Data Structures and Algorithms Notes

paraphrased by Tyler Wright

*An important note, these notes are absolutely **NOT** guaranteed to be correct, representative of the course, or rigorous. Any result of this is not the author's fault.*

0 Notation

We commonly deal with the following concepts in Data Structures and Algorithms which I will abbreviate as follows for brevity:

Term	Notation
The vertex set of a graph G	$V(G)$
The edge set of a graph G	$E(G)$

Contents

1 Data Structures

1.1 Stacks

A stack is a list of variables. It supports three operations:

Name	Description	Worst case runtime
`create()`	Creates a new stack	$O(1)$
`push(x)`	Adds `x` to the end of the stack	$O(1)$
`pop()`	Removes and returns the last element of the stack	$O(1)$

1.2 Queues

A queue is a list of variables. It supports three operations:

Name	Description	Worst case runtime
`create()`	Creates a new queue	$O(1)$
`add(x)`	Adds `x` to the end of the queue	$O(1)$
`serve()`	Removes and returns the first element of the queue	$O(1)$

1.3 Linked List

A linked list is a list of variables represented by nodes which point to the next and previous element in the list (or `null` if one does not exist). Each node has a unique identifier. It supports four operations:

Name	Description	Worst case runtime
`create()`	Creates a new linked list	$O(1)$
`insert(x, i)`	Inserts `x` after node `i`	$O(1)$
`delete(i)`	Removes node `i`	$O(1)$
`lookup(i)`	Returns node `i`	$O(1)$

1.4 Arrays

An array is a list of variables of fixed length. It supports three operations:

Name	Description	Worst case runtime
`create(n)`	Creates a new array of size `n`	$O(1)$
`update(x, i)`	Overwrites the data at position `i` with `x`	$O(1)$
`lookup(i)`	Returns the value at `i`	$O(1)$

1.5 Hash Tables

A hash table is an array of linked lists storing key-value pairs. We use a **hash function** to map data to a linked list. As we are using linked lists, if multiple keys map to the same index, we can just add them to the list - and when looking up data, we can find the right list with the hash function and then match our key.

It supports four operations:

Name	Description	Average runtime
`create(n)`	Creates a `n` sized array of linked lists and chooses a hash function `h`	$O(1)$
`insert(k, v)`	Inserts the pair (`k, v`), if $\frac{n}{2}$ pairs are stored, we create a hash table of double the size and copy the contents into it	$O(1)$
`delete(k)`	Deletes the pair corresponding to the key `k`	$O(1)$
`lookup(k)`	Returns the pair corresponding to the key `k`	$O(1)$

1.5.1 Markov's Inequality

For $X \geq 0$ a random variable with mean μ, for all t in $\mathbb{R}_{\geq 0}$:

$$\mathbb{P}(X \geq t) \leq \frac{\mu}{t}.$$

So, if X is the expected time it takes for an algorithm to terminate, we can say how likely it is for an algorithm to terminate based on our prediction.

1.6 Binary Heaps

Binary heaps are rooted binary trees where each level is full except possibly the last (which is filled from left to right). The elements of the tree are ordered according to a **heap property**. These have the following properties:

- For a heap of size n, the height of the heap is $\log_2(n)$
- For an index i:
  - The parent has index $\left\lfloor \frac{i}{2} \right\rfloor$,
  - The left child has index $2i$,
  - The right child has index $2i + 1$.

1.7 Priority Queues

A priority queue functions as queue with a notion of priority. We associate keys to priority in this case. In this course, we use a binary heap with the heap property that each parent has a value less than or equal to its children. This supports the following:

Name	Description	Runtime
`insert(x, k)`	Inserts `x` with key `k`	$O(\log_2(n))$
`decreaseKey(x, d)`	Decreases the key of `x` to `d`	$O(\log_2(n))$
`extractMin()`	Removes and returns the `x` in the queue with the smallest key	$O(\log_2(n))$

1.8 Disjoint Set

This data structure stores a collection of disjoint sets of $\{1, 2, \ldots n\}$ for some n in $\mathbb{Z}_{>0}$. This supports the following:

Name	Description	Runtime
`makeSet(x)`	Creates a new set containing only `x` this fails if `x` is already in a set	$O(1)$
`union(x, y)`	Merges the sets containing `x` and `y`	$O(\log_2(n))$
`findSet(x)`	Finds the identifier of the set containing `x` (the identifier is an element of the set)	$O(\log_2(n))$

This is stored as an array size n where each cell is empty or points to the identifier of the set it was originally added to.

1.9 Dynamic Search Structures

This structure stores a set of elements, each with a unique key. This supports the following:

Name	Description	Runtime
`insert(x, k)`	Inserts `x` with key `k`	$O(\log_2(n))$
`find(k)`	Returns the element with unique key `k`	$O(\log_2(n))$
`delete(k)`	Deletes the element with unique key `k`	$O(\log_2(n))$
`predecessor(k)`	Returns the element with unique key `n` such that `n` $<$ `k`	$O(\log_2(n))$
`rangeFind(a, b)`	Returns the elements with unique key `k` such that `a` $\leq$ `k` $\leq$ `b`	$O(\log_2(n))$

2 Graphs

A graph G is a set system (V, E) where the elements of E have size 2. Some definitions and facts follow from the definition:

- The elements of V are **vertices**,
- The elements of E are called **edges**,
- The size of V is often called the **order** of G,
- G is a 2-uniform set with ground set V,
- u, v in V are adjacent if $\{u, v\}$ is in E.

