5. Greedy and other efficient optimization algorithms

1. When the next step is easy to choose
2. Greedy algorithms on graphs
3. Space/time tradeoffs; dynamic programming
4. Transform and conquer

Inquiry

• Who here has worked a cash register? How do you make change?
• Which problems have a slow brute-force solution but also have a big-\(O(n)\) solution?
• How does a map app work?
• Is “always go uphill” *always* the fastest way to climb a hill?
5. Greedy and other efficient optimization algorithms

**Topic objective**

Explain and apply the greedy method of algorithm design.

**Relevant background objectives**

5.0a Describe a basic concept of graph theory**

5.0b Describe an operation on a graph**

*See Background slides and study questions for support with these*
5. Greedy and other efficient optimization algorithms

1. When the next step is easy to choose

- Have your courses been easy to decide on?
- Job-interview scheduling?
- Driving routes?

Subtopic objectives

5.1a Explain greedy algorithm design*

5.1b Explain the optimal-substructure property*
5. Greedy and other efficient optimization algorithms

Are BFS and DFS fast?

Exercise:
1. What is the least-cost route from A to D?

2. How do you know?
3. What routes would BFS and DFS find?

Making change

- Problem: make change with a minimum number of coins
- For each coin denomination (largest to smallest) pick out as many coins of that denomination as possible, without exceeding the remaining amount due, and subtract the amount added from remaining amount due
- This is greedy in that at each step the next denomination chosen is the largest that still fits
5. Greedy and other efficient optimization algorithms

**Make-change (amt, Denom)**

\[
i \leftarrow |Denom|
\]

while \(i > 0 \) and \( amt > 0 \)

write \[\lfloor \frac{amt}{Denom[i]} \rfloor\]

\( amt \leftarrow amt \mod Denom[i] \)

\( i \leftarrow i - 1 \)

- Assumes that \( amt \) is change due and \( denom \) is an array of denominations, e.g., \((1, 5, 10, 25)\)
- Outputs number of coins due for each denomination

**Solving some optimization problems**

- *Constraint problem*: To find some value that satisfies a set of constraints or conditions
- *Optimization problem*: to find maximum or minimum valued solution, among all solutions
- Minimizing cost function \( f \):

\[
\min(f) = m \text{ s.t. } f(m) = \min_{x \in \mathbb{N}} \{ y \mid y = f(x) \}
\]

- *Greedy algorithms* iteratively build a minimal-cost structure by choosing a step that minimizes the cost function
5. Greedy and other efficient optimization algorithms

**Greedy algorithms**

- Solve optimization problems by seeking *local optima* as the next step of a state-space search
- Greedy algorithms expand a partial solution until problem is solved
- Choice at each step is feasible, irrevocable
- Not all greedy algorithms produce optimal results, or even results meeting minimal constraints
- **Cases:** Making change; two algorithms to find minimal spanning tree; Dijkstra’s for shortest path; Huffman coding

**Compression and packing**

- **Compression problem:** Map alphabet to a set of variable-length bit encodings s.t. most frequently used characters are represented by shortest encodings
- **Packing problem** (knapsack): Find a maximum-valued set of items, each with weights and values, that fit into a container with a given maximum weight
5. Greedy and other efficient optimization algorithms

### Prefix-free variable-length codes

- **Problem:** Find a space-optimal variable-length bit encoding for an alphabet, given a distribution of occurrences of characters in strings to be encoded
- **Example:** Morse Code solves a simpler problem allowing *character delimiters*
- **Constraint:** Codes must be *prefix-free*, no codeword is a prefix of another codeword
- Binary tree can represent the code, where left branches are labeled 0 and right ones labeled 1

### Huffman’s compression algorithm

Create one-node trees, one for each character
Label nodes by character and its probability
Repeat until all trees are one:
- Find two trees with minimum weight so far
- Pair them as subtrees of a new node
- Weight (parent) \( \leftarrow \) sum of weights of subtrees

- This greedy algorithm achieves an encoding with optimum compression ratio by generating an optimal binary tree

