CSCI2100A/ESTR2102 – Data Structures (2024 Spring)

Algorithmic Paradigm

- Backtracking: Incrementally builds candidates to solutions and abandons a candidate (backtracks) as soon as it determines that candidate cannot possibly be completed to a valid solution.
- Brute-force Search: Systematically checking all possible candidates for whether each candidate satisfies the problem's statement.
- Divides-and-conquer: Recursively break down a problem into two or more subproblems of the same or related type, until these become simple enough to solve directly. The solutions to the subproblems are then combined to give a solution the original problem.
- Dynamic Programming: A problem that can be solved optimally by breaking it into subproblems and then recursively finding the optimal solutions to the subproblems.
- Greedy: Making the locally optimal choice at each stage, yielding a globally optimal solution.

Analysis

- Solving Homogeneous Recurrence Relations

Given an order-k homogeneous recurrence relation $x_n = a_1x_{n-1} + a_2x_{n-2} + \cdots + a_kx_{n-k}$. If its characteristic equation has distinct roots $r_1, r_2, \ldots, r_k$, then the general solution has the form of $x_n = c_1r_1^n + c_2r_2^n + \cdots + c_kr_k^n$. If the roots are $r_1, r_2, \ldots, r_s$ with multiplicities $m_1, m_2, \ldots, m_s$, respectively, then the general solution is the linear combination of the solutions:

$$\begin{matrix} r_1^n, & nr_1^n, & \ldots, & n^{m_1-1}r_1^n, \\ r_2^n, & nr_2^n, & \ldots, & n^{m_2-1}r_2^n, \\ & & \ldots, & \\ r_s^n, & nr_s^n, & \ldots, & n^{m_s-1}r_s^n. \end{matrix}$$

- Solving Non-homogeneous Recurrence Relations

For an order-k non-homogeneous recurrence relations $x_n = a_1x_{n-1} + a_2x_{n-2} + \cdots + a_kx_{n-k} + f(n)$. Let $x_n^{(s)}$ be a solution, called special solution. Then, the general solution is $x_n = x_n^{(s)} + x_n^{(h)}$, where $x_n^{(h)}$ is the corresponding homogeneous recurrence relation $x_n = a_1x_{n-1} + a_2x_{n-2} + \cdots + a_kx_{n-k}$. Example: if $k = 2, f(n) = cr^n$, then (1) if $r \neq r_1, r \neq r_2$, $x_n^{(s)} = Ar^n$, (2) if $r \neq r_1, r_1 \neq r_2$, $x_n^{(s)} = Anr^n$, (3) if $r = r_1 = r_2$, $x_n^{(s)} = An^2r^n$.

Divide-and-Conquer Analysis

Master Theorem: The recurrence $T(n) = aT(\frac{n}{b}) + cn^k, T(1) = c$, where a, b, c, k are all constants, solves to: $T(n) \in \Theta(n^k)$ if $a < b^k$; $T(n) \in \Theta(n^k \log n)$ if $a = b^k$; $T(n) \in \Theta(n^{\log_b a})$ if $a > b^k$.

Akra Bazzi Theorem: Let

$$T(x) = \begin{cases} \sum_{i=1}^{k} a_i T(b_i x + h_i(x)) + g(x), x \geq x_0 \\ \text{non} - \text{negative and bounded}, 0 \leq x \leq x_0 \end{cases}$$

Verify A1: x_0 is a constant; **A2:** $a_i > 0, \sum_{i=1}^{k} a_i > 1, 0 < b_i < 1$; **A3:** $g(x) \geq 0, |g'(x)| = O(x^p)$ for some integer p; **A4:** $|h_i(x)| = O(\frac{x}{(\log^2 x)})$. Let q satisfy $\sum_{i=1}^{k} a_i b_i^q = 1$. Then, $T(x) = (x^q(1 + \int_1^x \frac{g(u)}{u^{q+1}} \mathrm{d}u))$.