2.0.1 Graph Isomorphisms

For two graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, we say that G_1 and G_2 are isomorphic ($G_1 \cong G_2$) if there exists a bijection $\phi : V_1 \to V_2$ such that for each pair of vertices u, v in V we have that:

$$\{u, v\} \in E_1 \iff \{\phi(u), \phi(v)\} \in E_2.$$

2.1 Neighbourhood and Degree

For a graph $G = (V, E)$ the neighbourhood of v in V is the set of all adjacent vertices (denoted by $N_G(v)$). The neighbourhood of a set S is simply the union of the neighbourhoods of the elements of S (minus the vertices in S). The degree of v is simply the size of $N_G(v)$ denoted by $\deg(v)$.

2.1.1 Minimum and Maximum Degree

For a graph $G = (V, E)$ we have that the following to represent minimum and maximum degree:

$$\delta(G) := \min\{\deg(v) : v \in V\}$$
$$\Delta(G) := \max\{\deg(v) : v \in V\}.$$

2.1.2 k-regular Graphs

For a graph $G = (V, E)$, we have that G is k-regular for some k in $\mathbb{Z}_{>0}$ if for all v in V, we have $\deg(v) = k$.

2.2 Subgraphs

A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$ such that for all e in E' we have that $e \subseteq V'$.

2.2.1 Induced Subgraphs

An induced subgraph generated of $G = (V, E)$ is a subgraph $G' = (V', E')$ where:

$$E' = \{\{u, v\} \in E \text{ such that } u, v \in V'\}.$$

Induced subgraphs are generated from a subset of the vertices of a graph by selecting all the edges that are subsets of our chosen vertex set.

2.3 Complements of Graphs

For a graph $G = (V, E)$, we have that $\bar{G} = (V, \bar{E})$ is the complement of G where $\bar{E} = \{\{u, v\} : u, v \in V\} \setminus E$.

2.4 Walks

A walk of length is a set of vertices connected by edges. Its length is the number of edges it traverses.

A walk is closed if its first and last vertex are identical.

2.4.1 Types of Walks

Name	Closed?	Repeats vertices?	Repeats edges?
Walk	Not necessarily	Can	Can
Trail	Not necessarily	Can	Cannot
Paths	Not necessarily	Cannot	Cannot
Circuit	Yes	Can	Cannot
Cycles	Yes	Cannot	Cannot

2.5 Connected Graphs

A graph is connected if there exists a path between any two vertices in the graph.

2.5.1 Connected Components

A component of a graph G is a maximally connected subgraph of G.

2.6 Euler Circuits and Trails

An Euler trail in a graph $G = (V, E)$ is a trail in G that traverses every edge exactly once. An Euler circuit is a closed Euler trail.

2.6.1 Conditions for Euler Circuits and Trails

An Euler circuit in a connected graph $G = (V, E)$ exists if and only if each vertex in V has even degree. We can see from this that an Euler trail exists if and only if each vertex in V has even degree except exactly two vertices.

2.7 Hamiltonian Cycles and Paths

A Hamiltonian path is a path that visits each vertex exactly once. A closed Hamiltonian path is a Hamiltonian cycle.

2.7.1 Dirac's Theorem

For a graph $G = (V, E)$ where $n = |V| \geq 3$:

$$\delta(G) \geq \frac{n}{2} \Rightarrow G \text{ is Hamiltonian.}$$

3 Digraphs

A digraph (or directed graph) is a graph where each of the edges has a direction. This direction means the edge can only be traversed in a single direction.

3.1 Neighbourhoods in Digraphs

For a digraph $G = (V, E)$, we consider the edges entering and leaving a vertex (or set of vertices). Take v in V:

The in-neighbourhood of v is the set of edges that enter v denoted by $N^-(v)$.

The out-neighbourhood of v is the set of edges that leave v denoted by $N^+(v)$.

3.1.1 Degree in Digraphs

We can consider in-degree and out-degree:

$$\begin{aligned} \text{in-degree of } v = \deg^-(v) &= |N^-(v)| \\ \text{out-degree of } v = \deg^+(v) &= |N^+(v)|. \end{aligned}$$

3.2 The Directed Handshake Lemma

For a digraph $G = (V, E)$, we have that:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|.$$

3.3 Connectivity in Digraphs

3.3.1 Strong Connectivity

A digraph $G = (V, E)$ is strongly connected if for each u, v in V, there exists a path from u to v and from v to u.

3.3.2 Weak Connectivity

A digraph $G = (V, E)$ is weakly connected if for each u, v in V, there exists a path from u to v or from v to u.

3.3.3 Connected Components in Digraphs

A strong component of a digraph is the maximal strongly connected induced subgraph.

A weak component of a digraph is the maximal weakly connected induced subgraph.

3.4 Euler Circuits and Trails on Digraphs

For a digraph $G = (V, E)$, G has an Euler circuit if and only if G is strongly connected and every vertex has equal in and out degree. We can see from this that an Euler trail exist if and only if G is strongly connected and for some x, y in V:

- Every vertex in $V \setminus \{x, y\}$ has equal in and out degree,
- $\deg^-(x) = \deg^+(x) + 1$,
- $\deg^+(y) = \deg^-(y) + 1$.

4 Trees and Forests

A graph is a forest if it is acyclic. A tree is a connected forest.