[Pic Levitin, p. 326]
Greedy solution to continuous-knapsack problem

- **Problem:** Find maximum-valued subset of a set of weighted items $A[1..n]$, totaling less than $\text{max-wt}$
- **Solution:**
  1. Sort $A$ by ratio of value to weight
  2. While total weight so far is less than $\text{max-wt}$
     a. select max-ratio element of sorted $A$
     b. increment total weight so far
  3. Select some item of next element of $A$

Optimal-substructure property

- A problem has this property iff any structure that is an optimal solution is composed of optimal solutions to subproblems as well
- This property holds for *some* optimization problems, such as ones we have seen
- It is required for any greedy solution to produce an optimal solution
5. Greedy and other efficient optimization algorithms

### Hill climbing and the optimal-substructure property

- Imagine a hill that has no dips or flat ground anywhere from the base to the summit
- Thus, to get to the summit from any point on the side of the hill, the right way to go is always to choose an upward direction
- Thus any subpart of a path to a higher elevation is also a path to a relatively higher elevation
- The problem of climbing that hill has the optimal-substructure property

### Generalized greedy algorithm

\[
y \leftarrow \text{null} \\
\text{Repeat} \\
\quad \text{augment } y \text{ by appending the seemingly best available value from some set} \\
\quad \text{until } y \text{ satisfies the problem’s constraints} \\
\text{return } y
\]

- Normally \( y \) is a structure, e.g., a path in a graph
- Running time: \( \Theta(n \times T(\text{Find-best})) \)
5. Greedy and other efficient optimization algorithms

Limits of the greedy approach

• Some problems lack optimal-substructure prop.
• Examples: In baseball, chess, or checkers, a short-term sacrifice (e.g., bunt) may bring long-term net benefit
• Generalized state-space search is of an $n$-dimensional landscape full of local optima that are not global optima
• Example: Climbing to a higher ground or walking toward destination do not always optimally help us reach objective

Local optima in search

• State-space search: Finds global optimum in a very large set of possible solutions (states)
• Search goes from state to state
• Example: Finding highest location in a county
  – Greedy algorithm follows uphill path until at the top of a hill
  – Low hills are local optima; highest hill is global optimum
  – Greedy algorithm alone does not find global optimum
2. Greedy algorithms on graphs

- What is a graph?
- What graph-related algorithms did you study in Data Structures?
- How does a map app work?

Subtopic objective

5.2a Describe a greedy algorithm for graphs*
5.2b Apply a greedy algorithm*
5. Greedy and other efficient optimization algorithms

Graph problems

- *Graph coloring*: color the vertices so that no adjacent vertices have the same color
- *Minimal spanning tree*: Smallest-weight tree containing all vertices in a weighted graph
- *Single-source shortest path*: Dijkstra’s algorithm builds a tree of shortest paths, starting from the source vertex.
- The problem of finding the shortest path to one destination is reducible to the problem of finding shortest paths to all

Graph-coloring problem

- *An optimization problem*: What is the minimum number of colors needed to color all the vertices of a graph (equivalent to a map), such that no two adjacent vertices have same color?
- Country on a map is equivalent to a vertex in a graph; borders between countries are equivalent to edges in the graph
- The compiler-design problem of *register allocation* is equivalent to the graph coloring problem
5. Greedy and other efficient optimization algorithms

Greedy-coloring ($G$)

- Assumes an ordered sequence of colors
- Gives (possibly non-optimal) solution to graph-coloring problem
- Q: What is the fastest algorithm that solves the coloring problem optimally?

```
for i ← 1 to |G.V|
    v[i].color ← min { c | c is not the color of a vertex in G adjacent to v[i] }
```

Minimal spanning tree

- MST is a subgraph of a graph
- Example: building a road or rail system, connecting $n$ locations, with lowest construction cost
- Not every path in MST is minimal; only the whole tree is minimal
- [pic]
5. Greedy and other efficient optimization algorithms

Strategy for minimal spanning tree

Repeatedly add to a single, growing tree the lowest-weight available edge adjacent to a vertex already in the tree, without forming cycle

Prim’s algorithm

\[ \text{Prim} \left( G \right) \quad G = (V, E) \]

\[ V' \leftarrow \text{choose-element}(V) \quad \text{// any vertex} \]
\[ E' \leftarrow \emptyset \]

while \( |V'| < |V| \) do

\[ e \leftarrow \text{min-wt-edge} \quad (E) \text{ s.t.} \]
\[ e = (v, u) \land (v \in V') \land (u \in V - V') \]

\[ V' \leftarrow V' \cup \{ v \} \]
\[ E' \leftarrow E' \cup \{ e \} \]

