ch16: Greedy Algorithms

20 Nov 2012
CMPT231
Dr. Sean Ho
Trinity Western University

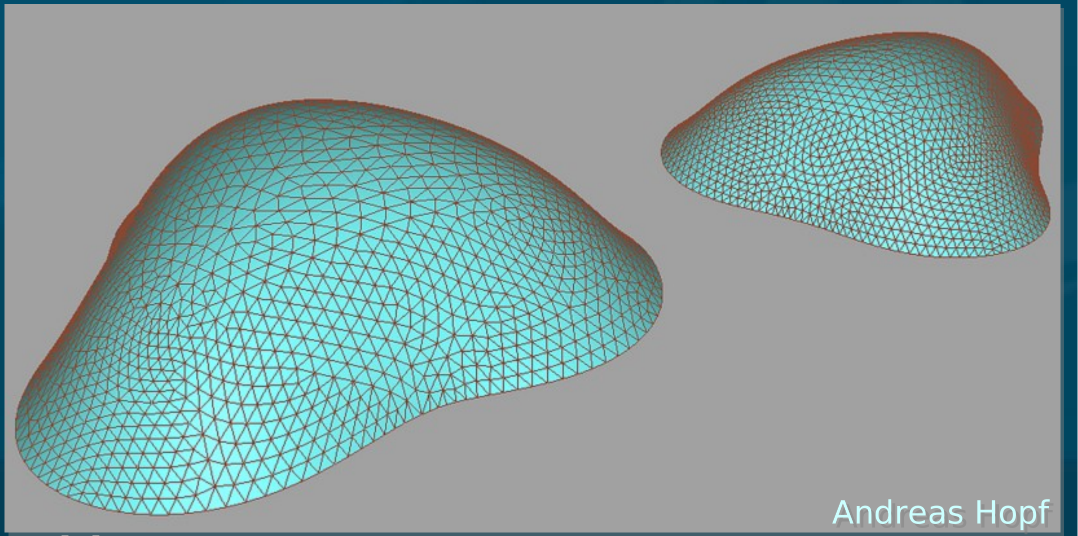
TRINITY WESTERN UNIVERSITY

Outline for today

- Greedy algorithms
  - Activity selection
  - Fractional knapsack problem
  - Huffman coding
- Intro to graph algorithms
  - Breadth-first search
- Review ch10,12,18,15

Greedy algorithms

- Another approach to optimisation
  - Faster than dynamic programming, when applicable
- At each decision point, go for immediate gains
  - Locally optimal choices ⇒ global optimum
- Not all problems have optimal substructure
  - Hybrid optimisation strategies use large jumps to get to right "hill", then greedy "hill-climbing" to get to the top



