

CSE 101, Winter 2002

Design and Analysis of Algorithms

Instructor: Andrew B. Kahng, <http://vlsicad.ucsd.edu>

Email: abk@ucsd.edu

Telephone: 858-822-4884 office, 858-353-0550 cell

Office Hours: MW noon-2pm, TuTh 8:30-9:30pm

Office: 3802 AP&M

Class webpage: <http://vlsicad.ucsd.edu/courses/cse101/>

Goals of Course

- Introduction to design and analysis of algorithms
- “Problem-solving”
- Classic Problems
 - Sorting, Path-Finding, String-Matching, Arithmetic, ...
- Tools
 - Recurrence Relations, Some Counting Techniques, Reduction, Probabilistic Analysis, NP-Completeness, ...
- Frameworks
 - Divide-and-Conquer, Greed, Dynamic Programming, Branch-and-Bound, Heuristics, ...
- Classic dilemmas
 - Ordering of material ? *We'll see a reasonable choice*
 - “Execution” or “Innovation” ? *More emphasis on latter*
 - Scope: Very broad – may feel like drinking from a firehose, but the material is coherent *You need to keep up – and I assume you are keeping up (notes, extra questions, readings, HW...)*

Course Logistics

- Textbook: Cormen et al., 2nd edition (2001)
- Lecture Room: likely to change (→ Center 101?)
- Discussion – 3 sections have been added:
 - Wed 09:05 A 09:55 A WLH 2204 (76 seats)
 - Fri 10:10 A 11:00 A CSB 002 (120 seats)
 - Wed 11:15 A 12:05 P CENTR 212 (146 seats)
 - **May develop material that is not covered in lecture** (e.g., solution of recurrence relations, structure of induction proofs, etc.)
- Four TAs: Joe Drish, Victor Gidofalvi, Eric Hall, Cynthia Sheng
- Homework: ~ 6 assignments with 7-10 day lead times
 - First assignment posted on Thursday, January 10 – check website!
 - Hard due dates (solutions posted on the web); **zero credit if late**
 - **THERE WILL BE IN-CLASS QUIZZES (2-3) + EC Problems**
- Grading
 - **40% HW AND QUIZZES (do not violate academic conduct rules)**, 25% in-class midterm (Feb 7), 35% final (March 22, 7pm)

Course Behavior

- Basic Courtesy
 - Cell phones and other distractions must be turned OFF
- Range of abilities
 - This is a required course
 - Everyone should learn something from it
 - If you are bored or know material already, realize that not everyone else may be in the same position

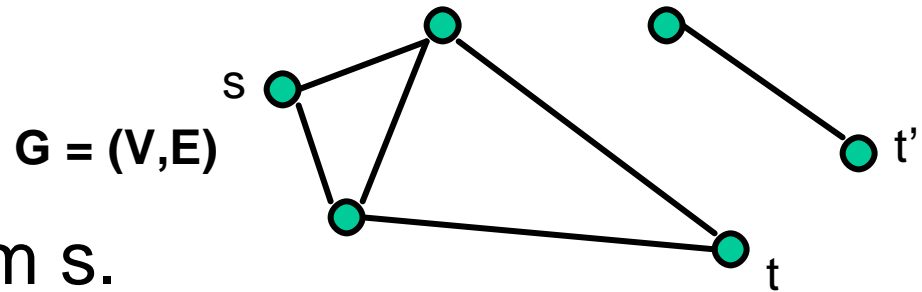
Introduction to Lecture

- Assigned reading:
 - Chapters 1-4 (background, asymptotic growth rates, recurrences)
 - Chapters 6, 7 (heapsort, quicksort)
- This week
 - Criteria for algorithms (correctness, efficiency, ...) and some example analyses
 - Asymptotic growth of functions (“Big-O notation”)
 - Examples of recurrences
 - A Sorting Excursion
 - Lower Bounds (worst- and average-case)
 - Practical issues
 - Simple methods
 - Closing the Gap (Heapsort, and D/Q Framework (Mergesort, Quicksort))
 - Example of Randomized Analysis
- Next week
 - Selection
 - The D/Q Framework

Introduction: What Is This Course About ?

- An algorithm is a method for solving a problem (on a computer)
- **Problem:** “Given fraction m/n , reduce to lowest terms.”
- An algorithm must be effective
 - In particular: give correct answer, halt
- **Problem:** “Given undirected graph $G = (V, E)$ and vertices $s, t \in V$, is there a path in G from s to t ?”
- *State an algorithm for this problem*
- *Other problem examples:*
 - *Given a set of points in the plane, find the closest pair*
 - *Given a set of points in the plane, find the shortest tour that visits each point exactly once (“Traveling Salesman Problem”)*

Undirected s-t Connectivity



- A1: BFS, DFS from s.
- A2: Take a random walk in G , starting at s.
 - *Is this an algorithm? (Does it halt?)*
- A3: Take a random walk in G for $5n^3$ steps starting at s ($n = |V|$); return NO iff we don't visit t.
 - *Is this an algorithm?*
 - *Does it “almost always” return the correct answer?*
- Do A3, A1 differ in terms of resources used?
 - A3 “trades” time for space, is “memoryless”.
 - A3: probabilistic effectiveness.

Course Overview

- Themes: Problem solving, “Spirit of Computing”, real-world necessity
- Examples of “real-world necessity”
 - DNA Sequencing
 - Evolutionary Trees (edit distance, Steiner trees...)
 - Finding homologues, evolutionary significance (string-match)
 - Conformational Analysis (min-energy state)
 - Autonomous Robots, Vehicles
 - (managing smart highways, collision avoidance / path planning, ...)
 - Logistics (scheduling, resource allocation, ...)
 - Design of VLSI circuits (placement, routing, partitioning, floorplanning, clock distribution, logic synthesis, ...)

Course Overview (cont.)

- What buys more, hardware or software?
 - FFT, Quicksort, etc.
 - Throwing hardware at a problem is *usually* not the right answer
- Patterns of problem-solving
 - e.g., Polya, How to Solve It
- Tools: Counting, Recurrence Relations, ..., **Data Structures, Problem-Solving Patterns, ...**
- Ideas:
 - Problem classes and “solution classes”
 - Lower bounds, reductions
 - *What do you think this means?*
 - Intractability (and reducibility, approximation)

Course Overview (cont.)

- Frameworks (“Paradigms”):
 - Divide-and-Conquer (D/Q)
 - searching, sorting, recurrences
 - Greed
 - Minimum spanning tree, coin changing, Huffman codes
 - Dynamic Programming
 - matrix chain product, shortest path, string processing
 - Backtrack and Branch-and-Bound
 - N-queens, game tree, search and planning
 - Heuristics
 - simulated annealing, evolutionary algorithms
 - Other
 - geometry, intractability / approximation, randomization

What is a Problem?