4.1 Leaves

For a tree $T = (V, E)$, for a vertex v in V, v is a leaf if $\deg(v) = 1$.

4.1.1 Existence of Leaves

If $|V| \geq 2$, we have that T has at least two leaves.

4.2 Characterisation of Trees

We have that for a graph $G = (V, E)$, the following are equivalent:

- G is a tree,
- G is maximally acyclic (G is acyclic and the addition of any edge forms a cycle),
- G is minimally connected (G is connected and the removal of any edge disconnects it),
- G is connected and $|E| = |V| - 1$,
- G is acyclic and $|E| = |V| - 1$,
- Any two vertices in G are connected by a unique path.

4.3 Rooted Trees

For a tree $T = (V, E)$, we can root T with some r in V.

In the rooting process, we take each v in $V \setminus \{r\}$ and define P_v to be the path from r to v. We then direct the edges from r to v for each P_v.

For u, v in $V \setminus \{r\}$, we say that:

- u is an ancestor of v if u lies on P_v,
- u is the parent of v if u is in the in-neighbourhood of v,
- $L_0 = \{r\}$ and $L_n = \{v \in V : |P_v| = n\}$ are the levels of T,
- The depth of a tree is the greatest n where L_n is non-empty.

5 Bipartite Graphs

A graph $G = (V, E)$ is bipartite if V can be partitioned into two vertex sets V_1, V_2 such that each edge connects a vertex from V_1 to a vertex in V_2.

5.1 Matchings

For $G = (V, E)$ a bipartite graph with bipartition X, Y, a matching from X to Y is a set of edges $M \subseteq E$ such that $f : X \to Y$ defined by:

$$f(x) := y \qquad \text{where } \{x, y\} \in M,$$

is injective on M. The matching is called perfect if $M = X$.

5.2 Augmenting Paths

An augmenting path is a path in a bipartite graph G that alternates between edges in a matching M for G and edges in $E(G) \setminus M$ with the first and last vertices not being matched in the matching M.

5.3 Hall's Marriage Theorem

For $G = (V, E)$ a bipartite graph with bipartition X, Y:

$$G \text{ has a perfect matching from } X \text{ to } Y$$
$$\iff$$
$$\text{For all } S \subseteq X, |N(S)| \geq |S|.$$

6 Graphs Algorithms

6.1 Data Representations of Graphs

6.1.1 Adjacency Matrix

We have for a graph $G = (V, E)$, the adjacency matrix is a $|V|$ by $|V|$ matrix $A = (a_{ij})$ where:

$$a_{ij} = \begin{cases} 1 & \text{if there's an edge from vertex } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

6.1.2 Adjacency List

We can represent a graph also by an array of linked lists or hash tables where each element in the array represents a vertex v and the corresponding list represents the vertices in the neighbourhood of v.

6.1.3 Comparision of Representations

We can compare some basic properties of the representations:

	Matrix	Linked Lists	Hash Tables
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$
Finding an edge from u	$O(1)$	$O(\deg(u))$	$O(1)$
Finding the neighbourhood of u	$O(\|V\|)$	$O(\deg(u))$	$O(\deg(u))$

This raises the question, why don't we always use hash tables? Due to the probability of collisions in hash tables, we opt for the linked list as it's more reliable for large graphs (additionally, we are almost always are looking for a neighbourhood not a specific edge).

6.2 Search

Generally with a graph searching algorithm, we have a data structure which is (in the general case) a structure that can store a set of vertices and return them one by one in some undefined manner. Starting with a graph G with some `u` in $V(G)$, naming our data structure `data`, we perform the following:

```
data Search(u) {
  add u to data;
  while (data is non-empty) {
    take x from data;
    if (x is not marked) {
      mark x;
      for (each edge {x, y}) {
        put y in data;
      }
    }
  }
}
```

We have that this process always terminates, visits every vertex in connected graphs, and has time complexity $O(|E(G)|)$ (assuming the data operations are $O(1)$).

6.2.1 Breadth-first Search

If our data structure is a queue, we get breadth-first searching. This causes vertices to be marked in distance order from the starting point.

By tracking distances, we can find shortest paths from a given vertex using this searching style. In a graph $G = (V, E)$, this takes $O(|V| + |E|)$ time.

6.2.2 Depth-first Search

If our data structure is a stack, we get depth-first searching. This causes vertices to be marked the further they are from the starting vertex.

6.3 Djikstra's Algorithm

For a non-negatively weighted, directed graph G, we have that Djikstra's algorithm returns the shortest path to all vertices from some starting vertex. Taking `distances` to be a relation between each v in $V(G)$ and the length of the shortest path to v, the algorithm is structured as follows:

```
distances Djikstra(s) {
  let pq be our priority queue;
  let dist be our array of distances;
  for (each v) {
    dist[v] = infinity;
  }
  dist[s] = 0;
  for (each v) {
    pq->insert(v, dist(v));
  }
  while (pq is non-empty) {
    u = pq->extractMin();
    for (each edge (u, v)) {
      if (dist[v] > dist[u] + weight(u, v)) {
        dist[v] = dist[u] + weight(u, v);
        pq->decreaseKey(v, dist(v));
      }
    }
  }
  return dist;
}
```

We have that the time complexity of the algorithm varies across different implementations of priority queues:

	Runtime
Linked List	$O(\|V\|^2 + \|V\|\|E\|)$
Binary Heap	$O((\|V\| + \|E\|) \log (\|V\|))$
Fibonacci Heap	$O(\|E\| + \|V\| \log (\|V\|))$

and has $O(|V| + |E|)$ space complexity across all queues.

