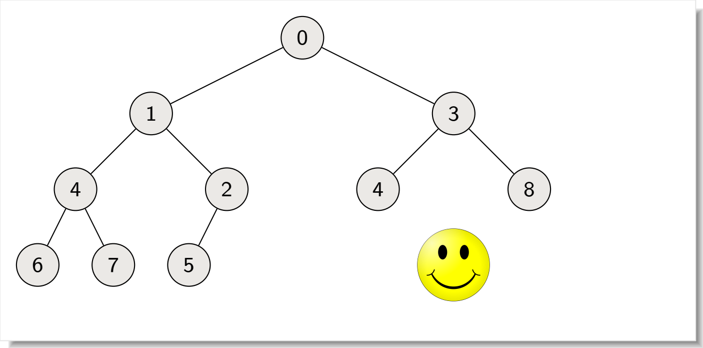
(Min)-Heap property: key of i's children is greater or equal to i's key



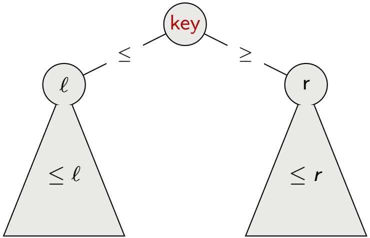
(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

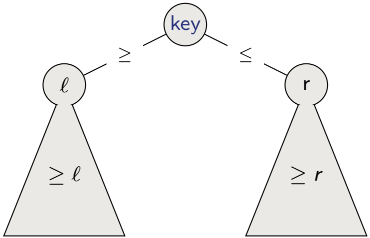
(Min)-Heap property: key of i's children is greater or equal to i's key



Max-Heap $\Rightarrow$ maximum element is the root

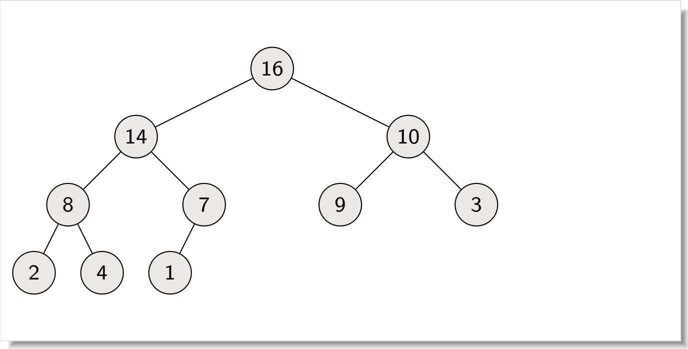


Min-Heap $\Rightarrow$ minimum element is the root



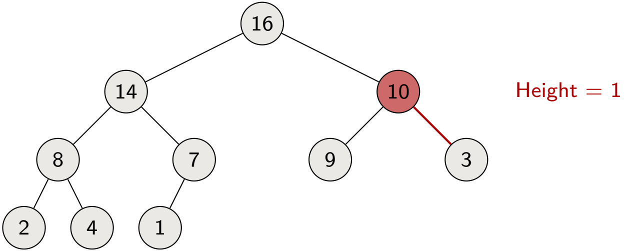
Height of a heap

Height of node = # of edges on a longest simple path from the node down to a leaf



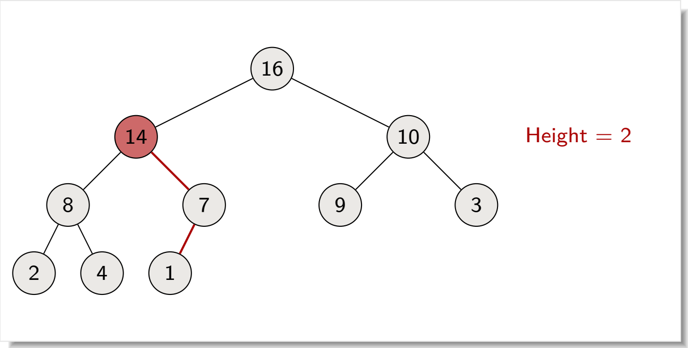
Height of a heap

Height of node = # of edges on a longest simple path from the node down to a leaf



