CS2040S
Data Structures and Algorithms

AY2021/22 Semester 2

Notes by Jonathan Tay

Last updated on April 17, 2022

Contents

1 Searching

1.1 Binary Search

Find the index of an element in a sorted array if it exists.

```
int search(A, key, n):
    lo = 0
    hi = n − 1
    while lo < hi:
        mid = lo + (hi − lo) / 2
        if key ≤ A[mid]: hi = mid
        else: lo = mid + 1
    return A[lo] == key ? lo : −1
```

`A[lo]` $\leq$ `key` $\leq$ `A[hi]` is a **loop invariant**, and on iteration k, `hi` $-$ `lo` $\leq \frac{n}{2^k}$.

This has a time complexity of $O(\log n)$.

1.2 Peak Finding

Find the index of a local maximum in an unsorted array.

A **peak** is a local maximum in `A` such that `A[i]` $\geq$ `A[i-1]` and `A[i]` $\geq$ `A[i+1]`. We also assume that `A[-1]` $=$ `A[n]` $=$ `-MAX-INT`.

```
int find_peak(A, key, n):
    if A[n/2 + 1] > A[n/2]:
        return find_peak(A[n/2 + 1 : n], n/2)
    else if A[n/2 − 1] > A[n/2]:
        return find_peak(A[1 : n/2 − 1], n/2)
    else A[n/2] is a peak:
        return n/2
```

There will always exist a peak in `A[lo:hi]`, and every peak in `A[lo:hi]` is a peak in `A[1:n]`.

The recurrence relation is $T(n) = T(\frac{n}{2}) + \theta(1)$ which gives a time complexity of $O(\log n)$.

2 Sorting

2.1 Bubble Sort

Repeatedly swaps adjacent elements that are out of order until there are no swaps in an iteration or after n iterations.

```
bubble_sort(A, n):
    for i in range(n − 1): // or no swaps
        for j in range(0, n − i − 1):
            if A[j] > A[j + 1]:
                swap(A[j], A[j + 1])
```

At the end of iteration j, the largest j items are correctly sorted in the final j positions of the array.

In the best case of a sorted array, this takes $O(n)$, and $O(n^2)$ otherwise. BubbleSort is stable.

2.2 Selection Sort

Maintains a sorted prefix, and repeatedly finds the smallest element in the unsorted remainder and swaps it with the first element in the remainder.

```
selection_sort(A, n):
    repeat i in range(n − 1):
        j = argmin(A[i : n])
        swap(A[i], A[j])
```

At the end of iteration i, the smallest i items are correctly sorted in the final i positions of the array. In all cases, this takes $O(n^2)$ comparisons, and is unstable.

2.3 Insertion Sort

Maintains a sorted prefix, and repeatedly inserts unsorted elements into the correct place in the sorted prefix.

```
insertion_sort(A, n):
    repeat i in range(1, n):
        key = A[i]
        j = i − 1
        while j > 0 and A[j] > key: // insert
            A[j + 1] = A[j]
            j = j − 1
        A[j + 1] = key
```

At the end of iteration i, the first i items are in sorted order. In the best case of a sorted array, this takes $O(n)$, in the average case of a random permutation $\theta(n^2)$, and in the worst case of an inverse sorted array, $O(n^2)$.

InsertionSort is stable as long as we do not swap identical elements.

2.4 Merge Sort

Using divide-and-conquer, we split the array into halves, recursively sorting both halves and then merging the two.

`merge` takes n iterations to move all elements to a final array, and each iteration takes $O(1)$ time to compare and copy elements. This gives a running time of $O(n) = cn$.

We get the recurrence for `merge-sort`:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

Therefore, we get an overall time complexity of $O(n \log n)$. `merge` also requires allocating space for the final array, which gives a total space complexity of $O(n \log n)$.

Merge sort is not in-place, but is stable if `merge` is implemented properly.