6.4 Bellman-Ford's Algorithm

For a weighted, directed graph G, we have that Bellman-Ford's algorithm returns the fastest path to all vertices from some starting vertex. Taking `distances` to be a relation between each v in $V(G)$ and the length of the shortest path to v, the algorithm is structured as follows:

```
distances BellmanFord(s) {
  let dist be our array of distances;
  for (each v) {
    dist[v] = infinity;
  }
  dist[s] = 0;
  do (|V| times) {
    for (each edge (u, v)) {
      // Relaxing (u, v)
      if (dist[v] > dist[u] + weight(u, v)) {
        dist[v] = dist[u] + weight(u, v);
      }
    }
  }
}
```

This runs in $O(|V||E|)$ time.

6.4.1 Negative Weight Cycles

Suppose our graph has a cycle which has a negative weight. This must mean that we can choose an arbitrarily negative path in the graph by traversing the cycle multiple times. This is why we require that there are no negative weight cycles.

We can run the algorithm on graphs with negatives cycles and simply run a final check at the end to see if we have a negative weight cycle. If we relax each edge again and decrease a path, there must be a negative cycle as we should already have all the shortest paths.

6.5 Johnson's Algorithm

For a weighted, directed graph we have that Johnson's algorithm returns the fastest path between all vertex pairs. It does this by re-weighting the graph and performing Djikstra's repeatedly.

6.5.1 Re-weighting based on Vertex Potential

For a graph $G = (V, E)$ with a weighting function $w : E \to \mathbb{Z}$, we define a potential function $h : V \to \mathbb{Z}$ to associate vertices with potentials. We define a re-weighting function $w' : E \to \mathbb{Z}$:

$$w'((u, v)) = w((u, v)) + h(u) - h(v).$$

We find the vertex potentials by adding a vertex s to the graph with an edge (s, v) for each v in V of weight zero forming a new graph G'. We then run Bellman-Ford on G' and define h as follows:

$$h(v) = \text{ the shortest path length from } s \text{ to } v \text{ in } G'.$$

Note that G' has the same number of negative weight cycles as G as all our edges are directed away from s.

6.5.2 The Algorithm

Starting with a graph $G = (V, E)$ with a weighting function $w : E \to \mathbb{Z}$, form $G' = (V \cup \{s\}, E \cup S)$ where:

$$S = \{(s, v) : v \in V\} \qquad \text{and} \qquad w(e) = 0 \text{ for all } e \text{ in } S.$$

Run Bellman-Ford on G' starting at s (detecting any negative weight cycles) to define our vertex potentials. Using the potentials, re-weight each edge as above in G. Run Djikstra's on every vertex in G to create our set of paired shortest paths. We can then convert our path weights back into their weights as inputted and retrieve the values if necessary. This takes $O(|V||E| \log_2(|V|))$ time.

6.6 Minimum Spanning Trees

In a graph $G = (V, E)$, a spanning tree $T = (V, E_T)$ is a tree with $E_T \subseteq E$.

A spanning tree on G is minimal if there is no other spanning tree on G with a lower weight.

6.6.1 Kruskal's Algorithm

For a weighted, connected, and undirected graph $G = (V, E)$, we have the following steps to the algorithm:

1. Generate a graph $T = (V, \emptyset)$
2. Generate a disjoint set data structure X of size $|V|$
3. For each v in V, perform `makeSet(v)` (where each vertex is defined by some unique integer in $\{1, \ldots, |V|\}$)
4. Sort the edges by weight
5. For each edge (u, v) (in increasing order):
    - If `findSet`$(u) \neq$ `findSet`(v), perform `union`(u, v) and add (u, v) to T.

Overall, this runs in $O(|E| \log_2(|V|))$ time.

7 Fast Fourier Transforms

7.1 Polynomials

A polynomial of degree n in $\mathbb{Z}_{\geq 0}$ is a function A:

$$A(x) = \sum_{i=0}^{n} a_i x^i,$$

where $a_1, \ldots, a_n$ are the coefficients of A. We can represent A by listing the coefficients $a_1, \ldots, a_n$, called the coefficient representation. Additionally, we say for $k > n$, k is a degree-bound of A.

7.1.1 Horner's Rule

We can evaluate polynomials quickly as follows. For a polynomial A degree n:

$$A(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_n))))).$$

This can be simplified in the following code:

```
int polynomial(coeffs, x) {
  output = 0;
  for (i = n; i >= 0; i--) {
    output = (output * x) + coeffs[i];
  }
  return output;
}
```

Taking $O(n)$ time.

7.1.2 Point Intersection with Polynomials

For a given set of points of size n, we have that there exists a unique polynomial with degree-bound n such that the polynomial intersects all the given points.

7.1.3 Point-Value Representation

We can represent a polynomial of degree n in $\mathbb{Z}_{\geq 0}$ by a set of points it intersects like so:

$$((x_1, y_1), \ldots, (x_{n+1}, y_{n+1})).$$

where for i, j in $[n+1]$ with $i \neq j$, $x_i \neq x_j$.

7.1.4 Polynomial Addition

For two polynomials A, B with degrees n, m in $\mathbb{Z}_{\geq 0}$ and $k = \max\{n, m\}$:

Under coefficient representation, taking:

$$A = (a_1, \ldots, a_n)$$
$$B = (b_1, \ldots, b_m),$$

we have that $(A + B) = (a_1 + b_1, \ldots, a_k + b_k)$ where we pad the polynomial of lesser degree with zeroes.

