Advanced Algorithms - Notes

Dom Hutchinson

March 19, 2020

Contents

1 Hashing

Definition 1.1 - *Dictionary*
A *Dictionary* is an abstract data structure which stores (*key*, *value*) pairs, with *key* being unique.
A *Dynamic Dictionary* can perform the following operations

Operation	Description
`add(k,v)`	Add the pair `(k,v)`.
`lookup(k)`	Return `v` if `(k,v)` is in dictionary, `NULL` otherwise.
`delete(k)`	Remove pair `(k,v)`, assuming `(k,v)` is in dictionary.

A *Static Dictionary* can only perform lookups, after it has been built.

Operation	Description
`lookup(k)`	Return `v` if `(k,v)` is in dictionary, `NULL` otherwise.

Proposition 1.1 - *Implementing a Dictionary*
Many data structures can be used to implement a *Dictionary*.
These include, but not limited to:

i) Linked lists.

ii) Binary Search, (2,3,4) & Red-Black Trees.

iii) Skip lists

iv) van Emde Boas Trees.

Remark 1.1 - *Motivation for Hashing*
None of the implementations of a *Dictionary* suggested in **Proposition 1.1** achieves a $O(1)$ run-time complexity in the worst case for all operations. To achieve this we introduce *Hashing*.

Definition 1.2 - *Hash Function*
A *Hash Function* takes in object's key and returns a value which is used to index the object in a *Hash Table*.
Let S be the set of all possible keys a hash function can recieve & m be the number of indexes in its *Associated Hash Table*. Then

$$h : S \to [m]$$

N.B. We want to avoid cases where $h(x) = h(y)$ for $x \neq y (collisions)$.

Remark 1.2 - *Hashing functions assign items to indices with a geometric distribution*

Remark 1.3 - *Avoiding Collisions in Hashing*
When indexing n items to m indicies using a *Hash Function* we only avoid *Collisions* if $m \gg n$.

Definition 1.3 - *Hash Table*
A *Hash Table* is an abstract data structre which extends the *Dictionary* in such a way that time complexity is reduced.
A *Hash Table* is comprise of an array & a *Hash Function*. The *Hash Function* maps an object's key to an index in the array. If multiple objects have the same *Hash Value* then a *Linked List* is used in that index, with new objects added to the end of the *Linked List* (Called *Chaining*).

Proposition 1.2 - *Time Complexity for Dictionary Operations in a Hash Table*
By building a *Hash Table* with *Chaining* we achieve the following time complexities for *Dictionary* operations

Operation	**Worst Case Time Complexity**	Comments
`add(k,v)`	$O(1)$	Add item to the end of *Linked List* if necessary.
`lookup(k)`	O(length of chain containing `k`)	We might have to search through the whole *Linked List* containing `k`.
`delete(k)`	O(length of chain containing `k`)	Only $O(1)$ to perform the actual deletion, but need to find `k` first.

Theorem 1.1 - *True Randomness*
Consider n fixed inputs for a *Hash Table* with m indices. (*i.e.* any sequence of n `add`/`lookup` /`delete` operations).
Pick a *Hash Function*, h, at random from a set of all *Hash Functions*, $H := \{h : S \to [m]\}$. Then

$$\mathbb{E}(\text{Run-Time per Operation}) = O\left(1 + \frac{n}{m}\right)$$

N.B. The expected run-time per operation is $O(1)$ if $m \gg n$.

Proof 1.1 - *Theorem 1.1*
Let x & $y \in S$ be two distincy keys & T be a *Hash Table* with m indexes.
Define $I_{x,y} \begin{cases} 1 & h(x) = h(y) \\ 0 & \text{otherwise} \end{cases}$.
We have $\mathbb{P}(h(x) = h(y)) = \frac{1}{m}$.
Therefore

$$\begin{aligned} \mathbb{E}(I_{x,y}) &= \mathbb{P}(I_{x,y} = 1) \\ &= \mathbb{P}(h(x) = h(y)) \\ &= \tfrac{1}{m} \end{aligned}$$

Let N_x be the number of keys stored in H that are hashed to $h(x)$.
Note that $N_x = \sum_{k \in T} I_{x,k}$.
Now we have that

$$\mathbb{E}(N_x) = \mathbb{E}\left(\sum_{k \in T} I_{x,k}\right) = \sum_{k \in H} \mathbb{E}(I_{x,k}) = n\frac{1}{m} = \frac{n}{m}$$

□

Remark 1.4 - *Why not hash to unique values*
Suppose we want to define a *Hash Function* which maps each key in S to a unique position in the *Hash Table*, T. This requires m unique positions, which in turn require $\log_2 m$ bits for each key. This is an unreasonably large amount of space.

Proposition 1.3 - *Specifying the Hash Function*
Consider a set of *Hash Functions*, $H := \{h_1, h_2, \dots\}$.
When we initialise a *Hash Table* we choose a hash function $h \in H$ at random and then proceed only to use h when dealing with this specific *Hash Table*.

Remark 1.5 - *Randomness in Hashing*
All the randomness in *Hashing* comes from how we choose the *Hash Function* & not from how the *Hash Function* itself runs.

Definition 1.4 - *Weakly Universal Set of Hashing Functions*
Let $H := \{h | h : S \to [m]\}$ be a set of *Hashing Functions*.

H is *Weakly Universal* if for any chosen $x, y \in S$ with $x \neq y$

$$\mathbb{P}(h(x) = h(y)) \leq \frac{1}{m} \text{ when varying } h(\cdot)$$

when h is chosen uniformly at random from H.

Theorem 1.2 - *Expected Run time for Weakly Universal Set*
Consider n fixeds to a *Hash Table*, T, with m indexes.
Pick a *Hash Function*, H, from a *Weakly Universal Set* of *Hash Functions*, H.

$$\mathbb{E}(\text{Run-Time per Operation}) = O(1) \text{ for } m \geq n$$

N.B. Proof is same as for *True Randomness*.

Proposition 1.4 - *Constructing a Weakly Universal Set of Hash Functions*
Let $S := [s]$ be the set of possible keys & p be some prime greater than s[1].
Choose some $a, b \in [0, p-1]$ & define

$$\begin{aligned} h_{a,b}(x) &= \underbrace{[\ (ax+b) \bmod p\ ]}_{\text{spread values over [0,p-1]}} \underbrace{\bmod m}_{\text{causes collisions}} \\ H_{p,m} &= \{h_{a,b}(\cdot) : a \in [1, p-1],\ b \in [0, p-1]\end{aligned}$$

N.B. $H_{p,m}$ is a *Weakly Universal Set* of *Hashing Functions*.
N.B. Different values of a & b perform differently for different data sets.

Remark 1.6 - *True Randomness vs Weakly Universal Hashing*

- For both *True Randomness* & *Weakly Universal Hashing* we have that when $m \geq n$ the expected `lookup` time in the *Hash Table* is $O(1)$.
- Constructing a *Weakly Universal Set* of *Hash Functions* is generally easier.

Theorem 1.3 - *Longest Chain - True Randomness*
If *Hashing Function* h is selected uniformly at random from all *Hashing Functions* to m indicies. Then, over m inputs we have

$$\mathbb{P}(\exists \text{ a chain length} \geq 3\log_2 m) \leq \frac{1}{m}$$

Proof 1.2 - *Theorem 1.3*
This problem is equivalent to showing that if we randomly throw m balls into m bins the probabiltiy of having a bin with at least $3\log_2 m$ balls is at most $\frac{1}{m}$.
Let X_1 be the number of valls in the first bin.
Choose any k of the M balls, the probabiltiy that all of these K balls go into the first bin is $\frac{1}{m^k}$.
By the *Union Bound Theorem* we have

$$\mathbb{P}(X_1 \geq k) \leq \binom{m}{k}\frac{1}{m^k} \leq 1k!$$

Applying the *Union Bound Theorem* again we have

$$\mathbb{P}(\text{at least 1 bin recieves at least } k \text{ balls}) \leq m\mathbb{P}(X_1 \geq k) \leq \frac{m}{k!}$$

[1] There is a theorem that $\forall\ n\ \exists\ p \in [n, 2n]$ st p is prime.

Observe that

$$\begin{aligned} k! &> 2^{k-1} \\ \text{Let } k &= 3\log_2 m \\ \implies k! &> 2^{(3\log_2 m - 1)} \\ &\geq 2^{2\log_2 m} \\ &\geq (2^{\log_2 m})^2 \\ &= m^2 \end{aligned}$$

Thus, setting $k = 3\log_2 m$ means

$$\frac{m}{k!} \leq \frac{1}{m} \text{ for } m \geq 2$$

□

Theorem 1.4 - *Longest Chain - Weakly Universal Hashing*
Let *Hashing Function* h be picked uniformly at random from a *Weakly Universal Set* of *Hashing Functions*.
Then, over m inputs

$$\mathbb{P}(\exists \text{ a chain length} \geq 1 + \sqrt{2m}) \leq \frac{1}{2}$$

N.B. This is a poor bound.

Proof 1.3 - *Theorem 1.4*
Let $x, y \in S$ be two keys and define $I_{x,y} \begin{cases} 1 & h(x) = h(y) \\ 0 & \text{otherwise} \end{cases}$.
Let C be a random variable for the total number of collision (*i.e.* $C = \sum_{x,y \in H, \mathbf{x}<y} I_{x,y}$).
Using *Linearity of Expectation* and that $\mathbb{E}(I_{x,y}) = \frac{1}{m}$ when h is *Weakly Universal*

$$\mathbb{E}(C) = \mathbb{E}\left(\sum_{x,y \in H,\ x<y} I_{x,y}\right) = \sum_{x,y \in H,\ x<y} \mathbb{E}(I_{x,y}) = \binom{m}{2}\frac{1}{m} \leq \frac{m}{2}$$

By *Markov's Inequality*

$$\mathbb{P}(C \geq m) \leq \frac{\mathbb{E}(C)}{m} \leq \frac{1}{2}$$

Let L be a random variable for the length of the longest chain in H.
Then, $C \leq \binom{L}{2}$. Now

$$\mathbb{P}\left(\frac{(L-1)^2}{2} \geq m\right) \leq \mathbb{P}\left(\binom{L}{2} \geq m\right) \leq \mathbb{P}(C \geq m) \leq \frac{1}{2}$$

By rearranging, we have that

$$\mathbb{P}(L \geq 1 + \sqrt{2m}) \leq \frac{1}{2}$$

1.1 Perfect Hashing

Remark 1.7 - *Motivation*
The *Hashing Schemes* discussed in the previous part perform well in the best & average cases but not necessarily in the worst cases (as they can have really long longest chains).