Return \((V', E')\)

> Post: \( T \) is a minimal spanning tree for \( G \)
5. Greedy and other efficient optimization algorithms

Kruskal’s algorithm

- Builds minimal spanning tree, bottom up, as an expanding sequence of subgraphs
- These are not necessarily connected until the algorithm terminates
- Greed applies via a weight-sorted list of all edges
- Each step connects two trees (possibly trivial) by the next edge in weight order
- Makes use of optimal-substructure property

Kruskal \( (G) \)  
\( G = (V, E) \)

\[ E_T \leftarrow \emptyset \]

Sort edges \( E \) into array \( e \) in weight order

\( k \leftarrow 1, \ count \leftarrow 0 \)

While \( count < \min\{|E|, |V| - 1\} \) do

- If \( (V_T, E_T \cup \{e_k\}) \) is acyclic
  - \( E_T \leftarrow E_T \cup \{e_k\} \)
  - \( k \leftarrow k + 1 \)

\( count \leftarrow count + 1 \)

Return \( (V, E_T) \)

- Sorting time dominates run time, hence run time is \( \Theta(|E| \ lg |E|) \)
Correctness of minimal spanning tree algorithms

- Shown by cut property:
- Let \( S \subseteq V, S \neq \emptyset \)
- Any edge \( e \) that is a minimal-cost bridge between \( S \) and \( (V - S) \) will be part of any MST
- Proof shows that any spanning tree lacking \( e \) is not minimal

Single-source shortest path

- In any weighted graph, from any source vertex, a tree exists composed of the set of shortest paths from the source to each other vertex
- The problem of a single shortest path reduces to building this tree (which is not the MST)
- Dijkstra’s algorithm builds this tree of shortest paths, starting from the source vertex
The Dijkstra algorithm

- Uses breadth-first search, beginning by examining the edge from the source to each adjacent vertex, then from each of these to its neighbors
- Builds solution tree along path of least immediate cost (greedy)
- If current path from source to a vertex \( v \) is found to be costlier than path to \( u \) plus path from \( u \) to \( v \), adds \( v \) to a minimum-path-so-far array, relaxing the estimate on path length

A shortest path’s subpaths are minimal

- Optimal-substructure property applies to shortest-path problem as follows
- If path \((u, \ldots, u', \ldots, v', \ldots, v)\) is minimal, then \((u', v')\) is minimal too (why?)
- So to minimize path \( u..v \), find shorter segments \( u'..v' \)
- \( u \) \( u' \) \( v' \) \( v \)
- \textit{Note:} more than one shortest path may connect two given vertices
5. Greedy and other efficient optimization algorithms

**Dijkstra (graph, weights, source)**

For $i \leftarrow 1$ to $|V|$ do

\[ \text{estimate}[i] \leftarrow \infty \]

\[ \text{PQ-insert}(Q, i) \]  // weight attribute is PQ key

\[ \text{estimate}[\text{source}] \leftarrow 0 \]

While $\neg \text{empty}(Q)$

\[ u \leftarrow \text{Extract-min}(Q) \]  //Greedy choice

For each vertex $v$ adjacent to $u$

If $\text{estimate}[v] > \text{estimate}[u] + \text{weight}(u,v)$

\[ \text{estimate}[v] \leftarrow \text{estimate}[u] + \text{weight}(u,v) \]

- Vector $\text{estimate}..|V|$ stores set of shortest known path lengths from $\text{source}$ to each vertex
- Key to priority queue $Q$ is $\text{estimate}[u]

**Application of Dijkstra algorithm**

<table>
<thead>
<tr>
<th>Step</th>
<th>$Q$</th>
<th>$u$</th>
<th>$v$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>abcde</td>
<td>a</td>
<td>b</td>
<td>0</td>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>bcde</td>
<td>b</td>
<td>c</td>
<td>0</td>
<td>3</td>
<td>9</td>
<td>$\infty$</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>cde</td>
<td>c</td>
<td>d</td>
<td>0</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4.</td>
<td>de</td>
<td>d</td>
<td>e</td>
<td>0</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5.</td>
<td>e</td>
<td>e</td>
<td>d</td>
<td>0</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Distance from $a$**