Under point-value representation, taking:

$$A = ((x_1, a_1), \ldots, (x_{k+1}, a_{k+1}))$$
$$B = ((x_1, b_1), \ldots, (x_{k+1}, b_{k+1})),$$

we have that $(A + B) = ((x_1, a_1 + b_1), \ldots, (x_{k+1}, a_{k+1} + b_{k+1}))$ where we pad the polynomial of lesser degree with zeroes.

7.1.5 Polynomial Multiplication

For two polynomials A, B with degrees n, m in $\mathbb{Z}_{\geq 0}$ and $k = 2 \cdot \max\{n, m\}$:

Under coefficient representation, taking:

$$A = (a_1, \ldots, a_n)$$
$$B = (b_1, \ldots, b_m),$$

we have that:

$$(A \cdot B)(x) = (c_1, \ldots, c_k) \qquad \text{where } c_i = \sum_{j=0}^{i} a_j b_{j-1}.$$

Taking $O(n^2)$ time.

We can do this with the point-value representation, taking:

$$A = ((x_1, a_1), \ldots, (x_{k+1}, a_{k+1}))$$
$$B = ((x_1, b_1), \ldots, (x_{k+1}, b_{k+1})),$$

We have that:

$$A \cdot B = \{(x_1, a_1 \cdot b_1), \ldots, (x_{k+1}, a_{k+1} \cdot b_{k+1})\}$$

Taking $O(n)$ time.

7.2 The Fast Fourier Transform

7.2.1 Roots of Unity

The idea is that we evaluate a polynomial to perform pointwise multiplication and then interpolate back into the coefficient representation. We need to evaluate a polynomial of degree n at $n+1$ points to convert it to point-value form. We use the $n+1$ roots of unity:

$$\omega_{n+1}^k = e^{\frac{2\pi i}{n+1}k},$$

for k in $\{0, 1, \ldots n\}$. So, we consider:

$$y_k = A(\omega_{n+1}^k),$$

The ordered vector $(y_0, \ldots, y_n)$ is the Discrete Fourier Transform (DFT) of the coefficient vector of A.

Cancellation Lemma: we have that $\omega_{dn}^{dk} = \omega_n^k$.

Halving Lemma: we have that if n is even, the set of all the squared roots of unity is just the set of roots of unity for $\frac{n}{2}$.

7.2.2 Method of the Fast Fourier Transform

For a polynomial A of even degree n, we define $A^{[0]}$ and $A^{[1]}$ as:

$$A^{[0]} = a_0 + a_2 x + \cdots + a_n x^{(n/2)}$$
$$A^{[1]} = a_1 + a_3 x + \cdots + a_{n-1} x^{(n/2)-1},$$

so we have that:

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2).$$

So, we can split a DFT computation into two equally sized parts, compute them, and then combine them in linear time.

7.3 Polynomial Multiplication

For polynomials A, B both with a degree bound n:

- Set the degree of A and B to $2n$, padding with zeroes,
- Perform the Fast Fourier transform,
- Form our point-value representation and multiply pointwise,
- Interpolate with the inverse Fast Fourier transform.

This process takes $O(n \log(n))$ time.

8 Dynamic Programming

Dynamic programming is the process of solving programming problems by breaking them down into overlapping subproblems, computing the base cases and storing the solutions to be later composed into a solution.

8.1 Largest Empty Square

This problem is about finding the largest square in a $n \times n$ black and white image such that the square does not contain a black pixel.

8.1.1 A Recursive Algorithm

To find the largest square at the position (x, y) (bottom-right corner at (x, y)), we use the following algorithm:

```
size LargestSquare(x, y) {
  if ((x, y) is black) return 0;
  if ((x == 1) or (y == 1)) return 1;
  return min(
    LargestSquare(x - 1, y - 1),
    LargestSquare(x - 1, y),
    LargestSquare(x, y - 1));
}
```

The time complexity of this algorithm, however, is exponential.

We get this as each cell barring the first and last columns and rows have cells where `LargestSquare` is computed three times (as they are checked from below, to the right, and below and to the right).

8.1.2 A Dynamic Algorithm

We now consider storing our solutions to cells so we do not repeat ourselves, take `A` to be an $n \times n$ array where each cell is undefined as first:

```
size LargestSquareStored(x, y) {
  if ((x, y) is black) A[x, y] = 0;
  if ((x == 1) or (y == 1)) A[x, y] = 1;
  if (A[x, y] is undefined) A[x, y] = min(
    LargestSquareStored(x - 1, y - 1),
    LargestSquareStored(x - 1, y),
    LargestSquareStored(x, y - 1));
  return A[x, y];
}
```

Giving `LargestSquareStored(n, n)` a time complexity of $O(n^2)$.

8.2 Weighted Interval Scheduling

We have a set of n intervals, a triple containing a start time s_i, finishing time f_i, and a weight w_i. A schedule is a set of intervals such that they do not overlap (with respect to their starting and finishing times).

The intervals are provided as an array A, sorted ascending by finishing times.

8.2.1 The Latest Compatible Interval

We define p as a function between intervals, taking an interval i and returning the latest interval that finishes before i starts.

This can be pre-computed in $O(n \log_2(n))$ time by using binary search.

