Advanced Algorithm Design and Analysis
CSc 140

Midterm 1 Review

Instructor: Hady Ahmady Phoulady

General Advice for Study

- **Understand** how the algorithms are working
  - Work through the examples we did in class
  - "Narrate" for yourselves the main steps of the algorithms in a few sentences
- Know **when** or **for what problems** the algorithms are applicable
- **Do not memorize** algorithms

Asymptotic notations

- A way to describe behavior of functions in the limit
  - Abstracts away low-order terms and constant factors
  - How we indicate running times of algorithms
  - Describe the running time of an algorithm as n goes to ∞
- O notation: asymptotic "less than": $f(n)$ "$\leq$" $g(n)$
- Ω notation: asymptotic "greater than": $f(n)$ "$\geq$" $g(n)$
- Θ notation: asymptotic "equality": $f(n)$ "$=$" $g(n)$

Exercise

- Order the following 6 functions in increasing order of their growth rates:
  - $n \log n$, $\log^2 n$, n^2, 2^n, $\sqrt{n}$, n.

$\log^2 n$

$\sqrt{n}$

n

$n \log n$

n^2

2^n

Running Time Analysis

- *Algorithm Loop1(n)*

  p=1

  for i = 1 to 2n

  p = p*i

  O(n)

- *Algorithm Loop2(n)*

  O(n^2)

  p=1

  for i = 1 to n^2

  p = p*i

Running Time Analysis

Algorithm Loop3(n)

```
s=0
for i = 1 to n
    for j = 1 to i
        s = s + I
```

$O(n^2)$

Recurrences

Def.: Recurrence = an equation or inequality that describes a function in terms of its value on smaller inputs, and one or more base cases

- Recurrences arise when an algorithm contains recursive calls to itself
- Methods for solving recurrences
  - Substitution method
  - Iteration method
  - Recursion tree method
  - Master method
- Unless explicitly stated choose the simplest method for solving recurrences

Master's method

- Used for solving recurrences of the form:

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

where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Compare $f(n)$ with $n^{\log_b a}$:

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:

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

regularity condition

Exercise

- $T(n) = 2T\left(\frac{n}{3}\right) + \frac{n}{2}$
  - Solve using the recursion tree method
  - Solve using the substitution method
  - Solve using the iteration method
  - Solve using the Master theorem

O(n)

Sorting

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

- Bubble Sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

Analysis of Insertion Sort

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1 ≈ $n^2/2$ comparisons	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1 ≈ $n^2/2$ exchanges	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

Analysis of Bubble-Sort

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A]

do for j ← length[A] **downto** i + 1

Comparisons: ≈ $n^2/2$

do if A[j] < A[j -1]

then exchange A[j] ↔ A[j-1]

Exchanges: ≈ $n^2/2$

$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n}(n-i+1) + c_3 \sum_{i=1}^{n}(n-i) + c_4 \sum_{i=1}^{n}(n-i)$$

$$= \Theta(n) + (c_2 + c_2 + c_4) \sum_{i=1}^{n}(n-i)$$

$$\approx \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

$T(n) = \Theta(n^2)$

Sorting

- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: $\Theta(n\lg n)$

Analysis of Selection Sort

Alg.: SELECTION-SORT(*A*)	cost	times
n ← length[A]	c_1	1
for j ← 1 **to** n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 **to** n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] $\boxed{\leftrightarrow}$ A[smallest]	c_7	n-1

≈$n^2/2$ comparisons

≈n exchanges

Analyzing Divide and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - $T(n)$ – running time on a problem of size n
  - **Divide** the problem into a subproblems, each of size n/b: takes $D(n)$
  - **Conquer** (solve) the subproblems $aT(n/b)$
  - **Combine** the solutions $C(n)$