Useful Formulas

$\log_A B = \frac{\log_C B}{\log_C A}$, $\sum_{i=1}^{N} A^i = \frac{A^{N+1}-1}{A-1}$, $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$, $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$.

Object-Oriented Programming Concepts

(1) Data encapsulation / information hiding: concealing of the implementation of a data object; **(2)** Data abstraction: separation between specification of a data object and its implementation

Abstract Data Types: a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations.

List: a finite sequence of elements of a data type T supporting: creation, empty query, full query, size query, retrieval of an element, insertion of an element, deletion of an element, clearing, traversal.

Arrays	Lists
Fast lookups	Slow lookup
Ordered	Ordered
Slow insertion	Fast insertion
Static, Fixed size	Dynamic, Flexible size
Cache friendly	More memory for pointers

Insertion: (1) Allocate memory and store data for new node; (2) Traverse to node just before the required position of new node; (3) Change next pointers to include new node in between.

```
newNode->next = temp->next; temp->next = newNode;
```

Deletion: (1) Traverse to element just before the element to be deleted; (2) Change next pointers to exclude the node from the chain.

```
temp->next = temp->next->next;
```

Doubly Linked List - Advantages: Allows traversing in both forward/backward directions; Deletion is straightforward; Reversing list is easy. Disadvantages: extra memory is required.

Circularly Double Linked List – Advantages: Can go to any node from any node without special handling; No null pointer; Disadvantages: Complex and can go out of control.

Stack: a stack of elements of a data type T supporting: creation, empty, full, size, pushing of an element from the top of a stack, popping an element from the top of a stack, finding the top most element in the stack, clearing, where all insertions are made at one end. LIFO: last-in-first-out. Applications: Balancing Symbols, Reverse Polish Calculator, Rearrangement of carts.

Queue: an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, front. Supports creation, empty, enqueue, dequeue. FIFO: first-in-first-out. Implementations: Linear array; Circular array, Double-ended queue.

Tree: (1) Connected; **(2)** Acyclic; **(3)** Number of edges = Number of nodes − 1. **Depth**: length of unique path from root to v. **Height**: longest path from v to leaf. Depth of a tree = Height of a tree.

Full binary tree: every node except leaves has two children.

Complete binary tree: every level except possibly the last, is completely filled, and all nodes are as far left as possible.

Binary search tree: key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. Supports `find()`, `find_min()`, `find_max()`, `insert()`, `delete()`.

Insertion: Start from the root of the binary tree. Compare key of the current node to the key of the inserting node, (1) if the key of the current node is exactly the key of the inserting node, then increment the count of the current node on the tree by 1; (2) if the key of the current node is larger than the key of the inserting node, then

Deletion: If the node is a leaf, just remove it; otherwise, if the node has only one child, adjust the parent's pointer to bypass the node to its child, then remove the node to be deleted. General case: replace the key of the node with smallest key of right subtree (it cannot have a left child), recursively delete that node (which is now empty).

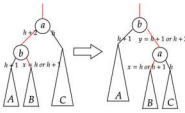
Average depth: Depth: $D(n) = D(i) + D(n-i-1) + n - 1$. All subtree sizes are equally likely:

$$D(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} D(j)\right] + n - 1.$$

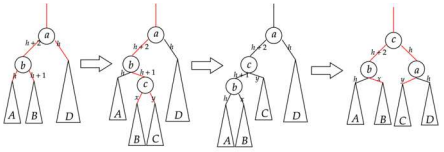
Preorder/Inorder/Postorder Traversals

AVL Tree: for every node in the tree, the height of the left and right subtrees can differ by at most 1.

