Algorithm Design Techniques (II)

Brute Force Algorithms.
Greedy Algorithms.
Backtracking

Brute Force Algorithms

- Distinguished not by their structure or form, but by the way in which the problem to be solved is approached
- They solve a problem in the most simple, direct or obvious way
  - Can end up doing far more work to solve a given problem than a more clever or sophisticated algorithm might do
  - Often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

Example: Counting Change

- Problem: a cashier has at his/her disposal a collection of notes and coins of various denominations and is required to count out a specified sum using the smallest possible number of pieces
- Mathematical formulation:
  - Let there be $\boldsymbol{n}$ pieces of money (notes or coins), $\boldsymbol{P}=\{\boldsymbol{p_1}, \boldsymbol{p_2}, \ldots, \boldsymbol{p_n}\}$
  - let $\boldsymbol{d_i}$ be the denomination of $\boldsymbol{p_i}$
  - To count out a given sum of money $\boldsymbol{A}$ we find the smallest subset of $\boldsymbol{P}$, say $\boldsymbol{S} \subseteq \boldsymbol{P}$, such that $\sum_{p_i \in S} d_i = A$

Example: Counting Change (2)

- Can represent $\boldsymbol{S}$ with $\boldsymbol{n}$ variables $\boldsymbol{X}=\{\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}\}$ such that

$$\begin{cases} x_i = 1 & p_i \in S \\ x_i = 0 & p_i \notin S \end{cases}$$

- Given $\{\boldsymbol{d_1}, \boldsymbol{d_2}, \ldots, \boldsymbol{d_n}\}$ our <u>*objective*</u> is:

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

- Provided that:

$$\sum_{p_i \in S} d_i = A$$

Counting Change: Brute-Force Algorithm

- $\forall\, \boldsymbol{x_i} \in \boldsymbol{X}$ is either a 0 or a 1 $\Rightarrow$ $\mathbf{2}^{\boldsymbol{n}}$ possible values for $\boldsymbol{X}$.
- A brute-force algorithm finds the best solution by enumerating all the possible values of $\boldsymbol{X}$.
  - For each possible value of $\boldsymbol{X}$ we check first if the constraint

$$\sum_{i=1}^{n} d_i x_i = A$$

    is satisfied.
  - A value which satisfies the constraint is called a *feasible solution*
  - The solution to the problem is the feasible solution which minimizes

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

    which is called the *objective function* .

Counting Change: Brute-Force Algorithm (3)

- There are $\mathbf{2}^{\boldsymbol{n}}$ possible values for $\boldsymbol{X} \Rightarrow$ running time of a brute-force solution is $\Omega(\mathbf{2}^{\boldsymbol{n}})$
- The running time needed to determine whether a possible value is a feasible solution is $\boldsymbol{O(n)}$, and
- The time required to evaluate the objective function is also $\boldsymbol{O(n)}$.
- Therefore, the running time of the brute-force algorithm is $\boldsymbol{O(n2^n)}$.

Applications of Greedy Strategy

- Optimal solutions:
  - Change making
  - Minimum Spanning Tree (MST)
  - Single-source shortest paths
  - Simple scheduling problems
  - Huffman codes
- Approximations:
  - Traveling Salesman Problem (TSP)
  - Knapsack problem
- Other combinatorial optimization

Counting Change: Greedy Algorithm

- A cashier does not really consider all the possible ways in which to count out a given sum of money.
- Instead, she/he counts out the required amount <u>beginning with the largest</u> denomination and proceeding to the smallest denomination
- Example: $\{\boldsymbol{d_1}, \boldsymbol{d_2}, \ldots, \boldsymbol{d_{10}}\}$ = $\{1, 1, 1, 1, 1, 5, 5, 10, 25, 25\}$. Count 32: 25, 5, 1, 1
- Greedy strategy: once a coin has been counted out it is never taken back
- If the pieces of money (notes and coins) are sorted by their denomination, the running time for the greedy algorithm is $\boldsymbol{O(n)}$.
- Greedy algorithm does not always produce the best solution. Example: set:$\{1, 1, 1, 1, 1, 10, 10, 15\}$. count 20: 15, 1, 1, 1, 1, 1. But best solution is: 20: 10, 10.

Example: 0/1 Knapsack Problem

- Given:
  - A set of n items from which we are to select some number of items to be carried in a knapsack.
  - Each item has both a *weight* and a *profit*.
  - The *objective:* chose the set of items that fits in the knapsack and maximizes the profit.
- Let:
  - w_i be the weight of item i,
  - p_i be the profit accrued when item i is carried in the knapsack, and
  - W be the capacity of the knapsack.
  - x_i be a variable which has the value 1 when item i is carried in the knapsack, and 0 otherwise

Example: 0/1 Knapsack Problem (2)

- Given $\{\boldsymbol{w_1, w_2, \ldots, w_n}\}$ and $\{\boldsymbol{p_1, p_2, \ldots, p_n}\}$, our *objective* is to maximize

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

- subject to the constraint

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

- Can solve this problem by exhaustively enumerating the feasible solutions and selecting the one with the highest profit.
  - since there are $\mathbf{2}^{\boldsymbol{n}}$ possible solutions, the running time required for the brute-force solution becomes prohibitive as $\boldsymbol{n}$ gets large.

