AAD

Master Method

- Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function. Define the recurrence $T(n)$ on n by

$$T(n) = aT(n/b) + f(n)$$

- Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

- The driving function $f(n)$ must be asymptotically larger than the watershed function $n^{\log_b a}$ by at least a factor of $\theta(n^{\varepsilon})$ for some constant $\varepsilon > 0$.

BFS

- Given a graph $G = (V, E)$ and a distinguished source vertex s, BFS systematically explores the edges of G to “discover” every vertex that is reachable from s.
- It computes the distance from s to each reachable vertex.
- It produces a “breadth-first tree” with root s that contains all reachable vertices.
- The algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.
- To keep track of progress, BFS colors each vertex white, gray, or black.

- Initially **WHITE**
- **GRAY** when discovered, but not yet had their adjacency lists fully examined.
- **BLACK** when all the adjacent elements examined.
- Whenever the search discovers a white vertex v in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree.

- Attaches three additional attributes to each vertex v in the graph:
  - $v.color$: WHITE, GRAY, or BLACK.
  - $v.d$: the distance from s to v , as computed by the algorithm.
  - $v.\pi$ is v 's predecessor in the breadth-first tree.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Analysis

- We use aggregate analysis.
- **Initialization**: The overhead for initialization is O(V)
- **Enqueuing and dequeuing** : Each vertex is enqueued at most once, and hence dequeued at most once. Enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$
- **Adjacency list scanning:** Listof each vertex is scanned only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\theta(E)$, the total time spent in scanning adjacency lists is $O(E)$.
- Thus, total running time of the BFS procedure is $\boldsymbol{O(V+E)}$