Rotation: Single (LL/RR)



Double (LR/RL)

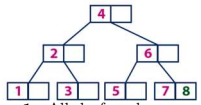


Height of AVL Tree is $O(\log n)$.

$$f_n = \begin{cases} 1, n = 1 \\ 2, n = 2 \\ f_{n-1} + f_{n-2} + 1, n > 2 \end{cases}$$

$$\Rightarrow f_n = \frac{5+2\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{5-2\sqrt{5}}{2}\left(\frac{1-\sqrt{5}}{2}\right)^n - 1$$

Trie: a prefix tree where all children of a node have a common prefix of the string associated with that parent node.



B-tree: Property 1: All leaf nodes are at the same level. **Property 2:** All nodes except root must have at least $\lceil m/2 \rceil - 1$ keys and maximum $m - 1$ keys. **Property 3:** All internal nodes must have at least $\lceil m/2 \rceil$ chlidren. **Property 4:** If the root is a non-leaf node, it must have at least two children. **Property 5:** A non-leaf node with $n - 1$ keys must have n children. **Property 6:** All key values in node are in ascending order.

Insertion Overflow: (1) Split the node (2) Send the middle element to the parent recursively.

Deletion Underflow: (1) Swap with the successor, delete from leaf (2) Redistribute keys in the same level with the right siblings.

Heap: a binary tree which satisfies the **order property** - for every node x, key of parent of $x \leq$ key of x.

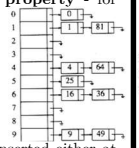
Insertion: Assign x to A[n+1], percolate-up()

Deletion: Swap with A[n], percolate-down().

Hashing:

Open Hash Table:

- whether the element is already in place.
- If the element turns out to be new, it is inserted either at the front of the list or at the end of the list. If the element exists, then use a linked-list deletion.



Find

- We use the hash function to determine which list to traverse.
- We then traverse this list in the normal manner, returning the position where the item is found.

Insertion/Deletion

- We traverse down the appropriate list to check whether the element is already in place.
- If the element turns out to be new, it is inserted either at the front of the list or at the end of the list. If the element exists, then use a linked-list deletion.

Closed Hash Table:

$h_i(x) = (hash(x) + f(i)) \bmod \text{H\_SIZE}, f(0) = 0.$

Insertion/Deletion:

If the corresponding location is empty, the record is inserted; otherwise, the insertion of new record will be determined by $h_i(x)$.

Lazy deletion:

Find: If the desired record is in the corresponding location, then the retrieval has succeeded; otherwise, while the location is nonempty, traverse through the collision function until encountering an empty location.

Lazy deletion: a special key indicating that the find function should not stop from here even though the entry is emptied.

Re-hashing: $O(n)$ running time, brute-force.

Hash types	Advantages	Disadvantages
Open hash table	Efficient use of memory (no resizing) Easy implementation	Additional memory for pointers Less cache friendly
Closed hash table with linear probing	Simple implementation Cache friendly	Primary clustering Degraded performance
Closed hash table with quadratic probing	Reduces primary clustering	Secondary memory Insertion could fail Table size is prime
Double hashing	Low clustering	Dependence on $h_2(x)$ Limited flexibility Long probing sequences

Bubble Sort $O(n^2)$

```
for i=1 to n-1 do
  for j=n to i+1 do
    if A[j].key < A[j-1].key then
      swap(A[j],A[j-1])
```

Requires little additional space. $O(n)$ scan determine sorted.

Original	34	8	64	51	32	21	Number of Exchange
After p = 1	8	34	21	64	51	32	4
After p = 2	8	21	34	32	64	51	3
After p = 3	8	21	32	34	51	64	2
After p = 4	8	21	32	34	51	64	0
After p = 5	8	21	32	34	51	64	0
After p = 6	8	21	32	34	51	64	0

Insertion Sort $O(n^2)$

```
for i=2 to n do
  key=A[i], i=j-1
  while (A[i]>key and i>0) do
    A[i+1]=A[i], i--
  A[i+1]=key
```

$O(n)$ scan determine sorted.

