Analysis and Design of Algorithms

Chapter 7: Transform and Conquer

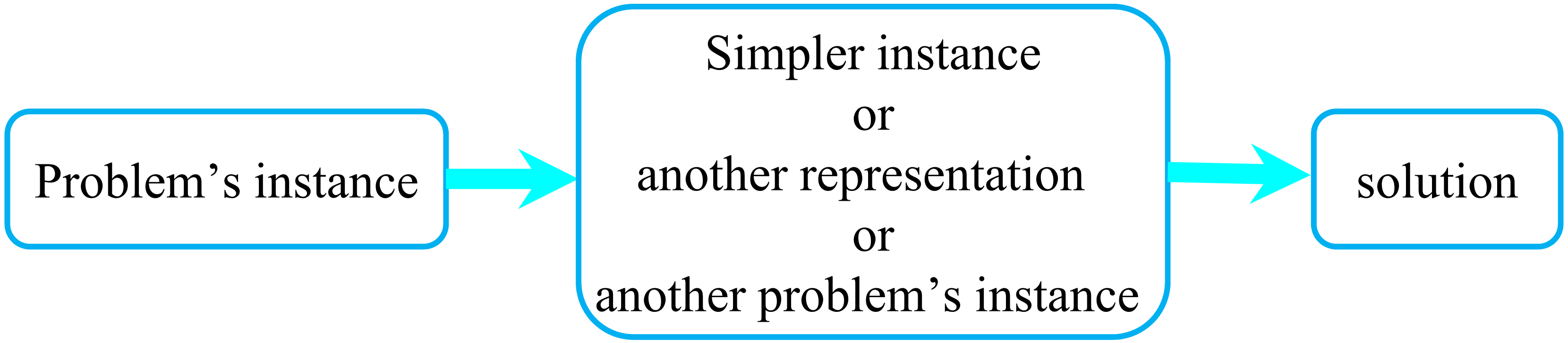
School of Software Engineering © Yanling Xu

Transform and Conquer

- ***Three variations of Transform and Conquer tech.***

  This group of techniques solves a problem based on a transformation.

  - *Two stage:*



Transform and Conquer

Three variations of Transform and Conquer tech.

Differ by what we transform a given instance to:

- *instance simplification:*

  to a simpler/more convenient instance of the same problem

- *representation change:*

  to a different representation of the same instance

- *problem reduction:*

  to a different problem for which an algorithm is already available

Presorting - Instance simplification

- *why interested in sorting?*

many questions about a list are easier to answer if the list is sorted.

- *benefit from sorting?*

☆ *the benefits of a sorted list should more than compensate for the time spent on sorting*

☆ generally comparison-based sorting alg. ***worst case, at least nlogn***

- ***Selection Sort*** $\Theta(n^2)$
- ***Bubble Sort*** $\Theta(n^2)$
- ***Insertion Sort*** $C_{worst}(n) = \frac{(n-1)n}{2}$ $C_{best}(n) = n-1$ $C_{avg}(n) \approx \frac{n^2}{4}$
- ***Mergesort*** $\Theta(n \log n)$
- ***Quicksort*** $C_w(n) = \Theta(n^2)$ $C_b(n) = \Theta(n \log n)$ $C_{avg}(n) = O(n \log n)$

Presorting

- ***Presorting --- Instance simplification***
  - *searching*
  - *computing the median (selection problem)*
  - *checking if all elements are distinct (element uniqueness)*

Presorting

Element Uniqueness with presorting

- *Element Uniqueness problem --a brute-force method*
  - *compare all pairs of the array's elements (see Chapt 2)*
  - *until either two equal elements found or no more pairs left*

$$C_{worst}(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

```
ALGORITHM UniqueElements(A[0..n − 1])
    //Determines whether all the elements in a given array are distinct
    //Input: An array A[0..n − 1]
    //Output: Returns "true" if all the elements in A are distinct
    //         and "false" otherwise
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] = A[j] return false
    return true
```

Presorting

Element Uniqueness with presorting

- Element Uniqueness problem -Presorting-based method
  - Stage 1: sort by efficient sorting algorithm (e.g. mergesort)
  - Stage 2: scan array to check pairs of adjacent elements
- Efficiency Analysis
  - sum of
  - time spent on sorting : at least nlogn comparisons –determine the overall efficiency
  - time spent on checking consecutive elements: no more than n-1 comparisons
  - **use a good sorting alg.**

$$C(n)=C_{sort}(n)+ C_{scan}(n)= \Theta(n\log n) + \Theta (n) = \Theta(n\log n)$$