![Diagram of graph with steps of Dijkstra's algorithm, showing distances from vertex a to other vertices]

**Problem**

- $a$ to $b$: 6
- $a$ to $c$: 1
- $a$ to $d$: 5
- $a$ to $e$: 3

**Solution**

- $d$ to $e$: 2
- $d$ to $c$: 5
- $b$ to $c$: 3
- $b$ to $e$: 2
Correctness and time of Dijkstra

- **Theorem:** The Dijkstra algorithm finds a shortest path from source to \( u \) for all \( u \in V \)
- How would this be proved?
- Output is estimate array
- How are paths extracted from this array?
- **Running time, using heap, worst case:**
  - \( O((n+m) \lg n) \)
  - \( O(n^2 \lg n) \) if \( m = n^2 \)
  where \( n \) is # vertices, \( m \) is # edges

Problem: Comparing graph searches

- Let’s compare Dijkstra, DFS, BFS for
  - Finding short paths
  - Traversal time
  - Finding an edge with a given label
5. Greedy and other efficient optimization algorithms

Problem: find shortest paths

<table>
<thead>
<tr>
<th></th>
<th>Wash</th>
<th>Bos</th>
<th>NY</th>
<th>Phila</th>
<th>Chi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wash</td>
<td>400</td>
<td>200</td>
<td>150</td>
<td>1050</td>
<td></td>
</tr>
<tr>
<td>Bos</td>
<td>225</td>
<td>300</td>
<td></td>
<td>1200</td>
<td>1000</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Phila</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Max. matching in bipartite graphs

- Simpler problem: $G = (V, E)$, where $V$ is partitioned into disjoint sets $V_1, V_2$, s.t. $E \subseteq V_1 \times V_2$ (every element of $V_1$ is paired with one in $V_2$ and vice versa)

- Suppose $M$ is a matching in $G$; then if every vertex in $V_1$ or every one in $V_2$ is matched, $M$ cannot be improved

- But $M$ can be improved if an augmenting path is found that yields a larger matching

- Complexity: $O(|V|^2 + |V||E|)$
Stable matching

- **Problem:** How can elements of two disjoint sets be matched 1-to-1 by mutual preference … such that no unmatched pair exists where each would prefer the other in the pair over the current partner.
- **Examples:** heterosexual men and women; students seeking internships and companies each seeking one intern.

Stable-marriage problem

- **A bipartite matching problem** (selecting pairs, one item from each of two sets).
- Each element of each partition has ordered prefer list.
- **Stable pairing:** one in which no unmatched pair exists whose the members would prefer each other to their current matches.
- **Algorithm:**
  
  While some element \( x \) of \( V_1 \) is unmatched
  
  \( x \) proposes to next-preferred \( y \in V_2 \)
  
  \( y \) accepts if unmatched or if matched with a less-preferred partner.

- **Complexity:** \( O(n^2) \)
5. Greedy and other efficient optimization algorithms

3. Space/time tradeoffs and dynamic programming

• What computational resources are of concern when choosing an algorithm?
• What resource other than time is used by the Dijkstra algorithm? (Give its name)

Subtopic objective

5.3 Explain an instance of dynamic programming
Use of memory for performance

- Time efficiency can sometimes be gained by making use of storage space (memory)
- *Tables* or *larger tree nodes* may be used to obtain improved running times
- **Cases:**
  - Sorting by counting
  - String matching
  - Hashing
  - B trees

Fibonacci using table

- Recall $Fib(x) = \begin{cases} 
1 & \text{if } x \leq 1 \\
Fib(x - 1) + Fib(x - 2) & \text{otherwise}
\end{cases}$
- Running time is $\Theta(2^x)$
- Algorithm is $\Theta(x)$ using table:

```plaintext
DP-Fib(x)
F[0] ← 1, F[1] ← 1
For i ← 2 to x do
    F[i] ← F[i - 1] + F[i - 2]
Return F[x]
```
5. Greedy and other efficient optimization algorithms