DFS

Algorithm

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1                  // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each vertex v in G.Adj[u]    // explore each edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  time = time + 1
9  u.f = time
10 u.color = BLACK                  // blacken u; it is finished
```

What is the running time of DFS?

- We use aggregate analysis.
- The loops on lines 1–3 and lines 5–7 of DFS take time $\theta(V)$.
- The procedure DFS_VISIT is called exactly once for each vertex $v \in V$ i.e. only if u is WHITE.
- During an execution of $DFS\_VISIT(G, v)$, the loop on lines 4–7 executes $|Adj[v]|$ times.
- $\sum_{v \in V} |Adj[v]| = \theta(E)$, the total cost of executing lines 4–7 of DFS_VISIT is $\theta(E)$
- So, The running time of DFS is $\theta(V + E)$

Properties of DFS

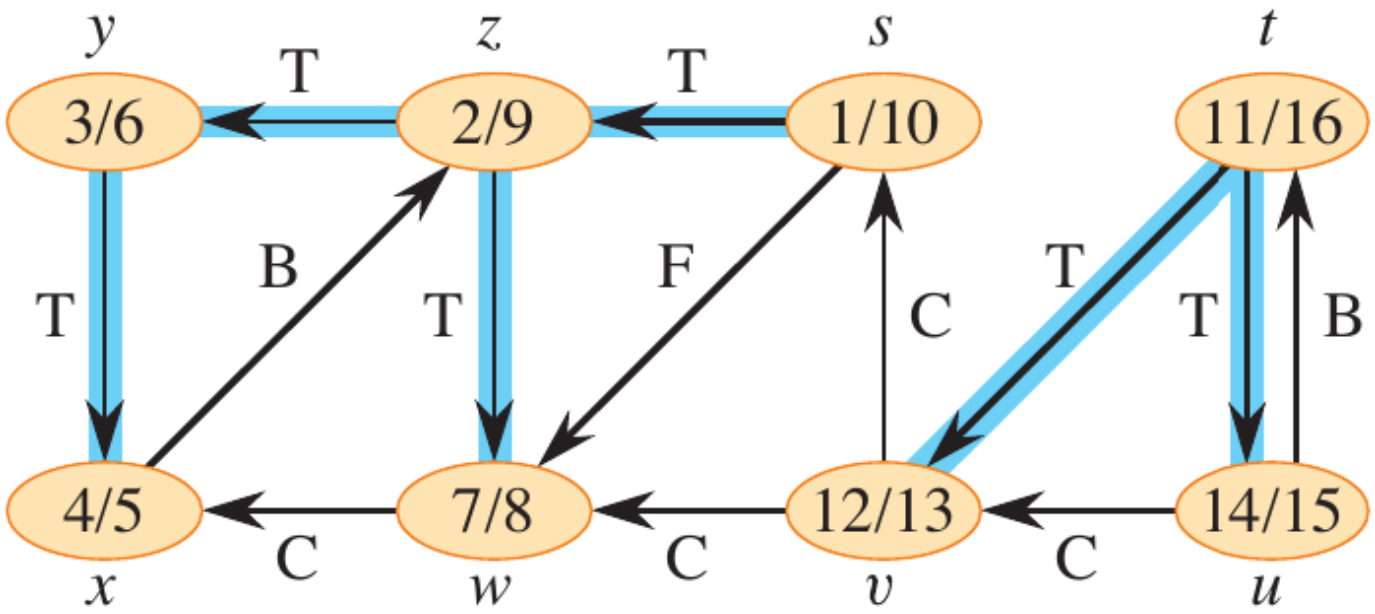
1. The predecessor sub-graph (G_π)form a forest of trees.
2. Discovery and finishing times have parenthesis structure.
   - If we represent the discovery of vertex u with a left parenthesis “(u” and represent its finishing by a right parenthesis “u)”, then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested.
3. The search can be used to classify the edges of the input graph $G = (V, E)$ (tree edges, forward edges, back edges and cross edges)

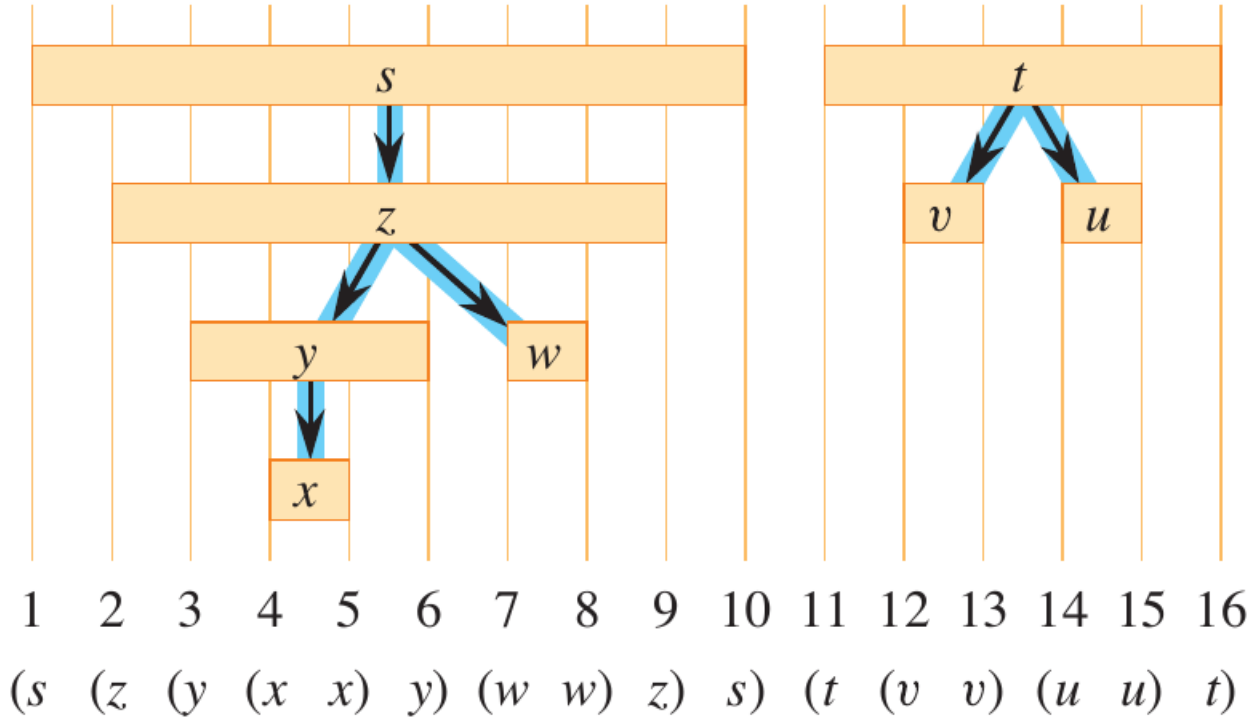
Parenthesis theorem

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

Example




s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

Classification of edges

- We can define four edge types

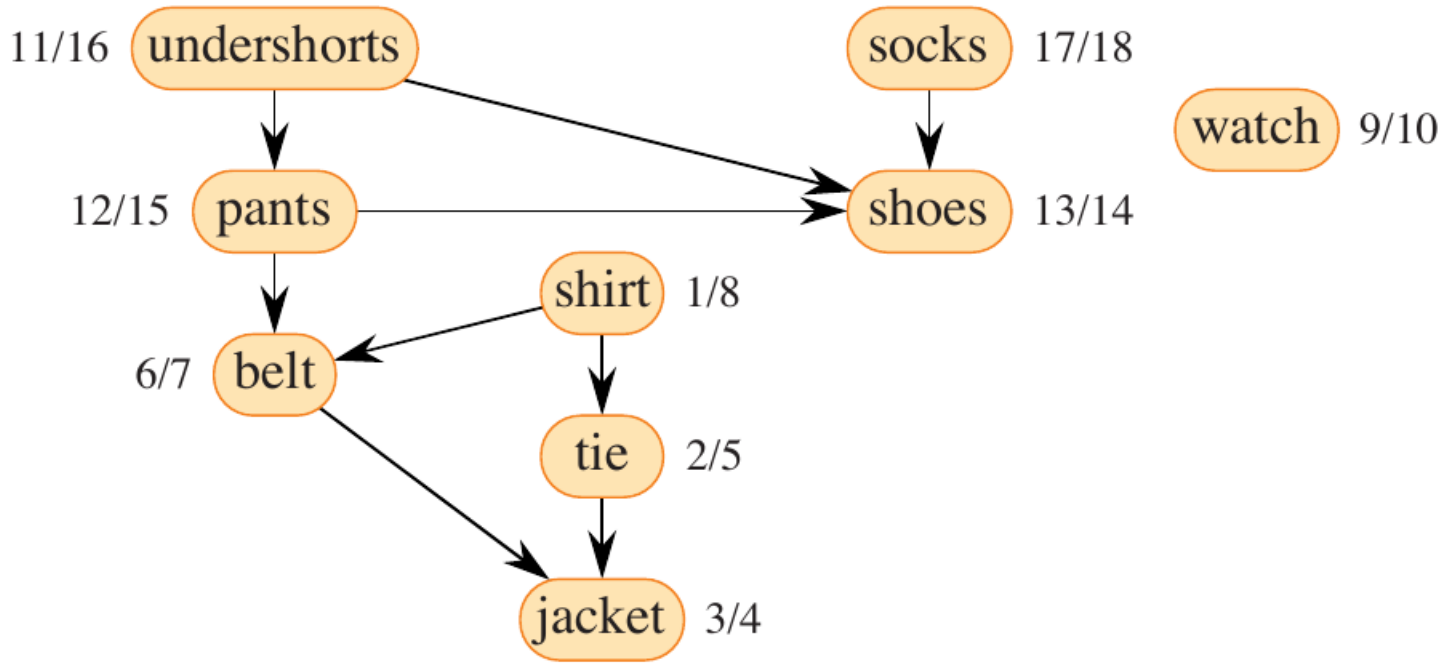
1. **Tree edges**: Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. **Back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. (Self loops are back edges)
3. **Forward edges** are those nontree edges (u, v) connecting a vertex u to a descendant v.
4. **Cross edges**: They can go between vertices in the same depth-first tree, (as long as one vertex is not an ancestor of the other) or they can go between vertices in different depth-first trees.

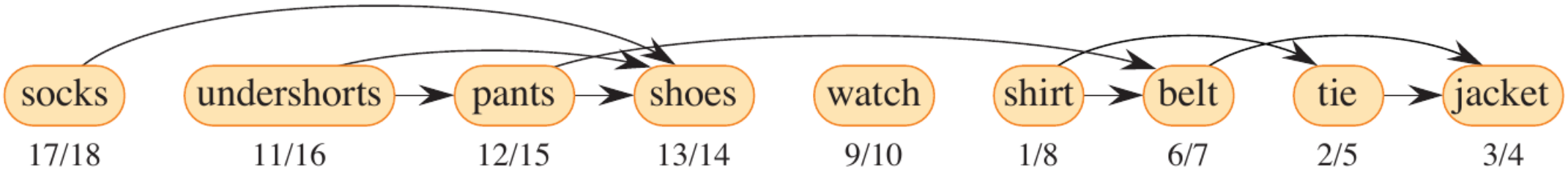
A key idea

- The key idea is that when we first explore an edge (u, v), the color of vertex v tells us something about the edge:
  1. WHITE indicates a tree edge
  2. GRAY indicates a back edge, and
  3. BLACK indicates a forward or cross edge.
     - edge (u, v) is a forward edge if $u.d < v.d$ and a cross edge if $u.d > v.d$.

Topological sort

- DFS can be used to perform a topological sort of a directed acyclic graph, or a “dag”.
- A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.
- An ordering of its vertices along a horizontal line so that all directed edges go from left to right.
- Application: to indicate precedences among events.


11/16
undershorts
socks
17/18
watch
9/10
12/15
pants
shoes
13/14
shirt
1/8
6/7
belt
tie
2/5
jacket
3/4


socks
undershorts
pants
shoes
watch
shirt
belt
tie
jacket
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finish times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Running Time

- DFS takes $\theta(V + E)$ time.
- $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.
- Thus, the TOPOLOGICAL-SORT procedure runs in $\theta(V + E)$ time

- **Lemma:** A directed graph G is acyclic if and only if a DFS of G yields no back edges.

Divide and Conquer

General Method

- Given a function to compute on n inputs
- The divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, $1 < k < n$, yielding k subproblems.
- These subproblems are solved, and then combine sub-solutions into a solution of the whole.
- If the sub-problems are still relatively large, then the $Dand\ C$ strategy reapplied until subproblems are small enough to be solved without splitting are produced.

Control abstraction for divide-and-conquer

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
9       }
10  }
```

$Small(P)$ is a Boolean-valued function that determines whether the input size is small enough to compute the answer without splitting.

- Computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

  - g(n) is the time to compute the answer directly for small inputs.
  - f(n) is the time for dividing P and combining the solutions to subproblem.

Merge Sort

Merge Sort

- Given a sequence of n elements (also called keys), $a[1], \ldots, a[n]$
- Split into two sets, $a[1], \ldots, a[\frac{n}{2}]\ and\ a[\frac{n}{2}+1] \ldots a[n]$
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

Strassen’s Matrix Multiplication

Strassen's Matrix Multiplication

- Let A and B be two $n\ x\ n$ matrices.
- The product matrix $C\ =\ AB$

$$C(i,j)\ =\ \sum_{1<k<n} A(i,k)B(k,j)$$

- The time complexity is $\theta(n^3)$.