- A problem is defined by:
 - (i) input domain
 - e.g., all ordered pairs of positive integers
 - (ii) output specification
 - e.g., equivalent fraction in lowest terms
- A problem with the input specified is a problem instance.
 - e.g., “reduce 343/56 to lowest terms”
- Types of Problems:
 - **Decision**
 - Y/N answer
 - **Computation**
 - How many acyclic $s - t$ paths in G ?
 - **Construction** (more than one answer)
 - Construct (exhibit) an $s - t$ path in G . *Any $s-t$ path, vs. shortest $s-t$ path, vs. ...*
 - **Optimization** (set of all alternatives; cost function)
 - Determine the shortest $s - t$ path in G .
- Correct Algorithm: for each input, an output is produced that meets specifications

Problem-Solving First Example

- **Tower of Hanoi**

- Rules: (i) One disk moves at a time, and (ii) Never put a larger disk onto a smaller disk
- If you move one disk / second, when will all 64 disks be moved?
- A more useful question: What is the minimum #moves needed to transfer a stack of n disks?
 - *Why more useful?* Assumes optimal strategy, ...

- **Define Notation:**

- For a stack of n disks, call this number T_n

- **Look At Small Cases:**

- $T_0 = 0$, $T_1 = 1$, $T_2 = 3$

Problem-Solving First Example (cont.)

- Can we reduce to a known problem?
 - $T_n \leq 2T_{n-1} + 1, n > 0$
 - Why?
 - Shift (n-1), move largest disk, shift again
 - Why \leq inequality?
 - $2T_{n-1} + 1$ suffices, but maybe can do better
 - Why does the lower bound (LB) $T_n \geq 2T_{n-1} + 1$ hold?
 - Must move largest disk sometime; at this instant, have (n-1) on single peg
 - $\rightarrow T_n = 2T_{n-1} + 1, T_0 = 0$
- What is a general (closed-form) solution for T_n ?
 - $\{ T \} = 0, 1, 3, 7, 15, 31, 63, \dots$

Problem-Solving First Example (cont.)

- Looks like $T_n = 2^n - 1$; let's guess this answer and try to prove it
 - Claim: $T_n = 2^n - 1$
 - Proof: (induction)
 - $T_0 = 0 = 2^0 - 1$ holds (basis)
 - $T_n = 2T_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$ (I.H.)
- Is there an easier way to see the result?
 - *Exercise*: Consider $U_n = T_n + 1$
- *Note the pattern of steps that we followed...*

Question: What Makes One Algorithm Better (or Worse) Than Another?

- *Efficiency with respect to resources (= one aspect)*
- Example: Determinant
 - Recall: $\det(2 \times 2 \text{ matrix}) = ad - bc$
 - Recall: recursive definition $\det M = (\dots)$
 - M_{1j} is the $(1, j)$ cofactor matrix of $n \times n$ matrix M
- Problem: Give an algorithm for computing $\det M$
 - A1: Use definition to get recursive algorithm (exercise)
 - *How many multiplications? (About $n!$)*
 - A2: Use Gaussian elimination to get LT M'
 - If M' is lower-triangular, $\det M' = \prod m'_{ii}$
 - Need to show $\det M' = \det M$ (by linearity of determinant)
 - For $n=20$, A1 takes 107 years; A2 takes 0.05 seconds

Problem-Solving Second Example

- Recall problem statement from above: m/n in lowest terms
- “Formal” Statement:
 - Input: integers $m \geq 0$, $n > 0$
 - Output: integers m' , n' s.t. $m/n = m'/n'$, $(m,n) = (m',n')$
- A1:
 - Cancel all 2's
 - Cancel all 3's
 - Cancel all 5's
 - etc. until $\min(m,n)$ exceeded
- Why is this silly?
 - What's the “worst case”?
 - We always have to check up to $\min(m,n)$

Problem-Solving Second Example (cont.)

- A1': *// try divisors starting with largest possible*
 - $i \leftarrow \min(m, n) + 1$
 - repeat $i \leftarrow i - 1$ until $((i|m) \text{ and } (i|n))$
 - return i
 - may get lucky and stop after only a few divisions
 - but, worst case: $m \approx n$, $(m, n) = 1$
- A2:
 - find $\text{gcd}(m, n)$ *// gcd = greatest common divisor*
 - return $m' = m / \text{gcd}(m, n)$, $n' = n / \text{gcd}(m, n)$
 - We have recast problem as gcd (!)

Problem-Solving Second Example (cont.)

– gcd(m,n) [Euclid's Algorithm] // *assumes m < n*

```
while m > 0 do  
    t ← n mod m  
    n ← m  
    m ← t  
return n
```

Example: Work through calculation of gcd(81,21)

– Claim: If $n > m$ then $\text{gcd}(m,n) = \text{gcd}(m,n-m)$

• *How do you prove an equality? Prove both inequalities.*

– Proof: (1st inequality) Want $\text{gcd}(m,n-m) \stackrel{3}{=} \text{gcd}(m,n)$

i.e., if $z|m$ and $z|n$ then $z|m$, $z|(n-m)$

$z|m$ and $z|n \Rightarrow m \bmod z = n \bmod z = 0$

$\Rightarrow (n-m) \bmod z = 0$

$\Rightarrow z|(n-m)$

Problem-Solving Second Example (cont.)

– gcd(m,n) [Euclid's Algorithm] // *assumes m < n*

```
while m > 0 do  
    t ← n mod m  
    n ← m  
    m ← t  
return n
```

Example: Work through calculation of gcd(81,21)

– Claim: If $n > m$ then $\text{gcd}(m,n) = \text{gcd}(m,n-m)$

• *How do you prove an equality? Prove both inequalities.*

– Proof: (2nd inequality) Want $\text{gcd}(m,n-m) \leq \text{gcd}(m,n)$

i.e., if $z|m$, $z|(n-m)$ then $z|m$, $z|n$

$z|m$ and $z|(n-m) \Rightarrow [m+(n-m)] \bmod z = 0$

$\Rightarrow z|n$

Proving That the Algorithm is “Good”

- Euclid’s Algorithm is correct. *Is it efficient?*
- How many times can we go through main loop of gcd (m,n)?
 - Suppose leading entry m halves each time? (It doesn’t...) \Rightarrow $\log m$ is an upper bound on # passes
 - *Is any geometric decrease good enough?*
- Notation:
 - (m_i, n_i) are values after i^{th} pass
 - Assume $m_0 \neq n_0$
 - Loop is executed a total of L times

Proving That the Algorithm is “Good”

```
gcd(m,n) [Euclid's Algorithm]  (assumes m < n)  
  while m > 0 do  
    t ← n mod m  
    n ← m  
    m ← t  
  return n
```

- Notation:

- (m_i, n_i) are values after i^{th} pass
- Assume $m_0 \neq n_0$
- Loop is executed a total of L times

- Claims:

- (i) $m_i \neq n_i \quad \forall 0 \leq i \leq L-1$ (true from algorithm statement)
- (ii) $n_{i+1} = m_i$ (true from algorithm statement)
- (iii) $m_{i+1} \leq n_i / 2$ [**Case 1:** $m_i \leq n_i / 2 \rightarrow m_{i+1} \leq n_i / 2$ since $m_{i+1} < m_i$.
Case 2: $m_i > n_i / 2 \rightarrow m_{i+1} = n_i \bmod m_i = n_i - m_i \leq n_i / 2$.]

