Algorithms: Strassen's Algorithm for Matrix Multiplication + Heaps and Heapsort

Alessandro Chiesa, Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 6, 05.03.2025

DIVIDE-AND-CONQUER

Merge Sort

Maximum-Subarray Problem

Matrix Multiplication

Recall Maximum-Subarray Problem

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

divide

-2	-4	3	-1

5	7	-7	-1

solve recursively

solve recursively

-2	-4	3	-1

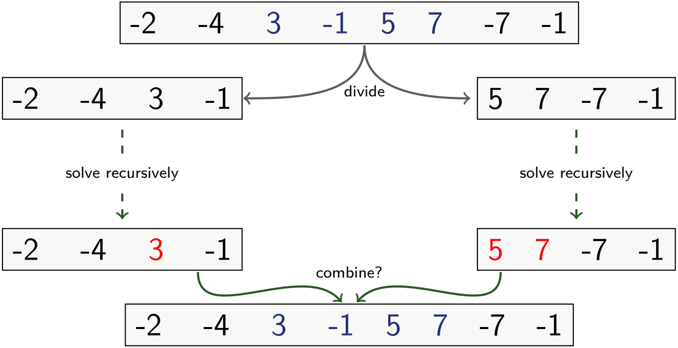
5	7	-7	-1

combine?

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

Solution

Also find the maximum subarray that crosses the midpoint!



Finding maximum subarray crossing midpoint

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

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

-2	-4	3	-1

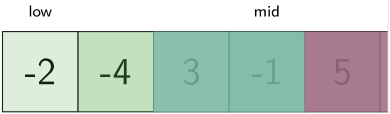
5	7	-7	-1

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

Crossing subarray

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)
```


```
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)
```

Analysis

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 $(cross\text{-}low, cross\text{-}high, cross\text{-}sum)$

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

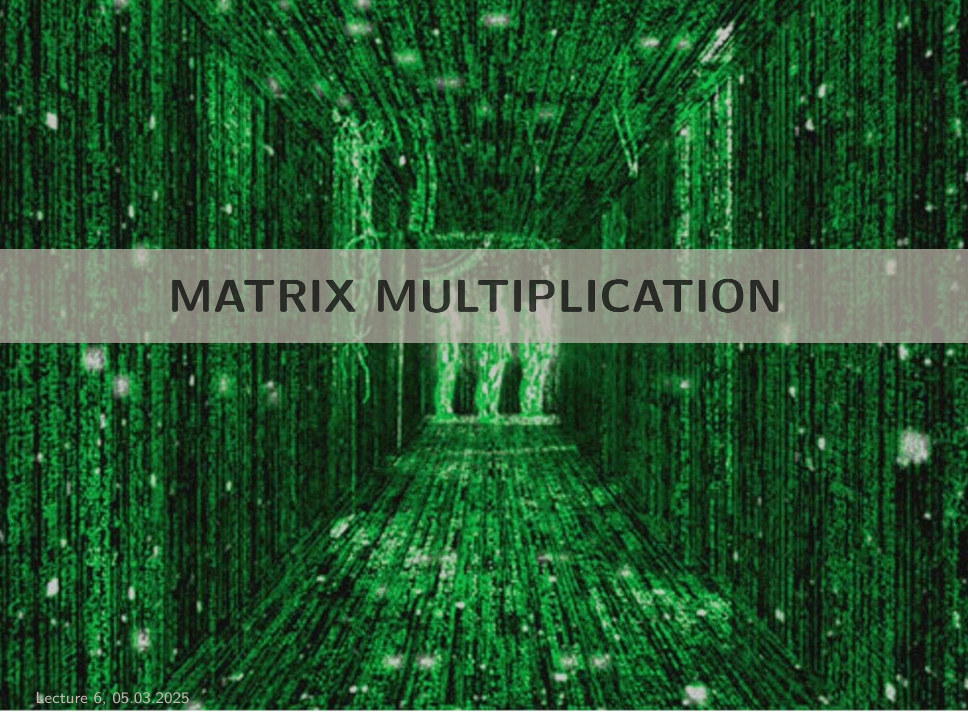
Conquer recursively solve two subproblems, each of size $n/2 \Rightarrow 2T(n/2)$

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



MATRIX MULTIPLICATION

Lecture 6, 05.03.2025

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 $C = A \cdot B$

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

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

How to multiply two matrices?

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

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

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

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

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

Naive Algorithm

Well simply multiply the matrices...

SQUARE-MAT-MULT(A, B, n)

let C be a new $n \times n$ matrix
for $i = 1$ **to** n
  for $j = 1$ **to** n
    $c_{ij} = 0$
    for $k = 1$ **to** n
      $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

Example:

$$\begin{pmatrix} ?35 & ?-13 \\ ?0 & ?7 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Naive Algorithm

Well simply multiply the matrices...

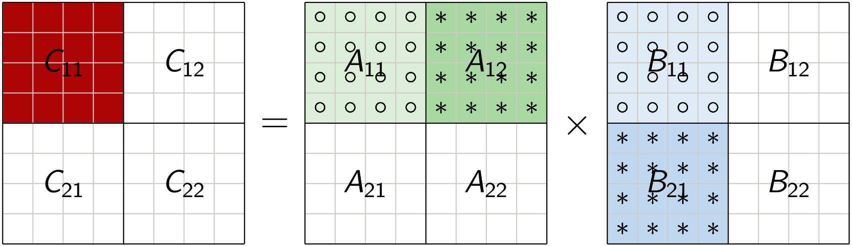
SQUARE-MAT-MULT(A, B, n)

let C be a new $n \times n$ matrix
for $i = 1$ **to** n
  for $j = 1$ **to** n
    $c_{ij} = 0$
    for $k = 1$ **to** n
      $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

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

Space? $\Theta(n^2)$

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

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

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

