Contents

1 Definitions to grind/cram

1.1 Runtimes

- Graphs
  - **Dijkstra's Algorithm:** $O((|V| + |E|) \log |V|)$
  - **Bellman-Ford Algorithm:** $O(|V||E|)$
  - **Johnson's Algorithm:** $|V||E| \log |V|$
- Minimum Spanning Trees
  - **Kruskal's Algorithm:** $O(|E| \log |V|)$
- Flow
  - **Edmonds-Karp Algorithm:** $O(|V||E|^2)$
  - **Ford-Fulkerson:** $O(v(f^*)|E|)$ ($v(f^*)$ is the maximum flow)

1.2 Lemmas

- Graphs
  - **Dirac's Theorem:** Any n-vertex graph G with minimum degree at least $n/2$ has a Hamilton cycle.
  - **Handshake Lemma:** Adding up the vertex degrees and dividing by 2 yields the number of edges in a graph.
  - **Berge's Lemma:** If M has no augmenting paths $\Rightarrow M$ is a maximum.
  - **Hall's Theorem:** A bipartite graph $G = (V, E)$ with bipartition (A, B) has a perfect matching iff $|A| = |B|$ and for all $X \subseteq A, |N(X)| \geq |X|$
- Hashing
  - **Markov's Inequality**: Let $X \geq 0$ be a random variable with mean μ. Then, for all $t \geq 0$, $P(X \geq t) \leq \mu/t$
- Flow
  - **Max-flow min-cut:** The value of a max flow is equal to the minimum capacity of the $c^+(A)$ over all cuts (A, B).
- Complexity Theory
  - **Cook-Levin Theorem:** SAT is NP-Hard, and therefore NP-Complete. Every problem that SAT cook-reduces to is also NP-hard.

1.3 Runtime of Dynamic Structures

	Insert	Delete	Find
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$
Binary Search Tree	$O(n)$	$O(n)$	$O(n)$
2-3-4 Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Skip List	$O(\log n)$	$O(\log n)$	$O(\log n)$

2 Graphs

2.1 Definitions

- Graphs are **isomorphic** if they are essentially the same graph (vertices have the same degrees and are connected to the same vertices) but they are labelled differently.
- Graphs are **equal** if they are *isomorphic* but the labels are also the same.
- The **neighbourhood** Of a vertex is the set of vertices joined to it by an edge.
- The **degree** of a vertex is the number of vertices joined to it.
- A **walk** is a sequence of vertices. An **Euler walk** is a walk that contains every edge in the graph exactly once. If there is an Euler walk in a graph then either:
  - Every vertex has even degree
  - All but two have an even degree (the start and end points).
- A **path** is a walk where no vertices repeat
- A graph is **connected** if any two vertices are joined by a path.
- A **subgraph** is a part of a some graph.
- An **induced subgraph** is a subgraph that is only created by removing vertices. That is, all the vertices in the subgraph must have the same edges as in the original graph.
- A graph is **Strongly connected** if there is a path from any vertex to all other vertices. A graph is **weakly connected** if for all vertices (u, v) there is a path from u to v **or** from v to u
- The **in-degree** of a vertex is the number of vertices pointed at it. The **out-degree** is the opposite.
- A **cycle** is a walk in which every vertex appears at *most* once (except the start and end nodes that appear twice because they're the same node). A **Hamilton cycle** is a cycle that has every vertex in the graph.
- A **tree** is a connected graph with no cycles.
- A **matching** is a collection of disjoint edges. It is **perfect** if every vertex is contained in some matching edge.
- A graph is **bipartite** if the vertices of the graph can be partitioned into two disjoint edges that contain no edges.
- An **augmenting path** is a path in a graph that alternates between matching and non-matching edges, which begins and ends in unmatched vertices.

2.2 Matchings

Given a bipartite graph, we can't form a matching by greedily adding edges (because sometimes it fails). We can repair the bad decisions we make with a greedy algorithm by using **augmenting paths**.