Definition 1.5 - *Static Perfect Hashing*
A *Perfect Static Hashing Scheme* is a scheme that produces a *Hash Table* where `lookup` has time complexity $\in O(1)$, even in the worst case. However this *Hash Table* is static so we cannot perform `insert` or `delete` after the table has been produced.

N.B. FKS Hashing Scheme is a *Perfect Static Hashing Scheme.*

Definition 1.6 - *FKS Hashing Scheme*
Below is an algorithm for the *FKS Hashing Scheme*

Algorithm 1: FKS Hashing Scheme

require: $n\{\#\text{ insertions}\}, T\{\text{Table with } n \text{ entries}\}$
Insert all n into T using h
while *Collisions in* $T \geq n$ **do**
    Rebuild T using a new h.
Let $n_i = |T[i]|$.
for $i \in [1, n]$ **do**
    Insert items of $T[i]$ into new table T_i of size n_i^2 using h_i.
    while *Collisions in* $T_i \geq 1$ **do**
        Rebuild T_i using a new h_i.
return T

N.B. $\mathbb{P}(\text{Collisions in } T_i \geq 1) \leq \frac{1}{2}$ and *N.B.* $\mathbb{P}(\text{Collisions in } T \geq n) \leq \frac{1}{2}$ so we expect to have to build each table twice.

Remark 1.8 - *If n items are mapped to the same index this counts as $\binom{n}{2}$ collision.*

Proposition 1.5 - *FKS Hashing Scheme -* `lookup`
Below is an algorithm for `lookup(x)` in the *Hash Tables* produced by the *FKS Hashing Scheme*

Algorithm 2: FKS - `lookup(x)`

require: T {main table}, $\{T_1, \dots, T_m\}$ {sub-tables}, x {key}
Compute $i = h(x)$.
Compute $j = h_i(x)$.
return $T_i[j]$

N.B. This runs in $O(1)$ time.

Proof 1.4 - *FKS Hashing Scheme - Space Requirements*
In the *FKS Hashing Scheme* the main table T requires space $O(n)$ and each sub-table T_i requires space $O(n_i^2)$, where $n_i = |T[i]|$.
Storing each task function, h_i requires space $O(1)$.
Thus the total space used is1

$$O(n) + \sum_i O(n_i^2) = O(n) + O\left(\sum_i n_i^2\right)$$

We know there are $\binom{n_i}{2}$ collisions in $T[i]$ so there are $\sum_i \binom{n_i}{2}$ collisions in T.
We know there are at most n collisions in T so

$$\sum_i \frac{n_i^2}{4} \leq \sum_i \binom{n_i}{2} < n \implies \sum_i n_i^2 < 4n$$

Thus

$$O(n) + O\left(\sum_i n_i^2\right) = O(n)$$

Proof 1.5 - *FKS Hashing Scheme - Expected Construction Time*
The expected construction time for the main table, T, is $O(n)$.

The expected construction time for reach sub-table, T_i, is $O(n_i^2)$ where $n_i := |T[i]|$.
Thus

$$\begin{aligned} \textit{expect}(\text{construction time}) &= \mathbb{E}\left(\text{construction time of } T + \sum_i \text{construction time of } T_i\right) \\ &= \mathbb{E}\text{construction time of } T) + \mathbb{E}\left(\sum_i \text{construction time of } T_i\right) \\ &= O(n) + \sum_i O(n_i^2) \\ &= O(n) + O\left(\sum_i n_i^2\right) \text{ see Proof 1.4} \\ &= O(n) \end{aligned}$$

Proposition 1.6 - *FKS Hashing Scheme - Properties*

- Has no collisions.
- `lookup` takes $O(1)$ time in worst-case.
- Uses $O(n)$ space.
- Can be build in $O(n)$ expected time.

1.2 Cuckoo Hashing

Remark 1.9 -
If our consruction has the property that $\forall\ x, y \in S$ with $x \neq y$ the probabiltiy that x and y are in the same bucket is $O\left(\frac{1}{m}\right)$, then for any n operations the expected run-time is $O(1)$ per operation.

Definition 1.7 - *Cuckoo Hashing*
In *Cuckoo Hasing* we use two hash functions, h_1 & h_2, to produce a single hash table.
When we `add` a value x to the hash table we place it in position $h_1(x)$. If there is already a value, y, already in this position then we move that value, y, to its alternative positione. We keep moving values until each value is in its position. If it is not possible (*i.e.* we have found a cycle) then we change h_1 & h_2 for new hash functions and rehash all the values.
This is formally descibed in the algorithm below

Algorithm 3: Cuckoo Hashing - Insert

```
require: {x_1, ..., x_n} {stream of keys}, T {Table with m entries}
choose h_1, h_2 for i ∈ [1, n] do
    pos = h_1(x).
    checked = [].
    while T[pos] not empty do
        if x ∈ checked then rehash;
        checked append x. y = T[pos].
        T[pos] = x.
        pos =alternative position for y.
        x = y.
    T[pos] = x.
return T
```

N.B. Rehash involves chooseing two new hash functions h_1 & h_2 are reinserting all keys, $\{x_1, \ldots, x_n\}$ into the table.

Proposition 1.7 - *Cuckoo Hashing Scheme - Properties*

i) An `add` takes *amortised exepcted* time $O(1)$.

ii) Every `lookup` and every `delete` has time complexity $O(1)$ in the worst-case.

iii) The space requirement is $O(n)$ where n is the number of keys stored.

Remark 1.10 - *Assumptions in Cuckoo Hashing*
In *Cuckoo Hashing* we make the following assumptions

i) h_1 and h_2 are indepenent.
i.e. $h_1(x)$ says nothing about $h_2(x)$, and visa-versa.

ii) h_1 and h_2 are truly random.
i.e. They map to each entry in the hash table with uniform probability.

iii) Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ time in the worst-case.

Definition 1.8 - *Cuckoo Graph*
A *Cuckoo Graph* is an interprettation of a *Hash Table* using *Graph Theory.*
Each vertex of the graph is an entry in the hash table and for each x_i we add an undirected-edge between $h_1(x_i)$ and $h_2(x_i)$.
If any cycles occur in a *Cucko Graph* then we know construction will fail for that pair of hash functions as no stable scenario can occur.
The length of the longest path tells us the time for the longest insert.

Theorem 1.5 - *Probability of Long Paths in Cuckoo Graphs*
Let m be the size of a hash table & n the number of entries we wish to insert Fr any pair of positions i and j, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from i to j with length $l \geq 1$ is at most $\frac{1}{c^l m}$.

Proof 1.6 - *Theorem 1.5*
TODO

Proof 1.7 - *Probability of a path between two positions in a Cuckoo Graph*
If a path exists from i to j, ther emust be a shortest path from i to j.
Therefore we can use **Theorem 1.5** and the *Union Bound* over all possible paths to show the probability of a path from i to j existing is at most

$$\sum_{l=1}^{\infty} \frac{1}{c^l m} = \frac{1}{m} \sum_{l=1}^{\infty} \frac{1}{c^l} = \frac{1}{m} \frac{1}{c-1} = O\left(\frac{1}{m}\right)$$

Definition 1.9 - *Buckets*
We say that two keys x & y are in the same *bucket* iff there exists a path from $h_1(x)$ to $h_1(y)$ in a *Cuckoo Graph.*
Note that this implies there is a path from $h_1(x), h_2(y)$; $h_2(x), h_1(y)$ and $h_2(x), h_2(y)$ as there are edges $(h_1(x), h_2(x))$ & $(h_1(y), h_2(y))$.

Remark 1.11 - *The time for an operation on x is bounded by the number of items in its bucket.*

Proposition 1.8 - *Probabiltiy of being in the same Bucket*
For $x, y \in S$ with $x \neq y$ the probability that they are in the same bucket is at most

$$\sum_{l=1}^{\infty} \frac{1}{c^l m} = \frac{1}{m} \sum_{l=1}^{\infty} \frac{1}{c^l} = \frac{1}{m} \frac{1}{c-1} = O\left(\frac{1}{m}\right)$$

If the size of a hash table is $m \geq 2cn$ then the expeceted time per operation is $O(1)$.
Further, `lookup`s take $O(1)$ time in the worst case.

Proposition 1.9 - *Probability of Rehashing*
The probabiltiy that a rehashing occurs in *Cuckoo Hashing* is equal to the probability of the *Cuckoo Graph* having a cycle.
A cycle is a path from x to x, via some intermidiary vertices.
Thus the probability that x is involved in a cycle is

$$\sum_{l=1}^{\infty} \frac{1}{c^l m} = \frac{1}{m(c-1)}$$

by **Proof 1.7**.
Thus the probabiltiy that there is at least one cycle in the whole hash table is

$$m\frac{1}{m(c-1)} = \frac{1}{c-1}$$

Proposition 1.10 - *Construction Time - Cuckoo Hashing*
Consider the result in **Proposition 1.9** when $c = 3$.
The probability of a rehashing occuring is $\frac{1}{2}$.
Thus we expected only one rehash to be necessary. The the expected time for a rehash is $O(n)$ then the expeceted construction time for the table is $O(n)$.
Therefore the *amortised expeced* time for rehashes over n insertions is $O(1)$ per insertion.
N.B. Checking for a cycle in a graph takes $O(n)$ time.

2 Bloom Filters

Definition 2.1 - *Bloom Filter*
A *Bloom Filter* is a data structure which is designed to be a space efficient way of storing a set S.
Bloom Filters support only the following operations

Operation	Description
`insert(k)`	Insert the key `(k)`.
`member(k)`	Returns `true` if `(k)`$\in S$, no otherwise.

Note that there is no way to remove objects from a *Bloom Filter* & you cannot ask which keys are in the *Bloom Filter*, only whether a particular key is.
N.B. Bloom Filters are meant to be used in cases where the size of the sample space is much larger than the number of keys being stored.

Remark 2.1 - *Motivation*
Bloom Filters can be used to build a blacklist of unsafe URLS. Whenever a new unsafe URL, `k`, is discovered we add it to the filter, `insert(k)`.
Whenever we want to visit to a new URL, `k`, we can query whether it is in the bloom filter, `member(k)`, and if we are returned *yes* then it is blocked.

Remark 2.2 - *Randomness in Bloom Filter*
A *Bloom Filter* is randomised in such a way that `member(k)` will sometimes return *yes* when in fact `k`$\notin S$. However it will never return *no* if `k`$\in S$.
N.B. The amount of space used by a *Bloom Filter* depends on the failure rate we allow it.

Proposition 2.1 - *Usefullness of Bloom Filter*
For a *Bloom Filter* `insert(k)` & `member(k)` both run in $O(1)$ and it requries $O(n)$ bits of space to store up to n keys.

Proposition 2.2 - *Building a Bloom Filter - Array*
A naïve approach to building a *Bloom Filter* is to use an array.
Suppose we have sample space $1, 2, \dots, |U|$ for key values.
We can store a set by mainting a bit string B where $B[k] = 1$ if $k \in S$ and $B[k] = 0$ otherwise.
e.g.