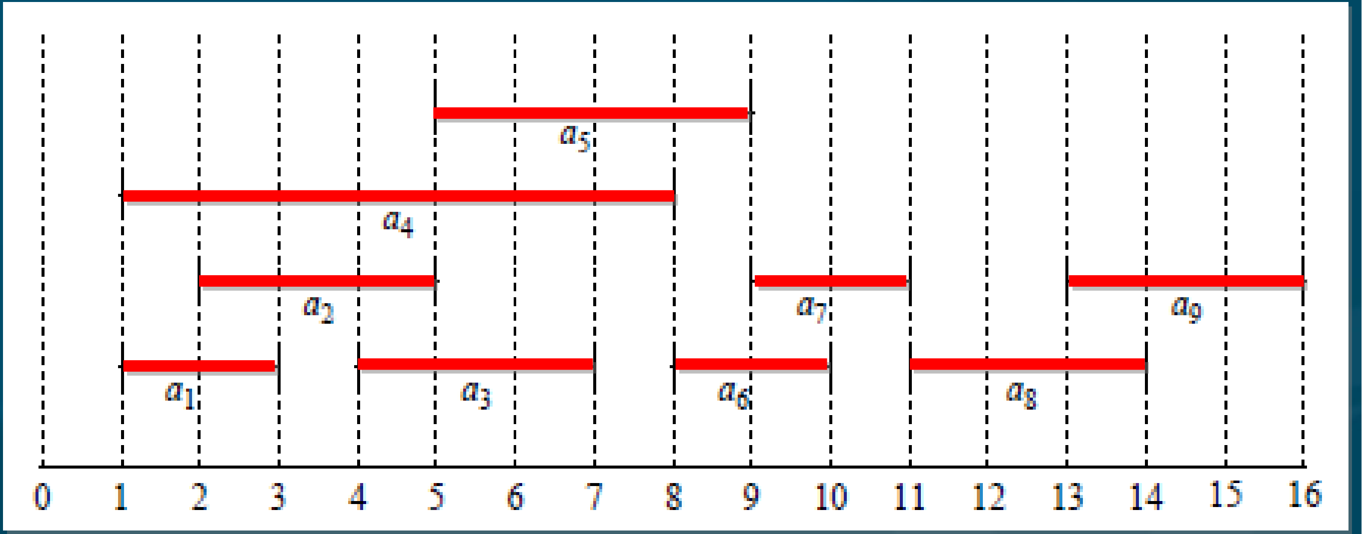
Problem-solving outline

- Find optimal substructure (e.g., recurrence)
- Convert to naïve recursive solution (code)
  - Could then be converted to dynamic prog.
- Use greedy choice to simplify the recurrence so only one subproblem remains
  - Don't have to iterate through all subproblems
  - Prove greedy choice yields global optimum!
- Convert to recursive greedy solution
- Convert to iterative greedy solution

Example: activity selection

- Activities S = {a_1, ..., a_n} which each require exclusive use of a shared resource
  - Each activity has start/finish times [s_i, f_i)
  - Activities are sorted by finish times
- ⇒ Find largest subset of S where all activities are non-overlapping
- e.g., a_2 and a_5 do not overlap:

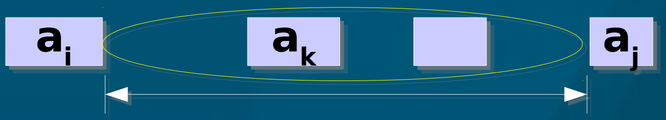
i	s	f
1	1	3
2	2	5
3	4	7
4	1	8
5	5	9
6	8	10
7	9	11
8	11	14
9	13	16



Solutions?

Solving: optimal substructure

- Let $S_{ij} = \{a_k \in S: f_i \le s_k < f_k \le s_j\}$: all activities that start after f_i and finish before s_j



  - Any activity in S_{ij} will be compatible with:
    - Any activity that finishes by f_i
    - Any activity that starts no earlier than s_j
- Let A_{ij} be a solution for S_{ij}: a largest mutually-compatible subset of activities

- Pick an activity $a_k \in A_{ij}$, and partition A_{ij} into
  - $A_{ik} = A_{ij} \cap S_{ik}$: those that finish before a_k starts
  - $A_{kj} = A_{ij} \cap S_{kj}$: those that start after a_k finishes

Proof of optimal substructure

- Claim: A_{ik} and A_{kj} are optimal solutions for S_{ik}, S_{kj}
- Proof (for A_{ik}): assume not:

  - Let A'_{ik} be a better solution:
    non-overlapping elements, and $|A'_{ik}| > |A_{ik}|$.
  - Then $A'_{ik} \cup \{a_k\} \cup A_{kj}$ would be a solution for S_{ij}, and its size is larger than $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$.
  - Contradicts the premise that A_{ij} was optimal.
- ⇒ Optimal substructure: split on a_k,
  recurse twice on S_{ik} and S_{kj},
  iterate over all choices of a_k and pick the best

Naive recursive solution

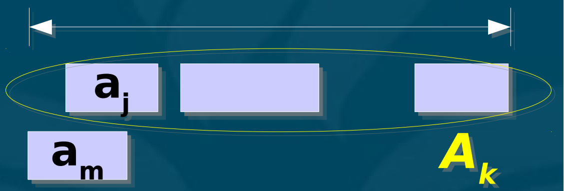
- Let c[i,j] = size of optimal solution for S_{ij}:
  - Splitting on a_k yields c[i,j] = c[i,k] + 1 + c[k,j]
  - Which choice of a_k is best? Naive: try all
- Recurrence: $c[i,j] = \max_{a\_k \in S\_ij} (c[i,k] + 1 + c[k,j])$
  - Base case: if $S_{ij} = \emptyset$, then c[i,j] = 0
- Could implement this using dynamic programming
  - Fill in 2D table for c[i,j], bottom-up
  - Auxiliary table storing the solutions A_{ij}
- With this problem, though, we can do better!