```
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
- 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

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

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 $\Theta(n^2)$
- 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

Karatsuba's algorithm

Problem: given two n-digit long integers x and y base b, find $x \cdot y$

Grade school algorithm: runtime $O(n^2)$

Can we do better than that?

Multiplying integers via divide and conquer

Divide: each number into two halves:

$$x = x_H \cdot b^{n/2} + x_L$$
$$y = y_H \cdot b^{n/2} + y_L$$

Then we have

$$\begin{aligned} x \cdot y &= (x_H \cdot b^{n/2} + x_L) \cdot (y_H \cdot b^{n/2} + y_L) \\ &= x_H \cdot y_H \cdot b^n + (x_H y_L + x_L y_H) \cdot b^{n/2} + x_L y_L \end{aligned}$$

Runtime?

Naive approach: compute $x_H \cdot y_H, x_H \cdot y_L, x_L \cdot y_H, x_L \cdot y_L$

All additions take $\Theta(n)$

$T(n) = 4T(n/2) + \Theta(n)$
$T(n) = \Theta(n^2)$

Multiplying integers via divide and conquer

Divide: each number into two halves:

$$x = x_H \cdot b^{n/2} + x_L$$
$$y = y_H \cdot b^{n/2} + y_L$$

Then we have

$$\begin{aligned} x \cdot y &= (x_H \cdot b^{n/2} + x_L) \cdot (y_H \cdot b^{n/2} + y_L) \\ &= x_H \cdot y_H \cdot b^n + (x_H y_L + x_L y_H) \cdot b^{n/2} + x_L y_L \end{aligned}$$

Runtime?

