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.*

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 (null if one does not exist). 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 is a set of distinct elements with associated value called the key. We can use a binary heap as a priority queue with the elements as the keys and the heap property that the parents are less than or equal to the 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

Stores a collection of disjoint sets where each set has elements $1, 2, \ldots n$ for some natural n. 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.

So, adding 3 *to* $\{7\}$ *would make* 3 *always point to* 7*. But creating a set with* 3*, will make it point to itself.*

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 Graph Theory

2.1 Definition of a Graph

A graph is a pair of sets $G = (V, E)$, where V is a set of vertices (or nodes) and E is a set of edges (or arcs).

2.2 Definition of an Edge

An edge of a graph $G = (V, E)$ is $e = \{u, v\}$ in E where u, v are vertices in V.

2.3 Definition of a Neighbourhood

For a graph $G = (V, E)$ with v in V, the neighbourhood of v is the set $V' \subseteq V$ of vertices connected to v by an edge in E.

The neighbourhood of v is denoted by $N(v)$.

The neighbourhood of a set of vertices is the union of the neighbourhoods of each vertex.

2.4 Definition of Degree

For a graph $G = (V, E)$ with v in V, the degree of v is the size of its neighbourhood.

The degree of v is denoted by $d(v)$.

2.5 The Handshake Lemma

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

$$|E| = \frac{\sum_{v \in V} d(v)}{2}.$$

This is because each edge visits two vertices, so by counting the degree of each vertex we count each edge exactly twice.

2.6 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:

$$d(v) = k.$$

We cannot have a k-regular graph where k is odd and $|V|$ is odd by the Handshake Lemma.

2.7 Isomorphic Graphs

Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called isomorphic if there exists a bijection $f : V_1 \to V_2$ such that:

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

This relationship is denoted by $G_1 \cong G_2$.

2.8 Definition of a Subgraph

A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

2.9 Definition of an Induced Subgraph

An induced subgraph generated from $G = (V, E)$ by $V' \subseteq V$ is the graph $G' = (V', E')$ where:

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

Essentially, you generate an induced subgraph from a subset of the vertices of a graph by selecting edges that join vertices in the subset.

2.10 Walks

2.10.1 Definition of a walk

A walk in a graph $G = (V, E)$ is a set of vertices in V connected by edges in E. The length of the walk is the number of edges traversed in the walk.

2.10.2 Definition of a path

A path is a walk where no vertices are repeated.

2.10.3 Definition of an Euler walk

An Euler walk is a walk such that every edge is traversed exactly once. Thus, for a graph $G = (V, E)$, the length is $|E|$.

2.10.4 Conditions for an Euler walk

For an Euler walk to be possible on a given graph, all vertices must have an even degree **or** exactly two vertices have odd degree.

If all vertices have even degree we have that the Euler walk is a cycle, if exactly two vertices have odd degree then we have that these vertices are the start and end points of our Euler walk.

2.11 Definition of a Connected Graph

A connected graph is a graph where for each pair of vertices, there is a path connecting them.

2.12 Definition of a Component

A component of a graph G is a maximal connected induced subgraph of G. This means an induced subgraph of G that is connected but is not longer connected if a vertex is removed.

2.13 Digraphs

2.13.1 Definition of a digraph

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.

2.13.2 The Directed Handshake Lemma

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

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

This is because if we consider the 'tail' of an edge (the vertex it leaves), each edge has exactly one tail.

2.13.3 Definition of a strongly connected digraph

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

2.13.4 Definition of a weakly connected digraph

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

2.13.5 Definition of components of 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.

So, these are induced subgraphs that are strongly/weakly connected but are no longer strongly/weakly connected once a vertex is removed.

2.13.6 Definition of neighbourhoods in digraphs

The neighbourhood of a vertex in a digraph can be considered by looking at the edges *from* the vertex and the edges *to* the vertex.

The in-neighbourhood of a vertex v are the edges that enter v. The out-neighbourhood of a vertex v are the edges that exit v. These are denoted by $N^-(v)$ and $N^+(v)$ respectively.

2.13.7 Definition of degrees in digraphs

For a vertex v, the in-degree of the vertex $d^-(v)$ is the size of the in-neighbourhood and the out-degree of the vertex $d^+(v)$ is the size of the out-neighbourhood.

It can be seen that the degree of a given vertex is the sum of its in and out degree (in a digraph).

2.13.8 Conditions for an Euler walk in a digraph

For an Euler walk to be possible on a given digraph, we have two cases, either:

- the digraph is strongly connected and every vertex has equal in and out degrees, or
- one vertex has an in-degree one greater than its out-degree, another has an out-degree one greater than its in-degree, and all remaining vertices have equal in and out degrees.

In the first case we have that the Euler walk is a cycle, in the second we have that the special vertices are the start and end points of our Euler walk.

2.13.9 Cycles

2.13.10 Definition of a cycle

A cycle is a walk where the first and last vertices are the same and each vertex appears at most once (barring the first and last vertex).