2.5 Quick Sort

Using divide-and-conquer, we partition the array around a pivot, and then recursively sort the two subarrays.

If there are no duplicates, we partition the array A into a new array B by using two pointers `lo` and `hi` starting from both ends of the array.

By iterating over A, if the element at $A[i]$ is less than the pivot, we copy it to `B[lo++]`, or `B[hi--]` otherwise.

Two invariants will hold:
$\forall i <$ `lo`, `B[i]` $<$ `pivot` and $\forall j >$ `hi`, `B[j]` $>$ `pivot`.

This implementation of `partition` takes $O(n)$ time, but is not in-place.

2.5.1 In-place Partitioning

```
partition(A, n):
    swap(pivot, A[0])
    lo = 1
    hi = n − 1
    while lo < hi:
        while A[lo] < pivot and lo < hi: lo++
        while A[hi] > pivot and lo < hi: hi−−
        if (lo < hi) swap(A[lo], A[hi])
    swap(A[lo], A[0])
    return lo
```

Two invariants will hold at the end of every iteration:
$\forall i \geq$ `hi`, `A[i]` $>$ `pivot` and $\forall\ 1 \leq j <$ `lo`, `A[j]` $<$ `pivot`.

2.5.2 Three-way Partitioning

Partition an array with duplicates in-place using four regions.

Three invariants will hold at the end of every iteration:
$\forall\ i <$ `lo`, `A[i]` $<$ `pivot`
$\forall$ `lo` $\leq j <$ `mid`, `A[j]` $=$ `pivot`
$\forall\ k >$ `hi`, `A[i]` $>$ `pivot`

Regardless of which `partition` algorithm is used, `quick-sort` is not stable.

It has a recurrence of `T(n) = 2T(n/2) + O(n)`. In the worst case (e.g. A[0] pivot), this is $O(n^2)$. However, the expected running time with high probability is $O(n \log n)$.

3 Hashing

3.1 Symbol Table

An abstract data type for key-value pairs with O(1) insertion, search, and deletion. Successor and predeccessor operations are unsupported.

Implementing a symbol table with an array (direct access table) indexed by integers is impractical for non-integer keys, and uses too much space.

Implementing a symbol with an AVL tree is slow, with lookup operations taking $O(\log n)$.

Furthermore, a dictionary with an AVL tree is a better choice for sorting in $O(n \log n)$ rather than $O(n^2)$ with a symbol table.

3.1.1 Hash Functions

Given a universe U of possible keys, we need to map them to a smaller number n of actual keys in m buckets.

A hash function $h : U \mapsto \{1, 2, \ldots, m\}$ is used to store key k in bucket $h(k)$. The time complexity depends on h and bucket access, but is usually assumed to be $O(1)$.

Two distinct keys k_1 and k_2 **collide** if $h(k_1) = h(k_2)$. It is impossible to choose a hash function that does not collide, due to the **pigeonhole principle**.

3.1.2 Chaining

Colliding keys are placed in the same bucket via a linked list.

For a table of size m and a linked list size of n, the total space used is $O(m + n)$.

For a hash function h with cost $\text{cost}(h)$, `insert` takes $O(1 + \text{cost}(h))$, while `search` takes $O(n + \text{cost}(h))$.

By the **simple uniform hashing assumption** where keys are equally likely to map to every bucket and are mapped independently, we can avoid the case where all keys hash to the same bucket.

Linearity of expectation states that $E(A+B) = E(A)+E(B)$. So, the expected number of items per bucket is $\frac{n}{m}$.

Therefore `search` is $1 + \frac{n}{m} = O(1)$ in expectation, but $O(n)$ in the worst case, and `insert` is still $O(1)$.

If we insert n items, the expected maximum cost is $O(\log n)$ and actually $\theta(\frac{\log n}{\log \log n})$.

3.1.3 Open Addressing

On collision, probe a sequence of buckets until we find an empty one.