1	2	3	4	5	6
0	1	0	0	0	1

here $|U| = 6$ and $S = \{2, 6\}$.
The *Bloom Filter* operations take $O(1)$ time but the array is $|U|$ long.

Proposition 2.3 - *Building a Bloom Filter - Hash Table*
Let $h : U \to [1, m]$ be a hash function.
We can implement a *Bloom Filter* using a *Hash Table* is we define that performing `insert(k)` sets $B[h(k)] = 1$ and `member(k)` returns `true` if $B[h(k)] == 1$, and `false` otherwise.
Using a hash function means the bit string B being maintained is much shorter than if we used the *Array* approach in **Proposition 2.2**, however we now have to deal with collisions & thus false-positive results from `member(k)`.
N.B. false-positives occur `member(k)` if $\exists k' \in S$ st $h(k') = h(k)$.

Remark 2.3 - *Reducing Collisions - Bloom Filter as Hash Table*
To ensure a low probability of collisions each operation, we pick the hash function h at random (Note that we don't change h after it is set).

Proposition 2.4 - *Probability of False Positive - Bloom Filter as Hash Table*
Suppose we have inserted n keys into the *Bloom Filter* and now we perform `member(k)` for $k \notin S$.
The bit string B contains at most n 1s among its m positions.
By definition the hash function assigns values uniformly $\mathbb{P}(h(i) = j) = \frac{1}{m}$ for $k \in [1, m]$.
Thus the probability of a false-positive is $\mathbb{P}(B[h(k)] = 1) \leq \frac{n}{m}$.
If we choose m to be $100n$ then $\mathbb{P}(B[h(k)] = 1) \leq .01$.

Proposition 2.5 - *Bloom Filter Complexities*
Both `insert(k)` and `member(k)` run in $O(1)$ and the structure uses m bits.
If we want a .01 false positive rate then $100n$ bits are required.

Proposition 2.6 - *Building a Bloom Filter - Proper*
Again we are maintaining a bit string B of length $m < |U|$.
Let $h_1, \dots, h_r$ be r hash functions which map $U \to [1, m]$ uniformly at random.

- `insert(k)` sets $B[h_i(k)] = 1$ for <u>all</u> $i \in [1, r]$.
- `member(k)` returns `true` iff $B[h_i(k)] = 1 \ \forall \ i \in [1, r]$.

Proposition 2.7 - *Probability of False Positive for* ***Proposition 2.6***
Assume that we have inserted n keys into the *Bloom Filter* and that we are performing `member(k)` for $k \notin S$.
This checks whether $B[h_i(k)] = 1 \ \forall \ i \in [1, r]$.
This is question whether r randomly chosen bits of B all equal 1 due to hash functions being uniformly random.
As there are n keys in the *Bloom Filter* at most nr bits of B are set to 1.
So $\frac{nr}{m}$ is the proportion of bits set to 1 in B.

So the probability that a ranomly chosen bit is 1 is $\leq \frac{nr}{m}$.
Thus the probability that r randomly chosen bits are all equal to 1 is $\leq \left(\frac{nr}{m}\right)^r$.

Proposition 2.8 - *Minimising False Positive Rate*
By differentiating $\leq \left(\frac{nr}{m}\right)^r$ we find it is minimised for $r = \frac{m}{ne}$.
If we substitute this back in we get a false positive rate of at most $\left(\frac{1}{e}\right)^{\frac{m}{ne}} \approx (0.69)^{\frac{m}{n}}$.
So for a failure rate of 1% we set $m \approx 12.52n$.
This is much better than $m = 100n$ required for **Proposition 2.4**.

Remark 2.4 - *Limitations of Hashing*
Using *Hashing* for dynamic dictionaries has a few draw backs

i) Randomness.

ii) Amortisation, *i.e.* expected complexities are averaged over n.

iii) Inflexible, hard to add new operations.

3 van Emde Boas Trees

Proposition 3.1 - *New operations for Dynamic Dictionary*
Below are some operations which are often used as an extension to the *Dynamic Dictionary* in **Definition 1.1**.

Operation	Description
`predecessor(k)`	Return the pair `(x,v)` in the dictionary with the largest key, `x`, such that `x`$\leq$`k`.
`successor(k)`	Return the pair `(x,v)` in the dictionary with the smallest key, `x`, such that `x`$\geq$`k`.

N.B. Hashing-based dynamic dictionaries are not suited to these operations.

Proposition 3.2 - *Implementing New Operations*
We could use self-balancing binary trees such as *2-3-4 Trees*, *Red-Black Trees* or *AVL Trees*. These can perform all five operations of our extended dynamic dictionary each in $O(\log n)$ worst case time and $O(n)$ space.

Definition 3.1 - *Van Emde Boas Trees*
Van Emde Boas Trees can support the five operations of our *Extended Dynamic Dictionary*. *Van Emde Boas Trees* can perform each operation in $O(\log \log u)$ time and use $O(u)$ space where u is the size of the universe.
N.B. Also known as *vEB Trees*.

Proposition 3.3 - *vEB Trees - Array*
Suppose we wishing to store the set $S \subset U$ keys with $u = |U|$.
Let A be a binary array of size u where $A[i] = 1$ iff $i \in S$.
We can use the following schema for each operation of the *Extended Dynamic Dictionary*

i) `add(x)` - Set $A[x] = 1$.

ii) `delete(x)` - Set $A[x] = 0$.

iii) `lookup(x)` - Return `true` if $A[x] = 1$, `false` otherwise.

iv) `predecessor(x)` - Increase i until `lookup(x-i)` returns `true`. Return $x - i$.

v) `successor(x)` - Increase i until `lookup(x+i)` returns `true`. Return $x + i$.

Here `add`, `delete` & `lookup` take $O(1)$ time, but `predecessor` & `successor` take $O(u)$ time. (Not great).

Proposition 3.4 - *vEB Trees - Blocks*
Let $S \subset U$ where U is a universe of size u.
Let $B_1, \dots, B_{\sqrt{u}}$ be bit vectors of length $\sqrt{u}$ which define sequential blocks of U.
Each of these vectors stores the values $\{0, \dots, \sqrt{u} - 1\}$. $B_i[j] \equiv 1$ iff $x := j + \sqrt{u} \cdot i$ is in S.
Let C be a bit vector of length $\sqrt{u}$ which summaries the blocks $B_1, \dots, B_{\sqrt{u}}$.
We have that $C[i] \equiv 1$ iff block B_i is non-empty.
With this set up be can fulfil the oeprations as follows

`add(x)`
i) Define $i := \lfloor x/\sqrt{u} \rfloor$ & $j := x - i \cdot \sqrt{u}$.
ii) Set $B_i[j] = 1$ & $C[i] = 1$.

`delete(x)`
i) Define $i := \lfloor x/\sqrt{u} \rfloor$ & $j := x - i \cdot \sqrt{u}$.
ii) Set $B_i[j] = 0$.
iii) Check if B_i is empty. If so set $C[i] = 0$.

`successor(x)`
i) Define $i := \lfloor x/\sqrt{u} \rfloor$ & $j := x - i \cdot \sqrt{u}$.
ii) Look for successor to x in B_i, starting at $B_i[j]$.
iii) If no successor exists
(a) Find succesor to j in C, starting at $C[j + 1]$.
(b) Return smallest element in that block.

N.B. In practive the blocks are implemented as a single bit-vector.

Remark 3.1 - *`successor(x)` for Blocks Implementation*
In the block implementation of *vEB Trees* we have $T(u) = 3T(\sqrt{u}) + O(1)$ as we make 3 recursive calls to `successor()`.
This gives time complexity $T(u) \in O(\sqrt{u})$.
We can improve this by reducing the number of recursive calls made.
To do this we restructure the data structure. (See **Proposition 3.5**).

Proposition 3.5 - *vEB Trees - Proper*
Consider the same set up as in **Proposition 3.4**.
Here the data structure is augment such that for each block B_i we store the `max` and `min` indexes used in it.
By doing this we can perform `successor(x)` as follows

i) Define $i := \lfloor x/\sqrt{u} \rfloor$ & $j := x - i \cdot \sqrt{u}$.
ii) If $j >$ `max`$_i$: # No successor in block
(a) $k =$`successor`$_C(i)$ # Find block with next value
(b) return B_k[`min`$_k$]
iii) Else: # Successor is in block
(a) return `successor`$_{B_i}(j)$

We are now only ever making a single recurisve call, giving us $T(u) = T(\sqrt{u}) + O(1)$.
Thus $T(u) \in O(\log \log u)$.
N.B. Each block & summary can be defined recursively.

Example 3.1 - *vEB Trees - Proper*
Consider a universe of size $u = 16$.
The following is an example of the datat structure that would be used

$$C = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \quad \texttt{min} = \begin{pmatrix} 0 \\ - \\ - \\ 1 \end{pmatrix} \quad \texttt{max} = \begin{pmatrix} 0 \\ - \\ - \\ 3 \end{pmatrix}$$

N.B. Each row relates to a single block.

Remark 3.2 - *Time Complexity of* `add` *for vEB Trees*
`add` contain a single recursive call.
We have that $T(u) = T(\sqrt{u}) + O(1)$.
By the Master's Theorem we have that $T(u) = O(\log \log u)$.
N.B. This is the same for all operations.

Remark 3.3 - *Space Complexity of vEB Trees*
Let $Z(u)$ be the space use by a *vEB Tree* over a universe of size u.
We have that

$$Z(u) = (\sqrt{u} + 1)Z(\sqrt{u}) + O(1) \implies Z(u) = O(u)$$

4 Orthogonal Range Searching

Definition 4.1 - *nD Range Searching Data Structure*
An *n-Dimensional Range Searching Data Structure* stores m distinct n-dimensional data points and supports a single operation

Operation	Description
`lookup`$(l_1, u_1, \ldots, l_1, u_1)$	Return all stored points $(z_1, \ldots, z_n)$ where $l_1 \leq z_1 \leq u_1$ and $\ldots$ and $l_n \leq z_n \leq u_n$.

Proposition 4.1 - *1D Range Searching Data Structure*
Consider an array of stored values $[x_1, \ldots, x_n]$.
Make a balanced binary tree by recursively taking the midpoint of the array as a new node and then partitioning the values around.
`lookup`(l_1, u_1) can be performed as follows

i) Find the successor of l_1. Takes $O(\log n)$ time.

ii) Find the predecessor of u_1. Takes $O(\log n)$ time.

iii) Find the path between the successor and predecessor.

iv) For each node on path:

(a) Consider its left & right subtree in turn (if they exist).

(b) If root of subtree is in $[l_1, u_1]$ then return all values in subtree.