**Binomial coefficient**

• $C(n, k)$ is the number of combinations (subsets) of $k$ elements chosen from a set of $n$ elements

• $C(n, k) =$

\[
\begin{cases}
1 & \text{if } k = 0 \text{ or } k = n \\
C(n - 1, k - 1) + C(n - 1, k) & \text{otherwise}
\end{cases}
\]

**Binomial** $(n, k)$

\[
\text{for } i \leftarrow 0 \text{ to } n \text{ do}
\]
\[
\hspace{1em} \text{for } j \leftarrow 0 \text{ to } \min\{i, k\} \text{ do}
\]
\[
\hspace{2em} \text{if } j = 0 \text{ or } j = k
\]
\[
\hspace{3em} C[i, j] \leftarrow 1
\]
\[
\hspace{2em} \text{else}
\]
\[
\hspace{3em} C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]
\]

\[
\text{Return } C[n, k]
\]

Time complexity: ________

Space complexity: ________
5. Greedy and other efficient optimization algorithms

Warshall’s algorithm

- Computes transitive closure (reachability matrix) of a digraph from its adjacency matrix
- Faster alternative to DFS or BFS for each pair
- Principle: If vertex \( j \) is reachable from \( i \), and \( k \) is reachable from \( j \), then \( k \) is reachable from \( i \)

\[
\text{Warshall} \left( M[n, n] \right)
\]

\[
\text{for } i \leftarrow 1 \text{ to } n \text{ do }
\]
\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do }
\]
\[
\text{for } k \leftarrow 1 \text{ to } n \text{ do }
\]
\[
\text{if } M[i, j] \land M[j, k]
\]
\[
M[i, k] \leftarrow \text{true};
\]

Return \( M \)

Floyd’s algorithm

- Finds shortest paths between any pair of vertices in a weighted graph
- Computes a distance, cost, or weight matrix
- Principle: Reduce cost estimate \( d_{ik} \) if shorter path found (greedy)

\[
\text{Floyd} \left( G[n, n] \right)
\]

\[
D \leftarrow G.W \quad \text{// weights matrix}
\]

\[
\text{for } i \leftarrow 1 \text{ to } n \text{ do }
\]
\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do }
\]
\[
\text{for } k \leftarrow 1 \text{ to } n \text{ do }
\]
\[
D[i, k] \leftarrow \min \{D[i, k], D[i, j] + D[j, k]\}
\]

Return \( D \)
Knapsack with table

- **Problem:** Given a set of $n$ items with weights $w_1 \ldots w_n$ and values $v_1 \ldots v_n$, find greatest-valued set of items that fit in knapsack of capacity $W$

- **Solution:** Let $V_{ij}$ be the optimal value of the first $i$ items in a knapsack of capacity $j$

  $$V[i, j] = \begin{cases} 
  \max \{ V[i-1, j], v_i + V[i-1, j-w_i] \} & \text{if } j > w_i \\
  V[i-1, j] & \text{otherwise}
  \end{cases}$$

- Time and space complexity: $\Theta(nW)$

---

Weighted interval scheduling

- Problem is to schedule weighted jobs in fixed intervals of time such that a jobs of maximum total weight are scheduled

- Solution entails double recursion

- Complexity of double recursion ($O(2^n)$) is reduced by use of table
**String matching**

- **Problem**: Find first occurrence of string of length \( m \) in string of longer length
- **Brute-force solution**: Perform \((n - m + 1)\) string comparisons, each of length \( m \)
- Faster **Boyer-Moore algorithm** (simplified):
  - A 26-element *shift table* for the search key
    store letters’ distances from the *right* of the key
  - Do string comparison from the right
  - Use the shift table to skip most comparisons
- **Average case**: \( \Theta(n) \) but “obviously faster”

**Longest common subsequence**

- Given sequences \( x_1, x_2 \), what is the longest subsequence \( y \) s.t. \( y \) is a subsequence of both \( x_1 \) and \( x_2 \)?
- Elements of subsequences are not necessarily contiguous, e.g., “dab” is a subsequence of “database”
- Dynamic programming solution: see Goodrich-Tamassia, pp. 568-572
5. Greedy and other efficient optimization algorithms