We re-define h as $h(key, i) : U \mapsto \{1, 2, \ldots, m\}$.

However, we cannot simply `null` items in `delete`, and instead use a **tombstone value** which allows `insert` to overwrite it.

A good $h(k, i)$ must enumerate all possible values buckets, such that for every bucket j there is some i such that $h(k, i) = j$, meaning it is some permutation of $\{1, 2, \ldots, m\}$. Otherwise, it would wrongly declare a table to be full even when there is stil space.

This is true for **linear probing**. However, LP does not obey the UHA (not SUHA!) where every key is equally

likely to map to every permutation, independent of other keys, due to **clustering**.

When the table is $\frac{1}{4}$ full, clusters of size $\theta(\log n)$ form which ruins $O(1)$ performance. For a load $\alpha = \frac{n}{m}$, assuming uniform hashing, each operation is expected to take $\leq \frac{1}{1-\alpha}$.

With **double hashing**, $h(k,i) = f(k) + i \cdot g(k) \bmod m$, if $g(k)$ has no common factors with m other than 1, then $h(k,i)$ hits all buckets.

Open addressing saves space, rarely allocates, and has better cache performance, but is more sensitive to hash function choice and load.

3.2 Table Resizing

If a hash table is too small, there are too many collisions, and if it is too large, we waste space.

Upon resizing from sizes m_1 to m_2, we need to choose a new hash function h re-compute all n hashes, and copy them over. This takes $O(m_1 + m_2 + n)$.

If we increment table size by 1, this takes $O(n)$, meaning `insert` takes $O(n)$.

If we double table sizes, each resize costs $O(n)$, but the average cost of insertion is now $O(1)$. Squaring works as well, but space usage is inefficient.

3.2.1 Amortization

An operation has amortized cost $T(n)$ if $\forall k \in \mathbb{Z}^+$, the cost of k operations $\leq k \cdot T(n)$.

Therefore, resize tables only if $n = m$, then $m = 2m$ and if $n < \frac{m}{4}$, then $m = \frac{m}{2}$.

In another example, incrementing a binary counter takes amortized $O(\log n)$ time.

4 Graphs

4.1 Topological Sort

A **topological sort** imposes a total ordering on all vertices, such that edges only point foward.

Only **directed acyclic graphs** have a topological order. Topological order is **not unique**.

4.1.1 Post-order Depth-first Search

Vertices are prepended to the list only after all of their children have been visited.

4.1.2 Kahn's Algorithm

Removes vertices without dependencies from the graph.

Each vertex tracks their in-degree, and is enqueued when it reaches zero. When dequeued, the vertex is removed and its children's in-degrees are decremented.

Both algorithms have a time complexity of $\mathbf{O(V + E)}$ and a space complexity of $\mathbf{O(V)}$.

4.2 Connected Components

Vertices v and w are in the same **connected component** iff there is a path from v to w.

Iterate over each vertex, and if it is unvisited, perform either BFS or DFS to flood-fill. This has a time complexity of $\mathbf{O(V + E)}$.

In the case of a directed graph, v and w are in the same **strongly connected component** iff there is a path from v to w and also from w to v.

4.2.1 Kosaraju's Algorithm

On the first pass, maintain a set of visited vertices and a stack of finished vertices. Explore vertices with DFS, pushing onto the stack when all of their children have been visited.

On the second pass, reverse the graph. Pop a vertex from the stack and flood-fill with DFS to get the SCC. Pop vertices from the stack until we get an unvisited vertex, then repeat the previous step.

This has a time complexity of $\mathbf{O(V + E)}$, and a space complexity of $\mathbf{O(V)}$.

4.3 Cycle Detection

A **cycle** is a path that visits a vertex more than once.

In a directed graph, keep parent pointers and sets for unvisited, visiting, and visited vertices. Flood fill vertices with DFS, and if we find a vertex in the visiting set, then there is a cycle.

