General Idea of Transform & Conquer

1. Transform the original problem instance into a different problem instance
2. Solve the new instance
3. Transform the solution of the new instance into the solution for the original instance
Varieties of Transform & Conquer
[Levitin]

- Transform to a simpler or more convenient instance of the same problem
  - “instance simplification”
- Transform to a different representation of the same instance
  - “representation change”
- Transform to an instance of a different problem with a known solution
  - “problem reduction”
Instance Simplification: Presorting

- Sort the input data first
- This simplifies several problems:
  - checking whether a particular element in an array is unique
  - computing the median and mode (value that occurs most often) of an array of numbers
  - searching for a particular element
    - once array is sorted, we can use the decrease & conquer binary search algorithm
  - used in several convex hull algorithms
Instance Simplification: Solving System of Equations

- A system of $n$ linear equations in $n$ unknowns:
  - $a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$
  - $\ldots$
  - $a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$

- Cast as a matrix problem:
  - $Ax = b$, where $A$ is $n \times n$ matrix, $x$ and $b$ are $n$-vectors

- To solve for all the $x$'s, solve $Ax = b$ for $x$
Motivation for Solving Systems of Linear Equations

- http://aix1.uottawa.ca/~jkhoury/system.html
  - geometry
  - networks
  - heat distribution
  - chemistry
  - economics
  - linear programming
  - games
Solving System of Equations

- One way to solve $Ax = b$ for $x$:
  - Compute $A^{-1}$
  - Multiply both sides by $A^{-1}$
  - $A^{-1}Ax = A^{-1}b$
  - $x = A^{-1}b$

- Drawback is that computing matrix inverses suffers from numerical instability in practice

- Try another approach...
LUP Decomposition

If $A$ is triangular, solving $Ax = b$ for $x$ is easy and fast using successive substitutions (how fast?)

Transform this problem into one involving only triangular matrices

- instance simplification!

Find

- $n \times n$ matrix $L$ with all 1’s on diagonal and all 0’s above the diagonal (“unit lower-triangular”)
- $n \times n$ matrix $U$ with all 0’s below the diagonal (“upper-triangular”)
- $n \times n$ matrix $P$ of 0’s and 1’s with exactly one 1 in each row and each column (“permutation matrix”)

such that $PA = LU$
Using LUP Decomposition

- We want to solve $Ax = b$.
- Assume we have $L$, $U$ and $P$ with desired properties so that $PA = LU$
- Multiply both sides of $Ax = b$ by $P$ to obtain $PAX = Pb$
  - Since $P$ is a permutation matrix, $Pb$ is easy to compute and is just a reordering of the vector $b$, call it $b'$
- Substitute $LU$ for $PA$ to obtain $LUx = b'$
- Let $y$ be the vector (as of yet unknown) that equals $Ux$; rewrite as $Ly = b'$
  - although $U$ is known, $x$ is not yet known
- Solve $Ly = b'$ for $y$
  - since $L$ is triangular, this is easy
- Now that $y$ is known, solve $y = Ux$ for $x$
  - since $U$ is triangular, this is easy
Solving $Ax = b$ with LUP Decomp.

- Assuming the L, U, and P are given, pseudocode is on p. 817 of [CLRS]
- Running time is $\Theta(n^2)$
- Example: <board>

- Calculating L, U and P is more involved and takes $\Theta(n^3)$ time. (See [CLRS].)
Instance Simplification: Balanced Binary Search Trees

- Transform an unbalanced binary search tree into a balanced binary search tree
- Benefit is guaranteed $O(\log n)$ time for searching, inserting and deleting as opposed to possibility of $\Theta(n)$ time
- Examples:
  - AVL trees
  - red-black trees
  - splay trees
Representation Change: Balanced Search Trees

- Convert a basic binary search tree into a search tree that is more than binary:
  - a node can have more than two children
  - a node can store more than one data item

- Can get improved performance (w.r.t. constant factors)

- Examples:
  - 2-3 trees
  - B-trees
B-Trees: Motivation

- Designed for very large data sets that cannot all fit in main memory at a time
- Instead, data is stored on disk
- **Fact 1:** Disk access is *orders of magnitude slower* than main memory access
- Typically a disk access is needed for each node encountered during operations on a search tree
  - For a balanced binary search tree, this would be about $c \log_2 n$, where $c$ is a small constant and $n$ is number of items
B-Trees: Motivation

- Can we reduce the time?
- Even if not asymptotically, what about reducing the constants?
  - Constants do matter
- Reduce the height by having a bushier tree
  - have more than two children at each node
  - store more than two keys in each node
- Fact 2: Each disk access returns a fixed amount of information (a page).
  - Size is determined by hardware and operating system
  - Typically 512 to 4096 bytes
- Let size of tree node be page size
B-Tree Applications