Given some matching in a bipartite graph, an *augmenting path* is a path such that:

- $\{v_i, v_{i+1}\} \in M$ for all odd i
- $\{v_i, v_{i+1}\} \notin M$ for all even i
- $v_i, v_k \notin \cup_{e \in M} e$

If we have an augmenting path for a matching, then we use that path instead of the current one that

we have. This means that we can now make a new greedy algorithm. Here, we have switch being a matching containing one more edge than M.

```
Input: A bipartite graph G = (V,E)
Output: A list of edges forming a matching in G of maximum size.

M = [] //empty matching
while G has augmenting path for M do {
    Find augmenting path P for M
    Update M = Switch(M,P)
}
```

We can't search for augmenting paths by brute force, so we instead reduce the problem to find any path from one set to another in a **directed graph** (breadth-first search). This gives us the final algorithm:

```
Find bipartition (A,B) in G
m = [] //initialise
while(true) {
    Form directed graph with matching
    P = new augmenting path if one exists
    else break
    Update M = switch(M,P)
}
return M
```

Overall, the runtime is $O(|E||V|)$. Using Berge's Lemma, we know that if there are no augmenting paths, then M is a maximum.

2.3 Semimatchings

A semimatching is a matching where each vertex in A has degree at most k and each vertex in B has degree at most 1. We want to reduce this problem to just finding a matching so that we can use the max matching algorithm we defined just now.

What we can do is to clone the people k times, let the clones be maximally matched using max matching, and then delete the clones (combining each matching of the clones).

2.4 Hall's Theorem

What if we just want to know whether a max matching exists or not? (The **decision problem** rather than the solution. More on this in complexity theory). We can use Hall's theorem for this that runs in constant time.

3 Hashing

3.1 Data structures

3.1.1 Stacks

Stacks are first in last out (**FILO**). They have 3 operations:

- **create()** ($O(1)$)
- **push(x)** ($O(1)$)
- **pop()** ($O(1)$)

3.1.2 Queues

Queues are first in first out (**FIFO**). They also have 3 operations:

- **create()** ($O(1)$)
- **add(x)** ($O(1)$)
- **serve()** ($O(1)$)

3.1.3 Double-ended linked lists

A linked list has four operations:

- **create()**: creates a new list and returns two IDs. ($O(1)$)
- **insert(x,i)**: inserts x after the node with ID i. ($O(1)$).
- **delete(i)**: deletes node with ID i. ($O(1)$).
- **lookup(i)**: returns the node with ID i. ($O(1)$).

3.1.4 Arrays

Arrays have three operations: `create()`, `update(x,i)`, `lookup(i)`. These all have $O(1)$.

3.1.5 Hash tables

Hash tables have four operations and have key-value pairs.

- **create()**: Creates new has table. ($O(1)$)
- **insert(k,v)**: Inserts (k, v) into the table. ($O(1)$ on **average**)
- **lookup(k)**: If (k, v) is in the table, returns v else `Null`. ($O(1)$ on **average**)
- **delete()**: Deletes pair (k, v). ($O(1)$)

3.2 Pros and cons

Task	Length-l list	Size-s array	Hash-table
Search for an element	$\Theta(l)$	$\Theta(s)$	$\Theta(1)$
Sequence of n insertions and/or removals	$\Theta(ln)$	$\Theta(sn)$ (holds $\leq s$)	$\Theta(n)$
Iterate over all elements in order	$\Theta(l)$	$\Theta(s)$	No order
Find an element by position	$\Theta(l)$	$\Theta(1)$	No order

Arrays: Good for storing *ordered* data with random access/update.

Lists: Good for storing ordered data that's being iterated over a lot.

Hash tables: Good for storing unordered data (maybe associated with keys.)

3.3 Hash tables

When we insert a value at point k, there might be another item there already (called a *collision*). We can solve this by **chaining**.

Instead of having one value at each point, we have an array of linked lists. This is good, but they do make the lists longer, impacting performance. We circumvent this by dynamic resizing.

On insertion, if there are at least $n/2$ values in the table, make a new table that's twice as big, and reinsert each value into the new table.

3.3.1 The Hash functino

We want our hash function to be good. We want the size of the array of linked lists to be a **prime number** (we use a lookup table for this). Then, to find the position of the linked list for a value (compute $h_p(k)$), we:

- Split k into base-p words. $k = \Sigma_{i=0}^{t} x_i p^i$ where $x_i = \lfloor k/p^i \rfloor \mod p$
- Output $h_p(k) = (\Sigma_i a_i x_i) \mod p$

This takes $O(\log N)$ time. So our runtime for a sequence of t operations will be $O(t \log N)$ time. This leaves us with this final implementation of hash tables:

create(n = 100): Finds a prime between p and $2n$.

insert(k,v): Increments the number of pairs stored, and adds (k, v) to the table (utilising the hash function discussed above). If at least $p/2$ pairs are stored, calls `create(2p)` to make a new table and copies everything over.

lookup(k): Searches for (k, v) in the list and returns v. **delete(k)**: Decrements the number of pairs stored and removes (k, v) from the list.