Original	34	8	64	51	32	21	Positions Moved
After p = 1	34	8	64	51	32	21	0
After p = 2	8	34	64	51	32	21	1
After p = 3	8	34	64	51	32	21	0
After p = 4	8	34	51	64	32	21	1
After p = 5	8	32	34	51	64	21	3
After p = 6	8	21	32	34	51	64	4

Selection Sort $O(n^2)$

```
for i=1 to n-1 do
  for j=i+1 to n do
    if A[i]>A[j] then
      swap(A[i], A[j])
```

Original	34	8	64	51	32	21	Positions Moved
After p = 1	34	8	64	51	32	21	1
After p = 2	8	34	64	51	32	21	4
After p = 3	8	21	64	51	32	34	2
After p = 4	8	21	32	51	64	34	2
After p = 5	8	21	32	34	64	51	1
After p = 6	8	21	32	34	51	64	0

Shell Sort $O(n^2)$

```
for k=1 to m do
  cur_gap=gap_k
  for i=1 to cur_gap do
    Apply Insertion Sort to chain
      [x[i + t(cur_gap + 1)] | t = 0,1,..., ⌊(n − i)/(cur_gap + 1)⌋]
```

$h(1) = 1$ and $h(i+1) = 3 \times h(i) + 1$. x be the smallest integer such that $h(x) \geq n$, and set *numinc*, the number of increments, to $x - 2$ and *incrmnts*[i] to $h(numinc - i + 1)$ for i from 1 to *numinc*.

Heap Sort $O(n \log n)$

Construct a min heap, apply n delete_min operations.

Merge Sort $O(n \log n)$

```
Algorithm MergeSort(Arr, start, end)
1. If start < end Then
2.   mid = (start + end)/2
3.   MergeSort(Arr, start, mid)
4.   MergeSort(Arr, mid + 1, end)
5.   Merge(Arr, start, mid, end)
```

```
Algorithm Merge(Arr, start, mid, end)
1.  temp = Create array temp of same size Arr
2.  i = start, j = mid + 1, k = start
3.  While i <= mid and j <= end
4.    If Arr[i] > Arr[j] Then
5.      temp[k++] = Arr[j++]
6.    Else
7.      temp[k++] = Arr[i++]
8.  While i <= mid
9.    temp[k++] = Arr[i++]
10. While j <= end
11.   temp[k++] = Arr[j++]
12. Loop from p = start to end
13.   Arr[p] = temp[p]
```

Quick Sort $O(n \log n)$

```
1: function PARTITION(A, lo, hi)
2:     pivot ← A[hi]
3:     i ← lo-1;
4:     for j = lo; j ≤ hi-1; j ← j+1 do
5:         if A[j] ≤ pivot then
6:             i ← i+1;
7:             swap A[i] with A[j];
8:         end if
9:     end for
10:    swap A[i+1] with A[hi];
11:    return i+1;
12: end function
```

```
function QUICKSORT(A, lo, hi)
    if lo < hi then
        p ← partition(A, lo, hi);
        quicksort(A, lo, p - 1);
        quicksort(A, p + 1, hi);
    end if
end function
```

Graph

Adjacency Matrix N^2

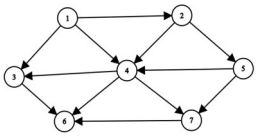
For each edge (u, v) with weight of w, set $a[u][v] = w$. Use 0 or $\pm\infty$ for non-existent edges.

Adjacency List $N + M$

Keep a list of all adjacent vertices for each vertex.