- Keeping index information for large amounts of data stored on disk
  - databases
  - file systems
B-Tree Definition

- B-tree with minimum degree $t$ is a rooted tree such that
  1. each node has between $t-1$ and $2t-1$ keys, in increasing order (root can have fewer keys)
  2. each non-leaf node has one more child than it has keys
  3. all keys in a node’s $i$-th subtree lie between the node’s $(i-1)$st key and its $i$-th key
  4. all leaves have the same depth

- Points 1-3 are generalization of binary search trees to larger branching factor
- Point 4 controls the height
B-Tree Example

- B-tree with minimum degree 2
  1. each node has between 1 and 3 keys, in sorted order
  2. each non-leaf node has 2 to 4 children, one more than number of keys
  3. keys are in proper subtrees
  4. all leaves have depth 1
B-Tree Height

- **Theorem:** Any n-key B-tree with minimum degree t has height $h \leq \log_t((n+1)/2)$.

- Height is still $O(\log n)$ but logarithm base is t instead of 2
  - savings in constant factor of $\log_2 t$, which is substantial since t is generally very large
  - Remember: $\log_2 x = (\log_2 t) \times (\log_t x)$

- **Proof:** Calculate minimum number of keys in a B-tree of height $h$ and solve for $h$. 
Searching in a B-Tree

- Straightforward generalization of searching in a binary search tree

- To search for $k$, start at root:
  1. Find largest $i$ such that $k \leq i^{th}$ key in current node
  2. If $k = i^{th}$ key then return “found”
  3. Elseif current node is a leaf then return “not found”
  4. Else recurse on root of $i^{th}$ subtree
Running Time of B-Tree Search

- **CPU time:**
  - Line 1 takes $O(t)$ (or $O(\log_2 t)$ if using binary search)
  - Number of recursive calls is $O(\text{height}) = O(\log_t n)$
  - Total is $O(t \log_t n)$

- **Number of disk accesses:**
  - each recursive call requires at most one disk access, to get the next node
  - $O(\log_t n)$ (the height)
B-Tree Insert

To insert a new key, need to

- obey bounds on branching factor / maximum number of keys per node
- keep all leaves at the same depth

Do some examples on a B-tree with minimum degree 2

- each node has 1, 2, or 3 keys
- each node has 2, 3, or 4 children
B-Tree Insert Examples

insert C
B-Tree Insert Examples

insert M

M goes in a full node; split the node in two; promote the median L; insert M
B-Tree Insert Examples

B goes in full leaf, so split leaf and promote median C. C goes in full root, so split root and promote median L to make a new root (only way height increases). But this is a 2-pass algorithm => twice as many disk accesses. To avoid 2 passes, search phase always recurses down to a non-full node...
To insert B, start at root to find proper place; proactively split root since it is full.
B-Tree Insert with One Pass

Recurse to node containing F; since not full no need to split.

Recurse to left-most leaf, where B belongs.
Since it is full, split it, promote the median C to the parent, and insert B.
B-Tree Insert with One Pass

Final result of inserting B.
Splitting a B-Tree Node

split(x,i,y)

input:
- non-full node x
- full node y which is the ith child of x

result:
- split y into two equal size nodes with t-1 keys each
- insert the median key of the old y into x
Splitting a B-Tree Node

\[ x: < 2t-1 \text{ keys} \]

\[ y: \begin{array}{ccc}
\alpha & m & \beta \\
\end{array} \]

\[ \rightarrow \]

\[ x: \begin{array}{c}
\alpha \\
\end{array}, \begin{array}{c}
m \\
\end{array}, \begin{array}{c}
\beta \\
\end{array}, \begin{array}{c}
t-1 \text{ keys} \\
\end{array} \]

\[ y: \begin{array}{c}
\alpha \\
\end{array}, \begin{array}{c}
\beta \\
\end{array} \]

\[ \leq 2t-1 \text{ keys} \]
B-Tree Insert Algorithm

- if root $r$ is full ($2t-1$ keys) then
  - allocate a new node $s$
  - make $s$ the new root
  - make $r$ the first child of $s$
  - split($s, 1, r$)
  - insert-non-full($s, k$)
- else insert-non-full($r, k$)
B-Tree Insert Algorithm (cont’d)

- procedure insert-non-full(x,k):
  - if x is a leaf then
    - insert k in sorted order
  - else
    - find node y that is root of subtree where k belongs
    - if y is full then split it
    - call insert-non-full recursively on correct child of x
      (y if no split,  
      1\textsuperscript{st} half of y if split and k < median of y,  
      2\textsuperscript{nd} half of y if split and k > median of y)
Running Time of B-Tree Insert

- Same as search:
  - $O(t \log_t n)$ CPU time
  - $O(\log_t n)$ disk access

- Practice (Homework?): insert F, S, Q, K, C, L, H, T, V, W into a B-tree with minimum degree $t = 3$
Deleting from a B-Tree