Naive approach: compute $x_H \cdot y_H, x_H \cdot y_L, x_L \cdot y_H, x_L \cdot y_L$
Suffices to compute $x_H y_H$, $x_L y_L$, $(x_H + x_L)(y_H + y_L)$

All additions take $\Theta(n)$

$T(n) = 3T(n/2) + \Theta(n)$
$T(n) = \Theta(n^{\log_2 3})$

Summary of divide-and-conquer

- 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

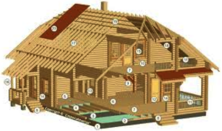
Heapsort:

- $O(n \lg n)$ 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”

Algorithm

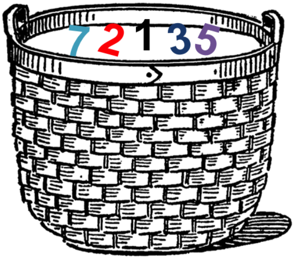


Algorithm

Data structures = dynamic sets of items

What kind of operations do we want to do?

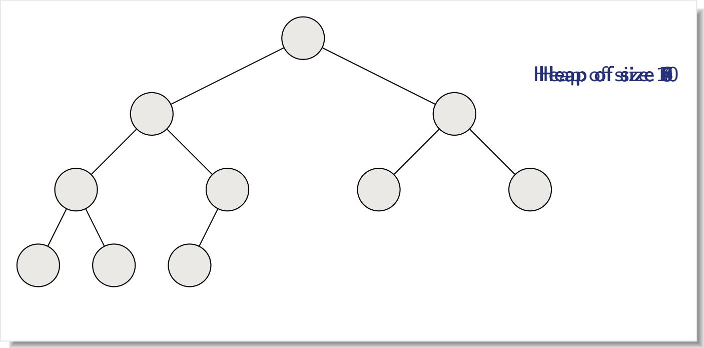
- Modifying operations: insertion, deletion, ...
- Query operations: search, maximum, minimum, ...



Data structure containing numbers

(Binary) heap data structure

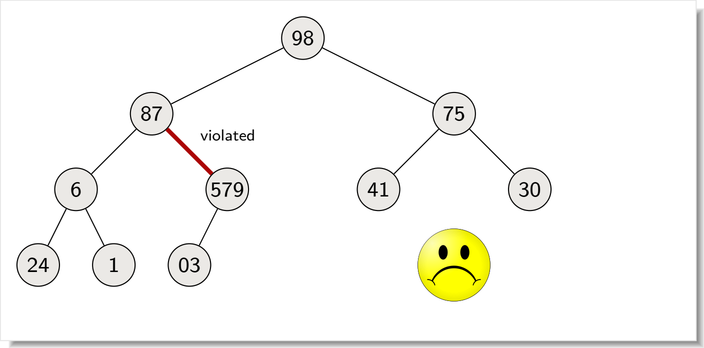
Heap A (not garbage-collected storage) is a nearly complete binary tree



(Binary) heap data structure

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

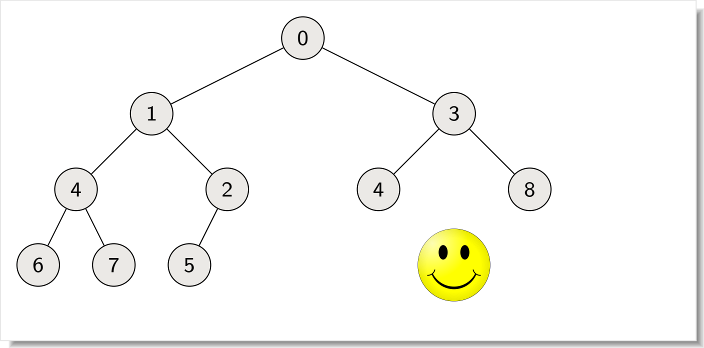
(Max)-Heap property: key of i's children is smaller or equal to i's key