Presorting

Computing a mode

Mode: a value that occurs most often in a given list of numbers

Eg. For {5, 1, 5, 7, 6, 5, 7} mode is 5

Brute-force method

- ***Idea:***

- Scan the list, compute the frequency of all its distinct values

- find the value with the largest frequency

Presorting

Computing a mode

- *Brute-force method (‘cont)*
  - ***implementation:***

- *Store the values already encountered, along with their frequencies, in an auxiliary list (the values in this auxiliary list are all distinct)*

- *On each iteration, the ith element of the original list is compared with the values already encountered by traversing this an auxiliary list*

- *If a matching value is found, its frequency is incremented;*

- *otherwise, the current element is added to the auxiliary list with frequency of* 1

Presorting

Computing a mode

Brute-force method ('cont)

- ***Worst case analysis***

- when a list with no equal elements,

- i th element is compared with i-1 elements of the auxiliary list

number of comparisons in creating the frequency auxiliary list

$$C(n) = \sum_{i=1}^{n}(i-1) = \frac{(n-1)n}{2} \in \theta(n^2)$$

number of comparisons to find the largest frequency in the auxiliary list n-1

Presorting

Computing a mode

Computing a mode with presorting

- ***idea :***

- sort the input firstly, then all equal values will be adjacent

- find the longest run of the adjacent equal values in the sorted array

- ***efficiency analysis***

sum of

- time spent on sorting : at least nlogn comparisons –determine the overall efficiency

- time spent on checking longest run of the adjacent : linear

- ***use a good sorting alg***

Presorting

Searching problem

Search for a given K in A[*0..n*-1]

- *Brute-force method*
  - ***sequential search :*** *(see Chapt2)*

$$T_{avg}(n) = \frac{p(n+1)}{2} + n(1-p) \qquad T_{\text{worst}}(n)=n \qquad T_{\text{best}}(n)=1$$

- *Binary Search* *(see Chapt5)*

$$C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil = \Theta(\log n)$$

$$C_b(n) = 1$$

$$C_{avg}(n) = \frac{1}{n}\sum_{i=1}^{k} i2^{i-1} \approx \log(n+1) - 1$$

Presorting

Searching problem

Searching with presorting

sum of

- time spent on sorting : at least nlogn comparisons –determine the overall efficiency

- time spent on binary search,

$$C_w(n) = \lfloor \log_2 n \rfloor + 1 = \Theta(\log n); \ , \ C_{avg}(n) = \Theta(\log n);$$

$$C(n)=C_{sort}(n)+ C_{search}(n)= \Theta(n\log n) + \Theta(\log n) = \Theta(n\log n)$$

- *if to search in the same list more than once, the time spent on sorting might be justified*

Gaussian Elimination - Instance simplification

Gaussian Elimination 高斯消去法

- *Idea*

Problem: Given: *a system of n linear equations in n unknowns with an arbitrary coefficient matrix*.

Idea:

*stage1: **Elementary operations**: Transform to an **equivalent** system of n linear equations in n unknowns with an **upper triangular** coefficient matrix.*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n &= b_n \end{aligned} \quad \longrightarrow \quad \begin{aligned} a_{1,1}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{nn}x_n &= b_n \end{aligned}$$

Gaussian Elimination - Instance simplification

Gaussian Elimination

- *Idea('cont)*

*stage2: Solve the latter by **backward substitutions** starting with the last equation and moving up to the first one.*

- find the value of x_n from the last equation immediately

- Substitute this value into the next to last equation to get x_{n-1}

- And so on, until we substitute the known values of the last n-1 variables into the first equation, to find the value of x_1

Gaussian Elimination

Applications of Gaussian Elimination

- *LU decomposition*
- *Computing a matrix inverse*
- *Computing a determinant*

Heaps and Heapsort

Heaps

- Heap is suitable for implementing priority queues

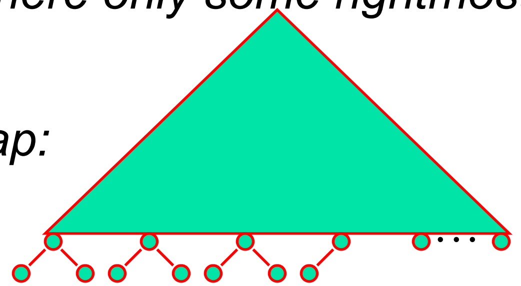
  maintaining a set S of elements, each with an associated value called a key/priority. It supports the following operations