Pitfalls:

- Be careful that a node does not end up with too few keys
- When deleting from a non-leaf node, need to rearrange the children (remember, number of children must be one greater than the number of keys)
B-Tree Delete Algorithm

delete(x,k):  // called initially with x = root

1. if k is in x and x is a leaf then
   delete k from x  // we will ensure that x has ≥ t keys

2. if k is in x and x is not a leaf then

```
x  k
  
  y    z
```
B-Tree Delete Algorithm (cont’d)

2(a) if y has \( \geq t \) keys then

- find \( k' = \text{pred}(k) \) // in y’s subtree
- delete(\( y, k' \)) // recursive call
- replace k with \( k' \) in x
B-Tree Delete Algorithm (cont’d)

2(b) else if z has ≥ t keys then

find $k’ = \text{succ}(k)$ // in z’s subtree

delete(z, k’) // recursive call

replace k with $k’$ in x
2(c) else // both y and z have < t keys
merge y, k, z into a new node w
delete(w,k) // recursive call
B-Tree Delete Algorithm (cont’d)

3. if k is not in (internal) node x then
   let y be root of x’s subtree where k belongs
3(a) if y has < t keys but has a neighboring sibling z with ≥ t keys then
   y borrows a key from z via x // note moving subtrees
B-Tree Delete Algorithm (cont’d)

3. if k is not in (internal) node x then
   let y be root of x’s subtree where k belongs
3(b) if y has < t keys and has no neighboring sibling z
   with ≥ t keys then
   merge y with sibling z, using intermediate key in x

whether (a), (b) or neither was done, call delete(y,k)
Behavior of B-Tree Delete

- As long as $k$ has not yet been found, we continue in a single downward pass, with no backtracking.
- If $k$ is found in an internal node, we may have to find pred or succ of $k$, call it $k'$, delete $k'$ from its old place, and then go back to where $k$ is and replace $k$ with $k'$.
- However, finding and deleting $k'$ can be done in a single downward pass, since $k'$ will be in a leaf (basic property of search trees).
- $O(\log_t n)$ disk access
- $O(t \log_t n)$ CPU time
Problem Reduction: Computing Least Common Multiple

- \( \text{lcm}(m,n) \) is the smallest integer that is divisible by both \( m \) and \( n \)
  - Ex: \( \text{lcm}(11,5) = 55 \) and \( \text{lcm}(24,60) = 120 \)

- One algorithm for finding \( \text{lcm} \): multiply all common factors of \( m \) and \( n \), all factors of \( m \) not in \( n \), and all factors of \( n \) not in \( m \)
  - Ex: \( 24 = 2*2*2*3, 60 = 2*2*3*5 \),
    \( \text{lcm}(24,60) = (2*2*3)*2*5 = 120 \)

- But how to find prime factors of \( m \) and \( n \)?
Reduce Least Common Multiple to Greatest Common Denominator

- Try another approach.
- \( \gcd(m,n) \) is product of all common factors of \( m \) and \( n \)
- So \( \gcd(m,n) \cdot \text{lcm}(m,n) \) includes every factor in both \( \gcd \) and \( \text{lcm} \) twice, every factor in \( m \) but not \( n \) exactly once, and every factor in \( n \) but not \( m \) exactly once
- Thus \( \gcd(m,n) \cdot \text{lcm}(m,n) = m \cdot n \).
- I.e., \( \text{lcm}(m,n) = \frac{m \cdot n}{\gcd(m,n)} \)
- So if we can solve \( \gcd \), we can solve \( \text{lcm} \)
- And we can solve \( \gcd \) with Euclid’s algorithm
Problem Reduction: Computing Number of Paths in a Graph

How many paths of length 3 are there in this graph between b and d?
Computing Number of Paths in a Graph

Claim: Adjacency matrix $A$ to the $k$-th power gives number of paths of length (exactly) $k$ between all pairs

Reduce problem of computing number of paths to problem of multiplying matrices!
Proof of Claim

- **Basis:** $A^1 = A$ gives all paths of length 1

- **Induction:** Suppose $A^k$ gives all paths of length $k$. Show for $A^{k+1} = A^k A$.