Proving That the Algorithm is “Good”

gcd(m,n) [Euclid’s Algorithm] (*assumes $m < n$*)

```
while m > 0 do  
    t ← n mod m  
    n ← m  
    m ← t  
return n
```

- Claims:

- (i) $m_i \leq n_i \quad \forall 0 \leq i \leq L-1$ (true from algorithm statement)
- (ii) $n_{i+1} = m_i$ (true from algorithm statement)
- (iii) $m_{i+1} \leq n_i / 2$ [**Case 1:** $m_i \leq n_i / 2 \rightarrow m_{i+1} \leq n_i / 2$ since $m_{i+1} < m_i$.
Case 2: $m_i > n_i / 2 \rightarrow m_{i+1} = n_i \bmod m_i = n_i - m_i \leq n_i / 2$.]

- Theorem: $m_{i+2} \leq m_i / 2$

- Proof:

- (ii) $\Rightarrow n_{i+1} = m_i$

- (iii) $\Rightarrow m_{i+2} \leq n_{i+1} / 2$

- Corollary: If $n_0 \geq m_0 \geq 1$, $L \leq 2 \log_2 m_0 + 1$

Do You Remember Recurrences?

- Fibonacci (basketball): UCSD 75, PLNU 64
- Assuming no 3-pointers, in how many ways can UCSD accumulate 75 points?
- Notation:
 - $S(n) \equiv$ # ways to score n points
- Small Cases:
 - $S(0) = 1$
 - $S(1) = 1$ 1 (free throw) \rightarrow 1 way to score 1 pt
 - $S(2) = 2$ 2 or 1|1 \rightarrow 2 ways to score 2 pts
 - $S(3) = 3$ 2|1 or 1|2 or 1|1|1 \rightarrow 3 ways to score 3 pts
 - $S(4) = 5$ 2|2 or 2|1|1 or 1|2|1 or 1|1|2 or 1|1|1|1

Is this familiar?

Do You Remember Recurrences? (cont.)

- Problem: What is $S(75)$?
 - **Notation**: write $F(n) = S(n-1)$
 $F(1) = F(2) = 1$; $F(n) = F(n-1) + F(n-2)$
 - **Guesses**: try $F(n) = a^n$ for some a
 $a^n = a^{n-1} + a^{n-2} \rightarrow a^2 = a + 1 \rightarrow a^2 - a - 1 = 0$
Roots: $a_1 = (1 + \sqrt{5})/2$; $a_2 = (1 - \sqrt{5})/2$
Inspection: $F(n)$ seems close to $(a_1)^n$ *What's missing?*
 - **Use all of the information**
 $F(1) = 1$; $F(2) = 1$ (initial conditions)
 - Homogeneous linear recurrence: any linear combination of $(a_1)^n$, $(a_2)^n$ is also a solution.
 - $c_1(a_1)^1 + c_2(a_2)^1 = F(1) = 1$; $c_1(a_1)^2 + c_2(a_2)^2 = F(2) = 1$
 - Get $c_1 = 1 / \sqrt{5}$, $c_2 = -1 / \sqrt{5}$
 - 1845 result of Lamé (see Knuth, volume 2, section 4.5.3): If $m, n \in \mathbb{N}$ and $k \in \mathbb{N}$, then L in $\gcd(m, n) \in k$, with equality when $(m, n) = (F(k-1), F(k))$.

Useful and Challenging Questions

- MaxMin

- Given a list of N numbers, return the largest and smallest.

- Finding a Celebrity

- Given a set S of N people, assume that for any pair I, J exactly one of the following is true: I “knows” J , or J “knows” I . Further, define a “celebrity” as someone who knows no one (and who is therefore known by everyone else). Given the “knows” relation over S , determine whether S contains a celebrity.

- Reduction

- SORTING problem:

Input: a set of numbers

Output: the elements of the set, in sorted order

- CONVEX HULL problem:

Input: a set of points in \mathfrak{R}^2

Output: the convex hull of these points, i.e., polygon vertices in order

→ Is “ease” of SORTING “related” to “ease” of CONVEX HULL?

Choosing Between Solutions

- Criteria:
 - Correctness
 - Time resources
 - Hardware resources
 - Simplicity, clarity (practical issues)
- Will need:
 - Size, Complexity measures
 - Notion of “basic” machine operation

Do You Remember Data Structures?

- Recall that we wanted $S(75) = F(76)$, i.e., the 76th Fibonacci number
- Give an efficient algorithm
 - For now, let's equate "efficient" with "using few 'elementary' machine operations"; we will not worry about size of operands, etc.
- $\text{fib1}(n)$ if $n < 2$ then return n
 else return $\text{fib1}(n-1) + \text{fib1}(n-2)$
 - Analysis: $T(n) = 1$ if $n < 2$; $T(n) = T(n-1) + T(n-2)$ otherwise
 $T(n) = F(n)$, i.e., around $(1.64)^n$
- ***What is wrong with fib1 ? It keeps on recomputing values that it has already computed.***

Do You Remember Data Structures? (cont.)

- `fib2(n)` $f[1] = 1; f[2] = 2;$
for $j = 3$ to n do
 $f[j] = f[j - 1] + f[j - 2]$
- Analysis: $T(n) = n$
 - Saving work (“caching”) can be useful !
 - Similar example: Pascal’s triangle (binomial coefficients)
 - But, can we do better?
- Idea: Use “natural structure”
 - We are applying the recurrence n times. Are there any shortcuts?

Do You Remember Data Structures? (cont.)