Example: 0/1 Knapsack Problem (3)

- **Possibilities** (provided the capacity of the knapsack is not exceeded)**:**
- **Greedy by Profit**
  - At each step select from the remaining items the one with the highest profit.
  - Chooses the most profitable items first.
- **Greedy by Weight**
  - At each step select from the remaining items the one with the least weight.
  - Tries to maximize the profit by putting as many items into the knapsack as possible.
- **Greedy by Profit Density**
  - At each step select from the remaining items the one with the largest *profit density*, p_i / w_i.
  - Tries to maximize the profit by choosing items with the largest profit per unit of weight.

Example: 0/1 Knapsack Problem (4)

- Assume $W = 100$

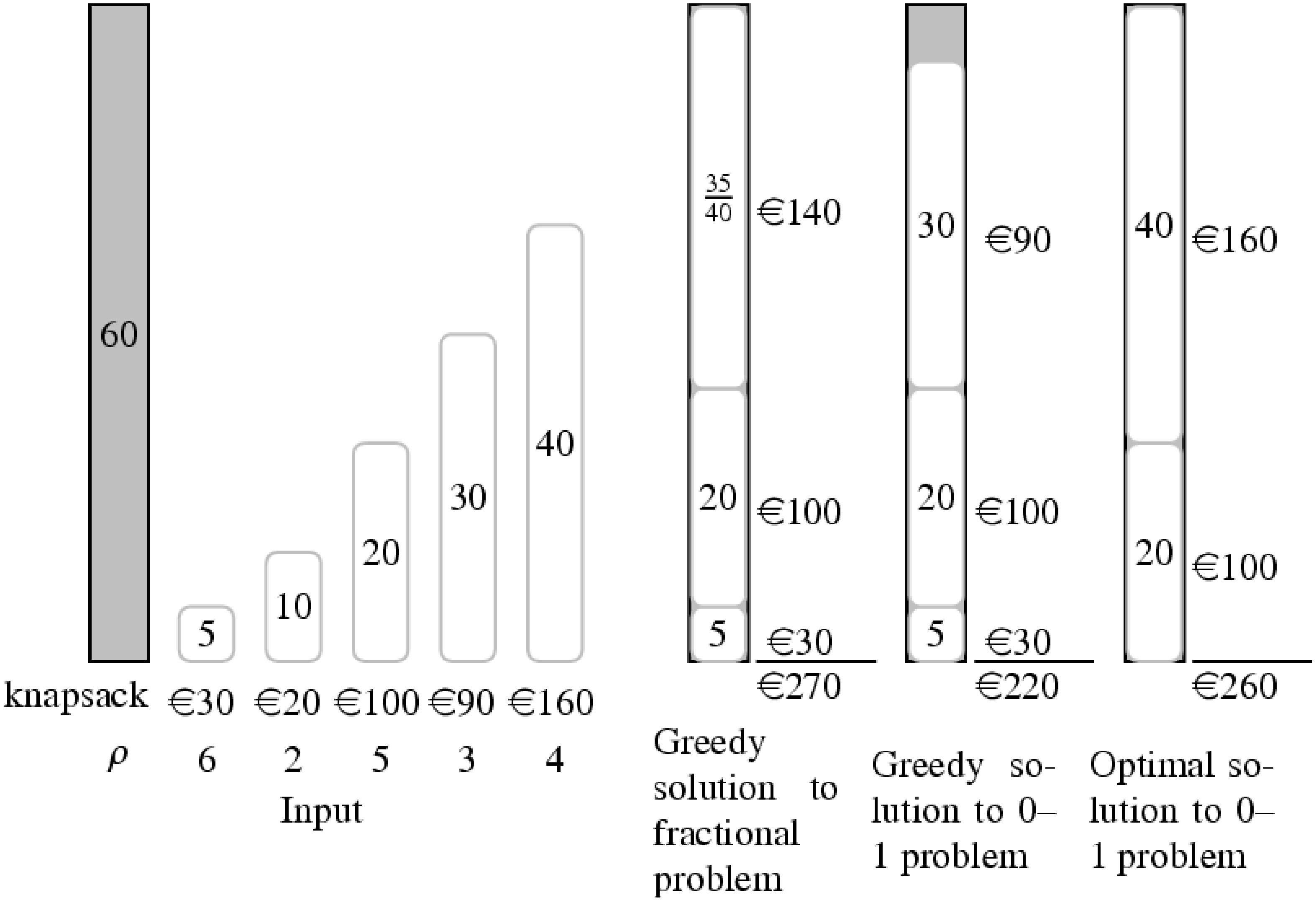
				greedy by			
i	w_i	p_i	p_i/w_i	profit	weight	density	optimal solution
1	100	40	0.4	yes	no	no	no
2	50	35	0.7	no	no	yes	yes
3	45	18	0.4	no	yes	no	yes
4	20	4	0.2	no	yes	yes	no
5	10	10	1.0	no	yes	yes	no
6	5	2	0.4	no	yes	yes	yes
			total weight	100	80	85	100
			total profit	40	34	51	55