  - Finding an item with the highest priority
  - Deleting an item with the highest priority
  - Adding a new item to the multiset

Notion of the Heap

- A binary tree with keys assigned to its nodes, one key per node
- Shape requirement: the binary tree is essentially complete, i.e. all its levels are full except possibly the last level, where only some rightmost leaves may missing
- Parental dominance requirement: for max-heap:

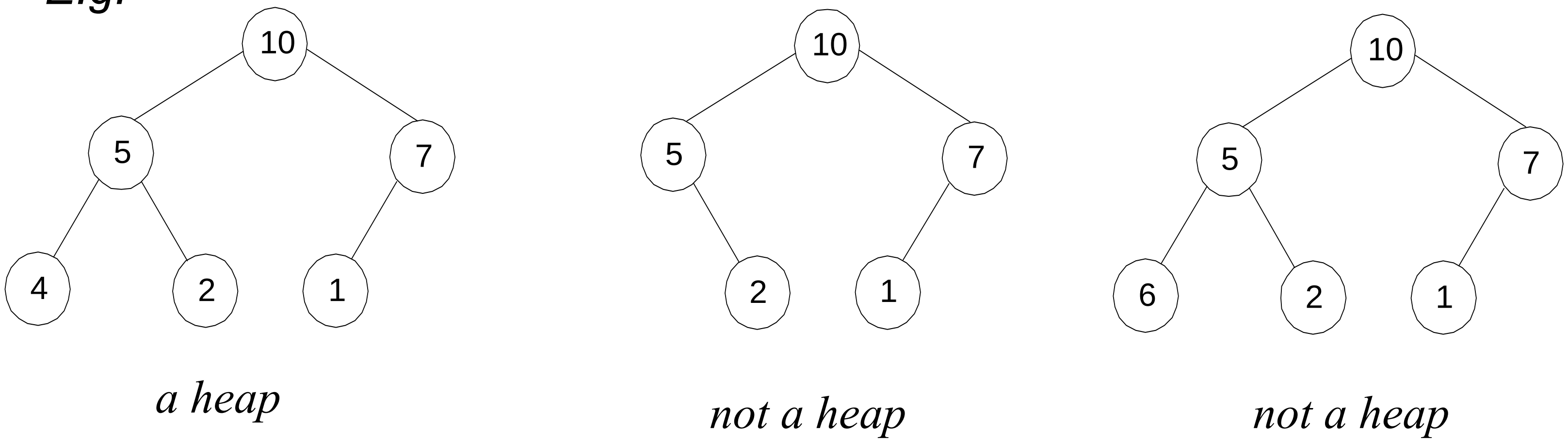
  key at each node $\geq$ keys at its children



Heaps and Heapsort

Heaps

E.g.



☆ *Heap's elements are ordered top down (a sequence of values along any path down from its root is decreasing or non-increasing if equal keys are allowed)*

☆ *but they are not ordered left to right*

Heaps and Heapsort

Heaps

Properties of Heaps

- *There exits exactly one essentially complete binary tree with n nodes, its height is* $\lfloor \log_2 n \rfloor$
  - *Height of a node: the number of edges on the longest simple downward path from the node to a leaf.*
  - *Height of a tree: the height of its root.*
  - *level of a node: A node's level + its height = h, the tree's height.*
- *The root of a heap always has the largest key (for a max-heap)*
- *A node of a heap considered with all its descendants is also a heap (The subtree rooted at any node of a heap is also a heap)*
- *Max-heap property and min-heap property*
  - *Max-heap: for every node other than root, A[PARENT(i)] >= A(i)*
  - *Min-heap: for every node other than root, A[PARENT(i)] <= A(i)*

Heaps and Heapsort

Heaps

Properties of Heaps

- *it is more efficient to implement a heap as an array, by storing the heap's elements in top-down left-to-right order*

  - Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations of the array

  - *Leaf keys occupy the last $\lceil n/2 \rceil$ locations*

  - *Relationships between indexes of parents and children.*

PARENT(i)