This implementation of `lookup`(l_1, u_1) takes $O(\log n + k)$ time, where k is the number of points to be returned, and $O(n)$ space and $O(n \log n)$ prep-time (to construct the tree).
N.B. Runtime depends on the size of the output.

Proposition 4.2 - *2D Range Searching Data Structure - Naïve*
Below is a naïve approach for performing `lookup(lx,ux,ly,uy)`

i) Find all the points with $l_x \leq x \leq u_x$ using **Proposition 4.1**.

ii) Find all the points with $l_y \leq y \leq u_y$.

iii) Find the points in both lists.

This approach has run time $O(\log n + k_x) + O(\log n + k_y) + O(k_x + k_y) = O(n + k_x + k_y)$ where k_x is the number of points found in step i) and k_y is the number of points found in step ii).

Proposition 4.3 - *2D Range Searching Data Structure - Proper*
For preprocessing build a balanced binary tree using the x value of each point.
Here is a better implementation of `lookup`(l_x, u_x, l_y, u_y).

i) Find the successor to l_x.

ii) Find the predeccessor to u_x.

iii) Find the path between these two points.
Each subtree on this path has the property that either <u>all</u> points in the tree have x values in the range $[l_x, u_x]$, or <u>none</u> of them do.

iv) For each subtree where <u>all</u> points have x values in the range $[l_x, u_x]$.

(a) Build a *1D Range Searching Structure* using the y-coordinates of each point in the tree.

(b) Perform `lookup`(l_y, u_y) on this subtree.

Remark 4.1 - *Time Complexity* ***Proposition 4.3***
Steps $i) - iii)$ take $O(\log n)$ time and the length of the path between the successor & predecessor is of length $O(\log n)$.
In step iv) we do $O(\log n)$ 1-D lookups. Each of these takes $O(\log n + k')$ time.
Thus the overall run-time is $O(log^2 n + k)$ as each lookup is disjoint.
N.B. Preprocessing takes $O(n \log n)$ time.

Remark 4.2 - *Space Complexity* ***Proposition 4.3***
The initial 1D structure used $O(n)$ space.
At each node we stored an array containing the points in its subtree.
The tree has depth $O(\log n)$.
Thus the total space used is $O(n \log n)$.

5 Pattern Matching

Definition 5.1 - *Exact Pattern Matching Problem*
Let T, P be strings of length n, m respectively.
The *Exact Pattern Matching Problem* is to find all occurences of P in T.
P matches at location i iff $\forall\ j \in [0, m)\ P[j] = T[i + j]$.
Proposition 5.1 - *Naïve Algorithm*
A *Naïve Algorithm* for solving the *Exact Pattern Matching Problem* is to align P with the start

of T; compare every character; return if a match has occured; slide on by one character; repeat until end of T is reached.
N.B. This takes $O(nm)$ time.

Proposition 5.2 - *Proper Algorithms*
Proper Algorithms can solve the *Exact Pattern Matching Problem* (*e.g.* KMP) in $O(n)$ time.
Note that this mean run times depends on the size of the text being analysed, not on the pattern which is being searched for.

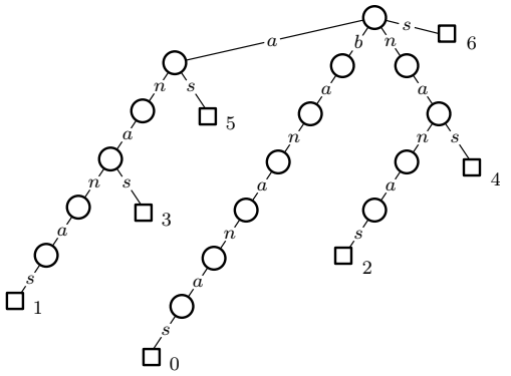
Remark 5.1 - *Preprocessing*
Since the text, T, is constant across all queries, it is acceptable to do a significant amount of preprocessing in order to speed up query times.

5.1 Suffix Trees

Definition 5.2 - *Suffix*
A *Suffix* is any substring of a string which includes the last character.

Definition 5.3 - *Atomic Suffix Tree*
An *Atomic Suffix Tree* is a tree where each edge is a labled by a character & each leaf is labelled with the index at which a given suffix starts.
Each root-to-leaf path is a suffix in the tree.

Example 5.1 - *Atomic Suffix Tree*
Below is an *Atomic Suffix Tree* for 'bananas'.



Proposition 5.3 - *Searching a Atomic Suffix Tree*
We can use an *Atomic Suffix Tree* to solve the *Exact Pattern Matching Problem.*
Given an pattern P, starting at the root, step down the path dictated by P.
If P requires a step which cannot be fulfilled then that pattern does not exist anywhere in T.
If the path of P does terminate successfully on a node: return all the leaf values in this nodes subtree.

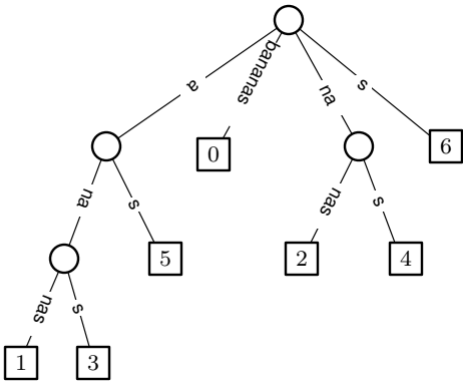
Remark 5.2 - *Runtime*
This approach has runtime complexity of $O(m)$.
It should be noted that the number of choices we have to check at each node does affect the runtime, however for constant size alphabets this is not of great concern.

Remark 5.3 - *Limitations of Atomic Suffix Tree*
Suffix Trees can have upto $\left(\frac{n}{2}+1\right)^2$ internal nodes, meaning its space requirements are $O(n^2)$ are quickly become unmanageable.

They can contain long paths.

Definition 5.4 - *Compacted Suffix Tree*
A *Compacted Suffix Tree* has the same structure as an *Atomic Suffix Tree*, the only difference being that any non-branching nodes (*i.e.* nodes with only one child) are merged into the node below and edges are now labelled with Substrings.
This means that every node has at least 2 children and we have $O(n)$ edges.
N.B. In practice we do not store the substring for each edge but rather the endpoints in T in order to save space.

Example 5.2 - *Compact Suffix Tree*
Below is an *Compact Suffix Tree* for 'bananas'.



Proposition 5.4 - *A Compact Suffix Tree does **not** always exist*
A complete *Compact Suffix Tree* does not always exist.
Suppose $T =$'*bb*' then the tree would contain a single edge and we would not successfully detect the pattern $P =$'*b*'.
This can be fixed by adding a character which does not appear in T onto the end (*e.g.* $T =$'*bb*\$').
Proposition 5.5 - *Searching a Compact Suffix Tree*
Same process as for an *Aromic Suffix Tree* except we are comparing multiple characters at a time.
Run time is $O(m + \#\text{occurences})$.

Proposition 5.6 - *Constructing a Compact Suffix Tree - Naïve*

i) Insert the suffixes one at a time, longest to shortest:

   (a) Search for the new suffix in the partial suffix tree.
   (Same technique as pattern matching).

   (b) Add a new edge and leaf for the new suffix.
   This may require the breaking of an edge into two.

This has run-time complexity $O(n^2)$.
N.B. This can be done in $O(n)$ time but is not covered on this course.

Proposition 5.7 - *Multiples Texts*
If we want to be able to search multiple texts we a character to the end of each text: this characters must be different & not appear in any of the texts.
We then append these texts and treat them as a single text.

5.2 Suffix Arrays

Definition 5.5 - *Lexicographical Sorting*
Lexicographical Sorting is a way of ordering words in alphabetical order.
Let T_1 and T_2 be two texts which we are sorting.

- Compare T_1 & T_2 character-by-character.
- Let i be the index of the first character to differ between T_1 & T_2.
- If $T_1[i] < T_2[i]$ alphabetically (or any order defined ordering) then T_1 is placed first, and visa-versa.
- If no characters differ but T_1 is shorter than T_2, then T_1 is placed first and visa-versa.

N.B. This is how words are sorted in the dictionary.

Definition 5.6 - *Suffix Array*
Let T be a text with $|T| = n$.
Lexicographic sort all n suffices of T.
We define a *Suffix Array*, A, st

$$A[i] = n - \text{length of suffix with was } i^{\text{th}} \text{ in lexicographic order}$$

Example 5.3 - *Suffix Array*
Consider the word *'bananas'*.

$n-l$	**suffix**
0	bananas
1	ananas
2	nanas
3	anas
4	nas
5	as
6	s

$n-l$	**suffix**
1	ananas
3	anas
5	as
0	bananas
2	nanas
4	nas
6	s

Thus, in this case, $A = [1, 3, 5, 0, 2, 4, 6]$.

Remark 5.4 - *Suffix arrays are much smaller than Suffix Trees*

Proposition 5.8 - *Suffix Array from Suffix Tree*
If we already have a *Suffix Tree* constructed then we can construct a *Suffix Array* by performing a depth first search on the tree and setting $A[i]$ to be the i^{th} element to be returned by this search.
N.B. Depth first search can be done in $O(n)$ time.

Proposition 5.9 - *Searching a Suffix Array*
Let T be a text we wish to search and P be the pattern we are looking for.
Let A be the *Suffix Array* of T.
Due A being ordered (lexicographically) we can perform a binary search on it to find occurences of P.
At each node we check we compare characters: if P is matched completly then we can return the index of the occurence.
N.B. Binary search can be used to find the first & last occurence of P and thus all occurences.

Remark 5.5 - *Runtime Seaching a Suffix Tree*
Comparing two strings takes $O(m)$ (where $|P| = m$) and the binary search checks $O(\log n)$ nodes (where $|T| = n$).
Thus overall runtime complexity for finding one occurence is $O(m \log n)$ and to find all occurences is $O(m \log n + \#\text{occurences})$.
N.B. This can be improved to $O(m + \log n + \#\text{occurences})$ using *LCP Queries*, which are discussed later.