- Note: solution not always optimal

Fractionary Knapsack Problem

- Thief robbing a store; finds n items which can be taken.
- Item i is worth € v_i and weighs w_i pounds ($v_i, w_i \in \boldsymbol{N}$)
- Thief wants to take as valuable a load as possible, but has a knapsack that can only carry W total pounds
- 0–1 knapsack problem: each item must be left (0) or taken (1) in its *entirety*
- Fractionary knapsack problem: thief is allowed to take *any fraction of an item* for a fraction of the weight and a fraction of the value. Greedy algorithm:
  - Let $\rho_i = v_i / w_i$ be the value per pound ratio (profit density)
  - Sort the items in decreasing order of ρ_i, and add them in this order

Knapsack Problem Example



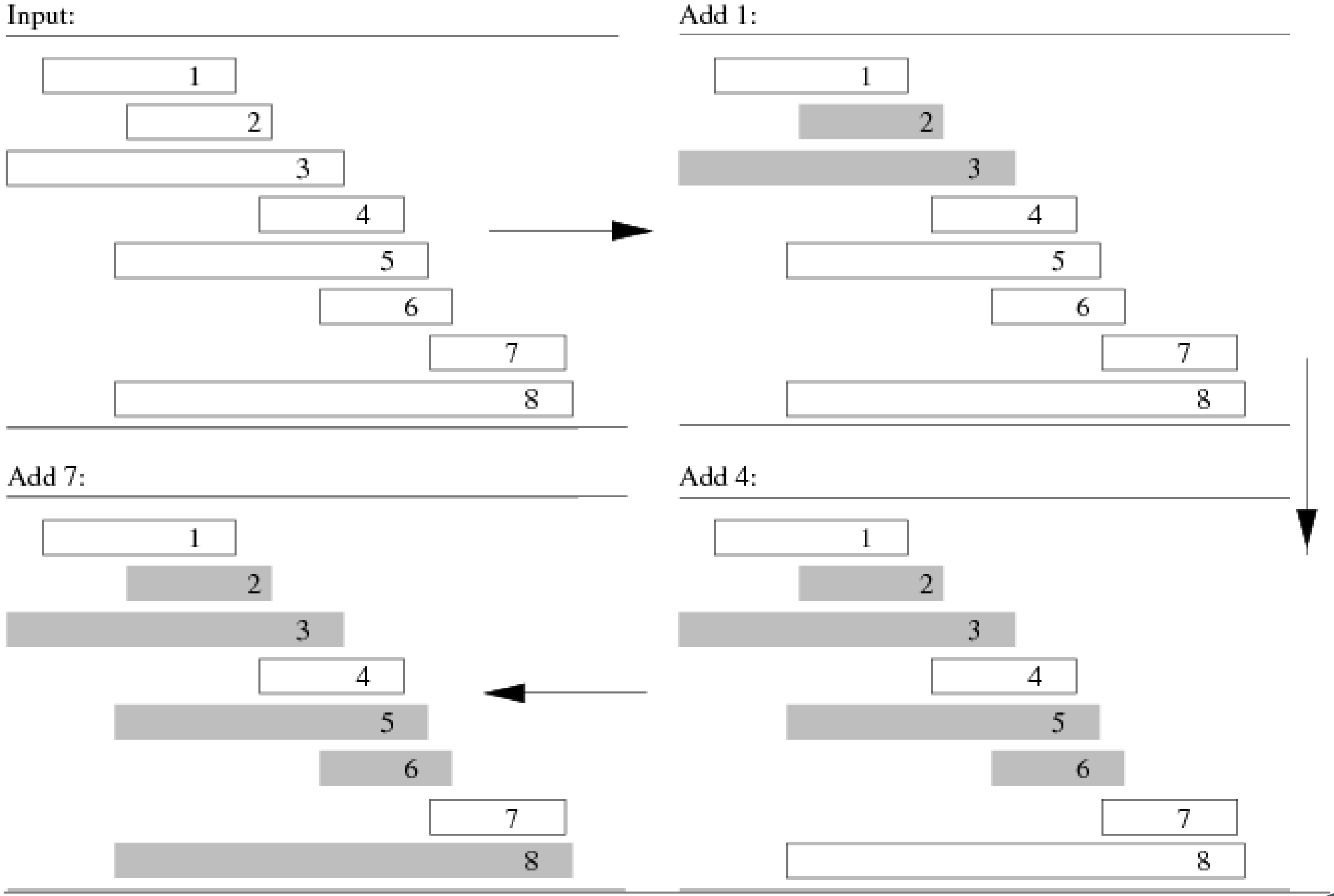
An Activity-Selection Problem

- Let S be a set of activities $S=\{a_1, a_2,\ldots, a_n\}$
  - They use resources, such as lecture hall, one lecture at a time
  - Each a_i, has a start time s_i, and finish time f_i, with $0\leq s_i< f_i < \infty$.
  - a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
- Goal: select maximum-size subset of mutually compatible activities.