- $\text{fib3}(n)$

- Consider 2×2 matrix M , with $m_{11} = 0$, $m_{12} = 1$, $m_{21} = 1$,
 $m_{22} = 1$

- Observe: $[F(k) \ F(k+1)]^T = M \times [F(k-1) \ F(k)]^T$

$$[F(n+1) \ F(n+2)]^T = M^n \times [F(1) \ F(2)]^T = M^n \times [1 \ 1]^T$$

- *How does this help?*

- *Hint: $76_{10} = 1001100_2$*

- $M^{76} = M^{64} \times M^8 \times M^4$

- \rightarrow fib3 uses “addition chains”

Quantifying “Better”, “Worse”

- Resources used depend on natural parameter n of input
 - search/sort list # items $x > y$
 - matrix mult largest dim $x * y ; x + y$
 - traverse tree # nodes follow ptr
- Asymptotic Notation “as n grows large”
 - $f \in O(g)$ if $\exists c_1, c_2 > 0$ s.t. $f(n) \leq c_1 g(n) + c_2 \quad \forall n > 0$
 - $f \in O(g)$ if $\exists c > 0, N$ s.t. $\forall n > N, f(n) \leq c g(n)$
 - e.g., $200x^2 \in O(x^{2.5})$
 - $f \in \Omega(g)$ if $g \in O(f)$
 - $f \in \Theta(g)$ if $g \in O(f), f \in O(g)$
 - f is $o(g)$ if $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$

Using “Big-Oh” Notation

- Definition: $f(n)$ is monotonically growing (non-decreasing) if $n_1 \geq n_2 \Rightarrow f(n_1) \geq f(n_2)$
- Theorem: For all constants $c > 0$, $a > 1$, and \forall monotonically growing $f(n)$, $(f(n))^c \in O(a^{f(n)})$
- Corollary (*take $f(n) = n$*): $\forall c > 0$, $a > 1$, $n^c \in O(a^n)$
 - Any exponential in n grows faster than any polynomial in n
- Corollary (*take $f(n) = \log_a n$*): $\forall c > 0$, $a > 1$, $(\log_a n)^c \in O(a^{\log_a n}) = O(n)$
 - Any polynomial in $\log n$ grows slower than $n^{c'}$, $c' > 0$
 - Exercise: $f \in O(s)$, $g \in O(r) \Rightarrow f+g \in O(s+r)$
 - Exercise: $f \in O(s)$, $g \in O(r) \Rightarrow f \cdot g \in O(s \cdot r)$
- *So, we can count operations in an asymptotic sense. What is an “operation”?*

Motivation for a Resource Model

- Suppose factorial, mod are “unit-cost” on some computer.

WILSON(n)

if $(n-1)! + 1 \neq 0 \pmod n$ then return TRUE

else return FALSE

- one-step primality testing \Rightarrow kind of fishy...
- Or, should return $\max_i x_i$ be “unit-cost”?
 - Propagation delay on wires, finite (non-zero) dimensions of transistors and wires \rightarrow physical models increasingly relevant

The RAM (Random-Access Machine) Model

- finite stored program
- finite collection of registers
 - each stores single integer or real
- array of n words of memory
 - each stores single integer or real
 - has unique address in $[1, \dots, n]$
- In one step:
 - Perform arithmetic, logical operation on register content
 - $R_j := M_{R_k}$ or $M_{R_j} := R_k$ (access contents of word whose address is in register)
 - JNZ, HALT, etc.

The RAM Model (cont.)

- Q: On a RAM machine, how large a number can be manipulated in constant time?
- Two variants:
 - uniform cost \rightarrow time is independent of the size of numbers
 - log cost \rightarrow time is proportional to #bits manipulated
- **Exercise:** What are costs for each, under the two variants?
 - (i) `sum_1_to_N(n)`

```
sum  $\leftarrow$  0
for i  $\leftarrow$  1 to n do sum  $\leftarrow$  sum + i
return sum
```
 - (ii) `fib4(n)`

```
i  $\leftarrow$  1, j  $\leftarrow$  0
for k  $\leftarrow$  1 to n do
  j  $\leftarrow$  i + j
  i  $\leftarrow$  j - i
return j
```
- Other: Turing, pointer machines; straight-line program, decision/comparison tree, ...

What Do We Measure?

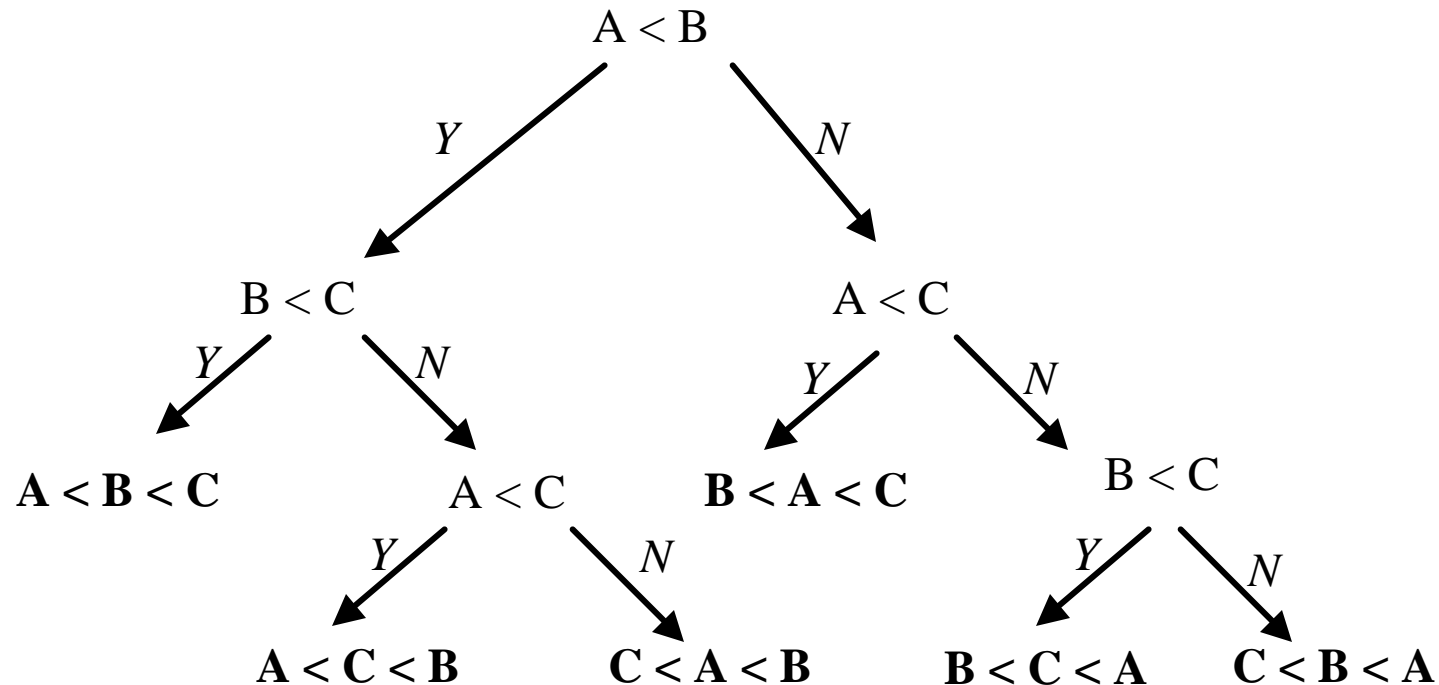
- Traditional metrics:
 - Program Size static
 - Runtime dynamic
 - Memory Usage dynamic
- Best Case (not informative)
 - e.g., Bubble Sort? Insertion Sort? Quicksort?
- Worst Case (easiest, most common)
 - $t_A(I) \equiv$ time used by alg A on instance
 - $D(n) \equiv$ set of all instances of size n
 - $WC_A(n) = \max \{t_A(I) \mid I \in D(n)\}$
- Average Case (useful, but often less tractable)
 - $p(I) \equiv$ probability that instance I occurs
 - $AC_A(n) = \sum_{I \in D(n)} p(I)t_A(I)$
- Amortized Analysis