The divide-and-conquer strategy

- A and B are each partitioned into four square submatrices, each having dimensions for $\frac{n}{2} X \frac{n}{2}$.
- Then the of product AB can computed by:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

- If $n = 2$, it will be multiplication of elements of A and B
- If n>2, it will be matrix multiplication and additions.
- This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitably small ($n = 2$) so that the product is computed directly.
- But, we need to perform eight multiplications matrices and four additions.

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

$$T(n) = 0(n^3)$$

No improvement at all !

- Volker Strassen has discovered a way to compute the C_{ij}'s using only 7 multiplications and 18 additions/subtractions.
- His method involves:
  - First computing the seven $\frac{n}{2} X \frac{n}{2}$ matrices P, Q, R, S, T, U, and V.
  - Then the C_{ij} 's are computed using the formulas as follows:

$$
\begin{aligned}
P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
Q &= (A_{21} + A_{22})B_{11} \\
R &= A_{11}(B_{12} - B_{22}) \\
S &= A_{22}(B_{21} - B_{11}) \\
T &= (A_{11} + A_{12})B_{22} \\
U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
V &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= P + S - T + V \\
C_{12} &= R + T \\
C_{21} &= Q + S \\
C_{22} &= P + R - Q + U
\end{aligned}
$$

- The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

  - By solving this we get

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

The Greedy Method

The General Method

- We have a problem having n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies those constraints is called a **feasible solution**.
- We need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an **optimal solution**.

- The greedy method devise an algorithm that works in stages, considering one input at a time.
- At each stage, a decision is made regarding whether a particular input is in an optimal solution by some selection procedure.
- If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then discarded.
- Otherwise, it is added

The control abstraction

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

Fractional Knapsack Problem

- We are given n objects and a knapsack (bag).
- Object i has a weight w_i and the knapsack has a capacity m.
- If a fraction x_i, $0 < x_i < 1$, of object i is placed into the knapsack, then a profit of $p_i X_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- We require the total weight of all chosen objects to be at most m.

- Formally, the problem can be stated as:

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

- A feasible solution(or filling) is any set $(x_i, \ldots, x_n)$ satisfying the above constraints.

Example

- Consider the following instance of the knapsack problem:

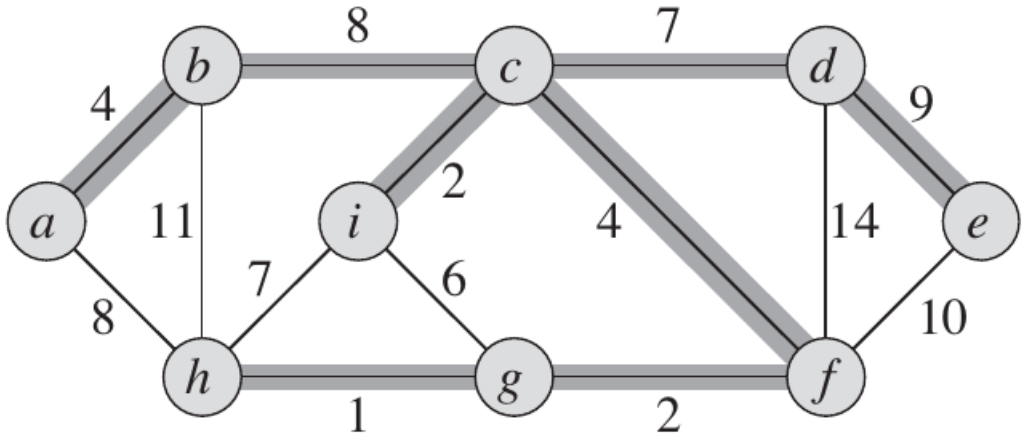
$n = 3,\ m = 20,\ (p_1, p_2, p_3) = (25,24,15),$ and $(w_1, w_2, w_3) = (18,15,10)$

Minimum Spanning Trees

- In Electronic circuit designs- the wiring problem.
- We have a connected, undirected graph $G = (V, E)$, where V is the set of vertices, E is the set of edges, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost to connect u and v.
- Find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$ is minimized.

- T is known as a **spanning tree** since it “spans” the graph G.
- The problem of determining T is called the **minimum-spanning-tree (MST) problem.**
- Example:



The generic method for MST

- We have a connected, undirected graph $G = (V, E)$, with a weight Function $w : E \rightarrow \mathbb{R}$ and we wish to find a minimum spanning tree for G.
- The generic method manages a set of edges A, maintaining the following loop invariant:
  - Prior to each iteration, A is a subset of some minimum spanning tree. $i.e,\ A \cup \{(u, v)\}$ is also a subset of MST.
- An edge a safe edge for A, since we can add it safely to A while maintaining the invariant.

GENERIC-MST(G, w)

```
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

Kruskal's algorithm

- Kruskal's algorithm finds a safe edge to add to the growing forest by finding an edge (u, v) of least weight that connect any two trees in the forest.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

- The running time of Kruskal's algorithm as $O(E \lg V)$.

Single-Source Shortest Paths

- Given a weighted, directed graph $G = (V, E)$, with weight function $w: E \rightarrow R$ mapping edges to real-valued weights.
- The weight $w(p)$ of path $p = < v_0, \ldots, v_k >$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

- We define the shortest-path weight $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v , \\ \infty & \text{otherwise} . \end{cases}$$

Variants:

- **Single-source shortest-paths problem**: find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.
- **Single-destination shortest-paths problem**: Find a shortest path to a given destination vertex t from each vertex v.
- **Single-pair shortest-path problem**: Find a shortest path from u to v for given vertices u and v.
- **All-pairs shortest-paths problem**: Find a shortest path from u to v for every pair of vertices u and v.

- ***Lemma***: Sub-paths of shortest paths are shortest paths:

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof If we decompose path p into $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ■

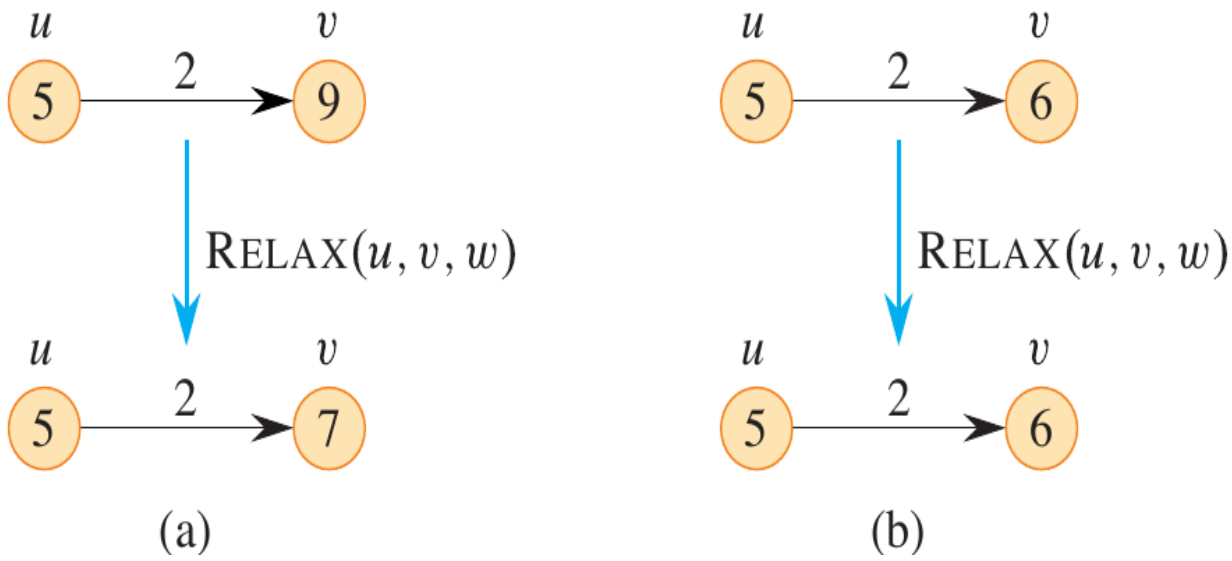
- **shortest-path estimate:** For each vertex $v \in V$, we maintain an attribute $v.d$ called shortest-path estimate, which is an upper bound on the weight of a shortest path from source s to v.
- We initialize the shortest-path estimates and predecessors by the following $\theta(V)$time procedure:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Relaxation:

- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$.