Greedy Activity Scheduler

```
GREEDYACTIVITYSCHEDULER(s, f)
n ← length[s]                 ▷ we assume f[1..n] is already sorted
A ← ⟨1⟩                       ▷ schedule activity 1 first
prev ← 1
for i ← 2 to n
      do if s_i ≥ f_prev            ▷ no interference?
            then append i to A
                 prev ← i           ▷ schedule i next
return A
```

Greedy Activity Scheduling Example



Prefix Codes

- A prefix code is a code in which no codeword is also a prefix of another codeword. For example, the variable-length $\boldsymbol{a}=0$, $\boldsymbol{b}=101$, $\boldsymbol{c}=100$, $\boldsymbol{d}= 111$, $\boldsymbol{e}= 1101$ $\boldsymbol{f}=1100$ code is a prefix code.
  - Variable-length prefix codes can be simply encoded and decoded.
- For example, the four characters ***cabf*** is encoded as: 10001011100
- For decoding it, we chose the prefix which forms a code, remove it, and continue with the rest.
  - Decoding a word can be understood as finding a leaf in a tree, starting from the root.

Huffman Codes

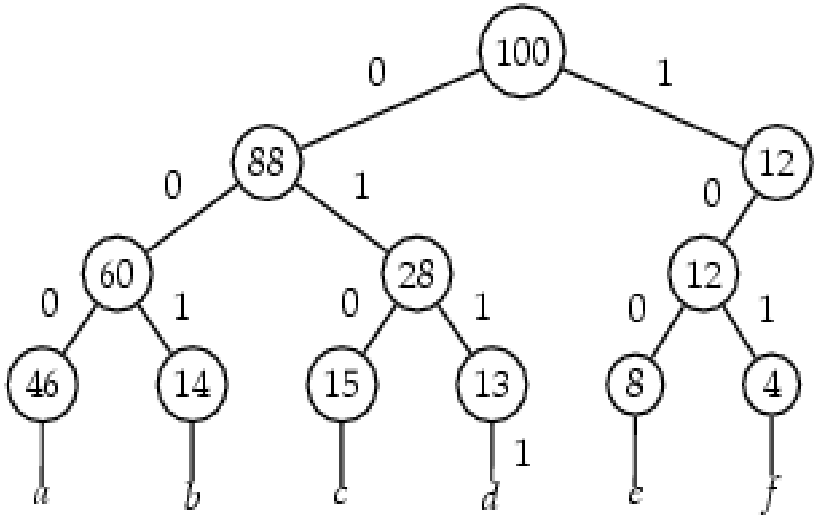
- Huffman codes are an - in a certain sense - optimal way for compressing data, typically for storage or transmission.
  - Suppose the input is a sequence of characters and suppose that – as is typically with text - certain characters appear more frequent than other characters.
  - The idea is that rather than using a fixed-length binary code for the text, using a variable-length code where more frequent characters have a shorter code than less frequent characters.
- Example:

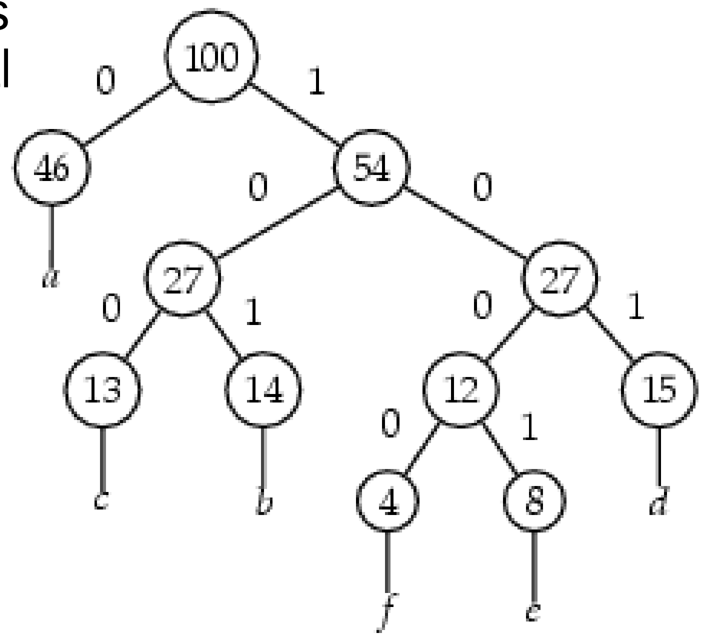
	a	b	c	d	e	f
Frequency	46	14	13	15	8	4
Fixed-length code	000	001	010	011	100	101
Variable-length code	0	101	100	111	1101	1100

  - A file with 100 characters takes 300 bits with the fixed-length code and 224 bits with the variable-length code.

Codes as Trees

- Trees corresponding to codes, with each node labeled by its frequency the leaves labeled with character coded by that patl





- Assume that ***f***(***z***) is the frequency of ***z*** and ***dT***(***z***) is the depth of ***c***'s leaf in the tree, which is the same as the length of the codeword for ***z*.** The number of bits for encoding a file can be computed as follows $B(T) = \left(\sum z \mid z \in C \bullet f(z) d_T(z)\right)$

Greedy Algorithm for Huffman Codes

- A code with tree T is optimal if $B(T)$ is minimal (for a fixed frequency of the characters)
  - Huffman codes are optimal prefix codes. For example, the code to the right on the previous slide is a Huffman code.
  - A greedy algorithm for constructing Huffman codes is based on following two properties:
    - If x and y are two characters with lowest frequency, then there exists an optimal prefix code in which the codes of x and y differ only in their last bit (hence have same length)
    - If T is an optimal prefix code for alphabet C and $x, y \in T$ are two leaves, then replacing x, y with a new parent z with $f(z)=f(x)+f(y)$ results in an optimal prefix code for $C - \{x, y\} \cup \{z\}$.