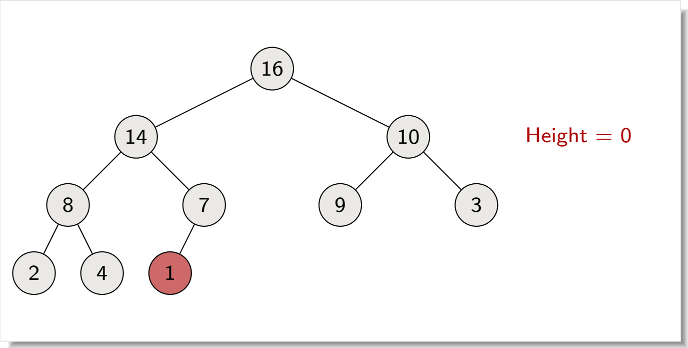
Height of a heap

Height of node = # of edges on a longest simple path from the node down to a leaf



Height of a heap

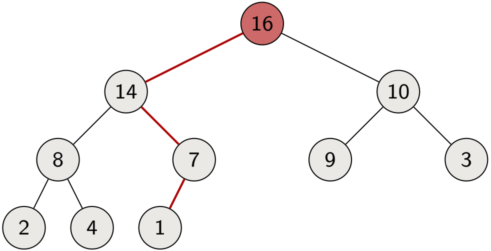
Height of node = # of edges on a longest simple path from the node down to a leaf



Height of a heap

Height of node = # of edges on a longest simple path from the node down to a leaf

Height of heap = height of root

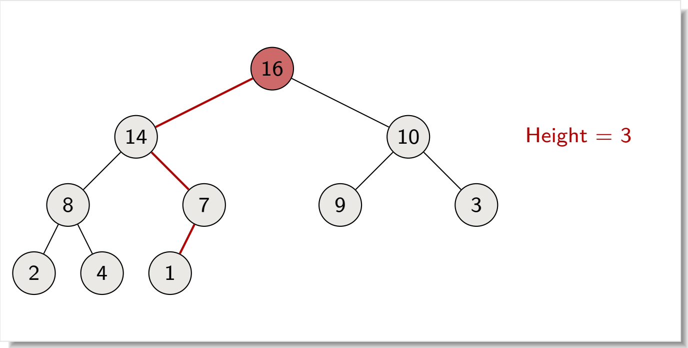


Height = 3

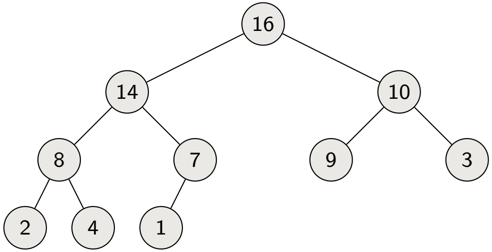
Height of a heap

Height of node = # of edges on a longest simple path from the node down to a leaf

Height of heap = height of root = $\Theta(\log n)$

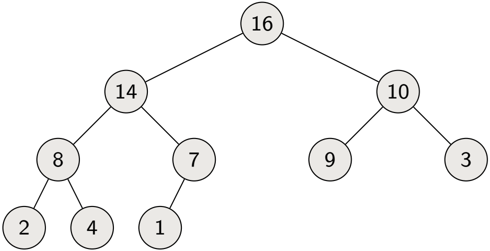


How to store a heap/tree?



How to store a heap/tree?

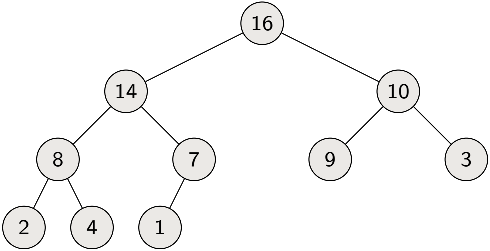
pointer to left and right children



How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array



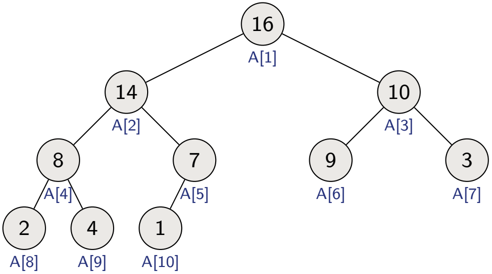
How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