Greedy choice

- Which choice of a_k leaves as much as possible of the resource available for other activities?
  - One which finishes the earliest
  - Since activities are sorted by finish time, just choose the first activity!
- Recurrence simplifies: to find optimal subset of S_{kj}, include a_k, then recurse on $S_k = \{a_i: s_i \geq f_k\}$: those that start after a_k finishes
  - Don't need to iterate over all choices of a_k
- We need to prove the greedy choice is optimal

Proof of greedy choice

- Let $S_k \neq \emptyset$ with $a_m \in S_k$ having earliest finish time.
  - Claim: $\exists$ optimal soln for S_k which includes a_m.
- Proof: Let A_k be an optimal solution for S_k.
  - If it includes a_m, then we're done.
- If not, let a_j be the first in A_k to finish.
  - Swap out a_m for a_j: let $A'_k = A_k - \{a_j\} \cup \{a_m\}$.
- Then A'_k is an optimal solution for S_k:
  - Size is same as A_k, and
  - Elements are non-overlapping: $f_m \leq f_j$



Recursive greedy solution

- Input: arrays s[], f[], with f[] sorted
  - Add a dummy entry f[0] = 0, so that $S_0 = S$.
- For each recursive subproblem S_k,
  - Skip over activities that overlap with a_k
  - Include the first activity that doesn't overlap, and recurse on the rest:
    - **def ActivitySel(s, f, k, n):**
      - **for m in k+1 .. n:**
        - **if (s[m] ≥ f[k]):**
          - **return {a_m} ∪ ActivitySel(s, f, m, n)**
      - **return NULL**
  - Initial call: ActivitySel(s, f, 0, n). ($\Theta(n)$!)

Iterative greedy solution

- Recursive solution is nearly tail-recursive, easy to convert to more efficient iterative solution:

```
def ActivitySel(s, f):
    A = {a_1}
    k = 1
    for m in 2 .. length(f):
        if (s[m] ≥ f[k]):
            A = A ∪ {a_m}
            k = m
    return A
```

i	s	f
1	1	3
2	2	5
3	4	7
4	1	8
5	5	9
6	8	10
7	9	11
8	11	14
9	13	16

- Complexity: $\Theta(n)$
  - If need to pre-sort on f[], then $\Theta(n \lg n)$

Greedy vs dynamic prog.

- Dynamic prog. more general
  - Not all problems have greedy property
- Dynamic prog. fills in table bottom-up
  - Greedy choice done top-down
- Choice in dyn. prog. needs all smaller subprobs
  - Greedy choice is simpler, so can make choice before solving subproblem
- Proving the greedy property:
  - Assume an optimal solution
  - Modify it to include the greedy choice
  - Show that it's still optimal

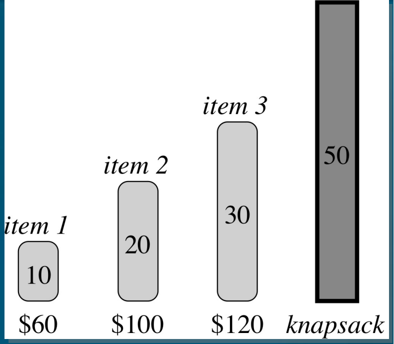
Optimising for greedy choice

- Often need to pre-process input to make the greedy choice easier
  - Sorted activities by finish time
  - Greedy choice can be done in O(1) each time
  - Sorting takes O(n lg n)
- If input is dynamically generated (can't sort whole list in advance), then
  - Priority queue: pop the most optimal choice

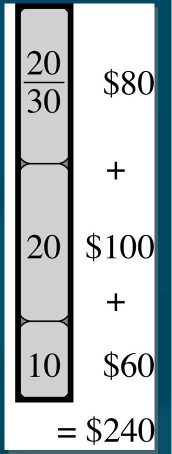
Outline for today

- Greedy algorithms
  - Activity selection
  - Fractional knapsack problem
  - Huffman coding
- Intro to graph algorithms
  - Breadth-first search
- Review ch10,12,18,15