Can Characterize Problem Complexity

- Upper Bounds:
 - Alg A has UB $f(n)$: $\forall I \in D(n), t_A(I) \leq f(n)$
 - Problem P has UB $f(n)$: \exists Alg A for P with UB $f(n)$
 - P has UB $O(f)$: \exists Alg A with UB $g(n)$; $g \in O(f)$
- Lower Bounds:
 - Alg A has LB $f(n)$: \exists infinitely many n s.t. $\exists I \in D(n)$ where $t_A(I) \geq f(n)$
 - Problem P has LB $f(n)$: \forall Alg A for P, \exists infinitely many m s.t. $\exists I \in D(m)$ for which $t_A(I) \geq f(m)$
- How Do We Argue UB?
 - Constructively (\leq , reductions)
- How Do We Argue LB?
 - e.g., comparison tree model, reductions

Sorting (With Comparisons)

- Input: sequence of numbers
Output: a sorted sequence
- Observe: Sorting == Identifying a Permutation



Comparison-based LB Arguments - Sorting

- Observe: Sorting \circ Identifying Permutation
- Binary Tree: Root at level (height) 0
- Theorem:
 - $\exists c > 0$ s.t. " algorithms which use comparisons to sort, and " input sizes n , at least one input requires $cn \log n$ comparisons
- Fact:
 - Binary tree of height h has at most 2^h leaves
- Observe:
 - $n!$ leaves needed \Rightarrow decision tree must have $h \geq \log(n!)$ and h is max # comparisons needed to sort input of size n using the corresponding algorithm

Sorting Lower Bound (cont.)

- Want: $\log(n!) \in \Theta(n \log n)$
- Claim: $\log(n!) \in O(n \log n)$
 $n! \leq n^n \Rightarrow \log n! \leq n \log n$
- Claim: $\log(n!) \in \Omega(n \log n)$
 $n! \geq (n/2)^{n/2} \Rightarrow \log n! \geq n/2 \log(n/2)$
 $\Rightarrow \text{BAD } 4 \log n! \geq n \log n$
Stirling: $\log n! \approx n \log n - 1.44 n$
- Worst-case analysis
 - Can be uninformative (e.g., QuickSort, Simplex Method)
- Can we lower-bound the “average case” complexity?
- Is “average case” well-defined?
 - Want $\sum_i p_i d_i \equiv$ expected depth of a leaf in the comparison tree
 - $d_i \equiv$ depth of leaf i ; $i =$ input with probability $= p_i$
 - Assume all input permutations equiprobable

Average-Case Complexity of Sorting

- Q: Even though all sorting algs have some input which requires $n \log n$ time, is there an algorithm with $o(n \log n)$ average-case performance?
- Theorem: If all $n!$ input permutations equiprobable, then *any* decision tree that sorts has *expected* depth $\Omega(n \log n)$.
 - Let $D(m) \equiv$ smallest sum of leaf depths over all binary trees with m leaves
 - Claim: $D(m) \geq m \log m$.
 - If Claim true, use $m = n!$ and fact that $\log n! \in \Theta(n \log n)$
 - $D(n!) \geq n! \log n! \rightarrow$ average leaf depth is $\Omega(n \log n)$

Average-Case Complexity of Sorting (cont.)

- Claim: $D(m) \geq m \log m$

Proof by induction on m .

($D(T) \equiv$ sum of leaf depths of tree T , where unambiguous.)

Claim trivial for $m = 1$; assume Claim " $m < k$ (*strong I.H.*).

Any tree T with k leaves can be viewed as a root and two subtrees T_i and T_{k-i} (with i and $k-i$ leaves respectively)

$$\rightarrow D(T) = i + D(T_i) + (k-i) + D(T_{k-i})$$

$$D(k) = \min_{1 \leq i \leq k} [k + D(i) + D(k-i)]$$

$$\geq k + \min_i [D(i) + D(k-i)]$$

$$\geq k + \min_i [i \log i + (k-i) \log (k-i)] \quad (\text{by I.H.})$$

which is minimized for $i = k/2$.

$$\text{P } D(k) \geq k + k \log (k/2) = k + k(\log k - 1) = k \log k$$

A Sorting Excursion

- Exercise: Use the comparison model to show an $\Omega(\log n)$ lower bound for deciding whether an n -element sorted list contains a given number k .
- A Sorting Excursion (cf. Sedgewick's text)
 - N.B.: Knuth \rightarrow Sedgewick \rightarrow Weiss
 - LB's (worst- and average-case) (done already)
 - Practical issues
 - Simple methods
 - Tradeoffs and special cases
 - Complexity
 - Closing the gap
 - D/Q paradigm: Mergesort and Quicksort
 - Quicksort randomized analysis

Practical Issues

- What is the best way to sort?
 - Not a well-posed question, but what are some key issues?
- Sorting Problem: Given a file of records, each record containing key(s), put the file into lexicographic order.
- Kinds of sorting algorithms
 - Internal → input file stays in main memory
 - External → records accessed sequentially in blocks
 - Stable → retain lexicographic order w.r.t. other keys
- Resources
 - $O(1)$ extra memory (“in place”)
 - $O(n)$ extra memory (auxiliary copy, etc.)
- Special cases, variations
 - Bounded key size; small number of possible key values; etc.
- For each method:
 - Why use the method? For which inputs is the method “good” or “bad”?

Insertion Sort

- Insertion Sort

for $j = 2$ to n do

$current = A[j]$

$i = j - 1$

 while $i > 0$ & $A[i] > current$ do

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = current$

next current

go left

find place for current

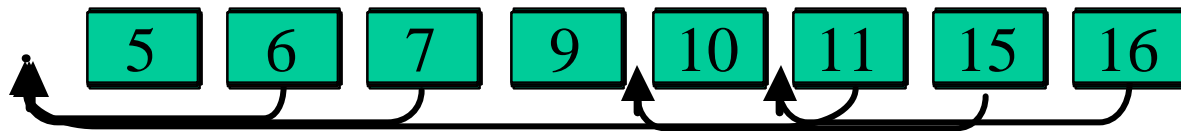
shift sorted right

go left

put current in place

- Like sorting a bridge hand
- What is a “good” input?