A =

16	14	10	8	7	9	3	2	4	1



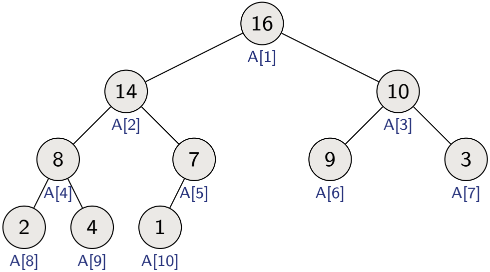
How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

A =

16	14	10	8	7	9	3	2	4	1



In this representation:

Root is $A[1]$

$\textsc{Left}(i) = ???$

$\textsc{Right}(i) = ???$

$\textsc{Parent}(i) = ???$

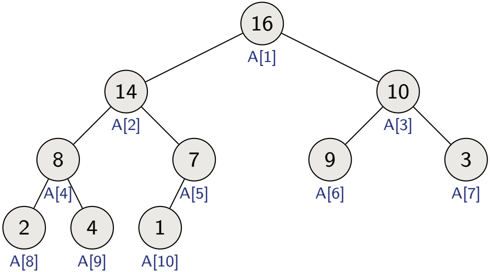
How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

A =

16	14	10	8	7	9	3	2	4	1



In this representation:

Root is $A[1]$

$\textsc{Left}(i) = 2i$

$\textsc{Right}(i) = ???$

$\textsc{Parent}(i) = ???$

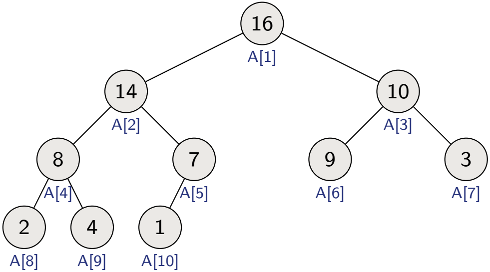
How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

A =

16	14	10	8	7	9	3	2	4	1



In this representation:

Root is $A[1]$

$\textsc{Left}(i) = 2i$

$\textsc{Right}(i) = 2i + 1$

$\textsc{Parent}(i) = ???$

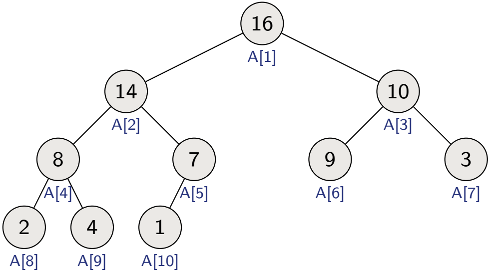
How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

$A =$

16	14	10	8	7	9	3	2	4	1



In this representation:

Root is $A[1]$

$\textsc{Left}(i) = 2i$

$\textsc{Right}(i) = 2i + 1$

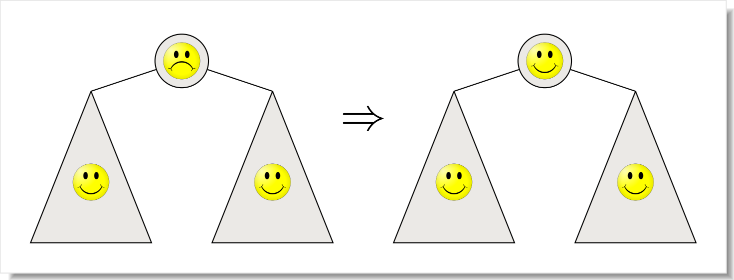
$\textsc{Parent}(i) = \lfloor i/2 \rfloor$

BUILDING AND MANIPULATING HEAPS

Maintaining the heap property

MAX-HEAPIFY is important for manipulating heaps:

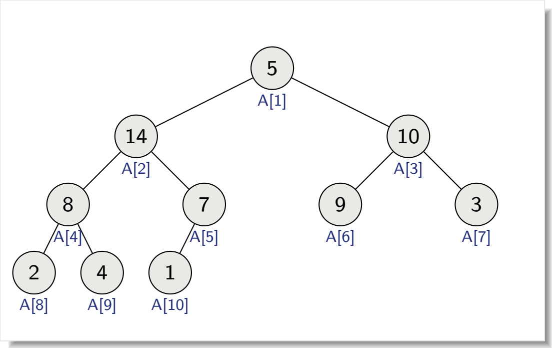
> Given an i such that the subtrees of i are heaps, it ensures that the subtree rooted at i is a heap satisfy the heap property



MAX-HEAPIFY(A, i, n)