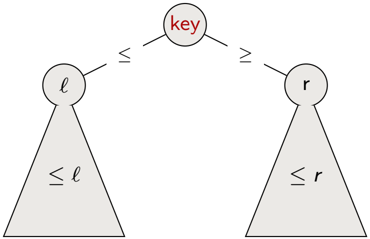
(Binary) heap data structure

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

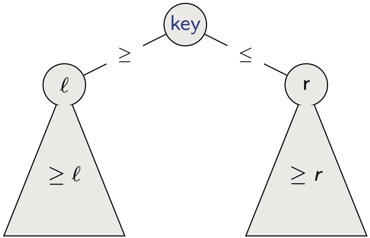
(Min)-Heap property: key of i's children is greater or equal to i's key



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

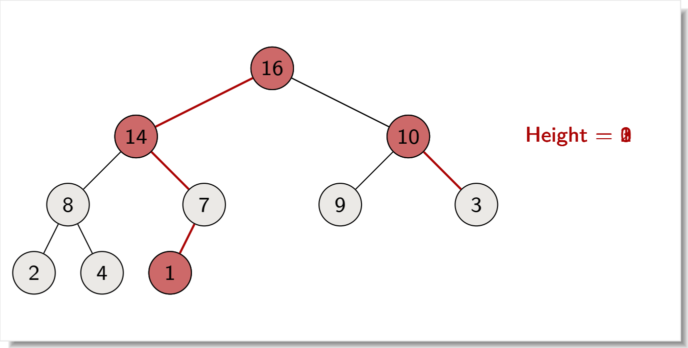


Min-Heap $\Rightarrow$ minimum element is the root



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

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



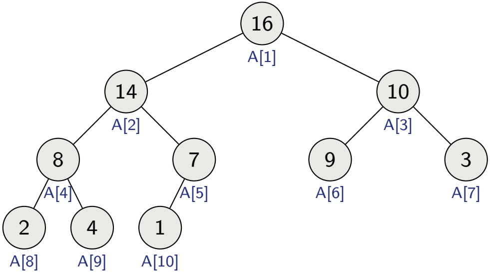
How to store a heap/tree?

~~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 $A[1]$

LEFT(i) =???= $2i$

RIGHT(i) =???= $2i + 1$

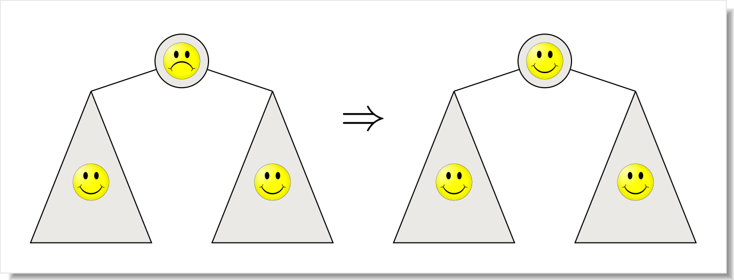
PARENT(i) =???= $\lfloor i/2 \rfloor$

BUILDING AND MANIPULATING HEAPS

Maintaining the heap property

MAX-HEAPIFY is important for manipulating heaps:

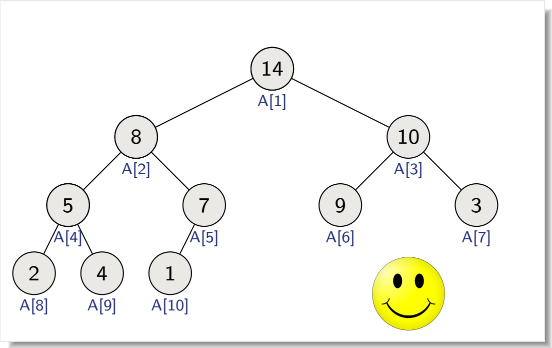
> Given an i such that the subtrees of i are heaps, it ensures that the subtree rooted at i is a heap satisfy the heap property