```
RELAX(u, v, w)
1   if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```



Dijkstra's algorithm

- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case where all edge weights are nonnegative. i.e, $w(u, v) \geq 0$.
- Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from s have already been determined.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum d value, adds u to S, and relaxes all edges leaving u.
- We use a min-priority queue Q of vertices, keyed by their d values. (replaces the FIFO queue of BFS)

```
DIJKSTRA(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  S = ∅
 3  Q = ∅
 4  for each vertex u ∈ G.V
 5      INSERT(Q, u)
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      S = S ∪ {u}
 9      for each vertex v in G.Adj[u]
10          RELAX(u, v, w)
11          if the call of RELAX decreased v.d
12              DECREASE-KEY(Q, v, v.d)
```

- $INSERT(Q, u)$ inserts u into the min-priority queue, Q
- $DECREASE\_KEY(Q, v, v.d)$ decreases the v's key to a new lesser value $v.d$.

Analysis

- The algorithm calls both $INSERT$ and $EXTRACT\_MIN$ once per vertex.
- Because each vertex u is added to set S exactly once, each edge in the adjacency list $Adj[u]$ is examined exactly once. Thus, $DECREASE\_KEY$ at most $|E|$ times overall.
- **The running time of Dijkstra's algorithm depends on how we implement the min-priority queue.**

Case1: Maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$.

- Store $v.d$ in the v^{th} entry of an array. Each $INSER$T and $DECREASE\_KEY$ operation takes $O(1)$ time, and each $EXTRACT - MIN$ operation takes $O(V)$ time (since we have to search through the entire array).
- Total time of $O(V^2 + E) = \boldsymbol{O(V^2)}$.

Case2: By implementing the min-priority queue with a binary min-heap.

- Each $EXTRACT\_MIN$ operation then takes time $O(\lg V)$. There are $|V|$ such operations.
- The time to build the binary min-heap is $O(V)$.
- Each $DECREASE\_KEY$ operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations.
- So, the total running time is therefore $\boldsymbol{O((V + E)\, lg\, V)}$.

Case3: By implementing the min-priority queue with a Fibonacci heap

- We can achieve a running time of $\boldsymbol{O(V\, lg\, V + E)}$.

Dynamic Programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

Divide-and-conquer	Dynamic programming
Partition the problem into disjoint subproblems, solve recursively, and then combine their solutions. (Sub problems are independent)	Applies when the subproblems overlap - that is, when subproblems share sub subproblems. (Sub problems are not independent)
Does more work than necessary, repeatedly solving the common sub subproblems.	Solves each sub subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing
Top-down approach	Bottom-up approach

Principle of optimality

- Dynamic programming typically applies to optimization problems.
- They can have many possible solutions. Each solution has a value, and you want to find a solution with the optimal (minimum or maximum) value.
- **The principle of optimality** states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- That is, if a problem can be solved by breaking it down into smaller subproblems, then the optimal solution to the overall problem can be constructed from the optimal solutions to its subproblems.
- Greedy method vs dynamic programming:
  - In greedy method only one decision sequence is ever generated.
  - In dynamic programming, many decision sequences may be generated.

- When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Matrix-chain multiplication

- We can multiply two matrices A and B only if they are compatible: the number of columns of A must equal the number of rows of B.
- If A is a p x q matrix and B is a q x r matrix, the resulting matrix C is a p x r matrix.
- The number of scalar multiplications is pqr.
- **Matrix-chain multiplication problem**:

  Given a chain $< A_1, A_2, \ldots., A_n >$ of n matrices, where $i = 1,2, \ldots n$ matrix A_i has dimension p_{i-1}x p_i , fully parenthesize the product $A_1 A_2 \ldots. A_n$ in a way that minimizes the number of scalar multiplications. The input is the sequence of dimensions $< p_0, p_1, p_2, \ldots., p_n >$

Dynamic Programming Solution

Step 1: The structure of an optimal parenthesization:

- To parenthesize the product, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$.
- The cost of parenthesizing is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1...j}$, plus the cost of multiplying them together.
- Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \ldots A_k$ within this optimal parenthesization of $A_i A_{i+1} \ldots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \ldots A_k$.
- Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems, finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions.

Step 2: A recursive solution

- We define the cost of an optimal solution recursively.
- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
- $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$ (each matrix A_i is $p_{i-1} X\ p_i$)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j : i \le k < j\} & \text{if } i < j\ . \end{cases}$$

  - $m[i, j]$ equals the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together.
  - We define $s[i,j]$ to be a value of k at which we split the product $A_i A_{i+1} \ldots A_j$ in an optimal parenthesization.

Step 3: Computing the optimal costs

- We compute the optimal cost by using a tabular, bottom-up approach.
- The procedure uses two auxiliary tables:
    1) $m[1 \ldots n, 1 \ldots n]$ for storing the $m[i, j]$ costs.
    2) $s[1 \ldots n-1, 2 \ldots n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$.

```
MATRIX-CHAIN-ORDER(p, n)
 1  let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
 2  for i = 1 to n                          // chain length 1
 3      m[i, i] = 0
 4  for l = 2 to n                          // l is the chain length
 5      for i = 1 to n - l + 1              // chain begins at A_i
 6          j = i + l - 1                   // chain ends at A_j
 7          m[i, j] = ∞
 8          for k = i to j - 1              // try A_{i:k} A_{k+1:j}
 9              q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
10              if q < m[i, j]
11                  m[i, j] = q             // remember this cost
12                  s[i, j] = k             // remember this index
13  return m and s
```

Step 4: Constructing an optimal solution

- Although $MATRIX\_CHAIN\_ORDER$ determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

Sample Qn:

- Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is <5, 10, 3, 12, 5, 50, 6>

Backtracking

General Method

- The desired solution is expressible as an n-tuple $(x_i, \ldots, x_n)$, where the x_i are chosen from some finite set S_i.
- It is used if there are multiple solutions and we want to find all those solutions.
- Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \ldots, x_i)$ (**called bounding functions**) to test whether the vector being formed has any chance of success.
- If it is realized that the partial vector $(x_1, \ldots, x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \ldots m_n$ possible test vectors can be ignored entirely.

```
1   Algorithm Backtrack(k)
2   // This schema describes the backtracking process using
3   // recursion. On entering, the first k − 1 values
4   // x[1], x[2], ..., x[k − 1] of the solution vector
5   // x[1 : n] have been assigned. x[ ] and n are global.
6   {
7       for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
8       {
9           if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
10          {
11              if (x[1], x[2], ..., x[k] is a path to an answer node)
12                  then write (x[1 : k]);
13              if (k < n) then Backtrack(k + 1);
14          }
15      }
16  }
```