8.2.2 A Recursive Algorithm

For n intervals indexed by $\{1, \ldots, n\}$:

```
weight WIS(i) {
  if (i == 0) return 0;
  return max(WIS(i - 1), WIS(p(i)) + w_i);
}
```

However, this leads to WIS(i) being calculated more than once for some i when we calculate `WIS(n)`.

8.2.3 A Dynamic Algorithm

For n intervals indexed by $\{1, \ldots, n\}$, we now consider a global array of n schedules S where each is entry is initially undefined:

```
weight WISStored(n) {
  if (n == 0) return 0;
  for (int i = 1; i <= n; i++) {
    S[i] = max(S[i - 1], S[p(i)] + w_i);
  }
  return S[n];
}
```

This takes $O(n)$ time.

8.2.4 Returning the Schedule

We can find the schedule using our stored S from the previous section:

```
schedule FindSchedule(i) {
  if (i == 0) return [];
  if (S(i - 1) <= S(p(i)) + w_i) {
    return FindSchedule(p(i)) ++ [i];
  }
  return FindSchedule(i - 1);
}
```

Thus, `FindSchedule(n)` gives us our schedule and takes $O(n)$ time.

8.3 Self-balancing Trees

8.3.1 Perfectly Balanced Trees

A tree where each path from the root to a leaf has the same length is perfectly balanced.

8.3.2 Parts of our Self-balancing Tree

We want to use self-balancing trees as an optimisation over linked lists in a dynamic search structure. Consider a tree where each node can have between 2 and 4 (inclusive) children (where a child can be empty) called a $2-3-4$ tree. Take note of the following:

2-node a node with value v, 2 children, and 1 key where the left child is less than or equal to v and the right child is greater than or equal to v.

3-node a node with values v_1, v_2, 3 children, and 2 keys where the left child is less than or equal to v_1, the middle child is between v_1 and v_2 (inclusive), and the right child is greater than or equal to v_2.

4-node a node with values v_1, v_2, v_3, 4 children, and 3 keys where the left child is less than or equal to v_1, the left-middle child is between v_1 and v_2 (inclusive), the right-middle child is between v_2 and v_3 (inclusive), and the right child is greater than or equal to v_3.

8.3.3 The Height of $2-3-4$ Trees

If we suppose all the nodes in the tree are 2/4 nodes we get the worst/best case heights for a $2-3-4$ tree with n elements:

Node Type	Height
2	$O(\log_2(n))$
4	$O(\log_4(n))$

8.3.4 Insertion on $2-3-4$ Trees

Splitting this operation works on a 4-node and requires its parent isn't a 4-node. The middle value of the node is added to the parent and two 2-nodes are formed from the left and right values and the four children. Splitting the root increases the height of the tree and is the only operation with this property.

When inserting an element k we search for where the element belongs whilst splitting any 4-nodes into 2-nodes as we recurse. We convert the bottom node from type t to $t+1$ ($t \neq 4$ by our algorithm structure) and insert our value.

8.3.5 Deletion on $2-3-4$ trees

Fusion this operation works on two 2-nodes with a shared parent that isn't a 2-node. A relevant key is taken from the parent and used to form a 4-node. Fusing the root decreases the height of tree and is the only operation with this property.

Transferring this operation works on a 2-node and a 3-node with a shared parent. A key from the parent is added to the 2-node whilst a key from the 3-node is added to the parent

We will consider the cases when deleting a value k. For leaves, we search for the value, transferring and fusing to convert 2-nodes on the path, we delete the value, converting the node from a node type t to a type $t-1$ ($t \neq 2$ by our algorithm structure). For non-leaves, we delete the predecessor of k, k' (always a leaf) and replace k with k'.

8.4 Skip Lists

We want to use skip lists as an optimisation over linked lists in a dynamic search structure. Building on a linked list, we require it is sorted and then we can add 'shortcut' levels. Each level is a subset of the linked list in the level below with the bottom level being the full list and each level containing the minimum and maximum.

8.4.1 Insertion in Skip Lists

When inserting an entry, we choose randomly whether it should appear in the level above. We insert it into the lowest level and essentially flip a coin to see if we should insert it into the level above. We repeat these coin flips until it fails to be inserted again (note that each level must contain the minimum and maximum of the list and the top level should be exactly the minimum and maximum).

If there is a level which isn't the bottom layer that contains all entries, we can delete all levels below it.

8.4.2 Deletion in Skip Lists

When deleting an entry, we simply delete all occurances of the entry. If this is the minimum or maximum, we ensure that each level contains the minimum or maximum unless the whole list is empty.

If there is a level which isn't the top layer that contains only the minimum and maximum entries, we can delete it.

8.4.3 Finding in Skip Lists

We start at the minimum of the top layer, iterating across, moving down layers as we encounter values greater than our desired value. We return when we can't go down anymore or we find our value:

```
value find(key) {
  while (true) {
    if (entry.key == key) {
        // We found the desired entry
        return entry.value;
    } else if (entry.key > key) {
        // We need to move down
        if (layer below) {
            move down;
        } else {
            return undefined;
        }
    } else {
        // We need to move to the right
        if (entry to the right) {
            move right;
        } else {
            return undefined;
        }
    }
  }
  return undefined;
}
```

8.4.4 Runtime of skip lists

All processes take $O(\log_2(n))$ time on average with randomised levels. Also, for large n, the amount of levels is also $O(\log_2(n))$ on average.