Knapsack problem



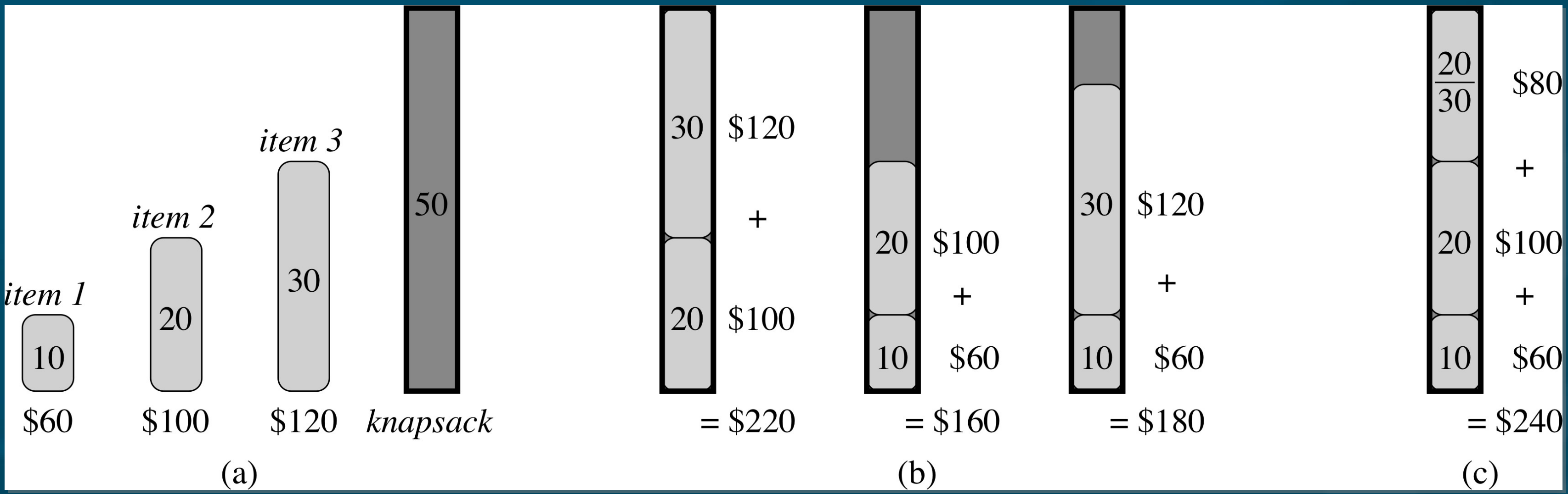
- Fractional knapsack problem:
  - n items, each with weight w_i and value v_i.
  - Maximise total value, subject to total weight W
  - Can take fractions of an item (think of liquids)
- Greedy soln: sort items by value-to-weight ratio
  - Greedy choice: take item with largest v_i / w_i.
  - Last spot may be filled with fractional item




```
def FractionalKnapsack(v, w, W):
    while totwt < W:
        add next item in decreasing order of value-to-weight
    replace last item with 1-(totwt-W) of itself
```

0-1 Knapsack

- Variant that does not allow fractions of an item
- Greedy strategy no longer works!
- Making initial locally-optimal choices locks us out of making later globally-optimal choices
- Still possible to solve using dynamic programming (Ex 16.2-2)