Proposition 5.10 - *DC3 Method*
Let T be a text with $n = |T|$.
The *DC3 Method* is an algoirthm for building *Suffix Arrays* in $O(n)$.
First we build a *Suffix Array* for two-thirds of the text & then for the final third.

i) Construct *Suffix Array* for two-thirds ($S_{1,2}$):
   (a) Define $B_1 := \{i \in [0, n) : i \ \% \ 3 \equiv 1\}$ & $B_2 := \{i \in [0, n) : i \ \% \ 3 \equiv 2\}$.
   (b) Define $R_1 := T[1, \dots, n]$ & $R_1 := T[2, \dots, n]$.
   (c) Pad the ends of R_1 & R_2 with a filler character ('\$') until their length is a multiple of 3.
   (d) Split R_1 & R_2 into blocks of length 3.
   (e) Define $R := R_1 * R_2$ (Concatenation) & split it into blocks of length 3.
   (f) Label the blocks of R in lexicographical order. (Takes $O(n)$ time with Radix Sort).
   (g) Define R' st $R'[i]$ = position of block i in lexicographical order. (*N.B.* $|R'| = \frac{2n}{3}$)
   (h) Compute the *Suffix Array* of R'.
   N.B. This is a recursive step & we are now sorting number suffices rather than letter suffices.

ii) Relabel the *Suffix Array* of R' in terms of first index (in T) of each block. (Call this $S_{1,2}$)
N.B. This gives the *Suffix Array* of $B_1 \cup B_2$.

iii) Construct *Suffix Array* for other third (S_0):
   (a) Define $B_0 := \{i \in [0, n) : i \ \% \ 3 \equiv 0\}$.
   (b) Write each suffix which is indexed by B_0 as the first character & the position of the suffix started by the next character in $S_{1,2}$.

$$\underbrace{T[B_0[i]]}_{\text{char}} + \texttt{index\_of}(B_0[i] + 1 \texttt{ in } S_{1,2}) \equiv T[B_0[i]] + \texttt{index\_of}(B_1[i] \texttt{ in } S_{1,2})$$

   (c) Sort these pairs lexicographically ($O(n)$ time in Radix Sort).
   (d) Relabel this array in terms of the index of the first character (in T). (Call this S_0).
   N.B. This is the *Suffix Array* of B_0.

iv) Merge the *Suffix Array*s to form S:
   (a) Set $i = 0$ and $j = 0$.
   (b) Compare $S_0[i]$ with $S_{1,2}[j]$:
   If $(T[S_0[i]] < T[S_{1,21}[j]])$: Insert $S_0[i]$ into S, increment i.
   Elif $(T[S_0[i]] > T[S_{1,2}[j]])$: Insert $S_{1,2}[j]$ into S, increment j.
   Else : (*i.e.* $T[S_0[i]] \equiv T[S_{1,2}[j]]$)

If $S_0[i]+1 \in S_{1,2}$ and $S_{1,2}[j]+1 \in S_{1,2}$:
- Add the element associated to whichever of these indices occurs first in $S_{1,2}$. Increment i or j as appropriate.

Else Add the element associated to whichever of $S_0[i]+2$ & $S_{1,2}[j]+2$ occurs first in $S_{1,2}$. Increment i or j as appropriate.
N.B. These indicies are guaranteed to be in $S_{1,2}$ due to the use of *mod* 3.

(c) Repeat until all characters have been positioned.
N.B. This is very similar to *Merge-Sort Merging*.

Proposition 5.11 - *DC3 Method - Run Time*
In the *DC3 Method* we have a recursive step at i) (h) which acts of two-thirds of the data.
The construction of S_0 takes $O(n)$ time as it uses *Radix Sorting*.
The merging of S_0 & $S_{1,2}$ takes $O(n)$ time.
Thus the *DC3 Method* has run-time equation $T(n) = T\left(\frac{2}{3}n\right) + O(n) \implies T(n) \in O(n)$.

6 Range Minimum Queries

Definition 6.1 - *Range Minimum Query Problem*
Let A be an array of length n and $i, j \in [0, n)$ with $i < j$.
A *Range Minimum Query* of indices i, j, $RMQ(i, j)$, outputs the index of the samllest value in the subarray $A[i, j] \equiv \big\{A[i], A[i+1], \dots, A[j-1], A[j]\big\}$.

Remark 6.1 - *Aim*
This can be naïvely be done in $O(n^2)$ space & $O(1)$ query time if we precompute a matrix of all solutions.
We want to do this in $O(n)$ space, $O(n)$ preprocessing time & $O(1)$ query time.

Proposition 6.1 - *Solution 1 - Block Decomposition*
Let A be an array of n elements.
The *Block Decomposition Solution* suggests preprocessing a *Minimum Binary-Heap* where each node gives the value & index of the smallest value of the leaves in its subtree.

i) Define $A_1 := A$.

ii) For $i \in [1, \log_2 n]$:

(a) Let $k := 2^i$.

(b) Define A_k st $\forall\ i \in [0, \frac{n}{k}],\ A_k[i] = (x, v)$ where $v := \min(A[ik, (i+1)k])$ and x is the position of v in A.

This takes $O(n)$ space since $\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$.
Each layer requires $\frac{n}{k}$ comparisons, thus preprocessing takes $O(n)$ time.

Proposition 6.2 - *Solution 1 - Querying*
Consider performing $RMQ(i, j)$ using the heap produced by **Proposition 6.1**

i) Find the largest block which lies completely within $[i, j]$.
i.e. The node with the largest subtree which lies completely in the $[i, j]$.

ii) Keep finding the largest block which does not overlap with any previous blocks.

iii) Return the minimum of root of each of these blocks.

We will never choose 2 consequitive blocks of the same height as there will be a large block with covers them.
There will be no gaps in a layer (for the same reason), thus we choose at most 2 blocks of each height.
Thus we pick at query at most $2\log_2 n$ blocks in total.
Query time is $O(\log_2 n)$.

Remark 6.2 - *We want faster queries*

Proposition 6.3 - *Solution 2 - Precompute solutions for intervals of length power of two*
Let A be an array of n elements.
Pre-compute the solutions for every interval of length 2^i for some i. Let A_k denote the array which stores the solutions to $RMQ(i, i + (k - 1)) \ \forall \ i$, where k is a power of 2.
N.B. $A_1 := A$.
$|A_k| = n - (k - 1) \in O(n)$ and we have $\log_2 n$ arrays so total space is $O(n \log_2 n)$.
We can build A_2 from A in $O(n)$ time & A_{2k} from A_k in $O(n)$ so pre-processing takes $O(n \log_2 n)$.

Proposition 6.4 - *Solution 2 - Querying*
Consider performing $RMQ(i, j)$ using the arrays produced by **Proposition 6.3**.

i) Define $l := j - i + 1$ (The interval length).

ii) If l is a power of 2: Return $A_l[i]$. (*i.e.* Just lookup the answer).
N.B. Takes $O(1)$ time.

iii) Otherwise:

   (a) Find power of two, k, st $k \leq l < 2k$.

   (b) Perform $RMQ(i, i + k - 1)$ and $RMQ(j - k + 1, j)$.
   N.B. Each query takes $O(1)$ time.

   (c) Return minimum of these two queries.
   N.B. Takes $O(1)$ time.

Each query can be done in $O(1)$ time.

Remark 6.3 - *Solution 2 has quicker queries but requires more space & pre-processing time*

Proposition 6.5 - *Solution 3 - Low-Resolution Range Minimum Queries*
Let A be an array of n elements.
Here we replace A with a *lower-resolution* array H and store many smaller arrays which give the detail.

i) Define $\tilde{n} := \dfrac{n}{\log_2 n}$.

ii) Define $H[i] = \min(A[i \log_2 n, (i + 1) \log_2 n - 1])$ for $i \in [0, \tilde{n})$.

iii) Define sub-arrays $L_i := A[i \log_2 n, (i + 1) \log_2 n - 1]$ for $i \in [0, \tilde{n})$.
N.B. In practice we just store the start & end indices of L_i and then query A.

We can use **Proposition 6.3** to construct H.
This requires $O(\tilde{n} \log_2 \tilde{n}) = O(n)$ space & pre-processing time, and we can perform queries in $O(1)$ time.
Using **Proposition 6.3** to construct each of the L_i requires $O((\log_2 n) \log_2 \log_2 n)$ space & pre-processing time, and each can be queried in $O(1)$ time.

Thus, total space is $O(n) + O(\tilde{n}(\log_2 n)\log_2\log_2 n) = O(n\log_2\log_2 n)$ & total pre-processing time is $O(n\log_2\log_2 n)$.

Proposition 6.6 - *Solution 3 - Querying*
Consider performing $RMQ(i,j)$ using the arrays produced by **Proposition 6.5**.

i) Define $i' := \left\lceil \frac{i}{\log_2 n} \right\rceil$ & $j' := \left\lfloor \frac{j}{\log_2 n} \right\rfloor$.

ii) Query $RMQ_H(i', j')$.

iii) If $i' \neq \frac{i}{\log_2 n}$: (*i.e.* the previous query does not cover the start of the interval)

- Query $RMQ_{L_{i'-1}}(i' - i, \log_2 n)$.

iv) If $j' \neq \frac{j}{\log_2 n}$: (*i.e.* the previous query does not cover the end of the interval)

- Query $RMQ_{L_{j'+1}}(0, j - j')$.

v) Return the minimum of these (at most 3) queries.

This requires at most 3 queries, each of $O(1)$ time, so can be done in $O(1)$ time.

6.1 ± 1 Range Minimum Queries

Definition 6.2 - ± 1 *Range Minimum Query*
Let A be an array of length n with the property that $\forall\ k \in [0, n-1),\ A[k+1] = A[k] \pm 1$.
Perform a *Range Minimum Query* of this array.

Definition 6.3 - *Equivalent* ± 1 *Arrays*
Let A_1 & A_2 be ± 1 arrays of length n.
We note that the values of each index are irrelevant, only the change in value between elements matters.
Thus, A_1 & A_2 are said to be *Equivalent* if they have the same pattern of value changes.
i.e. A_1 is equivalent to A_2 iff $\forall\ i, j\ RMQ_{A_1}(i,j) = RMQ_{A_2}(i,j)$.

Proposition 6.7 - *Number of Equivalent* ± 1 *Arrays*
Consider ± 1 arrays of length n.
We can denote the value change pattern of a ± 1 array as a binary string of length $n-1$ where the i^{th} digit represents the value change from $A[i]$ to $A[i+1]$.
Let 1 denote $+1$ and 0 denote -1.
Thus there are 2^{n-1} unique ± 1 arrays of length n.

Proposition 6.8 - *Range Minimum Queries for* ± 1 *Arrays*
Let A be a ± 1 array of length n.
Split A into $\tilde{n} := 2\frac{n}{\log_2 n}$ blocks.
Each block is of length $n\frac{\log_2 n}{2n} = \frac{1}{2}\log_2 n$ and is a ± 1 array, thus there are at most $\sqrt{n}$ different blocks.
Precompute the *Range Minimum Query* of each permutation & store in a table.
Store type of each block in a *Low-Resolution* array, H.
Querying requires a single query of H & a single query of the precomputed table of solutions.
Both take $O(1)$ time, so query time is $O(1)$.
This requires $O(n)$ space & pre-processing time.
N.B. This is similar to the *Low-Resolution* solution given in **Proposition 6.5**.

7 Lowest Common Ancestor

Definition 7.1 - *Ancestor*
Let T be a tree & i be a node in T.
Ancestors of i in T are the nodes which lie on the path from i to the tree-root.