- $(i,j)$ entry of $A^{k+1}$ is sum, over all vertices $h$, of $(i,h)$ entry of $A^k$ times $(h,j)$ entry of $A$:
Computing Number of Paths of length $k$

- We have to compute $A^k$.
- Do $k-1$ matrix multiplications
  - brute force or Strassen’s
  - $O(kn^3)$ or $O(kn^{2.8\ldots})$ running time
- Or, do successive doubling ($A^2$, $A^4$, $A^8$, $A^{16}$,...)
  - about $\log_2 k$ multiplications
  - $O(n^3\log k)$ or $O(n^{2.8\ldots\log k})$ running time
Problem Reduction Tool: Linear Programming

- Many problems related to finding an optimal solution for something can be reduced to an instance of the linear programming problem:
  - optimize a linear function of several variables subject to constraints
    - each constraint is a linear equation or linear inequality
Linear Program Example

- An organization wants to invest $100 million in stocks, bonds, and cash.
- Assume interest rates are:
  - stocks: 10%
  - bonds: 7%
  - cash: 3%
- Institutional restrictions:
  - amount in stock cannot be more than a third of amount in bonds
  - amount in cash must be at least a quarter of the amount in stocks and bonds
- How should money manager invest to maximize return?
Mathematical Formulation of the Example

- $x = \text{amount in stocks (in millions of dollars)}$
- $y = \text{amount in bonds}$
- $z = \text{amount in cash}$

maximize $(.10)*x + (.70)*y + (.03)*z$

subject to
- $x+y+z = 100$
- $x \leq y/3$
- $z \geq (x+y)/4$
- $x \geq 0, y \geq 0, z \geq 0$
General Linear Program

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

subject to

$a_{11}x_1 + \ldots + a_{1n}x_n \leq (or \geq or =) b_1$

$a_{21}x_1 + \ldots + a_{2n}x_n \leq (or \geq or =) b_2$

\ldots

$a_{m1}x_1 + \ldots + a_{mn}x_n \leq (or \geq or =) b_m$

$x_1 \geq 0, \ldots, x_n \geq 0$
Linear Programs with 2 Variables

maximize $x_1 + x_2$

subject to

$4x_1 - x_2 \leq 8$

$2x_1 + x_2 \leq 10$

$5x_1 - 2x_2 \geq -2$

$x_1, x_2 \geq 0$

feasible region

$x_1 = 2, x_2 = 6$ is optimal solution
Solving a Linear Program

Given a linear program, there are 3 possibilities:

- the feasible region is empty
- the feasible region and the optimal value are unbounded
- the feasible region is bounded and there is an optimal value

Three ways to solve a linear program: most common in practice

- simplex method: travel around the feasible region from corner to corner until finding optimal
  - worst-case exponential time, average case is polynomial time
- ellipsoid method: a divide-and-conquer approach
  - polynomial worst-case, but slow in practice
- interior point methods
  - polynomial worst-case, reasonable in practice
Use of Linear Programming

- Later we will study algorithms to solve linear programs.
- Now we’ll give some examples of converting other problems into linear programs.
Reducing a Problem to a Linear Program

- What unknowns are involved?
  - These will be the variables $x_1, x_2, \ldots$

- What quantity is to be minimized or maximized? How to express this quantity in terms of the variables?
  - This will be the objective function

- What are the constraints on the problem and how to state them w.r.t. the variables?
  - Constraints must be linear
Reducing a Problem to a Linear Program: Example

- A tailor can sew pants and shirts.
- It takes him 2.5 hours to sew a pair of pants and 3.5 hours to sew a shirt.
- A pair of pants uses 3 yards of fabric and a shirt uses 2 yards of fabric.
- The tailor has 40 hours available for sewing and has 50 yards of fabric.
- He makes a profit of $10 per pair of pants and $15 per shirt.
- How many pants and how many shirts should he sew to maximize his profit?
Reducing a Problem to a Linear Program: Example Solution

- **Variables:**
  - $x_1 = \text{number of pants to sew}$
  - $x_2 = \text{number of shirts to sew}$

- **Objective function:**
  - maximize $10x_1 + 15x_2$

- **Constraints:**
  - time: $(2.5)x_1 + (3.5)x_2 \leq 40$
  - fabric: $3x_1 + 2x_2 \leq 50$
  - nonnegativity: $x_1 \geq 0$, $x_2 \geq 0$
Knapsack Problem as a Linear Program

- Suppose thief can steal part of an object
  - “fractional” knapsack problem
- For each item $j$, $1 \leq j \leq n$,
  - $v_j$ is value of (entire) item $j$
  - $w_j$ is weight of (entire) item $j$
  - $x_j$ is fraction of item $j$ that is taken
- Maximize $v_1x_1 + \ldots + v_nx_n$
- Subject to
  - $w_1x_1 + \ldots w_nx_n \leq W$ (knapsack limit)
  - $0 \leq x_j \leq 1$, for $j = 1,\ldots,n$
A Shortest Path Problem as a Linear Program

- What is the shortest path distance from s to t in weighted directed graph $G = (V,E,w)$?
- For each $v$ in $V$, let $d_v$ be a variable modeling the distance from $s$ to $v$.

maximize $d_t$
subject to

$d_v \leq d_u + w(u,v)$ for each $(u,v)$ in $E$

$d_s = 0$