Algorithm:

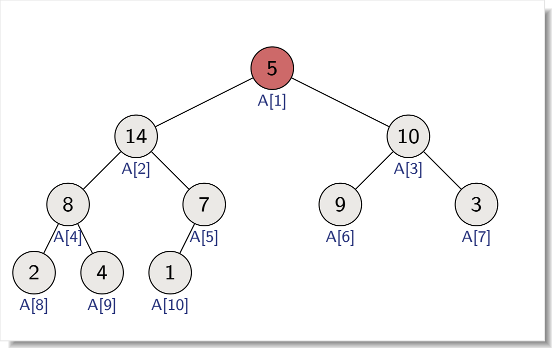
- Compare $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



MAX-HEAPIFY(A, i, n)

Algorithm:

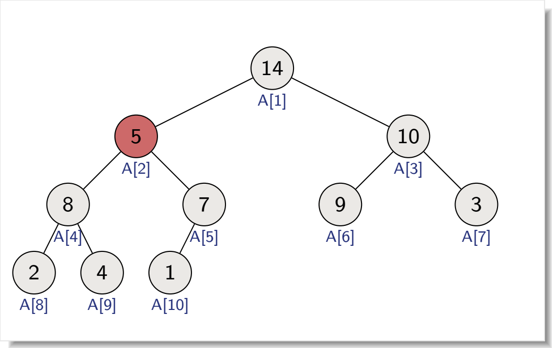
- Compare $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



MAX-HEAPIFY(A, i, n)

Algorithm:

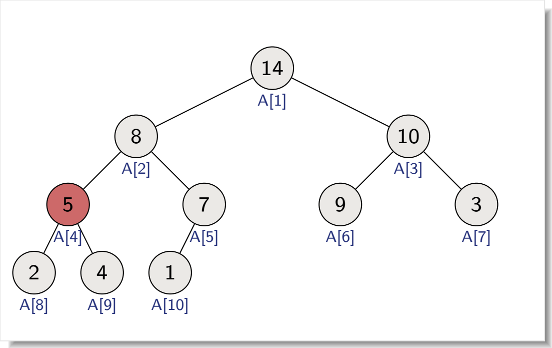
- Compare $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



MAX-HEAPIFY(A, i, n)

Algorithm:

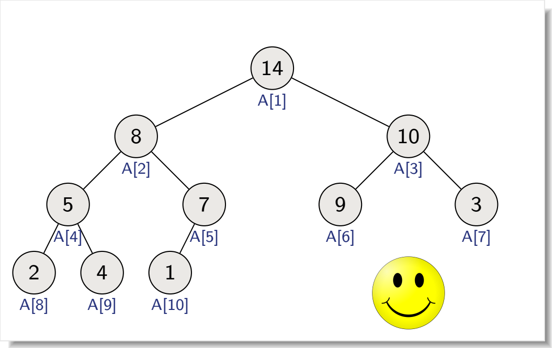
- Compare $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



MAX-HEAPIFY(A, i, n)

Algorithm:

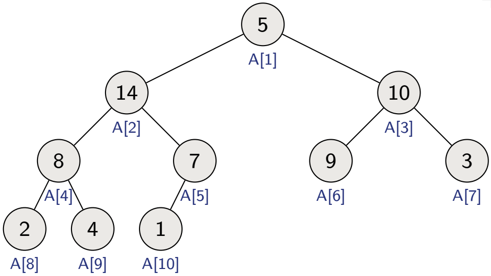
- Compare $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



Pseudo-code and analysis

MAX-HEAPIFY(A, i, n)

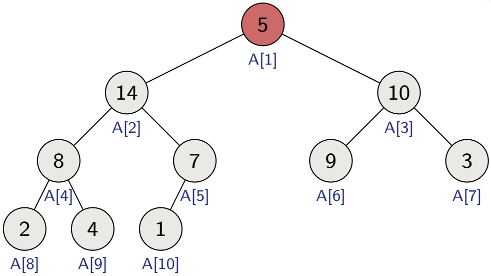
```
l = LEFT(i)
r = RIGHT(i)
if l ≤ n and A[l] > A[i]
    largest = l
else largest = i
if r ≤ n and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest, n)
```



Pseudo-code and analysis

MAX-HEAPIFY(A, i, n)