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



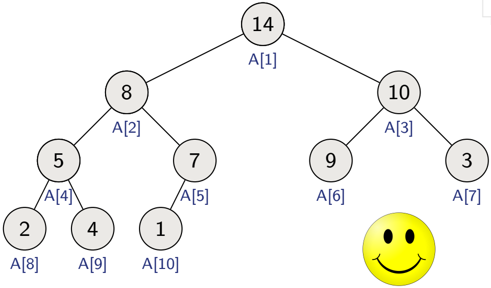
Pseudo-code and analysis

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

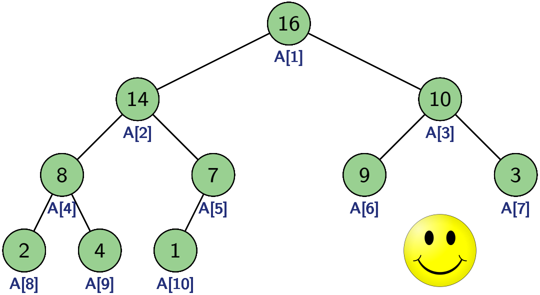
$\textsc{Build-Max-Heap}(A, n)$

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

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 $\textsc{Build-Max-Heap}$?

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

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

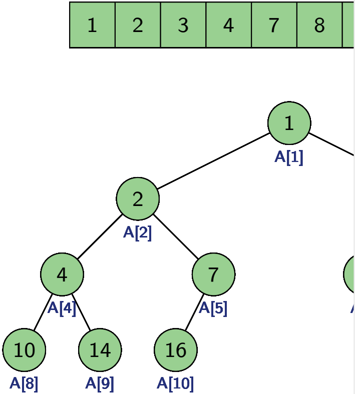
which is $O(n)$ since $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$.

HEAPSORT

The heapsort algorithm

- Builds a max-heap from the array
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array
- “Discard” this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root
- Repeat this “discarding” process until only one node (the smallest element) remains, and therefore is in the correct place in the array

Example



HEAPSORT(A, n)
 BUILD-MAX-HEAP(A, n)
 for $i = n$ **downto** 2
  exchange $A[1]$ with $A[i]$
  MAX-HEAPIFY$(A, 1, i-1)$

HEAPSORT(A, n)
 BUILD-MAX-HEAP(A, n)
 for $i = n$ **downto** 2
  exchange $A[1]$ with $A[i]$
  MAX-HEAPIFY$(A, 1, i-1)$

HEAPSORT(A, n)
 BUILD-MAX-HEAP(A, n)
 for $i = n$ **downto** 2
  exchange $A[1]$ with $A[i]$
  MAX-HEAPIFY$(A, 1, i-1)$

HEAPSORT(A, n)
 BUILD-MAX-HEAP(A, n)
 for $i = n$ **downto** 2
  exchange $A[1]$ with $A[i]$
  MAX-HEAPIFY$(A, 1, i-1)$