Topological Sort

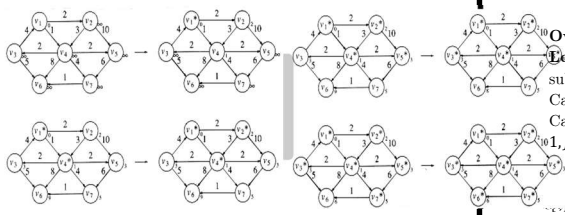
```
for i=1 to n do
  v=a vertex with indegree zero
  if (v does not exist) then
    report error
  top_num[v]=counter
  for each w adjacent to v do
    indegree[w]--
```



Indegree Before Dequeue #							
Vertex	1	2	3	4	5	6	7
v1	0	0	0	0	0	0	0
v2	1	0	0	0	0	0	0
v3	2	1	1	1	0	0	0
v4	3	2	1	0	0	0	0
v5	1	1	0	0	0	0	0
v6	3	3	3	3	2	1	0
v7	2	2	2	1	0	0	0
enqueue	v1	v2	v5	v4	v3	v7	v6
dequeue	v1	v2	v5	v4	v3	v7	v6

Dijkstra

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```



Prim's Algorithm $O(E \log V)$

```
PRIM-MST(V, E, w, s)
1  foreach v ∈ V
2      do key[v] ← ∞
3         P[v] ← NIL
4  key[s] ← 0
5  Q ← V
6  while Q ≠ ∅
7      do u ← EXTRACT-MIN(Q)
8         foreach v ∈ Adj[u]
9             do if v ∈ Q and w(u, v) < key[v]
10                then key[v] ← w(u, v)
11                     P[v] ← u
12 return P
```

Kruskal's Algorithm $O(E \log V)$

Sort weights of edges in descending order. If u and v are in the same set, the edge is rejected, because since they are already connected, adding (u, v) would form a cycle; Otherwise, the edge is accepted, and a union is performed on the two sets containing u and v.

Depth-First Search $O(n + m)$

Starting at some vertex, v, we process v and then recursively traverse all vertices adjacent to v. when we visit a vertex v, we mark it visited.

Breadth-First Search $O(n + m)$

Process vertices in layers: the vertices closest to the start are evaluated first, and the most distant vertices are evaluated last.

Randomized Algorithms:

Quick Sort: $T(n) = O(n) + \frac{1}{n}\sum_{i=1}^{n}(T(k-1) + T(n-k))$, $T(n) = O(n \log n)$.

Karger's Algorithm:

A cut in G is a pair $(S, V - S)$ of two sets S and $V - S$ that split the nodes into two groups. The size of a cut is the number of edges with one endpoint in S and one in $V - S$. A min cut is a cut in G with the least total cost.

Claim: the size of a min cut is at most the minimum degree in the graph. *Proof.* If v has the minimum degree, then the $cut(\{v\}, V - \{v\})$ has size equal to $\deg(v)$.

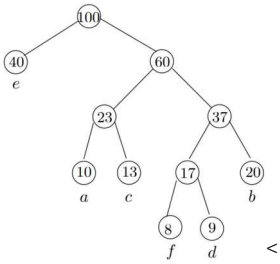
Choose any random edge (u, v) from the graph G to be contracted. Merge the vertices to form a supernode and connect the edges of the other adjacent nodes of the vertices to the supernode formed. Remove the self nodes, if any. Repeat the process until there is only two nodes left in the contracted graph. The edges connecting these two nodes are the min cut edges. Karger's algorithm produces cut C iff it never contracts an edge crossing C.

Advantages: simple, efficient, low complexity

Disadvantages: quality, reliability

Counting Money: At each step from descending amount of value, take the largest possible bill or coin that does not overshoot.

Huffman Encoding: $n = |\sum|, S$ be a set of nodes, with $\sigma \in \sum$ under $\text{freq}(\sigma)$. Repeat until S has one node left: (1) removal from S two nodes u_1, u_2 with smallest frequencies; (2) creation of a node v having u_1, u_2 as children, set $\text{freq}(v)$ to be sum of $\text{freq}(u_1) + \text{freq}(u_2)$; (3) insert v to S.



Coin Changing $< \cdots < d_k$, compute the minimum number of coins that changes n.