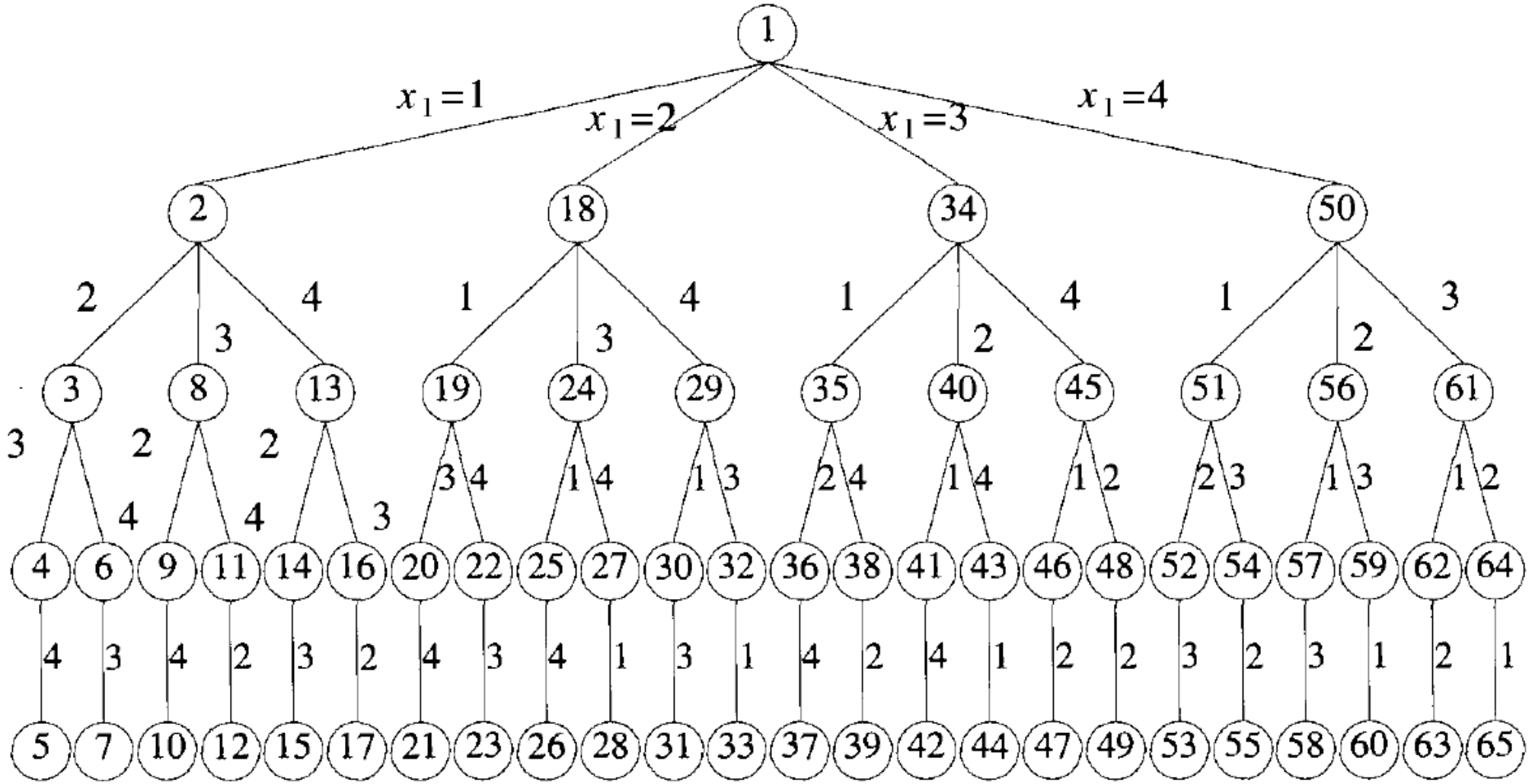
Bounding function B_{i+1}

- If $B_{i+1}(\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_{\mathrm{i}+1})$ is false for a path $(\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_{\mathrm{i}+1})$ from the root node to a problem state, then the path cannot be extended to reach an answer node.
- All the possible elements for the k^{th}position of the tuple that satisfy B_k are generated, one by one, and adjoined to the current vector $(x_1, \ldots, x_{k-1})$. Each time x_k is attached, a check is made to determine whether a solution has been found.

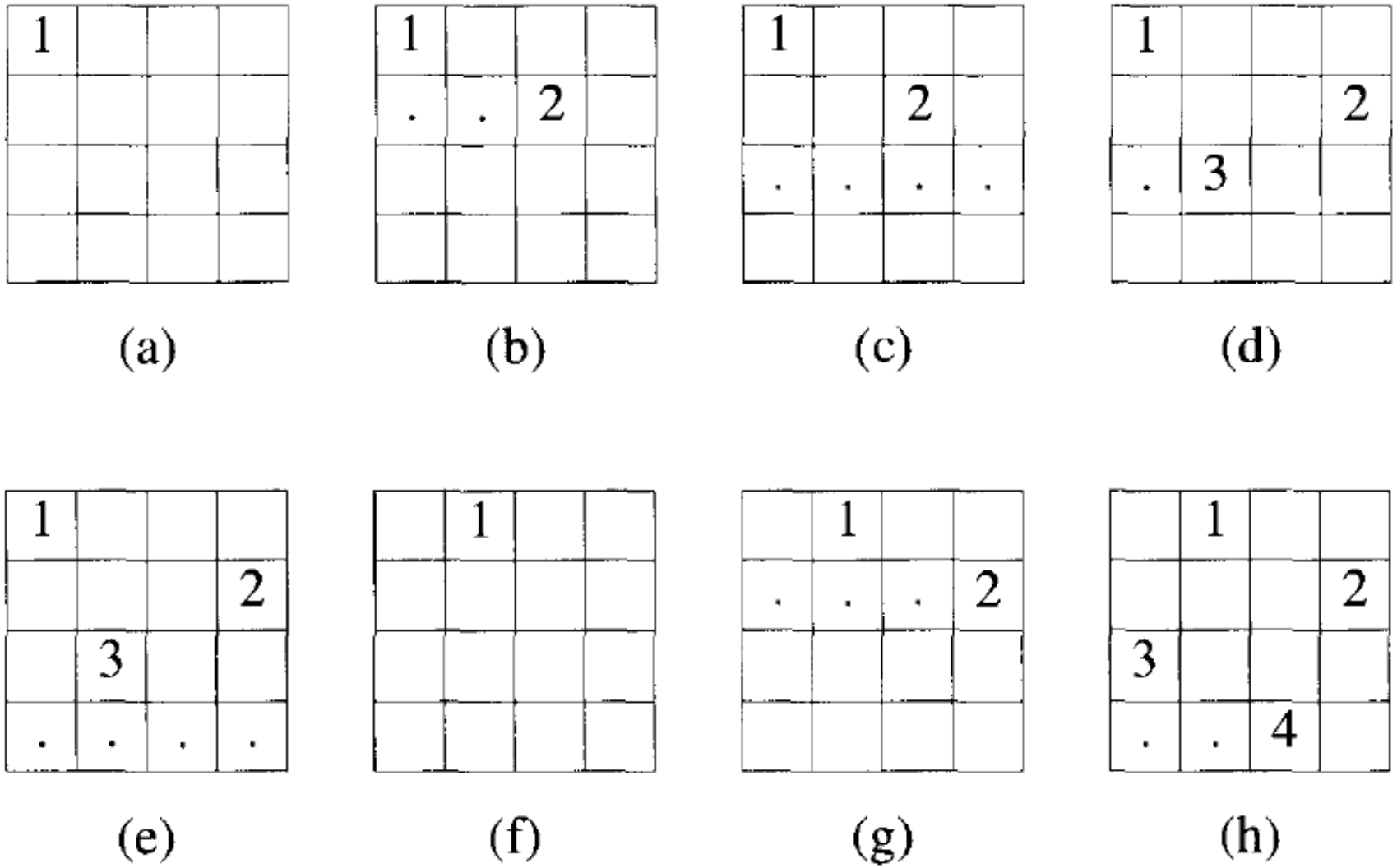
N-queens problem

- n queens are to be placed on an $n\ X\ n$ chessboard so that no two attack; that is, no two queens are on the same row, column, or diagonal.
- Since each queen must be on a different row, we can assume queen i is to be placed on row i.
- All solutions to the n-queens problem can there fore be represented as n-tuples $(x_1, \ldots, x_n)$, where x_i is the column on which queen i is placed.