Analysis of Heapsort

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i - 1)
```

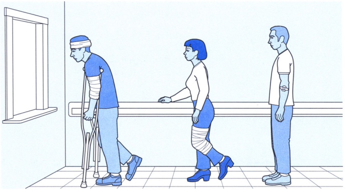
- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps are nice, next lecture we will see how to use them for priority queues and we will also start with other data structures

HEAP IMPLEMENTATION OF PRIORITY QUEUE



Priority Queue

- Maintains a dynamic set S of elements
- Each set element has a key — an associated value that regulates its importance

What kind of operations do we want to do?

INSERT(S,x): inserts element x into S

MAXIMUM(S): returns element of S with largest key

EXTRACT-MAX(S): removes and returns element of S with largest key

INCREASE-KEY(S,x,k): increases value of element x's key to k; assume $k \geq x$'s current key value

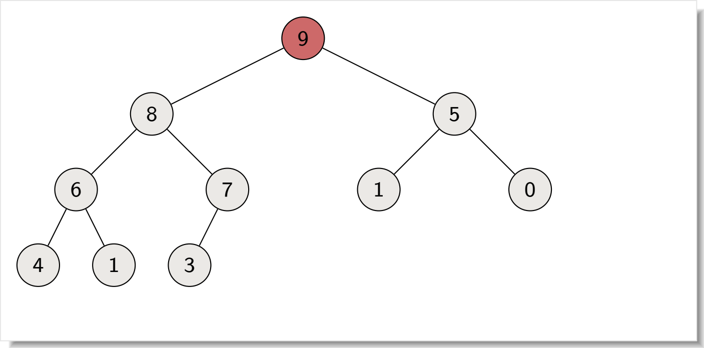
Example max-priority queue application: schedule jobs on shared computer

Heaps efficiently implement priority queues

Finding maximum element

HEAP-MAXIMUM(A)
return $A[1]$

Simply return the root in time $\Theta(1)$



Extracting maximum element

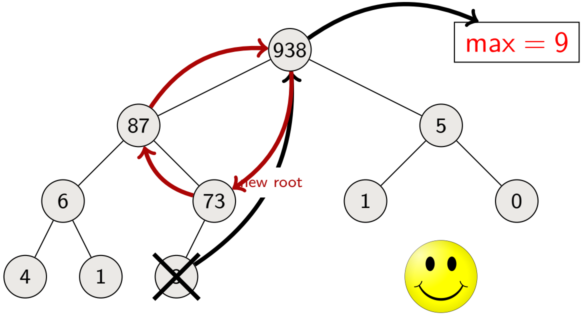
Heap-Extract-Max(A, n)

if $n < 1$
  error "heap underflow"
$max = A[1]$
$A[1] = A[n]$
$n = n - 1$
Max-Heapify$(A, 1, n)$
return max

Analysis: C

assignments plus
time for Max-Heapify

Henc



Increasing key value

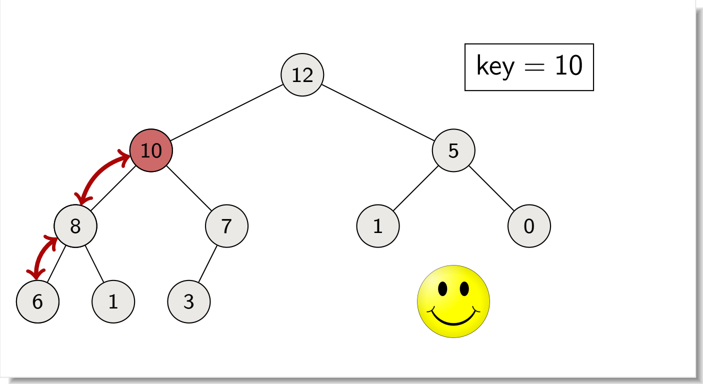
Given a heap A, index i, and new value *key*

HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

Upward path from node i has
lengt

Henc



Inserting into the heap

Given a new *key* to insert into heap

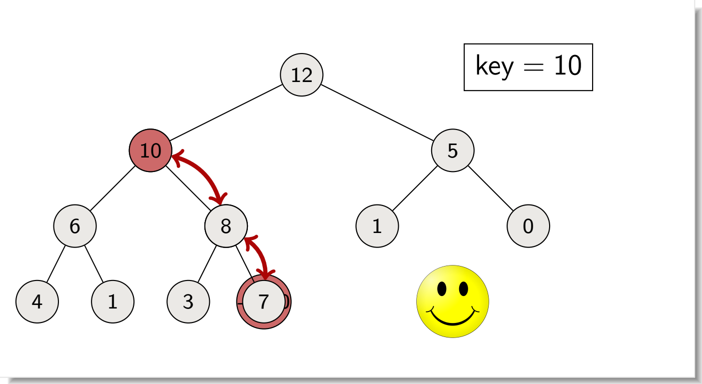
MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)

Analysis: Constant

time assignments

Henc



Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  Insert(S,x): $O(\lg n)$
  Maximum(S): $O(1)$
  Extract-Max(S): $O(\lg n)$
  Increase-Key(S,x,k): $O(\lg n)$
- Min-priority queues are implemented with min-heaps similarly