Insertion Sort Execution Example



Insertion Sort Complexity

- Insertion Sort

for j = 2 to n do

 current = A[j]

 i = j - 1

 while i > 0 & A[i] > current do

 A[i + 1] = A[i]

 i = i - 1

 A[i + 1] = current

next current

go left

find place for current

shift sorted right

go left

put current in place

- Like sorting a bridge hand
- Worst Case $O(n^2)$
- Average Case $O(n^2)$ (*exercise*)
- Best Case $O(n)$ (*define “almost-sorted”*)

Selection Sort

- Selection Sort // Bubble Sort = slow-motion Selection

```
for i = 1 to n-1 do
```

```
    min = i
```

```
    for j = i+1 to n do
```

```
        if a[j] < a[min] then min = j
```

```
    t = a[min]
```

```
    a[min] = a[i]
```

```
    a[i] = t
```

- Iteratively “select” smallest remaining element; swap with i^{th} element
 - How many times does a given element move?
“Once” \rightarrow only $n-1$ swaps total.
 - What is a “good” input? Small keys, large records \rightarrow linear-time sort if secondary storage access dominates complexity

(Fun/Useful) Questions to Think About

- Special sorting input: How should you sort if the key space is small? (e.g., letter grades A, B, C, D, F)
- Monty Hall
 - 3 boxes with one prize: you choose one box
 - Monty Hall shows you the empty box from the other two and offers to let you switch your choice
 - What is better: keep the same box, switch, or toss a coin?
- Death Row
 - 3 men on death row: one will not be executed tomorrow a.m.
 - Guard tells X that Y (among two others) will be executed
 - X thinks: “Before I heard this, my probability of survival was $1/3$, but now it is $1/2$.”
 - Is X correct?

Divide and Conquer for Sorting (2.3/1.3)

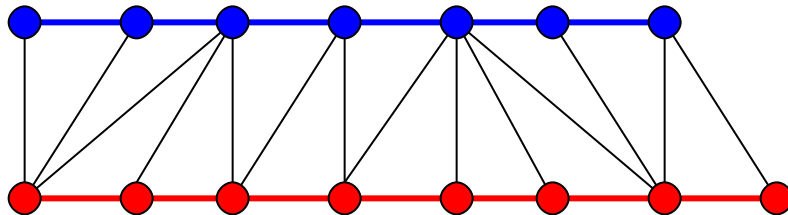
- Divide (into two equal parts)
- Conquer (solve for each part separately)
- Combine separate solutions
- Merge sort
 - Divide into two equal parts
 - Sort each part using merge-sort (recursion!!!)
 - Merge two sorted subsequences

Merging Two Subsequences

$x[1]-x[2]-\dots-x[k]$

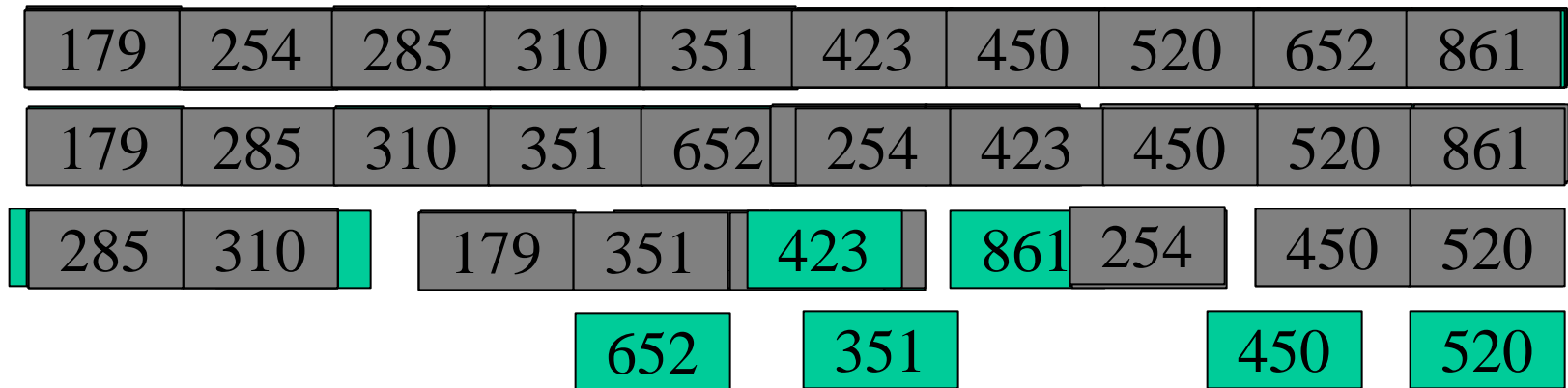
$y[1]-y[2]-\dots-y[l]$

if $y[i] > x[j] \Rightarrow y[i+1] > x[j]$

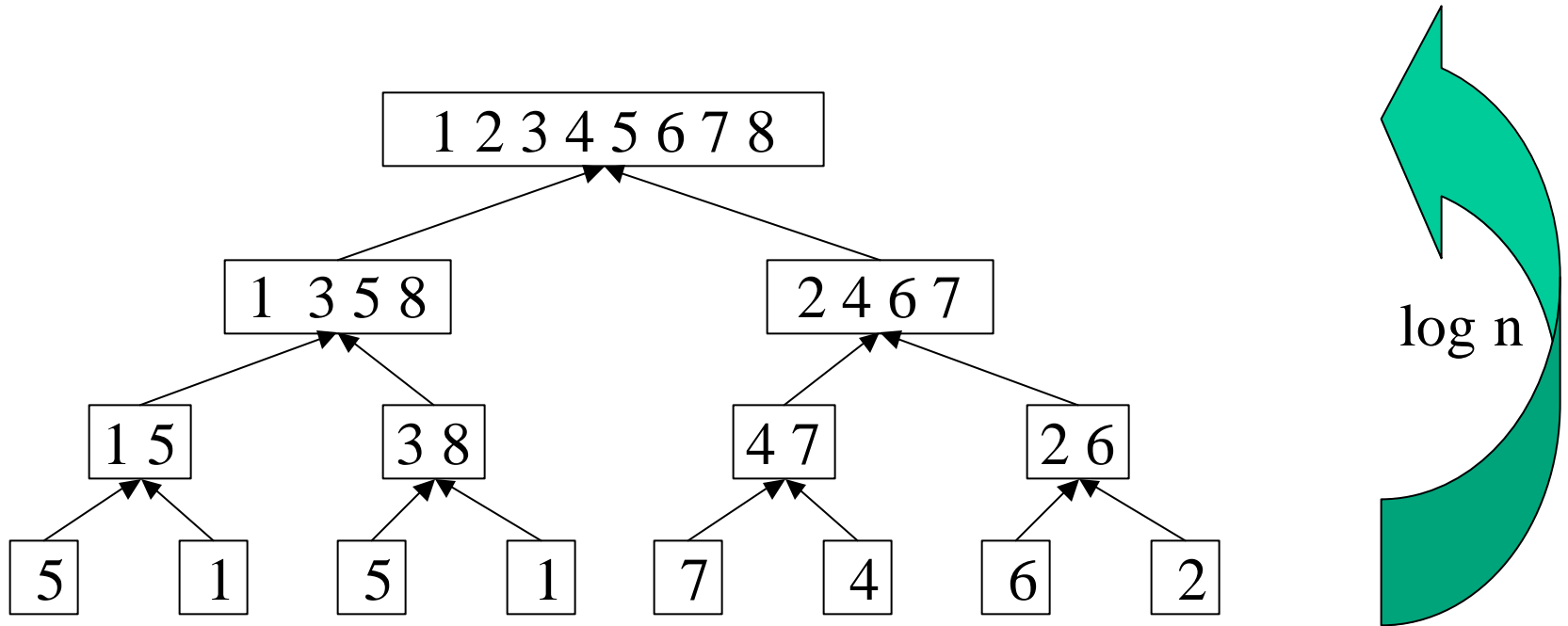


$\leq k+l-1$ edges = # (comparisons) = linear time

Merge Sort Execution Example



Recursion Tree



- n comparisons per level
- $\log n$ levels
- total runtime = $n \log n$

Quicksort (7.1-7.2/8.1-8.2)