5 Shortest Paths

5.1 Bellman-Ford Algorithm

Computes the shortest path for graphs with negative weights. Can be used to detect negative weight cycles.

Let $\delta(u,v)$ be the weight of the shortest path from u to v. By the triangle inequality, $\delta(s,v) \leq \delta(s,u) + \delta(u,v)$.

For each vertex v, we maintain an estimate $d(v)$ of the distance from a source s, and a parent pointer $\pi(v)$. All estimates are initialized to ∞, and $d(s) = 0$.

The algorithm performs $|V| - 1$ iterations over the graph, during which each edge is relaxed:

```
relax(u, v):
    if d(v) > d(u) + weight(u, v):
        d(v) = d(u) + weight(u, v)
        π(v) = u
```

Relaxation mainatains the invariant $d(v) \geq \delta(s, v)\ \forall v \in V$, i.e., we never under-estimate.

If an entire iteration does not change any estimates, then we can terminate early. After n iterations, the n-hop estimate on the shortest path (not every path!) is correct, i.e., $d(v_n) = \delta(s, v_n)$

Also, if P is the shortest path from s to v, and if P goes through u, then P is also the shortest path from s to u.

If we run a $|V|$-th iteration and the estimates still change, then we have found a negative weight cycle.

This algorithm has a time complexity of **O(EV)**.

5.2 Dijkstra's Algorithm

Computes the shortest path for graphs with only non-negative weights, including cyclic graphs.

For each vertex v, we maintain the distance $d(v)$ from a source s, and a parent pointer $\pi(v)$.

We also maintain a priority queue Q of vertices ordered by their distance s, i.e. the d-values. $d(s)$ is initialized to 0.

```
dijkstra():
    initialize() // O(|V| log |V|)
    while Q is not empty: // O(|V|)
        u = extract_min(Q) // O(log |V|)
        for each edge (u, v): // O(|E|)
            relax(u, v) // O(log |V|)
```

The relax operation is the same as in the Bellman-Ford algorithm, except that `decrease-key` is used to update the d-values.

The time complexity of this algorithm with an AVL tree is **O(E log V)**, but **O(E + V log V)** with a Fibonacci heap. The complexities are simplified as $|E| = \theta(|V|^2)$.

Graphs can be reweighted by multiplying the edge weights by a constant factor, but addition by a constant factor will break Dijkstra's algorithm.

We can also find the longest path by negating the edge weights, as long as the graph is acyclic.

If we wanted to find the path with shortest weight products rather than sum, it is easier to take the logarithm of the weights, rather than modify the relax operation.

5.3 Relaxation in Topological Order

Computes the shortest path for directed acyclic graphs.

Since the path is relaxed in order, the distance is correct after relaxation.

This has a time complexity of **O(V + E)**.

6 Union-Find

An abstract data type for disjoint sets and connected components.

`union` combines two objects, and `find` checks whether there is a path between two objects.

A naive implementation associates each object with an identifier. `find` checks whether their identifiers are the same in $O(1)$. `union` has to iterate over all identifiers and combine them in $O(n)$.

A better implementation associates each object with a parent. `find` now checks whether they have the same highest common ancestor (HCA) in $O(n)$. `union` has to find both their HCAs to link together in $O(n)$.

An optimization using **weighted union** associates a tree size with each object, so `union` links the smaller tree to the larger one in $O(\log n)$. Since the heights of the trees of size n is at most $\log(n)$, `find` takes $O(\log n)$ time.

The final optimization adds **path compression**. After each time the root is found, every node on the path is linked to the root. Both `union` and `find` take $O(\alpha(m, n))$ time each for m operations.

With path compression but without weighted union, both `union` and `find` take $O(\log n)$ time.

7 Minimum Spanning Trees

An acyclic subset of edges connecting all vertices, with minimum total weight.

MSTs cannot be used to find shortest paths.

A **cut** of a graph partitions the vertices into two disjoint subsets. An edge crosses a cut if it has one vertex in each of these two subsets.