Constructing Huffman Codes

- Huffman's algorithm uses a priority queue ***Q*** to identify the two least-frequent objects to merge together.

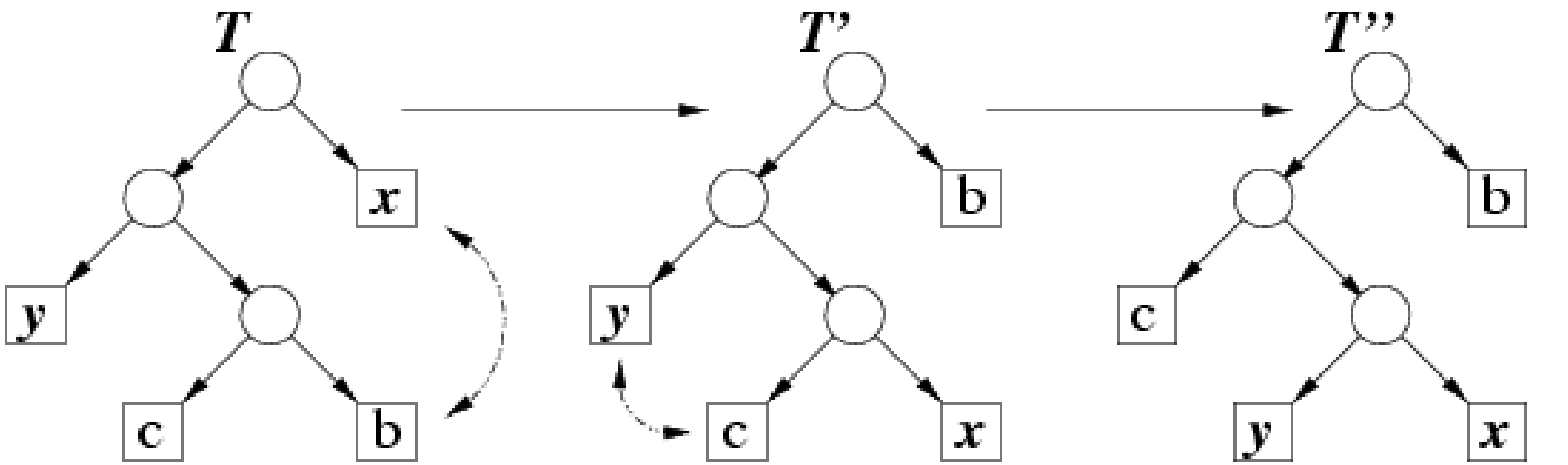
```
HUFFMAN(C)

n ← |C|
Q ← C
for i ← 1 to n − 1
        do z ← ALLOCATENODE()
           x ← left[z] ← EXTRACTMIN(Q)
           y ← right[z] ← EXTRACTMIN(Q)
           f[z] = f[x] + f[y]
           INSERT(Q, z)
return EXTRACTMIN(Q)
```

- The algorithm returns the single element left in the heap, which is the sum of all frequencies of all characters of ***C***.

Huffman Correctness

- Let T be any optimal prefix code tree, and let b and c be two siblings at the maximum depth of the tree.
- Assume without loss of generality that $p(b) \le p(c)$ and $p(x) \le p(y)$
- since x and y have the two smallest probabilities it follows that $p(x) \le p(b)$ and $p(y) \le p(c)$
- b and c are at the deepest level: $d(b) \ge d(x)$ and $d(c) \ge d(y)$
- Thus, we have $p(b) - p(x) \ge 0$ and $d(b) - d(x) \ge 0$, and hence their product is nonnegative.
- Now switch the positions of x and b in the tree, resulting in a new tree T'.

Hufman Correctness (2)



$$
\begin{aligned}
B(T') &= B(T) - p(x)d(x) + p(x)d(b) - p(b)d(b) + p(b)d(x) \\
&= B(T) + p(x)(d(b) - d(x)) - p(b)(d(b) - d(x)) \\
&= B(T) - (p(b) - p(x))(d(b) - d(x)) \\
&\leq B(T) \text{ because } (p(b) - p(x))(d(b) - d(x)) \geq 0.
\end{aligned}
$$

Hufman Correctness (2)

- Claim: Huffman's algorithm produces the optimal prefix code tree
- Proof (by induction on n)
  - Assume: < n characters, Huffman's algorithm is guaranteed to produce the optimal tree (OPT).
  - Suppose we have exactly n characters. The previous claim states that we may assume that in OPT, the two characters of lowest probability x and y will be siblings at the lowest level of the tree.
  - Remove x and y, replacing them with a new character z whose probability is $p(z) = p(x) + p(y)$. Thus $n - 1$ characters remain.