- Sorts **in place** like insertion sort, unlike merge sort
- **Divide** into two parts such that
 - elements of left part < elements of right part
- **Conquer**: recursively solve for each part separately
- **Combine**: trivial - do not do anything

Quicksort(A,p,r)

if $p < r$

then $q \leftarrow \text{Partition}(A,p,r)$

//divide

Quicksort(A,p,q)

//conquer left

Quicksort(A,q+1,r)

//conquer right

Divide = Partition

PARTITION(A,p,r)

//Partition array from A[p] to A[r] with pivot A[p]

//Result: All elements \leq **original** A[p] has index \leq i

x = A[p]

i = p - 1

j = r + 1

repeat forever

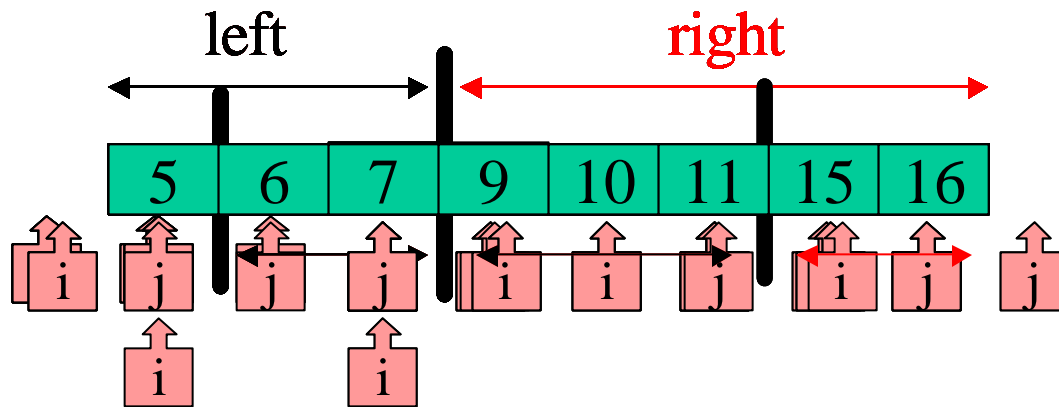
 repeat j = j - 1 until A[j] \leq x

 repeat i = i + 1 until A[i] \geq x

 if i < j then exchange A[i] \leftrightarrow A[j]

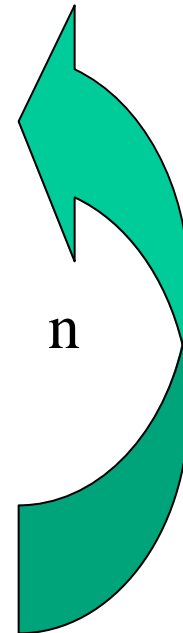
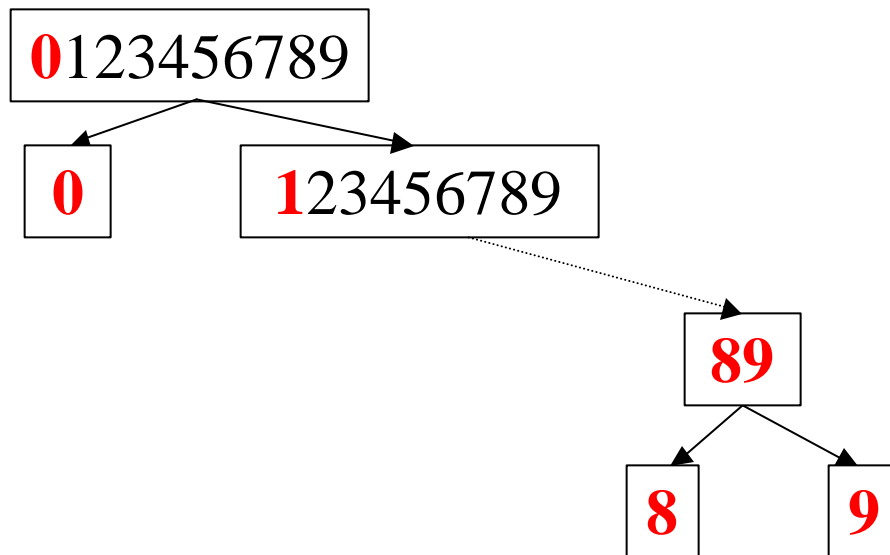
 else return j

How It Works



Runtime of Quicksort

- Worst case:
 - every time nothing to move
 - pivot = left (right) end of subarray
 - $O(n^2)$

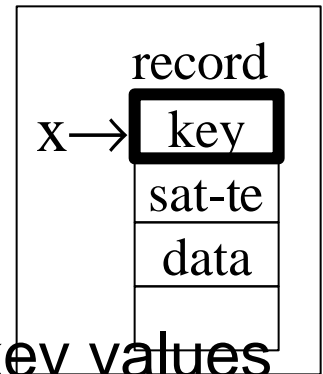


Runtime of Quicksort

- Best case:
 - every time partition in (almost) equal parts
 - no worse than in given proportion
 - $O(n \log n)$
- Average case
 - $O(n \log n)$???
 - How do we prove this?