Hashing

• Dictionary is array in which index is computed from key value

• Desirable attributes of hash function: speed, even distribution of keys

• Two implementations: Open addressing with linear probe; array of buckets (linked lists)

• Load factor: ratio of # of entries to table size

• Time/space tradeoff: High load factor costs time, low load factor wastes space

Dynamic programming

• Some problems (e.g., Fibonacci) have overlapping subproblems

• Dynamic programming suggests solving each subproblem only once and storing solution in a table for later reference

• Cases:
  – Fibonacci
  – Binomial coefficient
  – Warshall’s and Floyd’s algorithms (graphs)
  – Optimal BSTs
  – Knapsack problem
5. Greedy and other efficient optimization algorithms

DNA sequence matching

- A bioinformatics problem, in which phylogenetic (family) relationships among protein sequences in DNA are found by comparing
- It is a more sophisticated type of string comparison

Phylogenetic trees

- A way to represent the evolutionary relationship among genes or organisms
- Terminal nodes are from empirical data, internal nodes are inferred common ancestors
- Newick format: (((a, b), (c, (d, e)))) =

```
  a
 /|
/  \
 b  c
 /|
/  \
 d  e
```

- May reflect substitutions in sequences: ABCD (ZBCD (ABYD, ZBCQ), ABXD)
Alignment between 2 sequences

- **Definition**: “a pairwise match between the characters of each sequence”
  (Krane and Raymer, p. 35)
- **Significance**: An alignment corresponds to a hypothesis about the evolutionary history connecting the sequences
- **Objective**: To find the best alignments between two sequences
- **Techniques for alignment comparison of sequences** are “a cornerstone of bioinformatics”

Alignment techniques

- **Want to align** a given two elements of language: $\Sigma^*$ where $\Sigma = \{C, G, A, P\}$
- **Objective**: To insert gaps in either of two DNA sequences to maximize pairwise matches
- **Example**: align `AATCTATA` with `AAGATA`
- **Possible solution**: `AATCTATA` `AA--GATA`
- A scoring method accounts for matches, mismatches, and gaps
5. Greedy and other efficient optimization algorithms

Needleman-Wunsch algorithm

- **Overview:** Break down alignment problem into smaller problems by finding best alignment of subsequences; storing them in a table rather than computing repeatedly
- **Example:** Align CACGA, CGA (p. 42)
- There are 3 ways to start, beginning at the left:
  1. C (...A...C...G...A)
     C (...G...A)
  2. C (...A...C...A...G...A)
     C (...G...A)
  3. C (...A...C...G...A)
     C (...G...A)

Needleman-Wunsch

- Global sequence alignment algorithm
- Assume match score is 1, mismatch is 0, gap is (-1)
- To evaluate alignments above,
  
  \[ score(1) = +1 \text{ (C matches C) plus alignment score of ACGA and CGA} \]
  
  \[ score(2) = -1 + score(CACGA, CGA) \]
  
  \[ score(3) = -1 + score(ACGA, CCGA) \]
- Fill out table: […]
### Needlenman-Wunsch table

<table>
<thead>
<tr>
<th></th>
<th>g</th>
<th>t</th>
<th>c</th>
<th>a</th>
<th>t</th>
<th>a</th>
<th>g</th>
<th>a</th>
<th>c</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
<td>-5</td>
<td>-6</td>
<td>-7</td>
<td>-8</td>
<td>-9</td>
</tr>
<tr>
<td>t</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
<td>-5</td>
<td>-6</td>
<td>-7</td>
</tr>
<tr>
<td>c</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
<td>-5</td>
</tr>
<tr>
<td>a</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>t</td>
<td>-4</td>
<td>-3</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>a</td>
<td>-5</td>
<td>-4</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### Principles guiding dynamic-programming approach

- Subproblems exist that can be easily solved using solutions to smaller sub-sub-problems
- Solutions to all subproblems may be combined easily to solve the main problem
4. Transform and conquer

- What’s the simplest way to find the median of an array?
- What is worst case for BST search?
- Can this time risk be remedied?