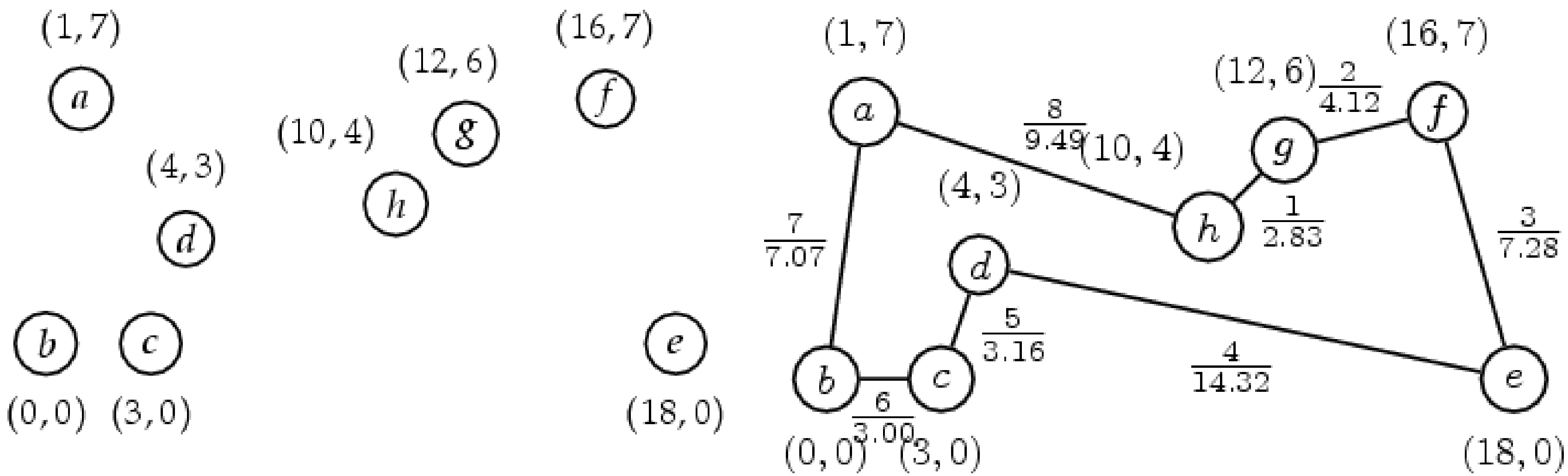
Hufman Correctness (3)

- Consider any prefix code tree T made with this new set of $n - 1$ characters
- We can convert it into a prefix code tree T' for the original set of characters by undoing the previous operation and replacing z with x and y (adding a "0" bit for x and a "1" bit for y). The cost of the new tree is

$$
\begin{aligned}
B(T') &= B(T) - p(z)d(z) + p(x)(d(z)+1) + p(y)(d(z)+1) \\
&= B(T) - (p(x)+p(y))d(z) + (p(x)+p(y))(d(z)+1) \\
&= B(T) + (p(x)+p(y))(d(z)+1-d(z)) \\
&= B(T) + p(x) + p(y).
\end{aligned}
$$

The Traveling Salesman Problem (TSP)

- Tour (Hamilton) (Hamiltonian cycle)
  - Given a graph with weights on the edges a tour is a simple cycle that includes all the vertices of the graph.
- TSP
  - Given a graph with weights on the edges, find a tour having a minimum sum of edge weights.



Here weights are Euclidean distances

A Greedy Algorithm for TSP

- Based on Kruskal's algorithm. It only gives a suboptimal solution in general.
- Works for <u>complete</u> graphs. May not work for a graph that is not complete.
- As in Kruskal's algorithm, first sort the edges in the increasing order of weights.
- Starting with the least cost edge, look at the edges one by one and select an edge only if the edge, together with already selected edges,
  1. does not cause a vertex to have degree three or more
  2. does not form a cycle, unless the number of selected edges equals the number of vertices in the graph.

The n-Queens Problem

- The problem is to place n queens (chess pieces) on an n by n board so that no two queens are in the same row, column or diagonal

		Q	
Q			
			Q
	Q		

  The 4 by 4 board above shows a solution for $n = 4$
- But first we will introduce an algorithm strategy called ***backtracking***, which can be used to construct ***all*** solutions for a given n.