MSTs have four properties:

1. MSTs are acyclic.
2. If an MST is cut, both subtrees are MSTs.
3. For every cycle, the maximum weight edge is not in the MST, but the minimum weight edge may or may not be.
4. For every vertex (and cut), the minimum weight edge is in the MST.

7.1 Prim's Algorithm

We maintain a set S for the vertices in the MST, a priority queue Q of vertices ordered by their distance d from the growing MST, as well as a parent pointer $\pi(v)$ for each vertex v.

We initialize $d(v)$ to ∞ for every vertex v, and start at some arbitrary vertex s such that $S = \{s\}$ and $d(s) = 0$.

Then, we repeatedly dequeue the vertex u with the smallest distance d, removing it from Q and adding it to S.

For every edge (u, v) in the graph, we relax it as so:

```
relax(u, v):
    w = weight(u, v)
    if d(v) > w:
        d(v) = Q.decrease_key(u, w) // O(log |V|)
        π(v) = u
```

This works because each added edge is the minimum across some cut, therefore each edge is in the MST.

Every vertex is only added and removed once from Q, taking $O(|V| \log |V|)$ in total.

Each edge is only relaxed once (calling `decrease-key`), taking $O(|E| \log |V|)$ in total.

Therefore, this has a time complexity of **O(E log V)**.

7.2 Kruskal's Algorithm

We will use union-find to keep track of the vertices in the MST.

We sort the edges by weight in ascending order and iterating through them. If the edge connects two vertices in the same connected component (with `find`), we ignore it. Otherwise, we add it to the MST and call `union`.

This works because the ignored edge would have been the heaviest edge on the cycle, which violates the MST property. Also, all other lighter edges have already been considered.

This has a time complexity of **O(E log V)** since $\log |E| = O(\log |V|^2) = O(2 \log |V|) = O(\log |V|)$.

With integer weights, $O(n)$ counting sort may be used such that the time complexity is improved to $O(|E|)$. With a $O(n^2)$ sort, the time complexity is $O(|E|^2)$.

7.3 Variants

If all edge weights are equal, any spanning tree found by DFS or BFS will be an MST since both the spanning tree an MSTs have exactly $V - 1$ edges.

Reweighting a graph has no effect on the MST.

A directed MST is a much harder problem, but in the special case with one root, an MST is the tree of all edges with the minimum weight, which takes $O(E)$ time.

A maximum spanning tree can be found by negating each edge weight, or by running Kruskal's in reverse.

8 Dynamic Programming

Construct an optimal solution to a problem from optimal solutions to smaller subproblems.

8.1 SRTBOT

1. **S**ubproblem definition.
2. **R**elate subproblem solutions recursively.
3. **T**opological order on subproblems (DAG).
4. **B**ase case(s) for subproblems.
5. **O**riginal problem, solved via subproblems.
6. **T**ime complexity.

Use **memoization** to re-use solutions to subproblems with a dictionary mapping subproblems to their solutions.

Recursive function computes the solution if and only if it is not already stored.

If the the input is some arbitrary sequence x of length n, good subproblems to consider are:

1. the $\theta(n)$ **prefixes** $x[:i]$,
2. the $\theta(n)$ **suffixes** $x[i:]$, and
3. the $\theta(n^2)$ **substrings** $x[i:j]$.

The use of subsequences as subproblems, where some elements can be omitted, is undesirable since there would be an exponential number of them.

8.2 Longest Common Subsequence (LCS)

Given two subsequences A and B, find the longest common subsequence L of A and B.

When there are **multiple inputs**, subproblems can be generated by taking the cross product of the subproblem spaces.