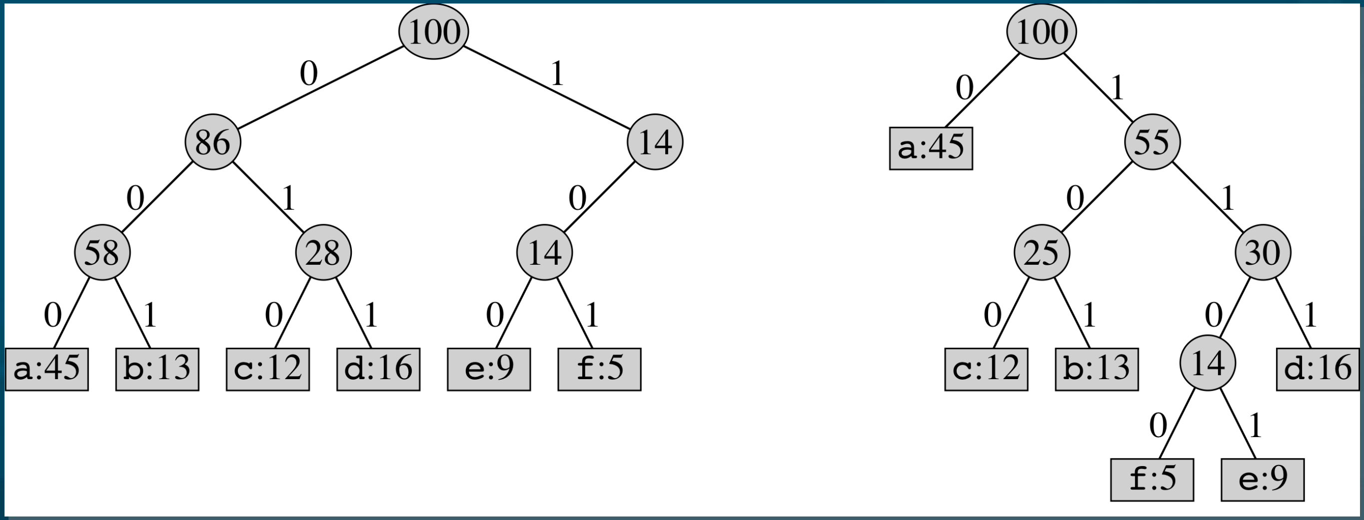
Encoding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Given a text with a known set of characters
  - Encode each character with a binary codeword
- Fixed-length code: all codewords same length
  - “cafe” ⇒ 010 000 101 100
- Variable-length code: some codes lower cost
  - “cafe” ⇒ 100 0 1100 1101
  - Compression: choose shorter codes for more frequent characters
- Prefix code: no code is a prefix of another
  - Unique parsing; don't need to delimit chars
  - “cafe” ⇒ 100011001101

Code trees

- Prefix code ⇒ code tree: binary tree where nodes represent prefixes; characters are at leaves
  - Fixed-length code ⇒ leaves all at same level
  - Decoding = walk down the tree
    - Cost of a char = depth in tree
- Total cost of encoding a file using a given tree:
  - Σ_c [freq(c) * depth(c)]



Huffman coding

- Build tree bottom-up
  - Start with two least-common chars
  - Merge to make new subtree with combined freq
- Min-priority queue manages the greedy choice
- Input: array of char nodes with .freq attribs

```
def huffman(chars):
    Q = new MinQueue(chars)
    for i in 1 .. length(chars)-1:
        z = new Node
        z.left = Q.popmin()
        z.right = Q.popmin()
        z.freq = z.left.freq + z.right.freq
        Q.push(z)
    return Q.popmin()
```

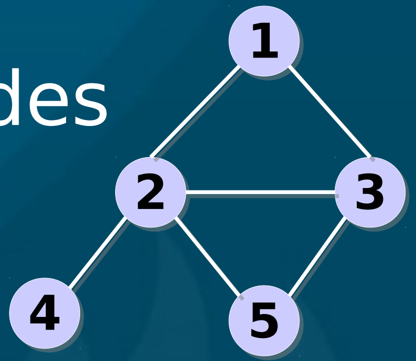
char	freq
a	15
b	5
c	9
d	7
e	18
f	10

Outline for today

- Greedy algorithms
  - Activity selection
  - Fractional knapsack problem
  - Huffman coding
- Intro to graph algorithms
  - Breadth-first search
- Review ch10,12,18,15

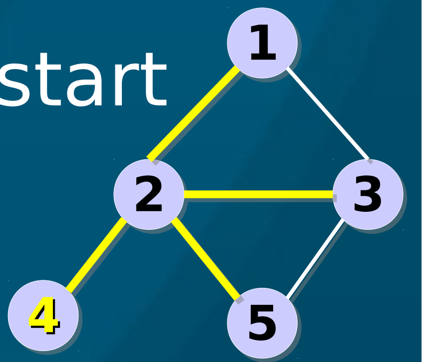
Intro to graph algorithms

- Representing graphs: G = (V, E)
  - V: vertices/nodes (e.g., via array or linked-list)
  - E: edges connecting vertices (directed or un)
- Representing edges:
  - Edge list: array/list of (u,v) pairs of nodes
  - Adjacency list: indexed by start node
    - What about undirected graphs?
    - How to find (out)-degree of every vertex?
  - Adjacency matrix: A[i,j]=1 if (i,j) is an edge
    - What about undirected graphs?
    - Weighted graph: A[i,j] not limited to 0/1