Subtopic objective

5.4 Explain an instance of transform-and-conquer
Algorithms using presorted arrays

- **Uniqueness verification** is linear-time after array is transformed by presorting
- Compare brute-force \(O(n^2)\) algorithm with algorithm using sorted array:

```
Uniqueness ( A[0 .. n – 1] )
Sort (A)
For i ← 0 to n – 2 do
    if A[ i ] = A[ i + 1 ]
        return false
Return true
```

Transformations of data

**Transformations:**
- Instance simplification
- Representation change
- Problem reduction

**Principle:** Performance advantages can be gained by changing the form of the input

**Problems:** Uniqueness, mode, matrix inverses, determinants, BST balancing, polynomial evaluation, least common multiple
## 5. Greedy and other efficient optimization algorithms

### Sorting by counting
- Suppose problem is to sort an array composed only of values in 1..\(m\)
- Then a solution is to count the occurrences of each value in 1..\(m\) and store in a table \(T\)
- Then write to the array \(T[1]\) 1’s, \(T[2]\) 2’s, etc.
- Running time \(O(n)\) is better than any comparison-based sort, provided that \(m \leq O(n)\)
- \(2 \ 5 \ 1 \ 2 \ 8 \ 7 \ 5 \ 1 \ 5 \ \Rightarrow \ 1 \ 1 \ 2 \ 2 \ 5 \ 5 \ 7 \ 8\)

### Finding mode
- **Mode**: most common element in array
- **Worst-case**: no duplications – brute force makes \(\Theta(n^2)\) comparisons to compile list of frequencies of elements [explain]
- Better algorithm using sorted array: Find longest run of equal values – \(\Theta(n)\)
- Complexity: \(\Theta(n \ lg \ n)\) including sort
5. Greedy and other efficient optimization algorithms

Mode \((A[0..n-1])\)

Sort \((A)\)

\(i \leftarrow 0, \ mode\_frequency \leftarrow 0\)

while \(i \leq n - 1\) do

\(run\_length \leftarrow 1\)

\(run\_value \leftarrow A[i]\)

while \(i + run\_length \leq n - 1\) and \(A[run] = run\_value\) do

\(run\_length \leftarrow run\_length + 1\)

if \(run\_length > mode\_frequency\) then

\(mode\_frequency \leftarrow run\_length\)

\(mode\_value \leftarrow run\_value\)

\(i \leftarrow i + run\_length\)

Return \(mode\_value\)

Reductions of problems

- Transform-and-conquer approach uses reducibility of some problems to others
- *Example: Least common multiple* problem is reducible to *greatest-common-divisor*:
  
  \[\text{lcm}(m, n) = mn / \text{gcd}(m, n)\]

- Finding extrema of *some* functions is reducible to finding derivative
- Problems like wolf-goat-cabbage (Levitin, p. 17, Problem 1) are reducible to state-space (graph) problems
### B trees
- Each node has $m$ children
- All data is stored in leaves
- All leaves are at same tree level
- Used to store very large indexes for databases stored on disk
- **Advantage**: extremely short paths to leaves ($\lg_m n$)
- **Disadvantage**: Wasted space

### Optimal BSTs
- **Problem**: Given probabilities that certain values will be search keys, find BST with minimum average search time
- **Solution**: Construct optimal subtree as one node with optimal left and right subtrees
- Dynamic-programming approach uses a table of average number of comparisons for a range of nodes
- Space complexity of transform process: $\Theta(n^2)$
- Time complexity of transform process: $\Theta(n^3)$
BST balancing

- **Problem:** Search of *degenerate* BST is $O(n)$
- **Tree balancing** is transform-conquer instance
- **Note:** Transformation from a set to a BST is itself a case of representation change
- **Problem:** preserve $O(lg\ n)$ properties of a balanced BST as it is built and updated
- **AVL tree:** BST with left, right subtrees differing in height by not more than 1

AVL trees

- Unbalanced BST subtree is transformed by *rotation* around root
- 4 kinds of rotation:
  - **Single**
  - **Double**

[Mirror images of Left]
AVL tree rebalancing

- Each node contains a value \textit{height}, denoting maximum path length to a leaf
- An imbalanced tree, with a chain of 3 nodes at root leading rightward, is rebalanced recursively by rotating so that root node becomes root of left subtree
- Each local trinode restructuring operation is O(1); an insertion or deletion may require O(\(\lg n\)) trinode restructurings

Red-black trees

\textit{Definition}: a kind of BST that maintains near-balance by the following property:

- All nodes are red or black
- Root is black
- Leaves are black
- Children of red nodes are black
- All paths from root to leaf have the same number of black nodes
Balancing red-black trees

- *Trinode restructurings* are used to restore the red-black property
- Only $O(1)$ structural changes are required to keep a red-black tree balanced after any insertion or deletion step
- Logarithmic time for insertion, deletion, search

Splay trees

- *Splay operation* on a BST moves a node to the root by a series of restructurings, e.g.:

  ![Splay Operation Diagram]

- Splaying improves balance
- Splay trees are splayed step by step at times of search, insertion, or deletion
- *Amortized analysis*: For a series of $m$ search/insert/delete operations, starting with an empty splay tree, with $n$ insertions, $T(m, n) = O(m \lg n)$
5. Greedy and other efficient optimization algorithms

(2, 4) trees

- **Definition**: balanced \( m \)-ary search trees such that \( m \in \{2, 3, 4\} \) for internal nodes
- Children are ordered by key; search goes to children of closest child node in case of no match at a given level
- On insertion into a 4-node, node is split into two: a 3-node and a 2-node
- Deletion may trigger merging of child nodes
- Search, insert, and delete are \( O(\lg n) \)

B trees

- Implements *Set* ADT
- Similar in principle to BST, but is balanced, by definition
- Branching factor is larger than 2; often > 100
- Nodes contain arrays of data elements and links to nodes; arrays are filled from left
- Nodes of a B tree of order \( d \) have between \( d/2 \) and \( d \) children; search or update is \( O(\lg_{d/2} n) \)
- **Motivation**: minimizes disk transfer time
5. Greedy and other efficient optimization algorithms

**More B tree rules**

- Tree has minimum branching factor \( m \)
- Each node has \( n \) data values, with \( m \leq n \leq 2m \); number of subtrees of a node is \( (n + 1) \)
- For nonleaf nodes, \( \text{subtree}[i] \) contains data values less than \( \text{data}[i] \), \( \text{subtree}[i+1] \) contains values greater than \( \text{data}[i] \)
- Every leaf has same depth
- Every subtree is a B tree

**B tree example**

- Minimum branching factor is 1
- *Search for ‘U’:* Right at HR; down at TW; success
- *Search for ‘K’:* down at HR; left at M; failure, since J is a leaf

![B tree diagram](image_url)
5. Greedy and other efficient optimization algorithms

B tree search algorithm

```plaintext
Search(node, key)

x ← min index of node.data s.t. key ≤ data[x]
If data[x] = key return true
else
  if node.link array is empty return false
  else return Search(link[x], key)
```

- This is equivalent to BST search, except that more than one value is searched in a node
- For large branching factor, first step may be done by binary search of node.data array

Inserting into a B tree

- If root.data array is not full, then insert in root; otherwise insert in the subtree node
- If data array is full, then split it and move middle element up to parent
- If root array is full, then create new root with middle element and others as its child nodes
- Hence B tree only gains height by this last root-adding step, thus leaves always stay at same level as each other
5. Greedy and other efficient optimization algorithms

Deleting from a B tree

- If root has no children, just delete target element from the data array in root
- Otherwise find largest data value stored in target’s subtree, replace target with that

Tries (pronounced “try”)

- Data structure to store a set of strings as a set of paths from the root
- Advantage: supports fast pattern matching
- If strings are stored as bits, trie is a binary tree
- Search time is (size of largest string stored) \times (size of alphabet)

Pattern matching with suffix tries

- The trie stores a collection of all the suffixes of substrings of a string
- Example (for “structure”):

```
structure                     e
  r u c t u e
  u c t u e
  c t u e
  t u e
  u e
  e
```

- Suffix trie enables fast determination whether a given string is a substring of the string stored in the trie

Horner’s Rule

- Algorithm to evaluate a polynomial:

```
Horner (P[0..n], x)
> P[0..n] are coefficients of degree-n polynomial
p ← P[n]
for i ← n – 1 downto 0 do
  p ← xp + P[i]
return p
```

- Complexity: Θ(n)
- Complexity of brute-force version: Θ(n^2)
- Rule can be used to do binary exponentiation in Θ(lg n) time
# References