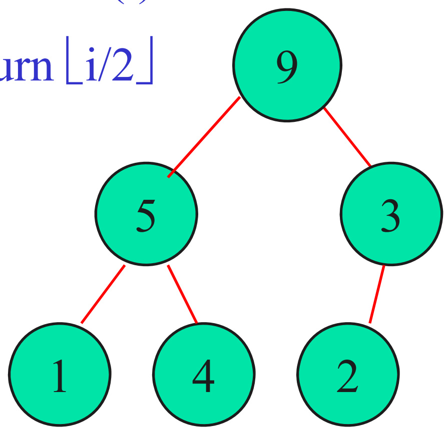
return $\lfloor i/2 \rfloor$

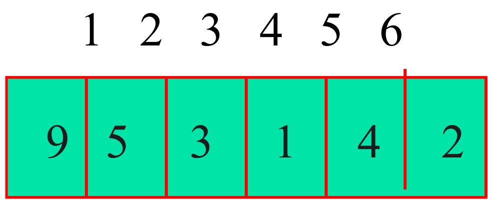
LEFT(i)

return 2i

RIGHT(i)

return 2i+1





Heaps and Heapsort

Heaps Construction

How to construct a heap with the given list of keys?

Bottom-up Heap construction

- *Build an essentially complete binary tree by inserting n keys in the given order.*

- *Heapify the tree*

  - *Starting with the last (rightmost) parental node, heapify/fix the subtree rooted at it; if the parental dominance condition does not hold for the key at this node:*

    - *exchange its key K with the key of its larger child*
    - *Heapify/fix the subtree rooted at the K's new position*
    - *until the parental dominance requirement for K is satisfied*

  - *Proceed to do the same for the node's immediate predecessor.*

  - *Stops after this is done for the tree's root.*

Heaps and Heapsort

Heaps Construction

Bottom-up Heap construction (A Recursive version)

```
ALGORITHM HeapBottomUp(H[1..n])
//Constructs a heap from the elements
//of a given array by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
        MaxHeapify(H, i)
```

Given a heap of n nodes, what's the index of the last parent? **$\lfloor n/2 \rfloor$**

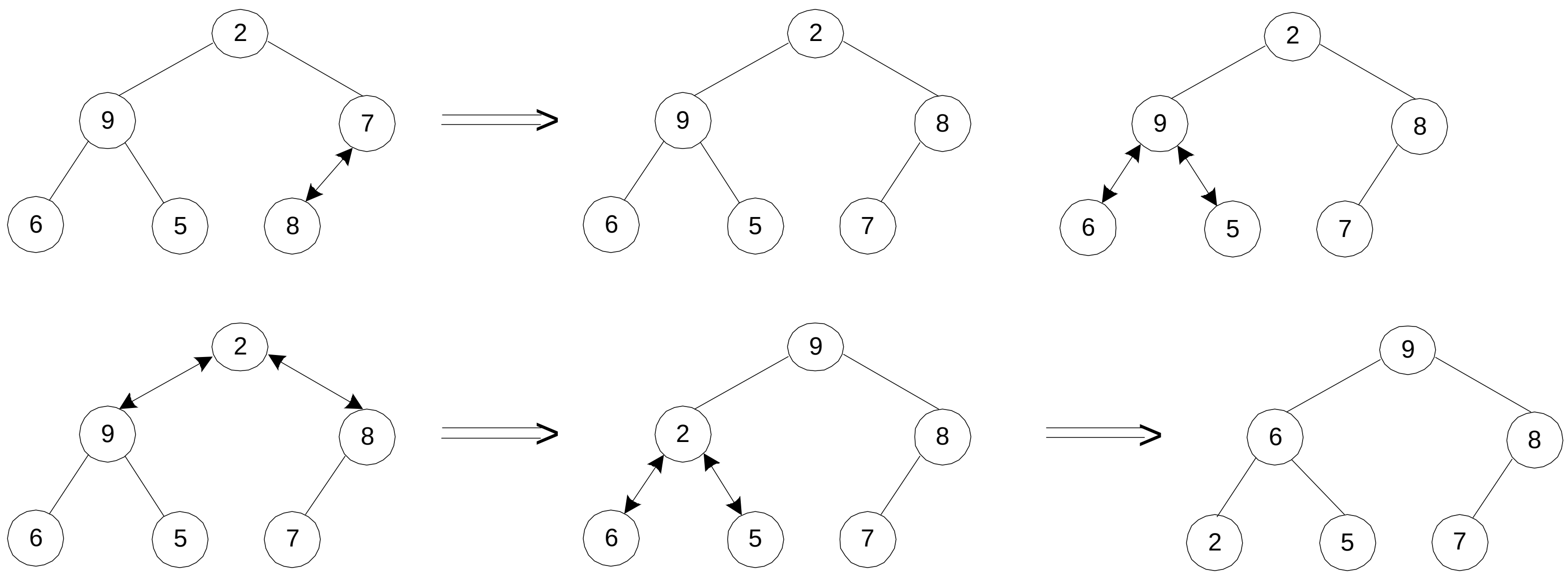
```
ALGORITHM MaxHeapify(H, i)
l ← LEFT(i)
r ← RIGHT(i)
if l <= n and H[l] > H[i]
   then largest ← l
   else  largest ← i
if r <= n and H[r] > H[largest]
   then largest ← r
if largest ≠ i
   then exchange H[i] ←→ H[largest]
        MaxHeapify(H, largest)
```