Definition 7.2 - *Lowest Common Ancestor*
Let T be a tree & i, j be nodes in T.
The *Lowest Common Ancestor* of i & j in T, $LCA(i, j)$, is the node furthest from the root which is also an ancestor of both i & j.

Remark 7.1 - *Aim*
We want to find a solution which works in $O(n)$ space, $O(n)$ pre-processing time & $O(1)$ query time.

Definition 7.3 - *Euler Tour*
Let T be a *Tree*.
An *Eurler Tour* of T is a depth-first search, where we record backtracking as well.
At each node take the leftmost edge which has not already been traversed, otherwise backtrack & repeat until all edges have been traversed.

Proposition 7.1 - *Length of Euler Tour*
Let T be a *Tree* with n nodes.
Since T has n nodes, it has $n-1$ edges.
In an *Euler Tour* each edge is traversed twice (once in each direction) & trivially start at the root.
Thus an *Euler Tour* has length $2(n-1)+1 = 2n-1$.

Remark 7.2 - *Euler Tours start & end on the root of the tree*

Proposition 7.2 - *Solution 1 - Using Range Minimum Queries*
Let T be a tree with n nodes & i, j be nodes in T.

i) Label each node of the tree incrementally with $[0, n-1]$ (left-to-right, then down a layer).

ii) Follow an *Euler Tour* of T. At each node:

- Append its value on the end of N.
- Append its depth on the end of D.

iii) Process D for *Range Minimum Queries* (Using **Proposition 6.5**).

This produces two arrays, N & D, each of length $2n-1$.
Using **Proposition 6.5** for iii) means that pre-processing time & total-space is

$$\underbrace{O(n \log_2 \log_2 n)}_{\text{RMQs}} + \underbrace{O(n)}_{\text{Euler Tour}} = O(n \log_2 \log_2 n)$$

Proposition 7.3 - *Solution 1 - Quering*
Let T be a *Tree* with n nodes and i, j be nodes in T.
Let N & D be the arrays produced by **Proposition 7.2**.
Consider making query $LCA(i, j)$ on T.

i) Find an occurence of i & j in N.

ii) Let i', j' be the indixes of these occurences.

iii) Return $N[\underbrace{RMQ_D(i', j')}_{\text{An Index}}]$.
N.B. $RMQ(\cdot, \cdot)$ takes $O(1)$ time.

Each of these steps takes $O(1)$ time. Overall query-time is $O(1)$.

Proposition 7.4 - *Correctness of Solution 1*
Consider *Euler Tours* recursively.
Let x be a node at depth x with children $c_1, \dots, c_k$.
We see the *Euler Tour* has recursive structure as

N	x	$\vdash$Tour of c_1 $\dashv$	x	$\vdash \cdots \dashv$	x	$\vdash$Tour of c_k $\dashv$	x
D	d	$\vdash$Tour of c_1 $\dashv$	d	$\vdash \cdots \dashv$	d	$\vdash$Tour of c_k $\dashv$	d

Claim that $RMQ(i, j) = y$ iff $LCA(i, j) = y$.
Let i', j' be any position of i, j in N.

i) Suppose i, j are in the same subtree of y.
Then y is not in interval i', j'.
Thus cannot be returned by $RMQ(i, j)$.

ii) Suppose i, j are not in subtrees of y.
Then y may be in the interval i', j'.
Since theses are not subtrees of y then a node higher than y will be traversed.
The index of this node will be returned by $RMQ(i, j)$ over that of y.

iii) Suppose i, j are in different subtrees of y.
Then y is in the interval i', j'.
Since these are both subtrees of y, y is the highest element in the trees.
Thus the index of y is returned by $RMQ(i, j)$.

Thus the claim holds. □

Proposition 7.5 - *Solution 2 - ± 1 Range Minimum Query*
Notice that the depth array, D, produced by **Proposition 7.2** is a ± 1 array.
Thus we can use the results from **Section 6.1**.
Specifically, replace D with a *Low-Resolution* array as described in **Proposition 6.8**.
Query time remains as $O(1)$.
Pre-processing time & total space is now $\underbrace{O(n)}_{\text{RMQs}} + \underbrace{O(n)}_{\text{Euler Tour}} = O(n)$.
This fulfils the aims in **Remark 7.1**.

7.1 Solving Range Minimum Queries using Leach Common Ancestor

Definition 7.4 - *Cartesian Tree*
Let A be an array.
The *Cartesian Tree* of is defined recursively

i) Set $\min(A)$ as root.

ii) Partition A around $\min(A)$ into A_1 & A_2.

iii) Repeat for A_1 & A_2, making the resulting trees subtrees.

N.B. This can be done in $O(n)$, but not covered.

Remark 7.3 - *$LCA(i,j)$ in the cartesian tree of $A \equiv RMQ_A(i,j)$*
This gives $O(n)$ space & pre-processing time, and $O(1)$ query time for the *Range Minimum Query* problem, using the solution to *Least Common Ancestors* in **Proposition 7.5**.

8 Approximation Algorithms

Remark 8.1 - *Look at NP-Completeness*

Definition 8.1 - *Approximation Algorithm*
Let A be an algorithm.
A is an *α-Approximation Algorithm* for a problem, P, if

i) A runs in *Polynomial-Time*.

ii) A always outputs a solution with a value, s, within an α factor of the *Optimal* solution.

The measure depends on the type of problem P is

- If P is a *Maximisation Problem*, then $\dfrac{\text{Optimal}}{\alpha} \leq s \leq \text{Optimal}$.
- If P is a *Minimisation Problem*, then $\text{Optimal} \leq s \leq \alpha \cdot \text{Optimal}$.

8.1 Constance Factor Approximations

Definition 8.2 - *Constant Factor Approximation Algorithm*
A *Constant Factor Approximation Algorithm* is an *Approximation Algorithm* where the approximation value, α, has a constant value, independent of the input.
Remark 8.2 - *Below are some examples of Constant Factor Approximations*

8.1.1 Bin Packing Problem

Definition 8.3 - *BinPacking Problem*
Consider an infinite number of bins of size 1 & a set of items with size $\in [0,1]$.
The *BinPacking Problem* is to find the fewest number of bins required to store this set of items.
This is *NP-Hard*.
The *Decision* version of this problem is "*can you pack the items into k bins?*".
The *Decision* version is *NP-Complete*.

Proposition 8.1 - *Solution 1 - First Fit*
We can approximate the *BinPacking Problem* as

If item i fits into bin j: pack it; next item;
else: start new bin.

Let I be the sum of the weights of the items, $fill(i)$ be the sum of the item sizes in bin i & s be the number of bins used.
Using the technique above we observe that $fill(2i-1) + fill(2i) > 1 \ \forall \ 2i \in [1,s]$.

$$\begin{aligned} &\implies \quad \left\lfloor \tfrac{s}{2} \right\rfloor < \textstyle\sum_{2i \in [1,s]} fill(2i-1) + fill(2i) \leq I < \texttt{Opt} \\ &\implies \quad s \leq 2 \cdot \texttt{Opt} \end{aligned}$$

Thus, we know that this solution uses no more that twice the optimal number of bins.
This is a 2-*Approximation Algorithm* of the *BinPacking Problem.*
N.B. This runs in $O(n)$ time.

Proposition 8.2 - *Solution 2 - First Fit Decreasing*
Consider sorting the items in *Non-Decreasing* order, then fit each item into the left-most bin it will fit in.
This runs in $O(n^2)$ time and uses s bins.
Consider bin $j = \lceil \frac{2s}{3} \rceil$

Case 1 Suppose bin j contains an item of size $> \frac{1}{2}$.
Then each bin $j' \leq j$ contains at item of size $> \frac{1}{2}$.
Thus at least $j = \lceil \frac{2s}{3} \rceil$ bins are required.
Thus, $s \leq \frac{3}{2} \cdot \texttt{Opt}$.

Case 2 Suppose bin j only contains times of size $\leq \frac{1}{2}$.
When the first item was packed into j all bins $j' \in (j, s]$ were empty and all unpacked items had size $\leq \frac{1}{2}$.
Thus each bin $j' \in (j, s)$ contains at least 2 items and bin s contains at least one item.
This means at least $2(s - j) + 1$ items don't fit into bins $j' \in [1, j)$.
Thus $\sum$item weights$> \min\{j - 1, 2(s - j) + 1\} \geq \lceil \frac{2s}{3} \rceil - 1$.
Since $\sum$item weights$\leq \texttt{Opt} \implies \lceil \frac{2s}{3} \rceil - 1 \leq \texttt{Opt}$.
Thus, $s \leq \frac{3}{2}\texttt{Opt}$.

Thus, in both cases, this solution is a $\frac{3}{2}$-*Approximation Algorithm.*

8.1.2 Job Scheduling on Parallel Machines

Proposition 8.3 - *Scheduling Jobs on Parallel Machines Problem*
Consider having m identical machines & n jobs which vary in time taken.
Our goal is to minimise the time taken to process all the jobs.
N.B. This problem is *NP-Hard.*

Remark 8.3 - *Assessing Success*
Let J be the set of all jobs & J_i be the set of jobs assigned to machine i.
Define $L_i := \sum_{j \in J_i} t_j$ to be the time for machine i to execute all its jobs.
Thus the total time to execute all jobs in J is $\sup L_i$.
Thus we want to minimise $\sup L_i$.

Proposition 8.4 - *Solution 1 - Greedy Algorithm*
Let J be the set of all jobs, with $n := |J|$ and there be m identical machines.

Assign job j to the machine i with the current smallest load.

Naively this takes $O(nm)$ time to produce a schedule, but can be done in $O(n \log m)$ time using a *Priority Queue.*
N.B. This is an *Online Algorithm.*

Proposition 8.5 - *Solution 1 - Approximation Value*
Let $\texttt{Opt}$ denote the time taken by the optimal schedul & T denote the time taken by the greedy schedule produced by **Proposition 8.4**.
Note that $\texttt{Opt} \geq \sup t_j$ where t_j is the time taken by job j.
And, $\texttt{Opt} \geq \frac{1}{m} \sum_j t_j$ since at least one machine will have a load $L_i \geq \frac{1}{m} \sum_j t_j$.
Consider the machine i with the greatest load (*i.e.* $L_i \equiv T$).

Let j be the last job assigned to i.
We know that when j was assigned to i, i had the lighestest load (*i.e.* $L_i - t_j \leq L_k \; \forall \; k \in [1, m]$).
Summing every machine we get

$$\begin{aligned} m(L_i - t_j) &\leq \sum_{k=1}^{m} L_k \\ \implies L_i - t_j &\leq \frac{1}{m} \sum_{k=1}^{m} L_k \\ &\leq \texttt{Opt} \text{ by second fact} \end{aligned}$$

Since $t_j \leq \texttt{Opt}$ we have

$$\begin{aligned} T &= L_i \\ &= L_i - t_j + t_j \\ &\leq \texttt{Opt} + \texttt{Opt} \\ &= 2 \; \texttt{Opt} \\ \implies T &\leq 2 \; \texttt{Opt} \end{aligned}$$

Thus **Proposition 8.4** is a 2-Approximation.