4 Fast Fourier Transform

4.1 Discrete Fourier Transform

$$A(x) = a_0 + a_1x + a_2x^2 + \dots$$
$$y_k = A(\omega_n^k)$$

4.2 Fast transforming

Fast Fourier Transform (FFT) is basically the same as the discrete FT, but it splits recursively splits the polynomial up into even and odd powers. For example, if we take $A[x] = 0 + 0x + 1x^2 - 1x^3$, then we get:

$$A^{[0]} \leftarrow (0, 1)$$
$$A^{[1]} \leftarrow (0, -1)$$
$$y^{[0]} \leftarrow FFT((0, 1), 2)$$
$$y^{[1]} \leftarrow FFT((0, -1), 2)$$

$FFT((0, 1), 2)$ is just the two squares of unity, and $FFT((0, -1), 2)$ is the negative of the two squares of unity $(-1, 1)$.

5 Traversing Graphs

5.1 Depth First Search

Depth First searches use a stack to store the nodes. Therefore in a binary tree, we would go down an entire branch before returning to search the children.

5.2 Breadth First Search

Breadth first searches use a queue to store the nodes. Therefore, in a binary tree, we would search all the children of each node before going down a level.

6 Shortest Path

6.1 Priority Queues

A **priority queue** stores a set of distinct elements. The following operations are supported by them:

- **insert(x,k)**: Inserts x with $x.key = k$.
- **decreasekey(x,k)**: Decreases the value of $x.key$ to k ($k < x.key$).
- **extractmin()**: Removes and returns the element with the smallest key.

We could use a number of methods of storing this data structure, but the best one is a binary heap. Each operation takes $O(\log n)$ time. We can also sort the structure in $O(n \log n)$ time.

6.2 Dijkstra's algorithm

To compute Dijkstra's algorithm, we:

1. Start at some node (A)
2. Set all distances to infinite
3. Then, for every edge, if `dist(v) > dist(u) + weight(u,v)`, then update the distance in the array.
4. Repeat until all nodes are **settled**

7 Minimum Spanning Trees

A **minimum spanning tree** is a subgraph such that every vertex is in the subgraph and it is a tree.

7.1 Kruskal's Algorithm

Kruskal's algorithm finds a minimum spanning tree in a connected, undirected graph using a disjoint set data structure:

1. For every vertex, do `MakeSet(v)`.
2. Sort the edges in order of increasing weight
3. For each edge (in order), if `FindSet(u) != FindSet(v)` then `Union(u,v)` and add to the minimum spanning tree.

These operations run in $O(\log |V|)$ time, so the overall runtime becomes $O(|E| \log |V|)$

8 Dynamic Programming

In the simplest sense, what we're trying to do with dynamic programming is:

- Find a recursive formula for the problem in terms of answers to the subproblems
- Write down a naive recursive algorithm
- Speed it up by storing the solutions to the subproblems
- Derive an iterative algorithm by solving the subproblems in a good order.

Sometimes, we need to start filling the array in from the top left. This allows us to fill in the rest of the array for computation.

9 Dynamic Search Structure

A dynamic search structure stores a set of elements. Each element x must have a unique key - $x.key$. The following operations must be supported:

- `insert(x,k)` - inserts x with the key $k = x.key$
- `find(k)` - returns the unique element x with $x.key = k$.
- `delete(k)` - deletes the unique element x with $x.key = k$.

9.1 2-3-4 Trees

The key idea here is that the nodes can have 2, 3, or 4 children (duh). Every path from the root to a leaf has the same length. Always.

9.1.1 Find

The `find` operation is easy. We just follow a path from the root. The time complexity of this operation is $O(h)$, with h being the height of the tree. This could be anywhere from $\log_4 n$ to $log_2 n$ (depending on the magnitude of each node).

9.1.2 Insert

To perform `insert(x,k)`, we search for the key as if performing `find(k)`. If the leaf is a 2-node, insert x, k, converting it into a 3-node. If the leaf is a 3-node, do the same. However, we cannot allow the node to be a 4 node.

To avoid this, we can **split** the 4-nodes into two 2-nodes (if the parent isn't a 4 node). We take the middle node and push it up to the parent node. Now, no path lengths have changed and this operation takes $O(1)$ time.