Heaps and Heapsort

Heaps Construction

Bottom-up Heap construction('cont)

- *Example 1: Construct a heap for the list 2, 9, 7, 6, 5, 8*



- *Example 2: 4 1 3 2 16 9 10 14 8 7 → 16 14 10 8 7 9 3 2 4 1*

Heaps and Heapsort

Heaps Construction

Worst-Case Efficiency for Bottom-up

- *assume n* $=2^k-1$, *so the heap is full, the maximum number of nodes occurs on each level*
- *Worst case: each key on level i will travel to the leaf level h*
  - *height of the tree* $h = \lfloor \log_2 n \rfloor$
  - *moving to the level down needs two comparisons*
    - *one to find the larger child*
    - *one to determine whether the exchange is required*
  - *number of key comparisons for a key on level i: 2(h-i)*

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{nodes at level i}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$$

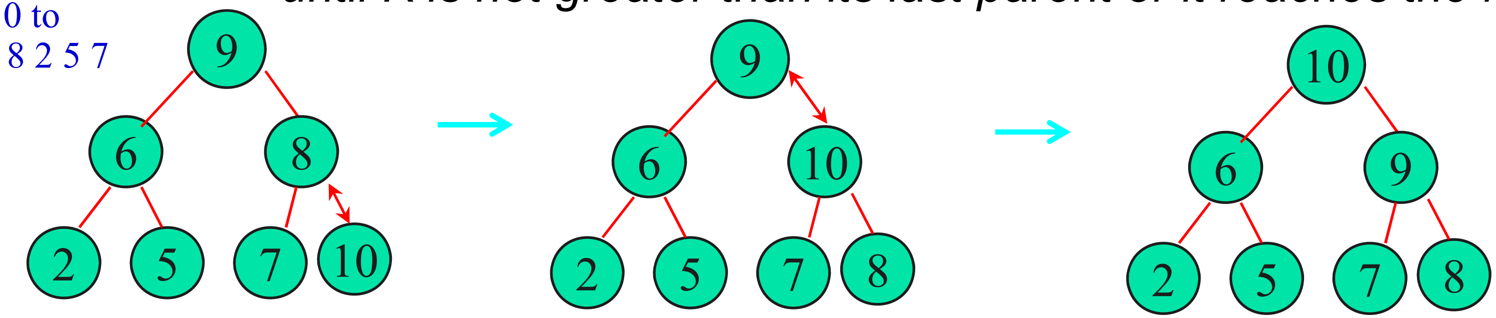
Heaps and Heapsort

Heaps Construction

Top-down Heap Construction

- *Successive insertions of new key into a previously constructed heap*
- *Insertion of a new key K*
  - *Insert the new node with key K at the last position in heap, i.e. after the last leaf of the existing heap*
  - *sift K up to its appropriate position*
    - *Compare with its parent, and exchange them if it violates the parental dominance condition.*
    - *Continue comparing the element with its new parent,*
    - *until K is not greater than its last parent or it reaches the root*

Ex: add 10 to
heap: 9 6 8 2 5 7



Heaps and Heapsort

Heaps Construction

Efficiency for Top-down

- *height of a heap with n node:* $h = \lfloor \log_2 n \rfloor$
- *Inserting one new element to a heap with* n-1 *nodes requires no more comparisons than the heap's height*
- *time efficiency for Top-down insertion is* $O(\log_2 n)$

Heaps and Heapsort

Heaps Construction

Root Deletion

- *swap the root with the last leaf K*
- *Decrease the heap's size by* 1
- *Heapify the smaller tree by sifting K down the tree, in exactly the same way in Bottom-up Heap construction*
  - *verify the parental dominance for K,*
  - *if it holds, we done.*
  - *if not, swap K with the larger of its children*
  - *and repeat this operation until parental dominance holds for K in its new position.*