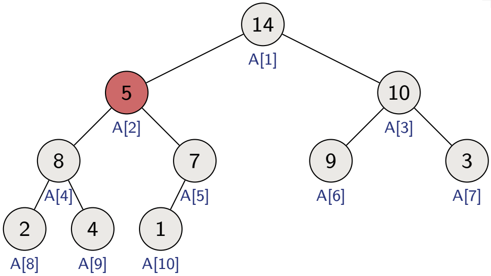
$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq n$ and $A[l] > A[i]$
  $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
  $largest = r$
if $largest \neq i$
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY$(A, largest, n)$



Pseudo-code and analysis

MAX-HEAPIFY(A, i, n)

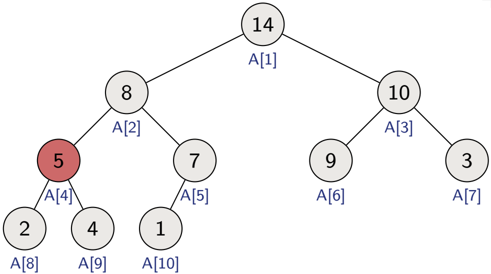
$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq n$ and $A[l] > A[i]$
  $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
  $largest = r$
if $largest \neq i$
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY$(A, largest, n)$



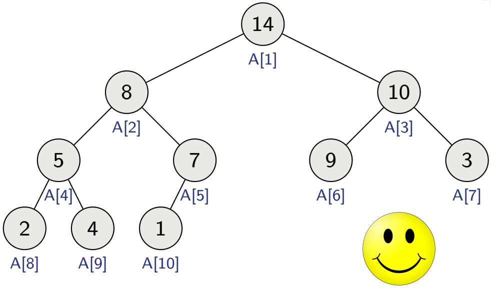
Pseudo-code and analysis

MAX-HEAPIFY(A, i, n)

$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq n$ and $A[l] > A[i]$
  $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
  $largest = r$
if $largest \neq i$
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY$(A, largest, n)$



Pseudo-code and analysis



MAX-HEAPIFY(A, i, n)

$l = $ LEFT(i)
$r = $ RIGHT(i)
if $l \leq n$ and $A[l] > A[i]$
  $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
  $largest = r$
if $largest \neq i$
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY$(A, largest, n)$

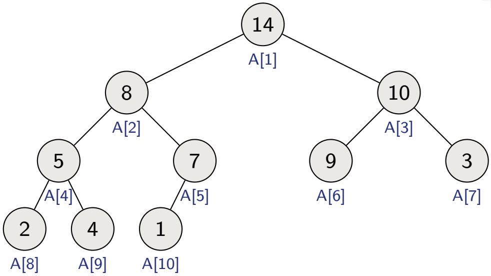
Pseudo-code and analysis

Running time?

Space?

MAX-HEAPIFY(A, i, n)

$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq n$ and $A[l] > A[i]$
  $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
  $largest = r$
if $largest \neq i$
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY$(A, largest, n)$



Pseudo-code and analysis

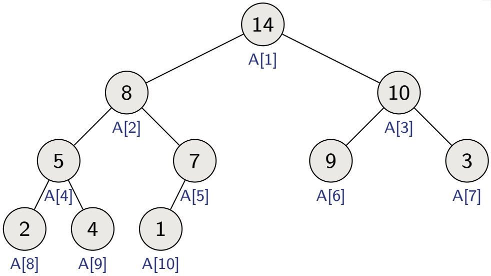
Running time?

$\Theta(\text{height of } i) = O(\log n)$

Space?

$\textsc{Max-Heapify}(A, i, n)$

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ n and A[l] > A[i]
    largest = l
else largest = i
if r ≤ n and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest, n)
```



Pseudo-code and analysis

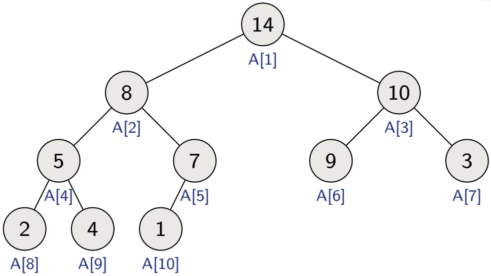
Running time?

$\Theta(\text{height of } i) = O(\log n)$

Space? $\Theta(n)$

MAX-HEAPIFY(A, i, n)

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ n and A[l] > A[i]
    largest = l
else largest = i
if r ≤ n and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest, n)
```