2.13.11 Definition of a Hamiltonian cycle

A Hamiltonian cycle is a cycle where each vertex is visited.

2.13.12 Conditions for a Hamiltonian cycle

Whilst the conditions necessary for a Hamiltonian cycle in general are unknown, by Dirac's theorem, we know that for a graph with n vertices, if every vertex has degree $\frac{n}{2}$ or greater then a Hamiltonian cycle exists.

2.14 Trees

2.14.1 Definition of a forest

A forest is a graph with no cycles.

2.14.2 Definition of a tree

A tree is a connected forest (or a connected graph with no cycles).

2.14.3 Path uniqueness of trees

For a tree $T = (V, E)$, we have that for any u, v in V, there exists a unique path from u to v.

To prove this, suppose there are two unique paths between u and v. These paths must diverge and if we connect them, they form a cycle which contradicts the definition of a tree.

2.14.4 The magnitude of edges in trees

For a tree $T = (V, E)$, we have that $|E| = |V| - 1$.

2.14.5 Rooted trees

For a tree $T = (V, E)$, we can root T with some r in V. For v in $V \setminus r$, we 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
- v is a **leaf** if it has degree 1
- $L_0 = \{r\}$ and $L_n = \{v : |P_v| = n\}$ are the **levels** of T
- The **depth** of a tree is the greatest n where L_n is non-empty.

2.14.6 Lower bound on the amount of leaves in a tree

For a tree with $T = (V, E)$, if $V > 1$, there must be at least 2 leaves.

2.14.7 Equivalent statements to the tree definition

For a graph $T = (V, E)$, we have that the following are equivalent:

- T is a tree
- T is connected and has no cycles
- $|E| = n - 1$ and T is connected
- $|E| = n - 1$ and T has no cycles
- T has a unique path between any two vertices

2.15 Bipartitions

2.15.1 Definition of a bipartite graph

For $G = (V, E)$, we have that G is bipartite if there exists $A \subset V$, $B \subset V$ such that A and B are disjoint and the induced subgraphs of A and B have no edges. A and B are bipartitions of G.

Saying G is bipartite is equivalent to saying G has no cycles of odd length.

2.15.2 Definition of a matching

A matching in a graph is a set of disjoint edges.

A matching is **perfect** if each vertex is contained in some matching edge.

2.15.3 Definition of a semi-matching

For k in $\mathbb{Z}_{>0}$, a k to 1 semi-matching in a bipartite graph G with a bipartition $\{A, B\}$ is a subgraph of G where each vertex in A has degree at most k and each vertex in B has degree at most 1.

2.15.4 Definition of an augmenting path

Given a matching M in a bipartite graph $G = (V, E)$, an augmenting path is a set of vertices in V connected by edges e_i in E such that:

$$e_i \text{ is } \begin{cases} \text{in } M & \text{for } i \text{ odd} \\ \text{not in } M & \text{for } i \text{ even.} \end{cases}$$

With the condition that the first and last vertices in the path are not in the matching.

2.15.5 Hall's Theorem

For a bipartite graph $G = (V, E)$ with the bipartition (A, B) has a perfect matching if and only if $|A| = |B|$ and for all $X \subseteq A$, $|N(X)| \geq |X|$.

3 Working on Graphs

3.1 Data Representations of Graphs

3.1.1 Adjacency matrix

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

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

3.1.2 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 and the corresponding list represents the vertices in the neighbourhood of the vertex.

3.1.3 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).

3.2 Search

Generally with a graph searching algorithm, we have a data structure which is left undefined here (besides the fact we can add vertices to it and take vertices out). Starting with a vertex `u`, 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 $O(|E|)$ (assuming the data operations are $O(1)$) where E is the edge set.

3.2.1 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.

Shortest paths By tracking distances, we can find shortest paths using this searching style. This is $O(|V| + |E|)$ in a graph $G = (V, E)$.

3.2.2 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.

3.3 Djikstra's Algorithm

For a weighted (non-negatively), directed graph (stored as a adjacency list) we have that Djikstra's algorithm (given a) starting vertex returns the fastest path to all other vertices (or a particular one if required). It 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 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.

3.4 Bellman-Ford's Algorithm

This algorithm solves the shortest path problem for weighted directed graphs. Note that there's no constraint on the parity of the weights. We assume that there are no negative weight cycles.

```
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.

3.4.1 Negative weight cycles

Suppose our graph has a cycle which has a negative weight. This must mean that we can choose an arbitrarily small/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.

3.5 Minimum Spanning Trees

3.5.1 Definition of a spanning tree

In a connected, undirected graph $G = (V, E)$, we have that a spanning tree $T = (V', E')$ of G is a subgraph of G where T is a tree and $V = V'$.

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

3.5.2 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.

4 Fast Fourier Transforms

4.1 Polynomials

4.1.1 Definition of a Polynomial

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_i are the coefficients of A. We say for $k > n$, k is a degree-bound of A. We can represent this by listing the coefficients, called the **coefficient representation**.

4.1.2 Fast Polynomial Evaluation

We can evaluate polynomials quickly using *Horner's Rule*, 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;
}
```

We have that this is $O(n)$.