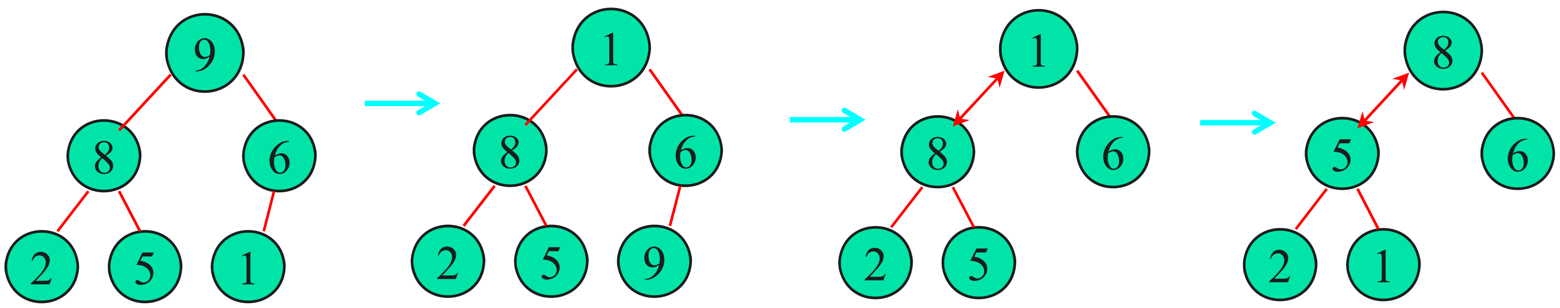
Heaps and Heapsort

Heaps Construction

Efficiency for Root Deletion

- *It can't make key comparison more than twice the heap's height*
- *Efficiency: $2h \in \Theta(\log n)$*

Ex: 9 8 6 2 5 1



Heaps and Heapsort

Heapsort

Heapsort algorithm

- Heap construction: Build heap for a given array (either bottom-up or top-down)
- Maximum deletion: Apply the root-deletion operation n-1 times to the remaining heap until heap contains just one node.

→ resulting: the array elements are eliminated in decreasing order

Heaps and Heapsort

Heapsort

Analysis of Heapsort

- *Bottom-up heap construction* $\mathrm{O}(n)$
- *Root deletion, Repeat n-1 times until heap contains just one node*

$$C_2(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \ldots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$

$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n\log_2 n \in O(n\log n)$$

- *Analysis shows that $C_1(n)+C_2(n)$=**Θ(nlogn),** in both the worst and average cases, the same class as mergesort*
- *But not require extra storage ---implemented with arrays*
- *Experiments show that heapsort runs more slowly than quicksort but competitive with mergesort*

Horner's Rule- Representation change

Horner's Rule For Polynomial Evaluation 霍纳法则

- Problem

  Polynomial Evaluation: Compute the value of a polynomial

  $$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \qquad (1)$$

  at a specific point x --- fast Fourier Transform, FFT

- Two brute-force algorithms

$p \leftarrow 0$
for $i \leftarrow n$ down to 0 do
  $power \leftarrow 1$
  for $j \leftarrow 1$ to i do
    $power \leftarrow power * x$
  $p \leftarrow p + a_i * power$
return p

$p \leftarrow a_0$; $power \leftarrow 1$
for $i \leftarrow 1$ to n do
  $power \leftarrow power * x$
  $p \leftarrow p + a_i * power$
  return p

Horner's Rule- Representation change

Horner's Rule For Polynomial Evaluation

Horner's Rule --Representation change

- *Obtained from* (1), *successively taking x as a common factor in the remaining polynomials of diminishing degrees*

$$p(x) = (\ldots (a_n x + a_{n-1})\, x + \ldots)\ \ x + a_0 \qquad (2)$$

E.g.: $p(x) = 2x^4 - x^3 + 3x^2 + x - 5 = x(2x^3 - x^2 + 3x + 1) - 5 =$

$= x(x(2x^2 - x + 3) + 1) - 5 = x(x(x(2x - 1) + 3) + 1) - 5$

To evaluate p(x) at x=3

coefficients	**2**	**-1**	**3**	**1**	**-5**
x=3	2	3*2+(-1) = 5	3*5+3=18	3*18+1=55	3*55+(-5)=160

Horner's Rule- Representation change

Horner's Rule For Polynomial Evaluation

Horner's Rule --Representation change

- *Such calculation could be organized with a two-row table*
- *first row: the polynomial's coefficients (including those equal to zero); listed from the highest a_n to lowest a_0*
- *second row: first entry is a_n; the next entry is computed as the x's value times the last entry plus the current coefficient in the first row*
- *The final entry is the value to sought*