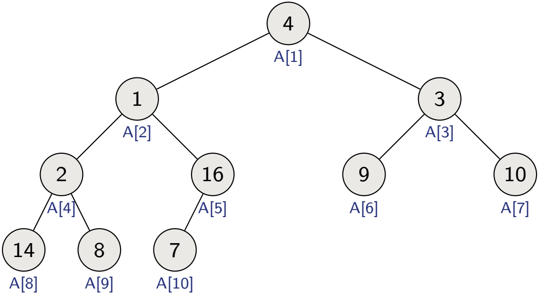


Building a heap

Build-Max-Heap(A, n)
for $i = \lfloor n/2 \rfloor$ **downto** 1
    Max-Heapify(A, i, n)

Given unordered array A of length n, Build-Max-Heap outputs a heap

4	1	3	2	16	9	10	14	8	7



Building a heap

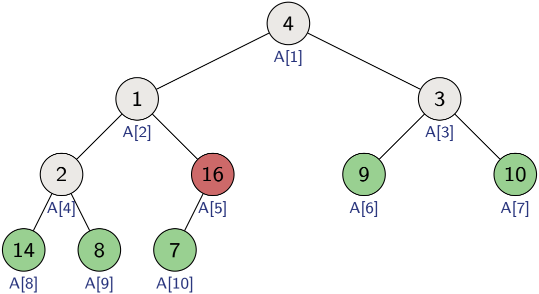
$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

  $\textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

4	1	3	2	16	9	10	14	8	7



Building a heap

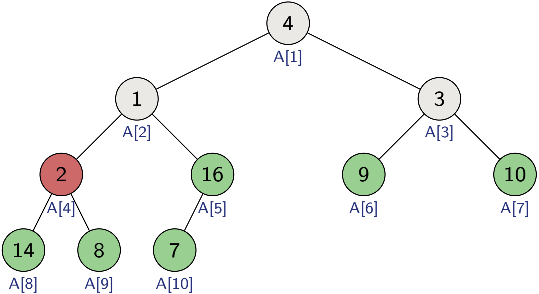
$\text{BUILD-MAX-HEAP}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

  $\text{MAX-HEAPIFY}(A, i, n)$

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

4	1	3	2	16	9	10	14	8	7



Building a heap

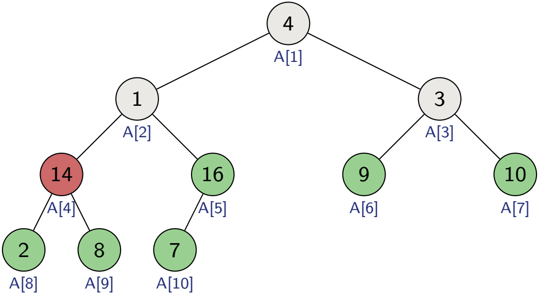
$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

$\quad \textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

4	1	3	14	16	9	10	2	8	7



Building a heap

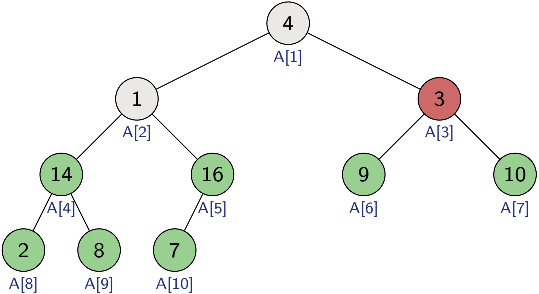
$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

  $\textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

4	1	3	14	16	9	10	2	8	7



Building a heap

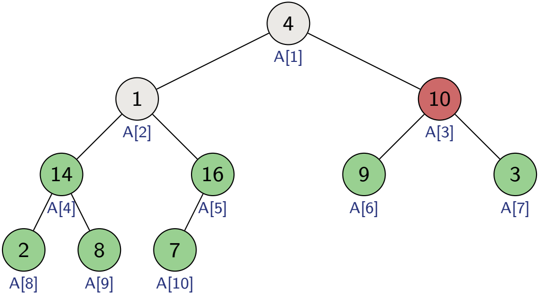
$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