9 Output Sensitivity

We have that some algorithms are output sensitive. This means that their runtime depends on what the answer to the problem is.

9.1 Line Intersections

Suppose we are given a set of line segments (as two coordinates), we would like to find all the coordinates of the Intersections between these line segments.

9.1.1 A Simple Algorithm

We iterate through all the pairs and output the intersections:

```
points intersections_simple(lines) {
  points ps;
  for (int i = 0; i < lines.size(); i++) {
    for (int j = i + 1; j < lines.size(); j++) {
      if (lines[i] intersects lines[j]) {
        ps.push(intersection);
      }
    }
  }
  return ps;
}
```

This algorithm takes $O(n^2)$ time.

9.1.2 Output Sensitivity in Line Intersections

It can be seen that certain inputs could potentially have $O(n^2)$ output but this would make it seem like the simple algorithm is optimal but we will see that if consider k to be the number of outputs, we can find an algorithm with $O(n \log_2(n) + k \log_2(n))$ time complexity. However, if we consider bounds for k:

$$\begin{aligned} k \leq 2n &\Rightarrow O(n \log_2(n)) \\ k \geq n^2 &\Rightarrow O(n^2 \log_2(n)), \end{aligned}$$

so for certain inputs this algorithm will be **worse** than the simple algorithm.

9.2 An Outline for Finding Intersections

It can be seen that for two line segments, they can only have an intersection if the spans of the segments in the y direction intersect also. Thus, we could consider sweeping a horizontal line through all our line segments picking up intersections as we go.

9.2.1 Adjacency

We say two line segments are adjacent if there is a contiguous horizontal line from one segment to the other (not interrupted by another line segment). It can be seen that two segments that are never adjacent can't intersect.

9.2.2 Event Points

We can't possibly iterate through all possible y points, thus we only consider 'event points' which are the end points of segments and line intersections but this requires that we calculate intersections as we go. If we have k intersections, this gives $O(n+k)$ event points.

We consider the set of event points as a priority queue with keys equal to their y value, allowing us to `extractMin` to get our next event point. However, our process could give rise to duplicate event points, but these can be dealt with by checking the queue beforehand.

9.2.3 Status

We consider the status of the sweep line to be the ordered set of line segments currently being interesected by the sweep line with respect to their x coordinates. The status can clearly only change at event points, so at each event point we query line segments that have newly become adjacent.

We consider status as a $2-3-4$ tree where:

- At the top of a line segment, we insert it,
- At the bottom of a line segment, we delete it,
- At an intersection, we swap the intersecting lines,

checking for new intersections as these changes occur.

9.2.4 The Full Process

We add all the start and end points of our line segments to our priority queue, and we iterate through them. We update the status and query for intersections as we progress, adding intersections to our output and the queue as necessary. This takes $O((n+k)\log_2(n))$ time.

10 Linear Programming

10.1 Vector Comparison

We have that for v, w in $\mathbb{R}^n$ for some n in $\mathbb{Z}_{>0}$ such that:

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \qquad w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix},$$

we have that $v \leq w$ if and only if for all i in $[n]$, $v_i \leq w_i$.

A result of this definition is that some vectors are incomparable.

10.2 Standard Form

The standard form of a linear programming problem is that we have an objective function $f : \mathbb{R}^n \to \mathbb{R}$, an $m \times n$ matrix A, and an m-dimensional vector b in $\mathbb{R}^m$. The desired output is a vector x in $\mathbb{R}^n$ that maximises $f(x)$ subject to $Ax \leq b$ and $x \geq 0$.

We can accomodate for minimisation and lower bounds by multiplying by negative one, and we can write negative variables as the subtraction of positive variables.

10.3 Integer Linear Programming

In a linear programming problem where we add the constraint that solutions must be integers, we can relax our constraints to form approximate solutions.

11 Flow Algorithms

11.1 Flow Networks

A flow network $F = (G, c, s, t)$ consists of a directed graph $G = (V, E)$, a capacity function $c : E \to \mathbb{R}$, a source vertex s in V (with empty in-neighbourhood), and a sink vertex t in V (with empty out-neighbourhood).

11.1.1 Flows

A flow in F is a function $f : E \to \mathbb{R}$ with the following properties:

- No edge has more 'flow' than capacity, for e in E:

$$0 \leq f(e) \leq c(e),$$

- Flow is conserved, for v in $V \setminus \{s, t\}$:

$$f^-(v) = \sum_{u \in N^-(v)} f((u, v)) = \sum_{w \in N^+(v)} f((v, w)) = f^+(v).$$

We denote the value of a flow as $v(f) = f^+(s)$. We can also define f^+ and f^- for sets by considering the flow entering and exiting the sets. Similarly, in and out flow of sets are identical.

11.1.2 Cuts

A cut is a partition A, B of V such that the source is in A and the sink is in B. We have that the flow in A minus the flow out is equal to the flow out to B minus the flow in:

$$(f^- - f^+)(A) = (f^+ - f^-)(B) = v(f).$$

11.2 Maximising Flow Value

We choose a greedy algorithm that works on a modified graph G_F, where we add backwards edges (edges corresponding to edges with non-zero flow) and forwards edges (edges corresponding to edges with non-capacity flow) that allows flow to be pushed back down the edge. So, edges are no longer considered when at capacity (except after we choose to push their flow backwards and re-route it).

11.2.1 Augmenting Paths in Flow Networks

An augmenting path is a directed path from s to t in G_F.