ALGORITHM *Horner*($P[0..n]$, x)
//Evaluates a polynomial at a given point by Horner's rule
//Input: An array $P[0..n]$ of coefficients of a polynomial of degree n
// (stored from the lowest to the highest) and a number x
//Output: The value of the polynomial at x
$p \leftarrow P[n]$
for $i \leftarrow n - 1$ **downto** 0 **do**
  $p \leftarrow x * p + P[i]$
return p

Horner's Rule- Representation change

Horner's Rule For Polynomial Evaluation

Efficiency of Horner's Rule

- *Number of multiplications =*

  Number of additions = $\sum_{i=0}^{n-1} 1 = n$

By-products

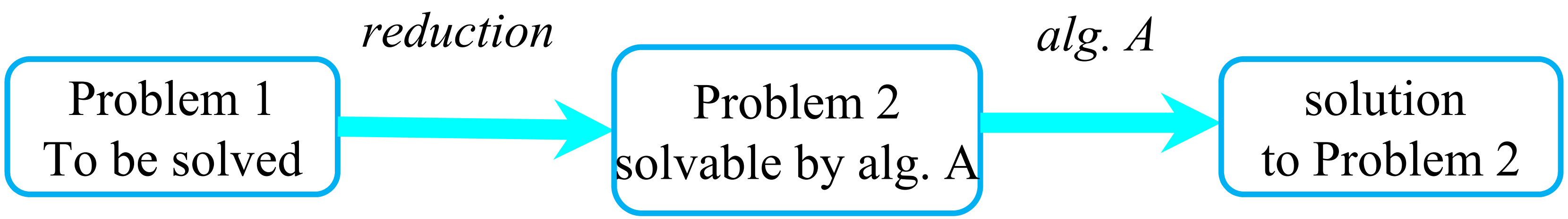
- ***synthetic division***: *The intermediate numbers generated by Horner's Rule alg. In the process of evaluating p(x) at some point x_0 are the coefficients of the quotient of the division of p(x) by x-x_0 ,*

- *The final result $p(x_0)$ equal to the remainder of the above division*

Problem Reduction

Problem Reduction

- *To solve a problem, reduce it to another problem that you know how to solve*



two points:

- *finding a problem to which the problem at hand should be reduced*
- *reduction-based algorithm to be more efficient than solving the original problem directly*

Problem Reduction

Problem Reduction

E.g. in analytical geometry, for three arbitrary points in the plane, $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, $p_3 = (x_3, y_3)$, the determinant is positive if and only if the point p_3 is to the left of the directed line through points p_1 p_2

$$\det\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_2y_3 + x_3y_1 - x_3y_2 - x_2y_1 - x_1y_3$$

i.e. we *reduce* a geometric problem about the relative locations of three points to a problem about the sign of a determinant.

☆ *the entire idea of analytical geometry is based on reducing geometric problems to algebra ones.*

Problem Reduction

Computing the Least Common Multiple

- *the Least Common Multiple of two positive integers m and n,* ***lcm (m, n) :***

  the smallest integer that is divisible by both m and n 最小公倍数

Middle-school method:

- *find the prime factorizations of m and n;*
- *lcm(m,n) be computed as:*

  *(product of all the common prime factors of m and n) **

  *(product of m's prime factors that are not in n) **

  (product of n's prime factors that are not in m)

$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$

$$lcm(24,60) = (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 = 120$$

$$\gcd(24,60) = 2 \cdot 2 \cdot 3 = 12$$

Problem Reduction

Computing the Least Common Multiple

- *Problem Reduction based method:*

  - *the product of lcm(m,n) and gcd(m,n) includes every factor of m and n exactly once, so*

$$lcm(m.n) = \frac{m \cdot n}{\gcd(m,n)}$$

  - *the problem of lcm is reduced to the problem of gcd and product*

Problem Reduction

Linear programming

- Linear programming:
  - a problem of optimizing a linear function of several variables subject to constraints in the form of linear equations and linear inequalities.