Proposition 8.6 - *Solution 2 - Longest Processing Time*
Let J be the set of all jobs, with $n := |J|$, let t_j be the time to execute job j and let there be m identical machines.

Sort jobs in non-increasing order.
Put job j into machine i with the smallest current load.

This takes $O(n \log n)$ time to produce a schedule.

Proposition 8.7 - *Solution 2 - Approximation Value*
Let J be the set of all jobs, with $n := |J|$, let t_j be the time to execute job j and let there be m identical machines. Let $\texttt{Opt}$ denote the time taken by the optimal schedul & T denote the time taken by the schedule produced by **Proposition 8.6**.
Note that if $n \leq m$ then **Proposition 8.6** is optimal.
Else (if $n > m$):
At least one of the machines is assigned two jobs (each taking at least t_{m+1} time).
Thus $\texttt{Opt} \geq 2t_{m+1}$.

Let i be the machine with the greatest load. (*i.e.* $T \equiv L_i$).
Let j be the last job assigned to i. We have $L_i - t_j \leq \texttt{Opt}$.
Assume $n > m$ and that i has assigned at least two jobs. Then $L_i - t_j > 0$.
We have that $t_j \leq t_{m+1} \leq \frac{1}{2}\texttt{Opt}$ (*i.e.* the time for last job is no more than the $(m+1)^{\text{th}}$ job's) by previous fact.
Therefore

$$\begin{aligned} T &= L - i \\ &= L_i - t_j + t_j \\ &\leq \texttt{Opt} + \tfrac{1}{2}\texttt{Opt} \\ &= \tfrac{3}{2}\texttt{Opt} \\ \implies T &\leq \tfrac{3}{2}\texttt{Opt} \end{aligned}$$

Thus **Proposition 8.6** is a $\frac{3}{2}$-Approximation.

8.1.3 k-Centres

Proposition 8.8 - *k-Centres Problem*
Consider n points in a 2D space.

We want to insert k centres into the space st the maximum distance from any point to the closest centre is minimised.
N.B. Distance is measured as the *Euclidean Distance* $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Proposition 8.9 - *Solution 1 - Greedy Algorithm*
Let $\{x_1, \dots, x_p\}$ be some points in 2D space.
Here is a *Greedy Algorithm* which offers a solution to **Proposition 8.8**

i) Pick any point to be a centre.

ii) Pick the point which has the greatest distance to its nearest centre to be a new centre.

iii) Repeat ii) until k centres have been picked.

This takes $O(nk)$ time.

Proposition 8.10 - *Solution 1 - Approximation Factor*
Let C be the set of centres produced by **Proposition 8.9** & $C_{\texttt{Opt}}$ be the optimal set of centres.
Let r be the greatest distance of a point from its nearest centre in C
and $r_{\texttt{Opt}}$ be the greatest distance of a point from its nearest centre in $C_{\texttt{Opt}}$.

Case 1 *No $c_i, c_{i'} \in C$ which have the same closest centre ,c_j, in $C_{\texttt{Opt}}$.*
Then all points must be within $r_{\texttt{Opt}}$ of a centre in $C_{\texttt{Opt}}$.
Thus all points are at most $2r_{\texttt{Opt}}$ from a centre in C. (opposite side of circle radius $r_{\texttt{Opt}}$)

Case 2 *$\exists\ c_i, c_{i'} \in C$ which have the same closest centre ,c_j, in $C_{\texttt{Opt}}$.*
Thus s_i is at most $2r_{\texttt{Opt}}$ from $s_{i'}$. (opposite side of circle radius $r_{\texttt{Opt}}$)
Assume that c_i was made a centre after $c_{i'}$, WLOG.
Thus c_i was the furthest from any exisiting centre in C when it was added.
Thus before adding s_i all centres in C were at most $2r_{\texttt{Opt}}$ from another centre in C.
Thus $r \leq r_{\texttt{Opt}}$.

Thus **Proposition 8.9** is 1 2-Approximation

8.2 Fully Polynomial Time Approximation Schemes

Definition 8.4 - *Pseudo-Polynomial Time Algorithm*
An algorithm is *Pseudo-Polynomial Time Algorithm* if it runs in *Polynomial Time* when all the numbers are integers $\leq n^c$ for some constant c.

Definition 8.5 - *Polynomial Time Approximation Scheme*
A *Polynomial Time Approximation Scheme*, PTAS, for a problem P is a family of algorithms, A, st $\forall\ \varepsilon > 0$ there is an algorithm $A_\varepsilon \in A$ st A_ε is a $(1+\varepsilon)$-Approximation algorithm for P.
N.B. A *PTAS* does not have to have a time complexity which is polynomial in $1/\varepsilon$.

Definition 8.6 - *Fully Polynomial Time Approximation Scheme*
A *PTAS* is called a *Fully PTAS*, FPTAS, if it has a time complexity which is polynomial in $1/\varepsilon$ (as well as in n).

8.2.1 Subset Sum

Definition 8.7 - *Subset Sum Problem*
Let S be a set of positive intergers & $t \in \mathbb{N}$ be a target value.
The *Decision Problem* of the *Subset Sum Problem* is to find if there is a subset, $S' \subseteq S$, st

$\sum_{s\in S'} = t$.
The *Optimisation Problem* of the *Subset Sum Problem* is to find the size of the largest subset, $S' \subset S$, st $\sum_{s\in S'} \leq t$. (Note the inequality).
N.B. The *Decision Problem* is *NP-Hard* and the *Optimisation Problem* is *NP-Complete*.

Proposition 8.11 - *Solution 1 - Exact Solution for Decision Problem*
Let S be a set of m positive intergers & define $S_i := S[1, i]$ and $t \in \mathbb{N}$ be a target value.
Let L_i be the set of all total sum values of $S' \subseteq S_i$ which are no larger than t.
Note that the largest subset of S with total-sum value $\leq t$ is the largest number in L_m.
We can compute L_i from L_{i-1} as

$$L_i = L_{i-1} \cup \{x \in L_{i-1} : x + S[i] \leq t\}$$

There are no duplicates in L_i so $|L_i| \leq t$.
We can now use the following algorithm

i) Let $L_0 = \{0\}$.

ii) For $i \in [1, m]$:

- Define $L_i = L_{i-1} \cup \{x \in L_{i-1} : x + S[i] \leq t\}$.

iii) Return $\sup L_m$.

The overall time-complexity is $O(mt) = O(n^{c+1})$.
Thus this a *Pseudo-Polynomial Time* algorithm.

Proposition 8.12 - *Solution 2 - PTAS*
Let S be a set of m positive intergers & define $S_i := S[1, i]$ and $t \in \mathbb{N}$ be a target value.
Let L_i be the set of all total sum values of $S' \subseteq S_i$ which are no larger than t.
We want to construct a *trimmed* version of L_i, L'_i, which is a subset of L_i and whose length is polynomial in the input length (*i.e.* $|L'_i| \leq n^c$ for some c).
We condition that $\forall\ y \in L_i\ \exists z \in L'_i$ which is almost as big as y.
We define the process $\texttt{Trim}(\cdot, \cdot)$.

$$\texttt{Trim}(L_i, \delta)\text{: } L_i[j] \in L'_i \text{ iff } L_i[j] > (1 + delta) \cdot \texttt{prev}$$

where `prev` is the previous value we included in L'_i.
Now we have an algorithm

i) Let $L'_0 = \{0\}$ and $d = \frac{1}{2m}\varepsilon$.

ii) For $i \in [1, m]$:

- Compute $U = L'_{i-1} \cup \{x \in L'_{i-1} : x + S[i] \leq t\}$.
- Define $L'_i := \texttt{Trim}(U, \delta)$.

iii) Return $\sup L'_m$.

This algorithm does throw away possible subsets but will always output a valid subset (but it may not be the largest).

Theorem 8.1 -
Consider the setup in **Proposition 8.12**.

$$\forall\ y \in L_i,\ \exists\ z \in L'_i \text{ st } \frac{y}{(1+\delta)^i} \leq z \leq y.$$

Proof 8.1 - *Theorem 8.1*
Proof by induction.
Base Case - $L_0 = L'_0 = \{0\}$.
Inductive Case: Assume **Theorem 8.1** holds for $i-1$.
As $y \in L_i$ we have that either $y \in L_{i-1}$ or $(y - S[i]) \in L_{i-1}$:

$(y \in L_{i-1})$ then $\exists\ x \in L'_{i-1}$ st $\dfrac{y}{(1+\delta)^{i-1}} \leq x \leq y$ by the *Inductive Hypothesis.*
By the definition of `Trim` $\exists\ z \in L'_i$ st $z \leq x \leq z \cdot (1+\delta)$.
$\implies z \leq x \leq y$ and $\dfrac{y}{(1+\delta^i)} \leq \dfrac{x}{1+\delta} \leq z$.
i.e. $\exists\ z \in L'_i$ st $\dfrac{y}{(1+\delta)^i} \leq z \leq y$, as required.

$(y - S[i] \in L_{i-1})$ Very similar to above.

Proposition 8.13 - *Solution 2 - Is a PTAS*
Using **Theorem 8.1** with $i := m$ and $\delta := \frac{1}{2m}\varepsilon$ we have that

$$\forall\ y \in L_m,\ \exists\ z \in L'_m \text{ st } \frac{y}{(1+\frac{\varepsilon}{2m})^m} \leq z \leq y.$$

We have that $\texttt{Opt} \in L_m$ so $\exists\ z \in L'_m$ st $\dfrac{\texttt{Opt}}{(1+\frac{\varepsilon}{2m})^m} \leq z \leq \texttt{Opt}$ For **Proposition 8.12** to be a PTAS we need to show that $(1+\frac{\varepsilon}{2m})^m \leq 1+\varepsilon$ for $\varepsilon \in (0,1]$.

$$\begin{aligned}\left(1+\frac{\varepsilon}{2m}\right)^m &\leq e^{\varepsilon/2} \\ &\leq 1 + \frac{\varepsilon}{2} + \left(\frac{\varepsilon}{2}\right)^2 \text{ since } e^x := \textstyle\sum_{i=0}^{\infty} \dfrac{x^i}{i!} \\ &\leq 1+\varepsilon\end{aligned}$$

Thus the output of **Proposition 8.12** is some z with

$$\begin{aligned}&\frac{\texttt{Opt}}{1+\varepsilon} \leq \frac{\texttt{Opt}}{(1+\frac{\varepsilon}{2m})^m} \leq z \leq \texttt{Opt} \\ \implies\ &\frac{\texttt{Opt}}{1+\varepsilon} \leq z \leq \texttt{Opt}\end{aligned}$$

Thus **Proposition 8.12** is a valid PTAS.