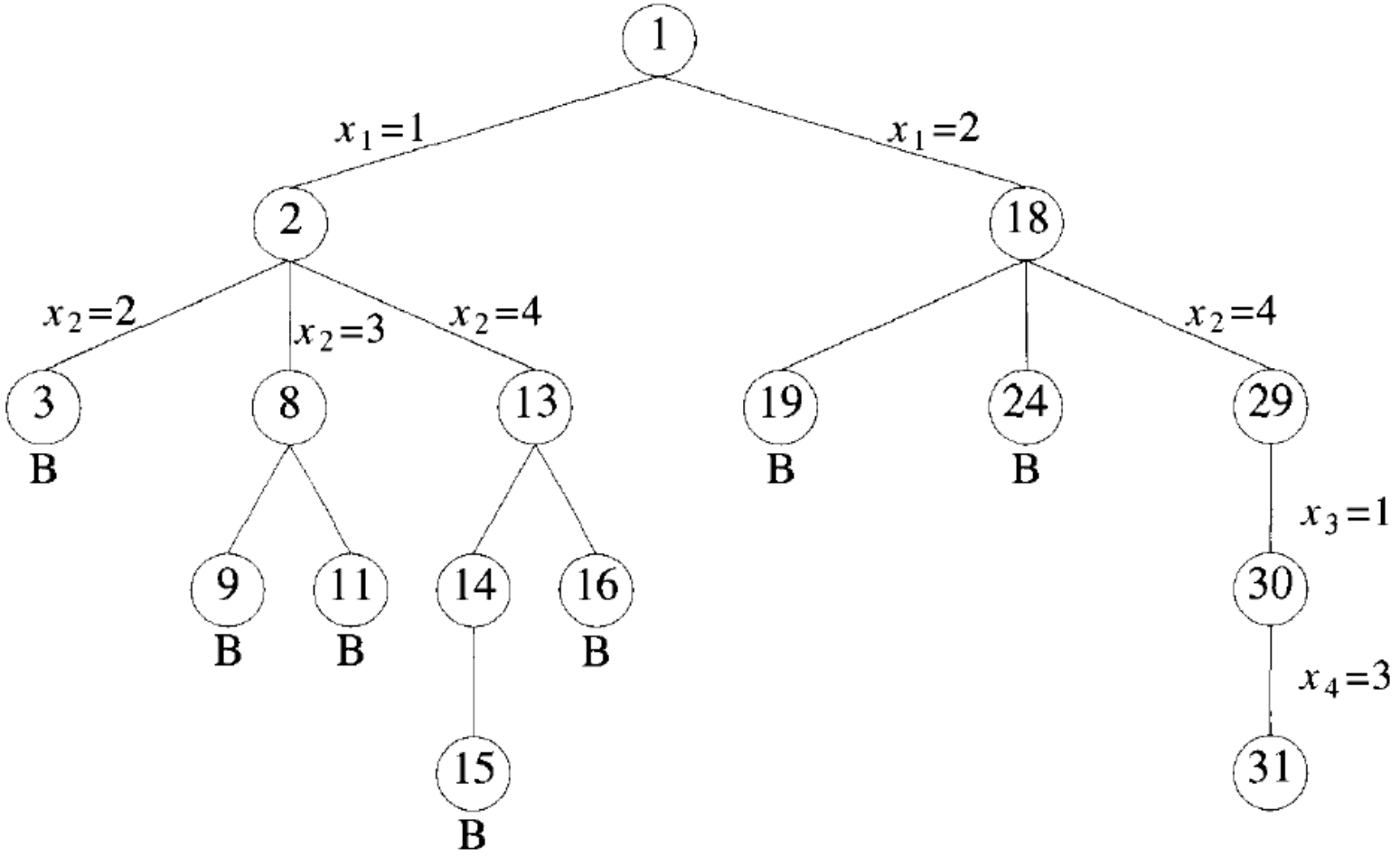
Tree organization of the 4-queens solution space