11.2.2 Residual Capacity

The residual capacity of an edge is the amount of flow you can add to the forward edge or the amount of flow you can remove from the backward edge (whichever is greater).

11.2.3 Pushing

Considering an augmenting path, pushing the path involves adding as much flow as possible to all the edges.

11.2.4 The Ford-Fulkerson Algorithm

We consider all augmenting paths (depth-first) and push each one, updating G_F as we go. The resulting flow is maximal. This takes $O(v(f^*)|E|)$ time where f^* is our maximal flow.

11.2.5 Flow Networks and Bipartite Graphs

Given a bipartite graph $G = (V, E)$ with bipartition A, B, we can form a new graph G' identical to G except:

- All edges are directed from A to B,
- All edges have capacity 1,
- We add a source vertex connected to all vertices in A,
- We add a sink vertex connected to all vertices in B,

in G', maximal flows corresponding to matchings.

11.2.6 The Edmonds-Karp Algorithm

If we pick augmenting paths with minimal edges (breadth-first) then we are guaranteed to finish Ford-Fulkerson in $O(|V||E|^2)$ time.

11.2.7 Vertex Capacities

We can add capacities to vertices too, forming a vertex flow network $F = (G, c_E, c_V, s, t)$ identical to the flow network except we restrict flow through vertices. We can form a regular flow network from this by changing each vertex v into two vertices v^+, v^- where the capacity of (v^-, v^+) is $c_V(v)$ and all the edges going into v go into v^- and all the edges going out of v go out of v^+.

11.3 Circulation Networks

A circulation network $C = (G, c, d)$ is a directed graph $G = (V, E)$ and a capacity function $c : E \to \mathbb{N}$ and demand function $d : V \to \mathbb{N}$. Vertices with positive demand are sinks, and vertices with negative demand are sources.

11.3.1 Circulations

A circulation is a function $f : E \to \mathbb{R}$ with $0 \leq f(e) \leq c(e)$ for each e in E and $f^-(v) - f^+(v) = D(v)$ for all v in V (flow is conserved except at sources and sinks).

We find circulations by attaching a source vertex to all sources in C with edges equal to the (negative) demand of the sources and similarly adding a sink vertex to all sinks in C with edges equal to the demand. This forms a flow network we can run our algorithms on.

12 Complexity Theory

12.1 Decision Problems

A decision problem is a problem such that the answer is in the set {`Yes`, `No`}.

12.2 Oracles

An oracle is a construct that given its corresponding problem, solves it in $O(1)$ time.

12.3 Cook Reductions

For the decision problems X, Y with O_Y the oracle for Y, we have that a Cook reduction from X to Y is an algorithm A_X which given an input of size n runs in $O(x^n)$ time and makes $O(x^n)$ calls to O_Y (all with inputs of size $O(x^n)$).

Suppose we have algorithms A_X, A_Y which solve the decision problems X and Y respectively. If whilst performing A_X we call A_Y as a subroutine a $O(x^n)$ number of times (for some finite n), we say we have a Cook reduction from X to Y denoted by $X \leq_c Y$.

12.3.1 Properties of Cook Reductions

We have that for the decision problems X, Y, Z:

- $X \leq_c Y$, $Y \leq_c Z$ implies that $X \leq_c Z$ (transitivity)
- $X \leq_c Y$ implies that if we have a polynomial time algorithm for Y, we have one for X
- $X \leq_c Y$ and there is no polynomial-time algorithm for X implies that there is no polynomial-time algorithm for Y.

12.4 Karp Reductions

For the decision problems X and Y, a Karp reduction from X to Y is a map f from the instances of X to the instances of Y such that:

- $f(x)$ can be computed in polynomial time (in x)
- $f(x)$ is a `Yes` instance of Y if and only if it's a `Yes` instance of X.

This is denoted by $X \leq_K Y$.

12.4.1 Properties of Karp Reductions

We have that Karp reductions are stronger than Cook reductions so that for any two decision problems X, Y:

$$X \leq_K Y \Rightarrow X \leq_C Y.$$

12.5 The Class, P

We have that **P** is the class of all decision problems which have a polynomial-time algorithm. We have that $\mathbf{P} \subseteq \mathbf{NP}$ as we can just process the solution (ignoring the witness).

12.6 The Class, NP

We have that **NP** is the class of all decision problems X such that there is some polynomial-time verification algorithm A_X such that for some input x, if x is a `Yes` instance of our problem, there is a witness bit string w such that $A_X(x, w) =$ `Yes`.

12.6.1 NP-hardness

We say a problem is **NP**-hard (under Cook reductions) if SAT Cook-reduces to it. Similarly, we say a problem is **NP**-hard (under Karp reductions) if SAT Karp-reduces to it.

12.6.2 NP-completeness

We say a problem is **NP**-complete if it is **NP**-hard and in **NP**.

12.7 The SAT Problem

The SAT problem is the problem that asks if when given some formula in conjunctive-normal form (consisting of AND and OR clauses) we can assign the variables such that the formula is satisfied (true).

12.7.1 Cook-Levin Theorem

We have that **SAT** is **NP**-hard and thus, **NP**-complete. Thus, every problem which **SAT** reduces to is **NP**-hard.

12.8 The 3-SAT Problem

We have that the width of a conjunctive-normal form is the number of literals of all the OR clauses. The 3-SAT problems asks if a width-3 conjunctive-normal form is satisfiable. We have that this is **NP**-complete.