Dynamic Sets (III, Introduction)

- Dynamic sets (Data structures):
 - we change a dictionary, add/remove words
 - reuse of **structured** information
 - on-line algorithms - very fast updating
- Elements:
 - **key** field is the element ID, dynamic set of key values
 - **satellite** information is not used in data organization
- Operations
 - **queries**: return information about the set
 - **modifying operations**: change the set



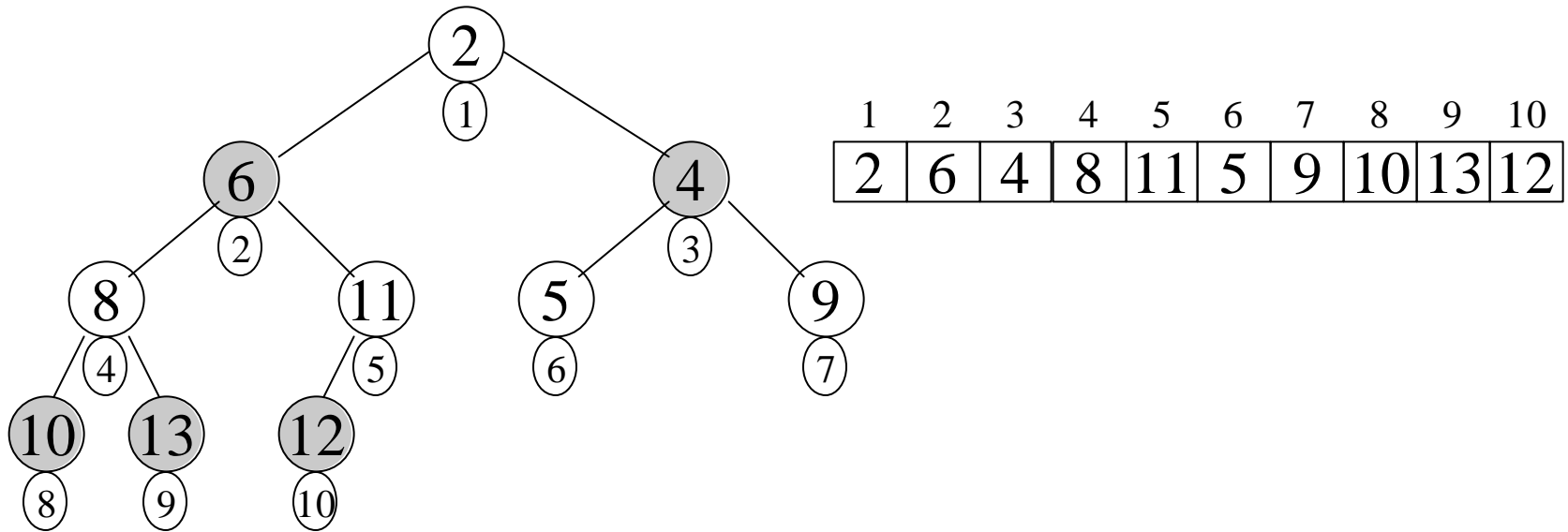
Operations (III, Introduction)

- Search(S, k) \rightarrow a pointer to element x with key k (query)
- Insert(S, x) add new element pointed to by x , assuming that we have $\text{key}(x)$ (modifying)
- Delete(S, x) delete x , x is a pointer (modifying)
- Minimum(S)/Maximum(S) \rightarrow max/min (query)
- Prede(suc)cessor(S, x) \rightarrow the next larger/smaller key to the key of element x (query)
- Union(S, S') \rightarrow new set $S = S \cup S'$ (modifying)

Priority Queues (6.5/7.5)

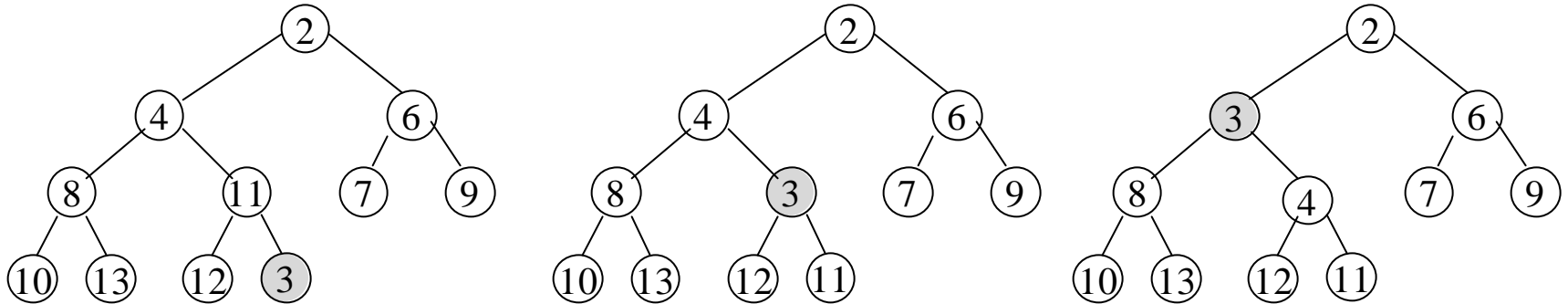
- Operations supported by **priority queue**
 - Insert(S, x) - inserts element with the pointer x
 - Minimum(S) - returns element with the minimum key
 - Extract-Min(S) - removes and returns minimum key
- Applications
 - job scheduling on shared computer
 - Dijkstra's finding shortest paths in graphs
 - Prim's algorithm for minimum spanning tree (next time)
- **Home Work:** **7-1** and **7-2**, p.152/6-1 p.142 and 6-2
p.143

Heaps (6.1/7.1)

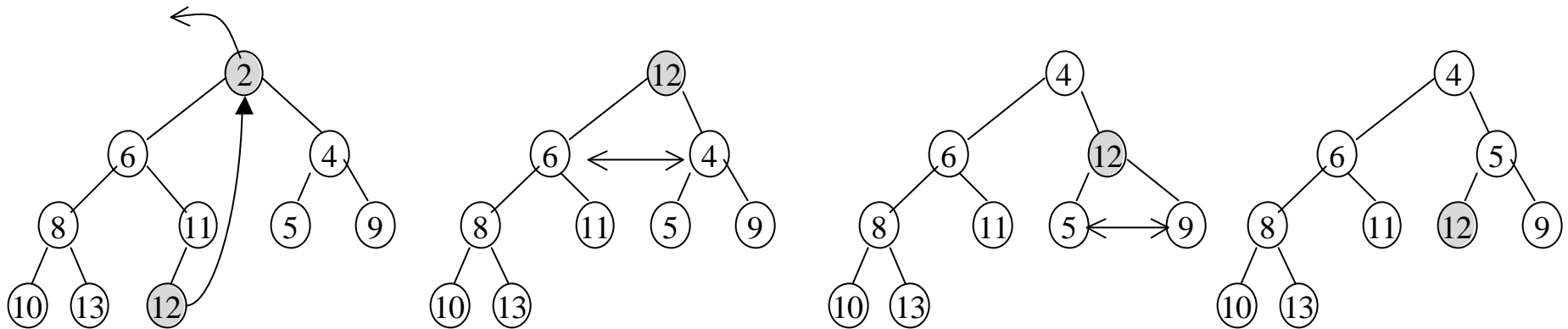


- **Pointers:**
 - Parent
 - Left(child), Right
- **Parent \leq Child**

Heap Operations (6.2-5/7.2-5)



Insert(S, x): $O(\text{height}) = O(\log n)$



Extract-min(S):

return head, replace head key with the last, float down, $O(\log n)$

Heapsort (6.4/7.4)

- Heapsort

- Build heap (for $i=1..n$) do insert ($A[1..i], A[i]$)
- For $(i=n..2)$ do
 - Swap ($A[1] \leftrightarrow A[i]$)
 - Heapsize = Heapsize-1
 - Float down $A[1]$