Example of a backtrack solution to the 4-queens problem



Portion of the state space tree generated during backtracking



```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k – 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k – 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] – i) = Abs(j – k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

- The number of nodes that will be generated by N-queen

$$1 + \sum_{j=0}^{n-1} \left[\prod_{i=0}^{j} (n - i) \right]$$

- For 4-queens problem, the count will be

  1+4+4.3+4.3.2+4.3.2.1 = 65

Branch and Bound

Some terminologies used:

- A node which has been generated and all of whose children have not yet been generated is called a **live node**.
- The live node whose children are currently being generated is called the **E-node.** (node being expanded).
- A **dead node** is a generated node which is not to be expanded further or all of whose children have been generated.

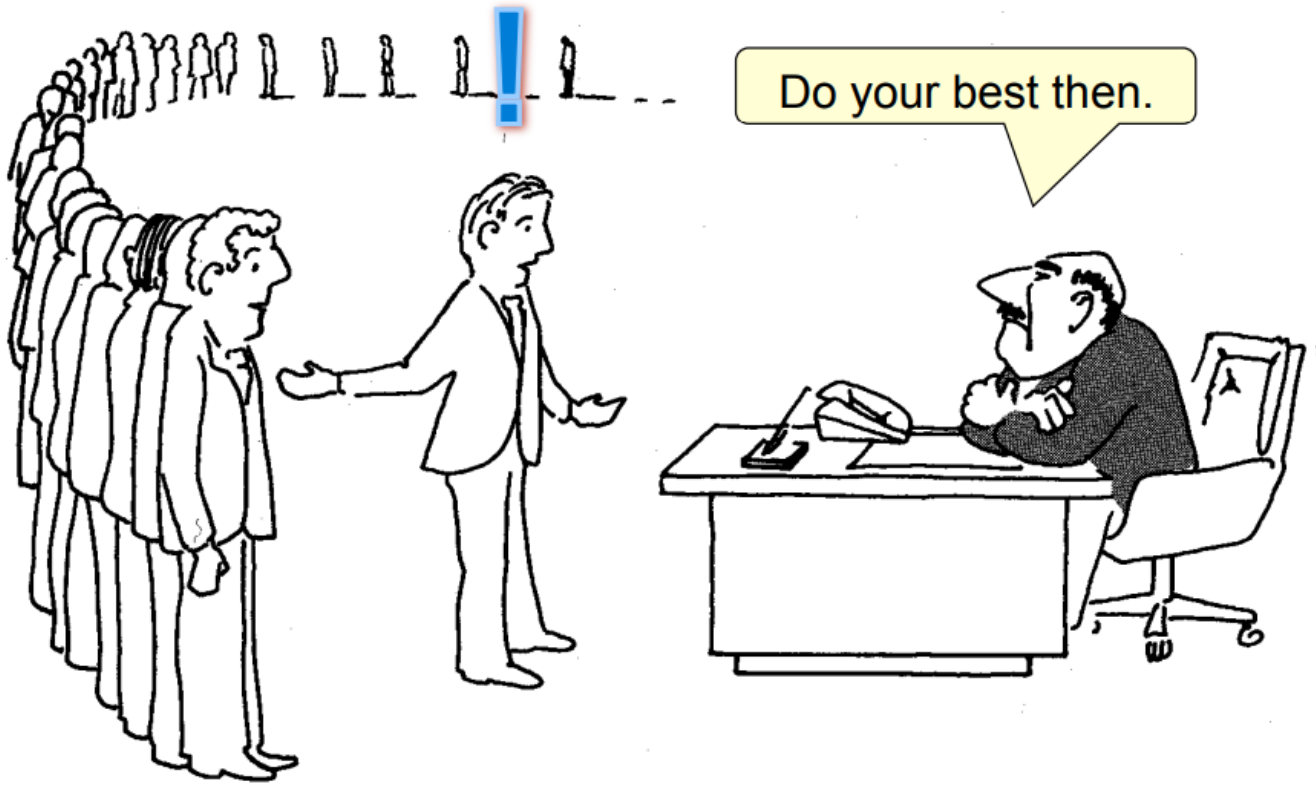
- Branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node. (BFS)
- Solve optimization problems, especially minimization problems.
- Two strategies use:
    1. BFS: FIFO, live nodes are placed in a queue.
    2. D-search: LIFO, live nodes are placed in a stack.
- As in the case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

LC Search

- The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(.)$ for live nodes.
- The next E-node is selected on the basis of this ranking function.
- The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node.
- Always choose for its next E-node a live node with least $\hat{c}(.)$. Hence called an LC-search(Least Cost search)

Approximation Algorithms

NP-completeness



"I can't find an efficient algorithm, but neither can all these famous people."

- An algorithm that returns near-optimal solutions are called approximation algorithm or heuristic algorithm.
- They do not guarantee to give a best solution but will give one that is close to optimal.
- It is good enough especially when the time required to find an optimal solution is considered.
  - $FS(I)$: represents the set of feasible solution for input I.
  - Value function, $val(I, x)$: returns the value of the optimization parameter achieved by the feasible solution x.
  - Optimum value, $opt(I)$: Best value achieved by any feasible solution. It is an x in $FS(I)$ such that $val(I, x) = opt(I)$

- An approximation algorithm for a problem is a polynomial time algorithm that, when given input I, outputs an element of $FS(I)$.

Performance ratios for approximation algorithms:

- The ratio between the value of the output and the value of an optimal solution.
- Let A be an approximation algorithm and $A(I)$, the feasible solution A chooses for input I, then

$$r_A(I) = \frac{val(I,A(I))}{opt(I)}, \text{ for minimization problems, and}$$

$$r_A(I) = \frac{opt(I)}{val(I,A(I))}, \text{ for maximization problems.}$$

In both cases, $r_A(I) \geq 1$

- If an algorithm achieves an approximation ratio of k, we call it k-approximation algorithm.

Bin Packing

- How to pack objects of various size with minimum wasted space.
- Let $S = \{s_1, s_2, \dots, s_n\}$ where $0 < s_i \leq 1$ and $1 \leq i \leq n$. The problem is to pack $s_1, s_2, \dots, s_n$ into as few bins as possible, where each bis has a capacity 1.
- Fortunately, there are a number of interesting heuristics we can apply to hopefully come close to optimality.
- The minimum number of bins required would be

$$\boldsymbol{Ceil(Total\ Weight/Bin\ Capacity)}$$

- **Applications**
  - ✓ Loading of containers like trucks.
  - ✓ Placing data on multiple disks.
  - ✓ Job scheduling.
  - ✓ Packing advertisements in fixed length radio/TV station breaks.
  - ✓ Storing a large collection of music onto tapes/CD’s, etc.

First Fit Decreasing strategy

- A simple heuristic greedy strategy.
- It is a modification of first-fit strategy that sorts the objects first so tat they are considered in the order of non-decreasing size.
- Steps:
  - Read the inputs of items
  - Sort the array of items in decreasing order by their sizes
  - Apply First-Fit algorithm
- Worst case time complexity: $O(n^2)$
- Average case time complexity: $O(n^2)$
- Best case time complexity: $O(n\ logn)$

Algorithm

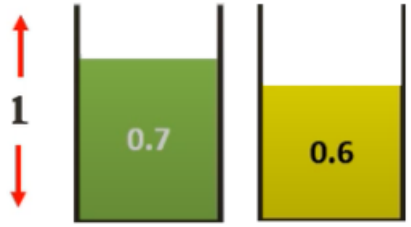
Input: A sequence $S = (s_1, \ldots, s_n)$ of type **float**, where $0 < s_i \leq 1$ for $1 \leq i \leq n$. S represents the sizes of objects $\{1, \ldots, n\}$ to be placed in bins of capacity 1.0 each.

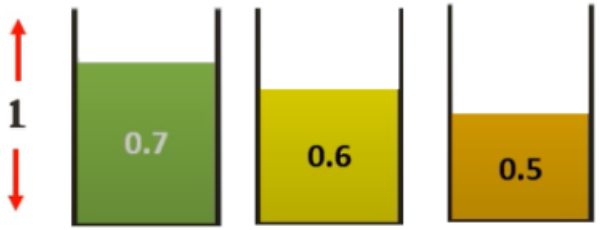
Output: An array bin where for $1 \leq i \leq n$, bin[i] is the number of the bin into which object i is placed. For simplicity, objects are indexed after being sorted in the algorithm. The array is passed in and the algorithm fills it.

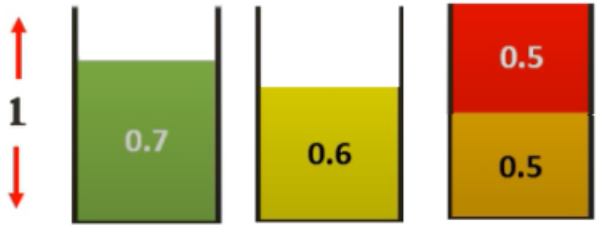
```
binpackFFD(S, n, bin)
    float[] used = new float[n+1];
    // used[j] is the amount of space in bin j already used up.
    int i, j;
    Initialize all used entries to 0.0.
    Sort S into descending (nonincreasing) order, giving the sequence
    s1 ≥ s2 ≥ ··· ≥ sn.