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

MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$

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

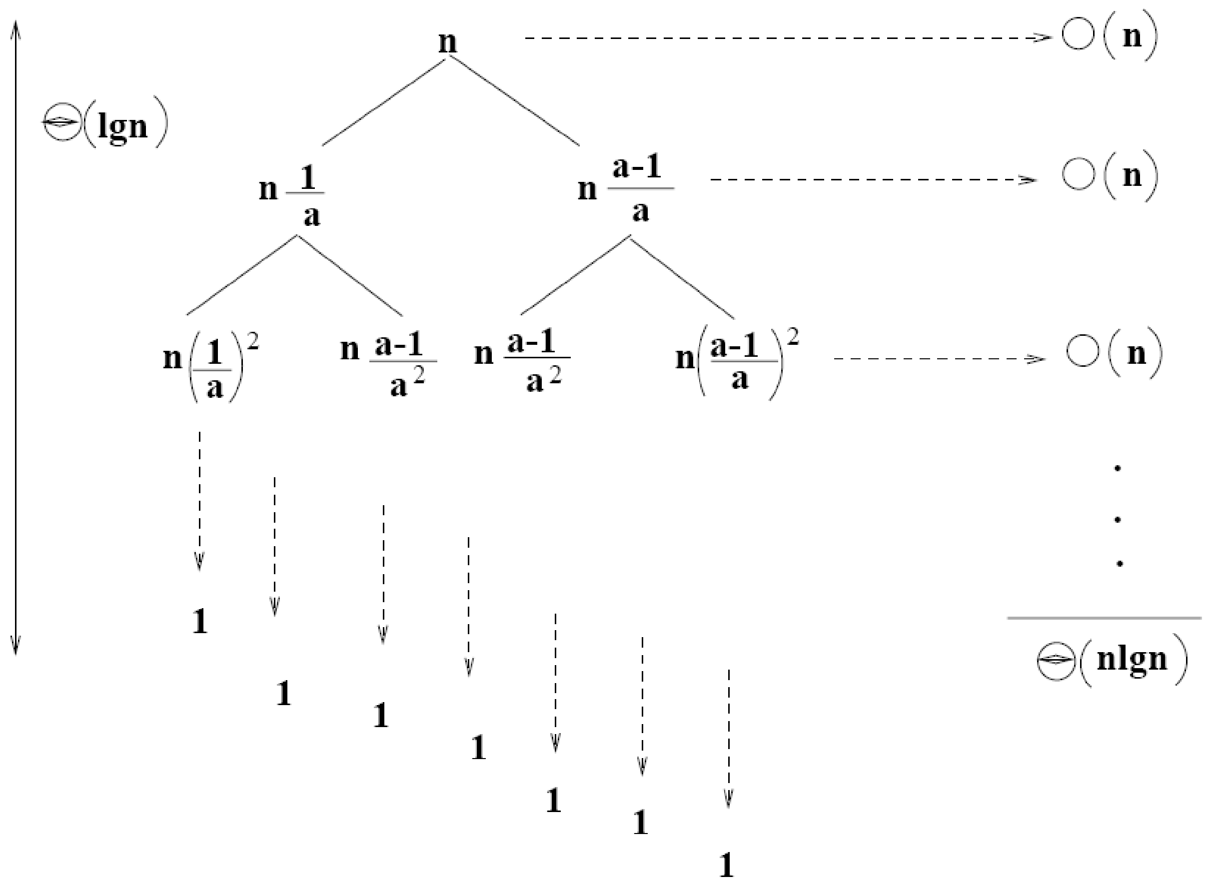
Quicksort

- Quicksort
  - Idea: Partition the array A into 2 subarrays A[p..q] and A[q+1..r], such that each element of A[p..q] is smaller than or equal to each element in A[q+1..r]. Then sort the subarrays recursively.
  - Design approach: Divide and conquer
  - Sorts in place: Yes
  - Best case: $\Theta(n\lg n)$
  - Worst case: $\Theta(n^2)$
- Partition
  - Running time $\Theta(n)$
- Randomized Quicksort $\Theta(n\lg n)$ – on average
  $\Theta(n^2)$ – in the worst case

Analysis of Quicksort

- Any $((a-1)n/a : n/a)$ splitting:

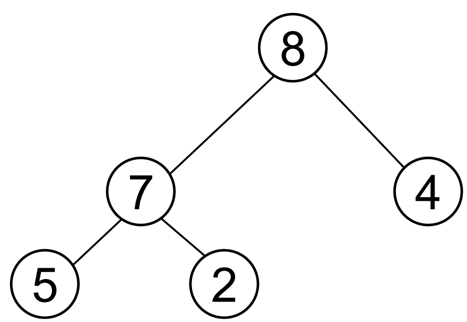
  ratio=$((a-1)n/a)/(n/a) = a-1$ it is a constant !!



The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

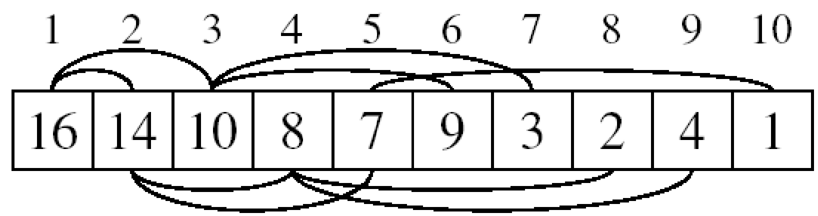
$$\text{Parent}(x) \geq x \qquad \text{(max heap)}$$

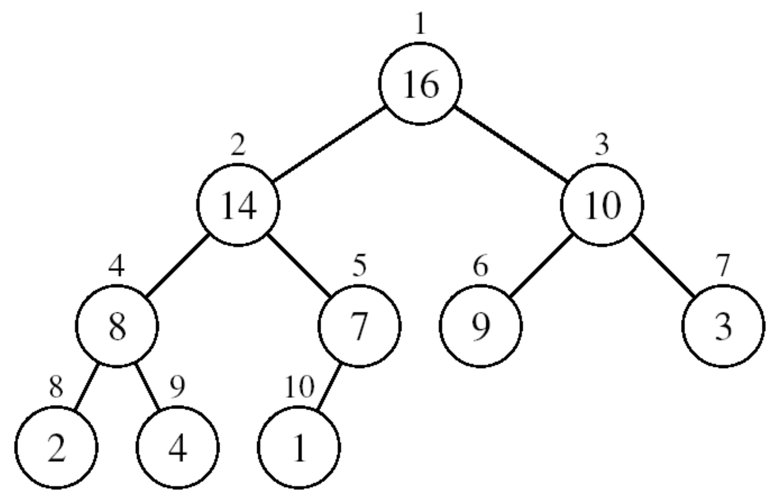


Heap

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves
- The root is the max/min element of the heap





A heap is a binary tree that is filled in order

Operations on Heaps (useful for sorting and priority queues)

- MAX-HEAPIFY $O(\lg n)$
- BUILD-MAX-HEAP $O(n)$
- HEAP-SORT $O(n\lg n)$
- MAX-HEAP-INSERT $O(\lg n)$
- HEAP-EXTRACT-MAX $O(\lg n)$
- HEAP-INCREASE-KEY $O(\lg n)$
- HEAP-MAXIMUM $O(1)$
- You should be able to show how these algorithms perform on a given heap, and tell their running time