Maximize(or minimize) $c_1x_1 + \ldots c_nx_n$

Subject to $a_{i1}x_1 + \ldots + a_{in}x_n \le (\text{or} \ge \text{or} =)\ b_i$, for $i=1\ldots n$

$x_1 \ge 0, \ldots, x_n \ge 0$

Problem Reduction

Linear programming

Algorithms for Linear programming:

- *simplex method: worst-case efficiency is to be exponential*
- *Ellipsoid algorithm: polynomial time.*
- *Interior-point methods: polynomial time*
- *Karmarkar's alg.: polynomial worst-case efficiency*
- *Integer Linear programming: the variables of a Linear programming problem are required to be integers.*
  - *no known polynomial-time alg.*
  - *branch-and-bound method for solving Integer Linear programming*

Problem Reduction

Linear programming

Investment Problem:

- **Scenario**
  - *A university endowment needs to invest \$100million*
  - *Three types of investment:*
    - *Stocks (expected interest: 10%)*
    - *Bonds (expected interest: 7%)*
    - *Cash (expected interest: 3%)*
- **Constraints**
  - *The investment in stocks is no more than 1/3 of the money invested in bonds*
  - *At least 25% of the total amount invested in stocks and bonds must be invested in cash*
- ***Objective**:*
  - *An investment that maximizes the return*

Problem Reduction

Linear programming

Investment Problem: (‘cont)

- ***mathematical model***

$$\text{Maximize} \quad 0.10x + 0.07y + 0.03z$$

$$\text{subject to} \quad x + y + z = 100$$

$$x \le (1/3)y$$

$$z \ge 0.25(x + y)$$

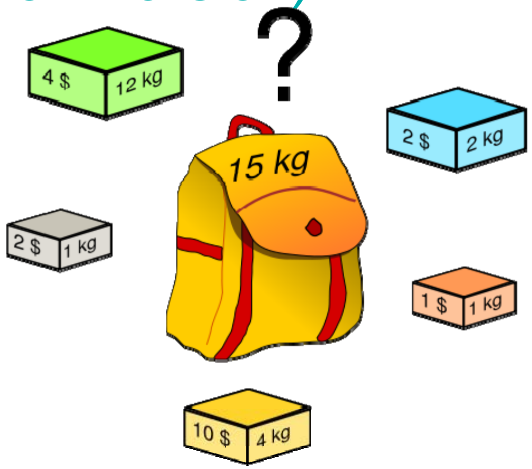
$$x \ge 0,\ y \ge 0,\ z \ge 0$$

optimal decision making problem ----→ linear programming problem

Problem Reduction

Linear programming

Knapsack Problem (Continuous/Fraction Version):



- ***Scenario***
  - *Given n items:*
    - *weights:* $w_1 \quad w_2 \ldots \quad w_n$
    - *values:* $v_1 \quad v_2 \ldots \quad v_n$
    - *a knapsack of capacity W*
- ***Constraints***
  - *Any fraction of any item can be put into the knapsack,* x_i
- ***Objective****:*
  - *Find the most valuable subset of the items*

Problem Reduction

Linear programming

- Knapsack Problem (Continuous/Fraction Version): (‘cont)

  - ***mathematical model***

Maximize

$$\sum_{i=1}^{n} v_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$0 \leq x_i \leq 1 \qquad \text{for } i = 1, \ldots, n$$

Problem Reduction

Linear programming

Knapsack Problem (Discrete Version)

- ***Scenario***
  - *Given n items:*
    - *weights:* $w_1 \quad w_2 \ldots \quad w_n$
    - *values:* $v_1 \quad v_2 \ldots \quad v_n$
    - *a knapsack of capacity W*
- ***Constraints***
  - *an item can either be put into the knapsack in its entirely or not be put into the knapsack.*
- ***Objective**:*

  Find the most valuable subset of the items

Problem Reduction

Linear programming

Knapsack Problem (Discrete Version) (‘cont)

- ***mathematical model***

Maximize

$$\sum_{i=1}^{n} v_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \le W$$

$$x_i \in \{0,1\} \qquad \text{for } i = 1,\ldots,n$$

Problem Reduction

Reduction to Graph

- *many problems can be solved by reduction to one of the standard graph problems*
- *state-space graph: vertices of a graph represent possible states of the problem, edges indicate permitted transitions among such states*
- *one of the graph's vertices represents the initial state, another represents a goal state of the problem*
- *puzzles and games*
- *not always a straightforward task*

problem ----→ a path from the initial-state vertex to a goal-state vertex

Problem Reduction

Reduction to Graph

- *River-crossing puzzle*

  • **Problem**: *The wolf, goat and bag of cabbage puzzle.*

  - *A peasant must transport a wolf, goat and bag of cabbage from one side of a river to another using a boat*
  - *the boat can only hold one item in addition to the peasant ,*
  - *subject to the constraints that the wolf cannot be left alone with the goat , and the goat cannot be left alone with the cabbage .*

Problem Reduction

Reduction to Graph

River-crossing puzzle

- ***state-space graph***