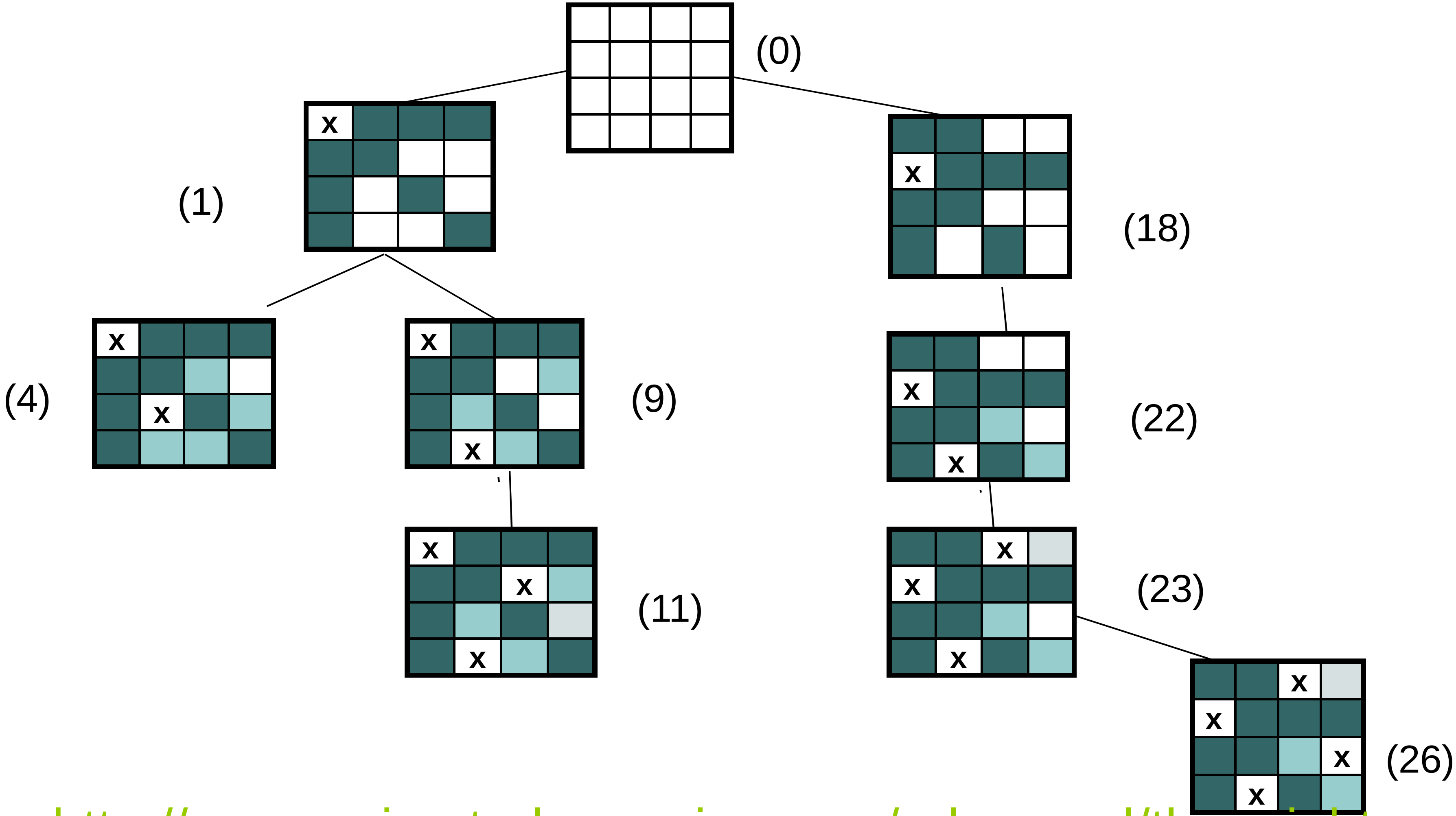
The n-Queens Problem

- We can solve this problem by generating a dynamic tree in a *depth-first* manner
  - Try to place a queen in each column from the first to the last
  - For each column, we must select a row
  - Start by placing a queen in row 1 of the first column
  - Then we check the rows in the second column for positions that do not conflict with our choice for the first column
  - After placing a queen in the second row, we continue to the third, etc.

The n-Queens Problem

- If at any point we find we cannot place a queen in the current column, we back up to the previous column and try a different row for that column
- We then continue trying to place more queens
- This process can be visualized by a tree of configurations
- The nodes are partial solutions of the problem
- The root is an empty board
- The children of the root will be boards with placements in the first column
- We will generate the nodes of the tree dynamically in a depth-first way

n-Queens Problem Instance



- http://www.animatedrecursion.com/advanced/the_eight_queens_problem.html

Backtracking Solution to n-Queens Problem

- We could continue with the previous example to obtain all possible solutions for $n = 4$
- Our goal now is to convert this approach into an explicit algorithm
- We track the position of the queens by using a array row
- $row[k]$ is the row in column k containing a queen
- The key function is the recursive function ***rnQueens***
- When $rnQueens(k, n)$ is called, queens have been successfully placed in columns 1 to $k - 1$
- The function then attempts to place a queen in column k
- If it is successful, then
  - if $k = n$, it prints the solution
  - if $k < n$, it makes the call $rnQueens(k+1, n)$
  - it $k > n$ then returns to the call $rnQueens(k-1, n)$

Backtracking Solution to n-Queens Problem

- To test for a valid position for a queen in column ***k***, we use a function ***positionOK*(*k,n*)** which returns true if and only if the queen tentatively placed in column ***k*** does not conflict with the queens in positions 1 to ***k*** –1
- The queens in columns ***i*** and ***k*** conflict if
  - they are in the same row: ***row*[*i*] = *row*[*k*]**
  - or in the same diagonal: **| *row*[*k*] – *row*[*i*] | == *k* – *i***
- We are thus led to our first version of the functions