4.1.3 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.

4.1.4 Point-Value Representation

We can represent a polynomial by a set of points it intersects like so:

$$\{(x_0, y_0), \ldots, (x_n, y_n)\},$$

for a polynomial degree $n + 1$.

4.1.5 Polynomial Addition

For two polynomials A, B with coefficients a_i, b_i and degrees n, m respectively, we have that:

$$(A + B)(x) = \sum_{i=0}^{\max(n,m)} (a_i + b_i)x^i.$$

If $m > n$ or vice versa, we pad out the shorter polynomial with zeroes. We can do this with the point-value representation by adding the 'y-values'. We have that addition as it's defined here is $O(n)$.

4.1.6 Polynomial Multiplication

For two polynomials A, B with coefficients a_i, b_i and degrees n, m respectively, we have that:

$$C(x) = (A \cdot B)(x) = \sum_{i=0}^{k} c_i x_i,$$

where $k = 2 \cdot \max(n, m)$ and:

$$c_i = \sum_{j=0}^{i} a_j b_{j-1}.$$

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

$$A = \{(x_0, y_0), \ldots, (x_n, y_n)\},$$
$$B = \{(x_{n+1}, z_0), \ldots, (x_{n+m}, z_m)\},$$

We have that:

$$C = A \cdot B = \{(x_0, y_0 \cdot z_0), \ldots, (x_k, y_k \cdot z_k)\}$$

This is much easier, yielding an $O(n)$ algorithm rather than an $O(n^2)$ algorithm.

4.2 Fast Fourier Transform

4.2.1 Roots of Unity

The idea is that we evaluate a polynomial to perform pointwise multiplication and then interpolate back into a polynomial. 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 $\{0, 1, \ldots n\}$. Therefore considering:

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

for A a polynomial, k as above, and the vector of all ordered y_k being 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 $\frac{n}{2}$.

This is true due to the Cancellation Lemma, we have:

$$(\omega_{2k}^j)^2 = \omega_{2k}^{2j} = \omega_k^j.$$

4.2.2 Method of the Fast Fourier Transform

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

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

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.

4.3 Polynomial Multiplication

So, the steps are laid out, for polynomials A, B with degree bound n, as follows:

- 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 is $O(n \log(n))$.

5 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.

5.1 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.

5.1.1 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).

5.1.2 Storing the solutions to subproblems

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 LargestSquare_Stored(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(
    LargestSquare_Stored(x - 1, y - 1),
    LargestSquare_Stored(x - 1, y),
    LargestSquare_Stored(x, y - 1));
  return A[x, y];
}
```

This can be adapted to work iteratively from $(1, 1)$ down to (x, y) with a complexity of $O(n^2)$.

5.2 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.

5.2.1 The rightmost compatible interval p

We define p as a function from **Interval** $\rightarrow$ **Interval**, which takes an interval i and returns the **latest** interval that finishes **before** i.

This can be precomputed beforehand in $O(n \log_2(n))$ time by using binary search ($O(\log_2(n))$).

5.2.2 A recursive algorithm

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

```
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.

5.2.3 Storing the solutions to subproblems

Now, we consider a global array of schedules S where $S[i]$ contains `WIS`(i) or is undefined:

```
weight WIS_Stored(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.

5.2.4 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);
}
```

This takes $O(n)$ time.

5.3 Self-balancing Trees

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

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.

5.3.1 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	$O(\log_4(n))$

5.3.2 Insertion on $2-3-4$ trees

Splitting this operation works on a 4-node. The middle value of the node is added to the parent and two 2-nodes are formed from the remains.

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 $t+1$ ($t \neq 3$ by our algorithm structure) and insert our value.

5.3.3 Deletion on $2-3-4$ trees

Fusion this operation works on two 2-nodes with a shared parent. 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'.

5.3.4 Binary Search Trees

For an element k in a binary search tree (acting as our dynamic search structure), the left child of an element is less than or equal to k and the right child is greater than or equal to k.

However, this results in $O(\log_2(n))$ time for `insert`, `find`, and `delete` for balanced trees like the $2-3-4$ tree above but becomes $O(n)$ for unbalanced trees.

5.4 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.

5.4.1 Insertion in skip lists

When inserting an entry, we choose randomly whether it appears in the level above. We insert it into the lowest level and 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.

5.4.2 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.

5.4.3 Finding in skip lists

We start at the minimum of the top layer, iterating across it until we find it or we find a greater value. If the next value is greater, we move to the layer below and repeat the process:

```
value find(int key) {
  while (entry.key != max_key) {
    if (entry.key == key) return entry.value;
    else if (entry.key > key) move down;
    else move right;
  }
  return undefined;
}
```

5.4.4 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.

6 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 segements.

6.1 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 $O(n^2)$.

6.2 Output Sensitivity

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:

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

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

6.3 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.

6.3.1 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.

6.3.2 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.

6.3.3 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.

6.3.4 The full process

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