Graph traversal: breadth-first

- Goal: touch all nodes in the graph exactly once
  - Overlays a breadth-first tree rooted at start
    - Path in the tree = shortest path in graph
- Graph ≠ tree: could have loops
  - ⇒ Need to track which nodes we've seen
- Assign colour: white = unvisited,
  grey = on border (some unvisited neighbours),
  black = no unvisited neighbours
- Use FIFO queue to manage grey nodes



Breadth-first search algorithm

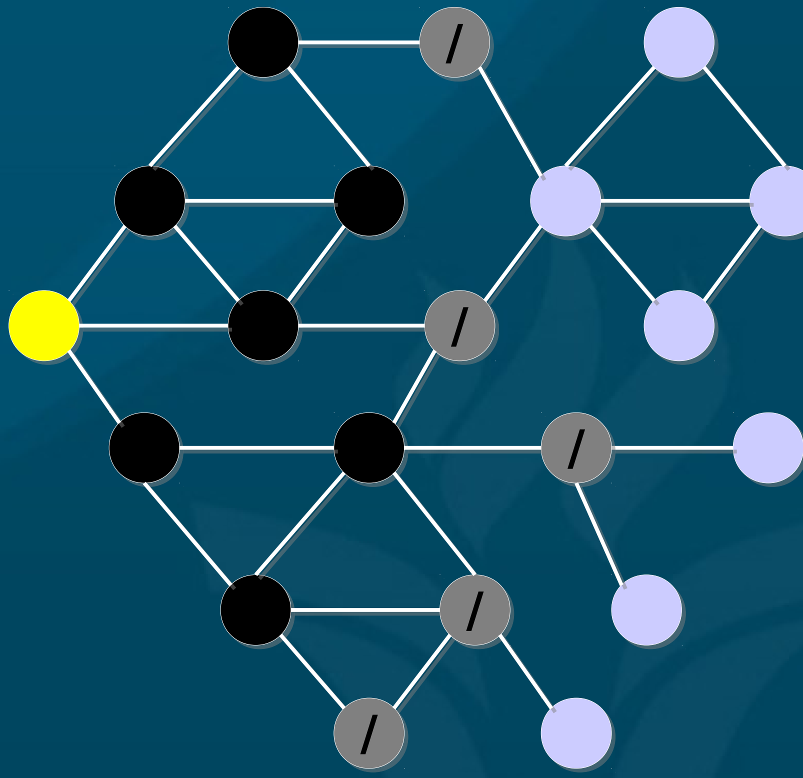
- Input: vertex list, adjacency list (linked lists), start
- Output: modify vertex list to add parent pointers
  - def BFS(V, E, start):

V

- initialise all vertices to be white, with NULL parent
- initialise start to be grey
- initialise FIFO: Q.push(start)

E

- while Q.notempty():
  - u = Q.pop()
  - for each v in E.adj[u]:
    - if v.colour == white:
      - v.colour = grey
      - v.parent = u
      - Q.push(v)
  - u.colour = black



- Complexity: O(V + E)

Outline for today

- Greedy algorithms
  - Activity selection
  - Fractional knapsack problem
  - Huffman coding
- Intro to graph algorithms
  - Breadth-first search
- Review ch10,12,18,15

Review for midterm 3

- ch10: Linked-lists (dbl, circ), stacks/queues
  - Implementation and complexity
- ch12: Trees (terms, expression trees)
  - BSTs (traversal, search, insert, del)
- ch18: B-trees (motivation, design, variants B*, B+)
  - Operations: search, insert, del
  - Complexity analysis: CPU and disk
- ch15: Dynamic programming
  - Optimal substructure ⇒ bottom-up solution
  - Rod-cutting, Fib, matrix-chain, optimal wt BST