S: $L(i,\ j) = \text{LCS}(A[i:],\ B[j:])\ \forall i \in [0, |A|]\ \forall j \in [0, |B|]$

R: $L(i,\ j) = \begin{cases} 1 + L(i+1,\ j+1) & \text{if} A[i] = B[j] \\ \max\{L(i+1,\ j),\ L(i,\ j+1)\} & \text{otherwise} \end{cases}$

T: for $i = |A| \ldots 0$ for $j = |B| \ldots 0$

B: $L(|A|,\ j) = L(i,\ |B|) = 0$

O: $L(0, 0)$

T: $\theta(|A| \cdot |B|)$ subproblems $\cdot\ \theta(1)$ each $= \theta(|A| \cdot |B|)$

We can use **parent pointers** to reconstruct the solution.

8.3 Longest Increasing Subsequence (LIS)

Given a sequence A, find the longest strictly increasing subsequence L of A. Generalizes to non-strict as well.

S: $L(i) = \text{LIS}(A[i:])$ that starts with $A[i]$ (**constraint**).

R: $L(i) = 1 + \max\{L(j) \mid i < j < |A|,\ A[i] < A[j]\} \cup \{0\}$

T: for $i = |A| \ldots 0$

B: $L(|A|) = 0$

O: $\max\{L(i) \mid 0 \leq i \leq |A|\}$

T: $\theta(|A|)$ subproblems $\cdot\ \theta(|A|)$ choices $+\ \theta(|A|) = \theta(|A|^2)$

8.4 All-Pairs Shortest Path (APSP)

Find the shortest path between all pairs of vertices in a graph, using the Floyd-Warshall algorithm.

First, number every vertex from 1 to $|V|$.

S: $d(u, v, k)$ = weight of the shortest path from u to v using only the vertices $\in \{u, v\} \cup \{1, 2, \ldots, k\}$ $\forall u, v \in V\ \forall k \in [0,\ |V|]$

R: $d(u, v, k) = \min \begin{cases} d(u, v, k-1) & k \in SP \\ d(u, k, k-1) + d(k, v, k-1) & k \notin SP \end{cases}$

T: increasing k for $k = 0 \ldots |V|$ for $u \in V$ for $v \in V$

B: $d(u, v, 0) = \begin{cases} 0 & \text{if } u = v \\ w(u, v) & \text{if } (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$

O: $d(u, v, |V|)$ assuming no negative weight cycles

T: $\theta(|V|^3)$ subproblems $\cdot\ \theta(1)$ each $= \theta(|V|^3)$

8.5 Partitioning (Rod Cutting)

Given a rod of length $L \in \mathbb{Z}^+$ and a value $v(l)$ for a rod of length $l \in [1,\ L]$, what is the max-value partition for the rod?

S: $X(l) =$ max-value partition of length $l \forall l \in [0, L]$

R: $X(l) = \max\{v(p) + X(l - p) \mid 1 \leq p \leq l\}$

T: increasing l for $l = 0, 1 \ldots, L$

B: $X(0) = 0$

O: $X(L)$

T: $\theta(L)$ subproblems $\cdot\ \theta(|L|)$ choices $= \theta(|L|^2)$

This problem runs in polynomial time, but the next problem will run in **psuedo-polynomial** time.

8.6 Subset Sum

Given a multiset $A = \{a_0, a_1, \ldots, a_{n-1}\}$ with n positive integers, and a target sum T, does any subset $S \subseteq A$ sum to T?

This is an example of a **decision problem** where the answer is binary.

S: $X(i, t)$ = does any subset $S \subseteq A[i\ :]$ sum to T $\forall i \in [0, n]\ \forall t \in [0, T]$?

R: $X(i, t) = \text{OR} \begin{cases} X(i+1, t) & a_i \notin S \\ X(i+1, t - a_i) \text{ if } a_i \leq t & a_i \in S \end{cases}$

T: decreasing i for $i = n, n-1, \ldots 0$

B: $X(n, t) = \begin{cases} \text{true} & \text{if } t = 0 \\ \text{false} & \text{otherwise} \end{cases}$

O: $X(0, T)$

T: $\theta(nT)$