Optimal Structure:

$$MinNumCoins(n) = \min \begin{cases} MinNumCoins(n - d_1) + 1 \\ MinNumCoins(n - d_2) + 1 \\ \cdots \\ MinNumCoins(n - d_k) + 1 \end{cases}$$

Overlapping Subproblems: Optimize by **table look-ups**

Longest Common Subsequence: Longest subsequence of characters in both X and Y.

Case 1: if $X[i] = Y[j]$, then $C[i, j] = 1 + C[i-1, j-1]$;

Case 2: If $X[i]! = Y[j]$, then $C[i, j] = \max\{C[i-1, j], C[i, j-1]\}$;

Formulation:

$$C[i,j] = \min \begin{cases} 0, \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1, \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{C[i-1, j], C[i, j-1]\}, \text{if } X[i] \neq Y[j] \text{ and i, j} > 0 \end{cases}$$

Edit Distance: The minimum number of edits (insertion, removal and replacement) needed to convert X into Y.

Case 1: if $X[i] = Y[j]$, then $D[i, j] = D[i-1, j-1]$.

Case 2: if $X[i] \neq Y[j]$, then $D[i, j] = \min\{D[i-1, j], D[i, j-1], D[i-1, j-1]\} + 1$.

Longest Increasing Path in Matrix

$P[i, j] = \max\{P[x, y] \text{ where } M[x, y] \text{ is adjacent to } M[i, j] \text{ and } M[x, y] < M[i, j]\} + 1$.

algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	E + V	E + V	V
Dijkstra (binary heap)	no negative weights	E log V	E log V	V
Bellman-Ford	no negative cycles	E V	E V	V
Bellman-Ford (queue-based)	no negative cycles	E + V	E V	V

```
Initialize d[s]=0 and d[v]=∞ for all other vertices.
for i=1 to |V|-1 do
  for each edge (u, v) ∈ E do
    if d[v] > d[u] + w(u, v) then
      d[v] ← d[u] + w(u, v)
for each edge (u, v) ∈ E do
  if d[v] > d[u] + w(u, v) then
    report that a negative-weight cycle exists
```

Ford-Fulkerson Algorithm

- Idea: increase flow along augmenting paths
- Find an undirected path from s to t such that:

Can increase flow on forward edges (not full).

Can decrease flow on backward edge (not empty).

Net flow across a cut (A, B) is the sum of the flows on its edges from A to B minus the sum of the flows on its edges from B to A.

Maxflow-mincut theorem: Value of the maxflow = capacity of mincut. Fattest path can avoid bad augmentation.

Maximum Cut Problem: Find a partition of V into two non-empty sets that maximizes the number of crossing edges.

1. Let (A, B) be an arbitrary cut of G;
2. While there is a vertex v with $d_v(A, B) > c_v(A, B)$:
   - Move v to other side of the cut
   - increases number of crossing edges by $d_v(A, B) - c_v(A, B)$
3. Return final cut (A, B)

Convex Hull

Algorithm: Graham Scan (Upper Hull)

```
Sort the points according to increasing order of their
x-coordinates, denoted (p_1, p_2, ..., p_n).
Push p_1 and then p_2 onto S.
For i ← 3 to n do:
    While |S| ≥ 2 and orient(p_i, S[t], S[t − 1]) ≤ 0 do:
        pop S.
    Push p_i onto S.
```

Invariant: Let $U(i)$ be the upper hull of $\{p_1, \ldots, p_i\}$, in step i, the stack contains (from bottom to top order) $U(i)$.

Time Complexity: $O(n \log n)$

A **forward index** is a map from documents to terms (and positions). These are used when you search within a document.

An **inverted index** is a map from terms to documents (and positions). These are used when you want to find a term in any document.

and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2
fishkeepers	2:1			
found	1:1			
fresh	2:1			
freshwater	1:1	4:1		
from	4:1			
generally	4:1			
in	1:1	4:1		
include	1:1			
including	1:1			
iridescence	4:1			
marine	2:1			
often	2:1	3:1		