    for (i = 1; i ≤ n; i ++)
        // Look for a bin in which s[i] fits.
        for (j = 1; j ≤ n; j ++)
            if (used[j] + si ≤ 1.0)
                bin[i] = j;
                used[j] += si;
                break;  // exit for (j)
        // Continue for (i)
```

Example

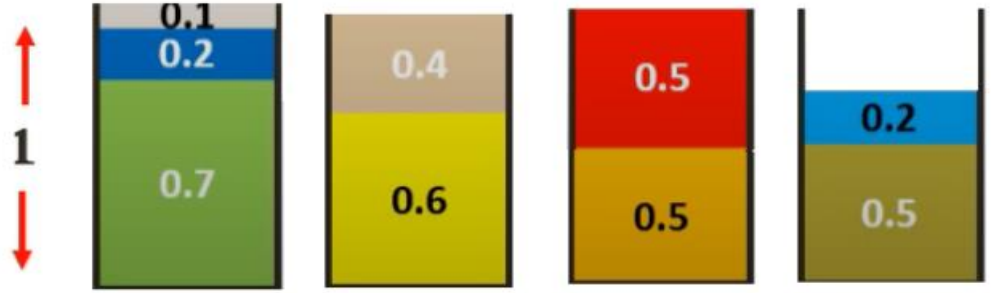
- Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.
- The minimum number of bins required would be
  Ceil(3.7/1) = 4 bins.
- Sorting them we get {0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1}







Doing the same for all items, we get.



- Thus only 4 bins are required which is the same as the optimal solution.

Other heuristics

Next Fit:

- The simplest approximate approach to the bin packing problem is the Next-Fit (NF) algorithm which is explained later in this article. The first item is assigned to bin 1. Items 2,... ,n are then considered by increasing indices : each item is assigned to the current bin, if it fits; otherwise, it is assigned to a new bin, which becomes the current one.
- Approximation ratio arbitrarily close to 2. So, it is 2-approximate.

First Fit:

- A better algorithm, First-Fit (FF), considers the items according to increasing indices and assigns each item to the lowest indexed initialized bin into which it fits; only when the current item cannot fit into any initialized bin, is a new bin Introduced.

Best Fit:

- Best-Fit (BF), is obtained from FF by assigning the current item to the feasible bin (if any) having the smallest residual capacity. That is, put the next item in the bin so that the smallest empty space is left.

You can refer https://iq.opengenus.org/bin-packing-problem/ for more detailed illustrations.

Randomization

- A technique for designing approximation algorithms.
- Randomized algorithms are algorithms that makes random decision during their execution.
- Specifically, they are allowed to use variables, such that their value is taken from some random distribution.
- In addition to input, algorithm takes a source of random numbers and makes random choices during execution.
- Unlike deterministic algorithms, which produce the same output for the same input every time they are run, randomized algorithms may produce different results on different runs due to the use of randomness.
- Randomized algorithms are classified in two categories: Monte Carlo and Las Vegas .
- Monte Carlo and Las Vegas are two different types of randomized algorithms that employ randomness to solve problems, but they do so in distinct ways.

Monte Carlo algorithms

- A Monte Carlo algorithm runs for a fixed number of steps and produces an answer that is correct with probability > 1/3.
- Such an algorithm has a fixed running time, but its correctness is random.
- They rely on randomization to make probabilistic decisions that might lead to incorrect outputs in some cases.
- However, they can be designed to produce accurate results with high probability.

- **When to Use Monte Carlo Algorithms?**
  - Monte Carlo algorithms are appropriate for problems where a small probability of error is acceptable. These algorithms are widely used in simulations, statistical analysis, and optimization.
  - When dealing with problems that are extremely time-consuming to solve deterministically, Monte Carlo algorithms can provide relatively quick results.

Example:

- Primality Testing: Algorithms like the Miller-Rabin primality test use randomness to determine whether a number is likely prime or composite. While they may occasionally produce incorrect results, the probability of error can be made extremely low by adjusting the number of iterations.

Las Vegas Algorithms

- Las Vegas algorithms are randomized algorithms that always provide the correct result, but the running time is subject to randomization.
- In other words, while they guarantee the accuracy of the output, the time it takes to obtain this result may vary.
- They guarantee the correctness of the solution but do not guarantee the time it will take to find it.
- **When to Use Las Vegas Algorithms?**
  - Approximation Problems: Las Vegas algorithms are ideal when dealing with optimization problems where you need an approximation to a solution. They provide solutions that are typically close to the optimal result while still ensuring correctness.
  - NP-Hard Problems: When dealing with NP-hard problems that are computationally intractable in the worst-case, Las Vegas algorithms can provide practical solutions within a reasonable amount of time.

  Example: Quick sort

Randomized version of quicksort

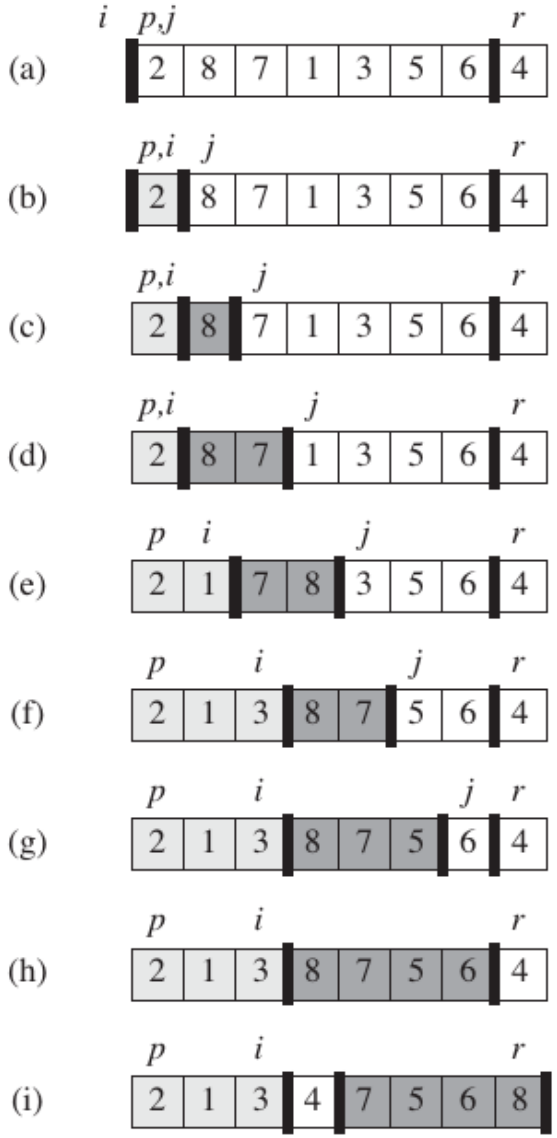
- Sorting array $A[p..r]$
- Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$.
- Compute the index q as part of this partitioning procedure.
- Sort the two subarrays $A[p..q-1]$and $A[q+1..r]$ by recursive calls to quicksort.

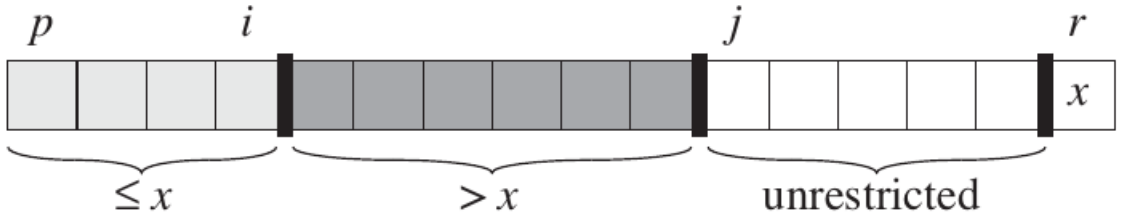
```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

PARTITION always selects an element $x = A[r]$ as a pivot element around which to partition the subarray $A[p..r]$.

Example


(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8


p
i
j
r
x
≤ x
> x
unrestricted

Algorithm

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2       q = RANDOMIZED-PARTITION(A, p, r)
3       RANDOMIZED-QUICKSORT(A, p, q − 1)
4       RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Worst-case analysis

- Using the substitution method, we can show that the running time of quicksort is $O(n^2)$
- Let $T(n)$be the worst-case time for the procedure QUICKSORT on an input of size n. We have the recurrence,

$$T(n) = \max_{0<q<n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

where the parameter q ranges from 0 to $n - 1$ because the procedure PARTITION produces two subproblems with total size $n - 1$.

We guess that $T(n) \leq cn^2$ for some constant c.

Substituting this guess into recurrence, we obtain

$$\begin{aligned} T(n) &\leq \max_{0\leq q\leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0\leq q\leq n-1} (q^2 + (n-q-1)^2) + \Theta(n) \ . \end{aligned}$$

By solving this, we get $T(n) \leq \mathrm{cn}^2$.