9.1.3 Delete

To perform `delete(k)` on a leaf, we search for the k using `find(k)`. If the leaf is a 3 or 4-node, delete (x, k). We can't delete 2 nodes, so we need to either **fuse** or **transfer**.

Fusing a node is where we merge two nodes together with the parent key to create a 4-node. If the parent is the root, then we can also fuse them together by combining the root with the two children to make a 4-node.

Transferring is where we have a 2-node and a 3-node. We can transfer the parent key down, and place the other key from the 3-node where the parent node was.

Each of these functions take $O(\log n)$ time.

9.2 Skip lists

A skip list is essentially a linked list with some shortcuts put into place to make it a bit faster to find things. There are n levels to a skip list.

9.2.1 Find

To `find(k)`, we start at the top left. We then check the next node on that level. If the node is larger than k, then we go down a level and repeat until we find k.

9.2.2 Insert

To `insert(x,k)`, we simply find the place where the key should be and insert. Then, we simulate a coin toss. If heads, we replicate the key a level up, and repeat this until we hit tails.

9.2.3 Delete

`Delete(k)` is straightforward, we just find the key and delete from all levels.

Again, all operations will take $O(\log n)$ time. This is expected time, while the one for 2-3-4 trees are the *worst* case time.

10 Shortest Path 2

Earlier, we looked at the shortest paths for graphs that don't have any negative weights. But, some graphs do actually have negative weights. We can't just add a constant because this would mess

with the shortest path. Instead, we have another algorithm for it.

10.1 Bellman-Ford algorithm

This algorithm essentially repeatedly asks if it can find a shorter route. It runs $|V|$ iterations of itself, and in each iteration we relax every edge in the graph. After enough iterations (enough being every edge being processed $|V|$ times), we will get the shortest path for each node.

Unfortunately, negative weight cycles break this algorithm, since we could go around the cycle infinite times and get a shorter path every time. So, we just add a final check to see if the algorithm would find a shorter path (which would be impossible without there being a negative weight cycle). If there is, then we flag it up.

```
For all v, set dist(v) = infinity
set dist(s) = 0
for i = 1 to size(V) {
    for each edge (u,v) in E
        if(dist(v) > dist(u) + weight(u,v)) {
            dist(v) = dist(u) + weight(u,v)
        }
}

for each edge (u,v) in E {
    if(dist(v) > dist(u) + weight(u,v)) {
        output "negative weight cycle found"
    }
}
```

The algorithm sets all distances to infinity (which takes $O(|V|)$ time), and then begins the main loop (which will take $O(|V||E|)$ time), before doing a final check ($O(|E|)$ time) leaving us with a total runtime of $O|V||E|$.

10.2 All-pairs shortest paths

We've only focused on single source shortest paths, but what if we wanted to find shortest paths from every vertex to every other vertex? If the graph was all positive weights, then we could run Dijkstra's algorithm $|V|$ times, giving us $O(|V||E|\log|V|)$ time. If the graph has both negative and positive weights, we could run Bellman-Ford's algorithm $|V|$ times, givng us $O(|V|^2|E|)$ time. We know that the output will be in $O(|V|^2)$ time, so we can't expect to do any better than $O(|V|^2)$ time in the algorithm.

To solve this problem, we'll use a combination of both of the algorithms we've seen before.

10.2.1 Johnson's algorithm

As previously discussed, we can't just slap a constant on all of the weights because this messes up the correct answer for the shortest path. So instead, we're going to associate a value $h(v)$ with each vertex (this is called the **potential** of v). Now, we reweight the graph so that each edge (a, b) has weight $weight(a, b) + h(a) - h(b)$. Note that reweighting does not affect negative cycles.

To choose h, we add one additional vertex called s to the graph, and add an edge from it to every vertex in the graph. Each of the edges has weight 0. Note that this cannot add any new negative cycles.

Now, for each v, let $\delta(s, v)$ denote the length of the shortest path from s to v. The key observation is that $\delta(s, v) \leq \delta(s, u) + weight(u, v)$. Rearranging this, we get $weight(u, v) + \delta(s, u) - \delta(s, v)$. Looks familiar, right? Now, the weight of every edge becomes: $weight(u, v) + \delta(s, u) - \delta(s, v) \geq 0$.

We can now piece together the whole algorithm.