$\quad \textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

4	1	10	14	16	9	3	2	8	7



Building a heap

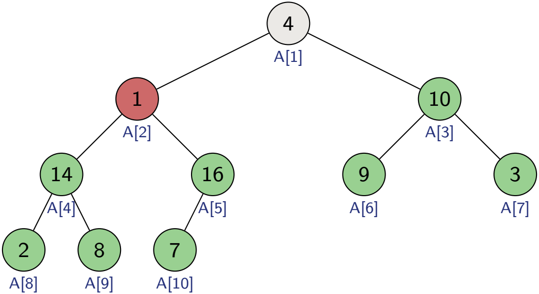
BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

4	1	10	14	16	9	3	2	8	7



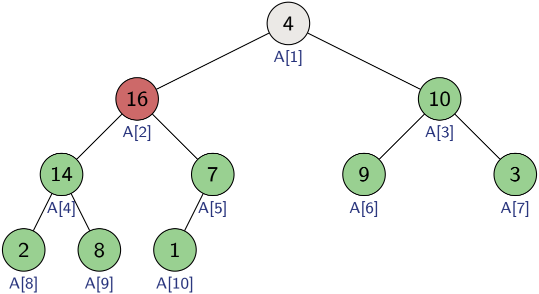
Building a heap

$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1
    $\textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

4	16	10	14	7	9	3	2	8	1



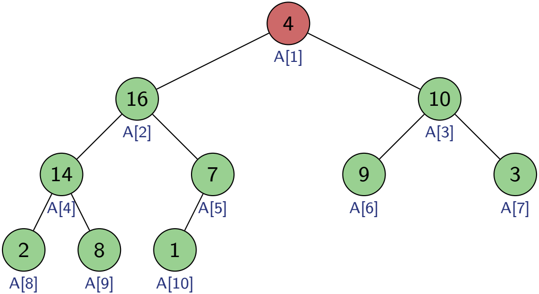
Building a heap

$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1
  $\textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

4	16	10	14	7	9	3	2	8	1



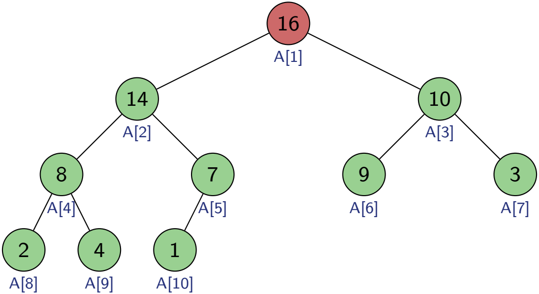
Building a heap

$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1
    $\textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

16	14	10	8	7	9	3	2	4	1



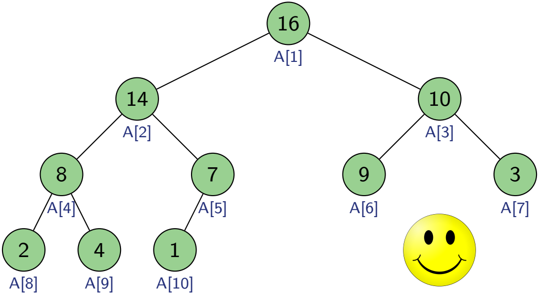
Building a heap

$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1
  $\textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

16	14	10	8	7	9	3	2	4	1



Analysis

$\textsc{Build-Max-Heap}(A, n)$
  for $i = \lfloor n/2 \rfloor$ **downto** 1
    $\textsc{Max-Heapify}(A, i, n)$

What is the worst-case running time of $\textsc{Build-Max-Heap}$?

Analysis

$\text{BUILD-MAX-HEAP}(A, n)$
for $i = \lfloor n/2 \rfloor$ **downto** 1
    $\text{MAX-HEAPIFY}(A, i, n)$

What is the worst-case running time of BUILD-MAX-HEAP?

Simple bound: $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$ in total

Analysis

$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

$\quad \textsc{Max-Heapify}(A, i, n)$

What is the worst-case running time of $\textsc{Build-Max-Heap}$?

Simple bound: $O(n)$ calls to $\textsc{Max-Heapify}$, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$ in total

Tighter analysis: Time to run $\textsc{Max-Heapify}$ is linear in the height of the node it's run on. Hence, the time is bounded by

$$\sum_{h=0}^{\lg n} \{\text{\# nodes of height } h\} O(h) = O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h}\right),$$

which is $O(n)$ since $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$.