Backtracking Solution to n-Queens Problem

```
nQueens(n)
{
  rnQueens(1, n)
}

positionOK(k)
{
  for i = 1 to k–1
      if (row[k] = row[i] ||
           abs(row[k]–row[i]) = k–i)
              return false;
  return true;
}

rnQueens(k,n) {
   for row[k]= 1 to n
     if (positionOK(k) )
       if ( k = n )
       {
          for i = 1 to n
             print(row[i] + " ");
          println;
       }
     else
         rnQueens(k+1, n)
}
```

http://www.animatedrecursion.com/advanced/the_eight_queens_problem.html
http://en.wikipedia.org/wiki/Eight_queens_puzzle

n-Queens Problem: Running Time

- Improvements can be made so that ***positionOK***(***k,n***) runs in $O(1)$ time rather than $O(k)$ time.
- We will obtain an upper bound for the running time of our algorithm by bounding the number of times ***rnQueens***(***k,n***) is called for each $k < n$
  - $k = 1$: 1 time, by ***nQueens***
  - $1 < k < n$: $n(n-1)\ldots(n-k+2)$ at most, since there are $n(n-1)\ldots(n-k+2)$ ways to place queens in the first k-1 columns in distinct rows
- Ignoring recursive calls, ***rnQueens***(***k,n***) executes in $\Theta(n)$ time for $k < n$.
- This gives a worst-case bound for ***rnQueens***(***k,n***) of

$$n\ [\ n(n-1)\ldots(n-k+2)\ ]\ \text{for}\ 1 < k < n$$

- For $k = n$, there is at most one placement possible for the queen. Also, the loop in ***rnQueens***(***k,n***) executes in $\Theta(n)$ time.

n-Queens Problem: Running Time (2)

- There are $n(n-1)\ldots2$ ways for the queens to have been placed in the first $n-1$ columns, so the worst case time for *rnQueens*(n, n) is

$$n(n-1)\ldots2$$

- Combining the previous bounds, we get the bound

$$n\,[\,1 + n + n(n-1) + \ldots + n(n-1)\ldots2]$$
$$= n\cdot n!\,[\,1/n! + 1/(n-1)! + \ldots + 1/1!\,]$$

  A result from calculus:

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} = 1 + \sum_{i=1}^{\infty} \frac{1}{i!}$$

- This means that $n\cdot n!\,[\,1/n! + 1/(n-1)! + \cdots + 1/1!\,] \leq n\cdot n! \cdot (e-1)$
- We thus have that our algorithm runs in $O(n\cdot n!)$ time

General Form of a Backtracking Algorithm

- Solution is of the form ***x*[1], . . . , *x*[*n*],** where the values of ***x*[*i*]** are from some set ***S*** – this set would be {1,…, ***n***} for the ***n***-queens problem
- General form:

```
// invocation
backtrack(n)
{
      rbacktrack(1, n)
}
```

```
rbacktrack(k, n)
{
  for each x[k] ∈ S
    if ( bound(k) )
      if (k = n)
       {//output a solution;
        //stop if one solution is desired
        for i = 1 to n
           print(x[i] + " ")
        println( )
       }
      else
        backtrack(k+1, n)
}
```

General Form of a Backtracking Algorithm

- The function $bound(k)$ assumes that $x[1]$,…, $x[k-1]$ is a partial solution and that $x[k]$ has been given a value
- The key is to choose a bounding function that eliminates many potential nodes from the tree (idea of branch –and-bound)

Hamiltonian-Cycle Problem

- The problem of determining if a graph has a Hamiltonian cycle is known to be NP-complete
- This means it is extremely unlikely that there is a polynomial time algorithm for this problem
- We can use backtracking to get an algorithm for this problem with decent running time for moderate size graphs
- We number the vertices of the graph from **1** to ***n*** and use the adjacency matrix representation
- We want $\boldsymbol{x[1], \ldots, x[n]}$ to be vertices such that $\boldsymbol{x[1] \ldots x[n]\ x[1]}$ is a Hamiltonian cycle for the graph
- Thus they must be distinct and each vertex and its successor in the list must be adjacent
- Since a Hamiltonian cycle passes through every vertex, we may assume it starts at vertex **1**

Backtracking Algorithm for Hamiltonian Cycles

```
hamilton(adj, x) {
  n = adj.last
  x[1] = 1
  used[1] = true
  for i = 2 to n
     used[i] = false
  rhamilton(adj, 2, x)
}
pathOK(adj, k, x) {
  n = adj.last
  if ( used[x[k]] )
     return false
  if ( k < n )
    return ( adj[ x[k–1], x[k] ] )
  else
    return (adj[ x[k–1], x[k] ]  ∧ adj[ x[k], x[1] ] )
}
```

```
rhamilton(adj, k, x)  {
  n = adj.last
  for x[k] = 2 to n
     if ( pathOK(adj, k, x) ) {
       used[k] = true
       if ( k = n )
         return true
       else if ( rhamilton(adj, k+1, x))
            return true
     }
  return false
}
```

http://en.wikipedia.org/wiki/Knight's_tour

Reading

- AHU, chapter 10, sections 3, 4 and 5
- Preiss, chapter: Algorithmic Patterns and Problem Solvers, sections Brute-Force and Greedy Algorithms
- CLR, chapter 17, CLRS chapter 16
- Notes