1. Add one additional vertex s to the graph.
2. For every vertex, add an edge (s, v) with weight 0.
3. Run Bellman-Ford with source s. If there's a negative weight cycle, it will be detected in this step.
4. Reweight each edge so that $weight'(u, v) = weight(u, v) + h(u) - h(v)$. Where $h(v) = \delta(s, v)$.
5. For each vertex, run Dijkstra's algorithm.
6. For each pair of vertices, compute $\delta(u, v) = \delta'(u, v) + h(v) - h(u)$.

Is this really faster? Well, here are the breakdowns of the time complexity of each step:

- $O(1)$
- $O(|V|)$
- $O(|V||E|)$
- $O(|E|)$
- $O(|V||E| \log |V|)$
- $O(|V|^2)$

So the overall time complexity is $O(|V||E| \log |V|)$ time.

11 Linear Programming

11.1 Standard form

Here are some of the rules you can use to turn a linear function into standard form:

- Convert all $\cdots \to \min$ to $-\cdots \to \max$
- Convert all $=$ relations to $\leq$ and a $\geq$
- Convert all $\geq$ to negated $\leq$.
- Remove non-negativity. That is, provided that a variable doesn't have to be non-negative, then split it up into x_1, x_2, where the x_1 is the positive part and x_2 is the negative.

- Finally, put it into a matrix.

12 Flow

12.1 Ford-Fulkerson Algorithm

```
Input: A (weakly connected) flow network (G,c,s,t)
Output: A flow f with no augmenting paths

construct flow f with f(e) = 0 for every edge
construct residual graph Gf
while Gf contains a path P from s to t do
    Find P using (depth/breadth)-first search
    Update f <- Push(G,c,s,t,f,P)
    Update Gf on edges of P
end while
return f
```

Runtime: $O(v(f^*)|E|)$ where f^* is the maximum flow.

12.2 Edmonds-Karp Algorithm

Same as Ford-Fulkerson, but we always pick P with few edges as possible. Basically we only need to use **breadth-first** search on the residual graph to find the augmenting paths rather than depth-first search.

Runtime: $O(|V||E|^2)$.

13 Complexity Theory

13.1 The class NP

NP is the class of all decision problems such that there is a polynomial-time algorithm Verify that if and only if x is a yes instance of the decision problem, then there is some bit string w (called a **witness**) with `verify(x,w) = yes`

P is the class of all decision problems with a polynomial-time algorithm.

13.2 Cook Reduction

A cook reduction from X to Y is a polynomial-time algorithm for a problem X which, given an input of size s makes poly calls to an oracle for y whose input instances are all of size poly. We write this: $X \leq_c Y$

The point of this is that if there **is** a poly-time algorithm for Y, then there's one for X too. If there isn't one for X, then there's none for Y either.

13.3 Cook-Levin Theorem

The **SAT** problem asks: 'is the input CNF formula satisfiable?'. That is, CNF being **conjunctive normal form** ($x \vee (\neg y) \vee z \mapsto x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$).

The **Cook-Levin Theorem** states that every problem in NP is Cook-reducible to SAT. So, if there's a polynomial algorithm for SAT, then there's a polynomial algorithm for **every** problem in NP, so P = NP.

13.4 NP-Completeness

We say a problem is **NP-hard** if any problem in NP is Cook-reducible to it, and **NP-complete** if it is also in NP. So, SAT is NP-complete.

Important: By the Cook-Levin Theorem, a problem is NP-hard if and only if SAT reduces to it. This is how we normally prove it.

- If a problem is NP-Hard, there's probably no poly-time algorithm for it.
- Even if there is one, you won't be able to find it.
- In practice, almost every problem either has a poly-time algorithm or is NP-hard.

Basically, NP-Hardness means that we need to find an alternative to the thing you were trying to prove.

13.5 Church-Turing Thesis

The **Church-Turing Thesis** says that any computable function is computable with a *Turing machine*.

The **Strong Church-Turing Thesis** says that any function computable in **polynomial time** is computable in **polynomial time** using a *Turing machine*.

This changes the problem to modeling the entire start to finish, step by step. Where with some Turing machine, we can compute a CNF formula that is satisfiable iff after running the machine on some input, it halts with the output `yes`.

The long story is that we check consistency with an AND of **many** clauses that are in the form:

$$[P_{i,t} = 1 \wedge S_{3,r=1} \wedge C_{i,T} = 0 \Rightarrow C_{i,r+1} = 1]$$