Proposition 8.14 - *Solution 2 - Run-Time*
Assessing the run-time of **Proposition 8.12** amounts to assessing the size of L'_i.
By the definition of `Trim` we have that for any successive elements $z, z' \in L'_i$

$$\frac{z'}{z} \geq 1 + \delta = 1 + \frac{\varepsilon}{2m}$$

Further, $\forall\ z \in L'_i,\ z \leq t$. Thus, L'_i contains at most $O(\log_{1+\delta} t)$ elements.
We have

$$\begin{aligned}|L'_i| &= \log_{1+\delta} t \\ &= \frac{lnt}{\ln(1+\frac{\varepsilon}{2m}}} \\ &\leq \frac{2m(1+\frac{\varepsilon}{2m})\ln t}{\varepsilon} \text{ since } \ln(1+x) > \tfrac{x}{x+1} \\ &= O\left(\frac{m\log t}{\varepsilon}\right)\end{aligned}$$

Thus the algorithm runs in

$$O(m|L'_i|) = O\left(m^2\frac{\log t}{\varepsilon}\right) = O\left(n^3\frac{\log n}{\varepsilon}\right)$$

8.3 Asymptotic Polynomial Time Approximation Schemes

Definition 8.8 - *Asymptotic Polynomial Time Approximation Schemes*
An *Asymptotic Polynomial Time Approximation Schemes*, APTAS, for a problem P is a family of algorithms A where

$$\exists c > 0 \text{ st } \forall\ \varepsilon > 0,\ A_\varepsilon \in A \text{ runs in polynomial-time and ouputs a solution } s \text{ with}$$

- $\texttt{Opt} \leq s \leq (1+\varepsilon)\texttt{Opt} + c$ for minimisation problems.
- $\dfrac{\texttt{Opt}}{(1+\varepsilon)} \leq s \leq \texttt{Opt} + c$ for maximisation problems.

Note that an APTAS does not have to runs polynomial in $\frac{1}{\varepsilon}$.

Definition 8.9 - *Fully Asymptotic Polynomial Time Approximation Schemes*
A *Fully Asymptotic Polynomial Time Approximation Schemes*, AFPTAS, is an FPTAS which runs polynomially in $\frac{1}{\varepsilon}$

8.3.1 Partition

Definition 8.10 - *Partition Problem*
Let S be a set of positive integers & $t \in \mathbb{N}$ be a target value.
The *Partition Problem* is a special case of the *Subset Sum Problem* where $t := \frac{1}{2}\sum_{s\in S} s$.
Thus the *Decision Problem* is whether there is a subset $S' \subseteq S$ st $\sum_{s'\in S'} s' = \frac{1}{2}\sum_{s\in S} s$.
Equivalently, can we pack S into two bins of size $\frac{1}{2}\sum_{s\in S} s$.
N.B. This problem is *NP-Complete*.

Proposition 8.15 - *Approximating Partition problem as BinPacking*
Here is a solution to the *Partition Problem* by approximating the *BinPacking Problem*.
We covert the *Partition Problem* into a *BinPacking Problem* by dividing all items & bin sizes by t.
Now all bins have size 1 and items $in(0, 1]$.
If the optimal number of bins, `Opt`, is 2 then the answer to the *Partition Decision Problem* is `true`.

Proposition 8.16 - *Solution 1 - Bin Packing*
Suppose we have n items which take one of c different sizes & at most d items fit in each bin, for some constants c, d.
We can describe any packing of items into a single bin by its type (the type determines num items in bin).
There are $i \in [0, c]$ items of any size & d different sizes. Thus

$$\#\text{ types} \leq (c+1)^d$$

Ignoring rearrangements of bins we have $j \in [0, n]$ bins of each type. Thus

$$\#\text{ packings} \leq (n+1)^{(c+1)^d}$$

However, not all of these are valid.
We can check each of the different packings to check if they are valid & return the one with the fewest bins.
This takes $O(n \cdot (n+1)^{(c+1)^d})$ time & exactly solves the *BinPacking Problem*.

Proposition 8.17 - *APTAS for BinPacking*
Suppose we have n items which take one of c different sizes & at most d items fit in each bin, for some constants c, d.

i) Remove all small items ($\leq \frac{\varepsilon}{2}$) so only $c = \frac{2}{\varepsilon}$ of the remaining large ($> \frac{\varepsilon}{2}$) items will fit into a single bin.

ii) Divide the items into $k = \left\lfloor n\frac{\varepsilon^2}{2} \right\rfloor$ a constant number of groups.
The sizes of items in each group are then rounded up to match the size of the largest member (and the largest group is removed).
This will leave only $d = \frac{4}{\varepsilon^2}$ different item sizes.

iii) Use the polynomial time algorithm to optimally pack the remaining special case.
N.B. This takes $O\left(n(n+1)^{(\frac{4}{\varepsilon^2}+1)^{2/\varepsilon}}\right)$ time.

iv) Reverse the grouping from ii) and greedily pack all the small items remvoed in i)

Proposition 8.18 - *Packing Large Items*
Each large item is $> \frac{\varepsilon}{2}$ so at most $\frac{2}{\varepsilon}$ bins are required.

Proposition 8.19 - *ii) - Reducing the number of items sizes*
Given a set of items, order then in non-increasing order & split them into continuous groups of k items (the last group may have less items).
Define S' to be the set of these items but each item is rounded to the maximum size in its group and does not include the largest group.
This means S' contains $\frac{n}{k}$ distinct items.
By the way we define k, it is big enough st $\frac{n}{k} \leq \frac{4}{\varepsilon^2}$ (a constant).
Note that $\texttt{Opt}_{S'} \leq \texttt{Opt}_S \leq \texttt{Opt}_{S'} + k$ by **Proof 8.2**.

Setting $k := \lfloor n(\frac{\varepsilon^2}{2}) \rfloor$ and S to be the set of 'large items'.
Then $k \leq \varepsilon \cdot \texttt{Opt}_S$ since each of the n items in S is $\geq \frac{\varepsilon}{2}$.
$\implies \texttt{Opt}_{S'} + k \leq (1+\varepsilon)\texttt{Opt}_S$.
$\implies \texttt{Opt}_S \leq \texttt{Opt}_{S'} \leq (1+\varepsilon)\texttt{Opt}_S$.

Proof 8.2 - $\texttt{Opt}_{S'} \leq \texttt{Opt}_S \leq \texttt{Opt}_{S'} + k$

- If S can be packed into b bins, then S' can be packed into b bins.
Each item from S is replaced by an item from the next group down. This item is no larger than it.
Since S' does not have the largest group of S, the elements in the smallest group of S are discarded. So the packing is valid & $\texttt{Opt}_{S'} \leq \texttt{Opt}_S$.

- If S' can be packed into b bins, then S can be packed into $b + k$ bins.
This is because we can replace each item in S' with a smaller item from S.
This leaves the largest group in S to be added, this group is of size k thus at most k bins are required.
$\texttt{Opt}_S \leq \texttt{Opt}_{S'} + k$.

N.B. Each of these transformations takes polynomial time.

Theorem 8.2 - *iv)*
Let $\varepsilon \in [0, 1]$ and suppose we are given a packing of the items $s \in S$ with $s > \frac{\varepsilon}{2}$ (large items) into b bins.
Then, we can pack all items in S into either b or $(1+\varepsilon)\texttt{Opt} + 1$ bins (in polynomial time).
This is done by placing small items anywhere but not starting an extra bin unless we have to.
$(1+\varepsilon)\texttt{Opt} + 1$ bins are required iff all the bins in the given packing (except possibly the last) is $1 - \frac{\varepsilon}{2}$ full.

0 Reference

0.1 Definitions

Definition 0.1 - *Amortised Expected*
Amortised Expected is a term for the complexity of something. It describes the total complexity to execute a sequence of instructions, divided by the number of instructions.
e.g. If n instructions take $O(n)$ time to execute completley, then the *amortised expected* time is $O(1)$.

Definition 0.2 - *Decision Problem*
A *Decision Problem* is a problem which is answered using either *yes* or *no*, only.

Definition 0.3 - *Greedy Algorithm*
A *Greedy Algorithm* is an algorithm which makes locally optimal choices at each stage, with the intent of finding a global optimum.

Definition 0.4 - *Online Algorithm*
An *Online Algorithm* is an algorithm which can process its input in the order it is fed into the algorithm without having the entire input available at the start.
This means you can keep passing more information to the algorithm.

Definition 0.5 - *Offline Algorithm*
An *Offline Algorithm* requires the whole input before it beings processing. This algorithm then needs to totally rerun if you want to increase the input size.

0.2 Probability

Definition 0.6 - *Sample Space,* Ω
A *Sample Space* is the set of possible outcomes of a scenario. A *Sample Space* is not necessarily finite.
e.g. Rolling a dice $\Omega := \{1, 2, 3, 4, 5, 6\}$.

Definition 0.7 - *Event*
An *Event* is a subset of the *Sample Space*.
The probability of an *Event*, A, happening is

$$\mathbb{P}(A) = \sum_{x \in A} \mathbb{P}(x)$$

Definition 0.8 - *Disjoint Events*
Let A_1 & A_2 be events.
A_1 & A_2 are said to be *Disjoint* if $A_1 \cap A_2 = \emptyset$.

Definition 0.9 - σ*-Field,* $\mathcal{F}$
A *Sigma Field* is the set of possible events in a given scenario.
A *Sigma Field* must fulfil the following criteria

i) $\emptyset, \Omega \in \mathcal{F}$.

ii) $\forall\ A \in \mathcal{F} \implies A^c \in \mathcal{F}$.

iii) $\forall\ \{A_1, \dots, A_n\} \subseteq \mathcal{F} \implies \bigcup_{i=1}^{n} A_i \in \mathcal{F}$.

Definition 0.10 - *Probability Measure,* $\mathbb{P}$
A *Probability Measure* maps a *σ-Field* to $[0,1]$ which satisfies

i) $\mathbb{P}(\emptyset) = 0$ & $\mathbb{P}(S) = 1$; and,

ii) If $\{A_1, \dots, A_n\} \subseteq \mathcal{F}$ are pair-wise disjoint then $\mathbb{P}\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} \mathbb{P}(A_i)$. [$\sigma$-Additivity]

Definition 0.11 - *Random Variable*
A *Random Variable* is a function from the sample space, S, to the real numbers, $\mathbb{R}$.

$$X : S \to \mathbb{R}$$

The probability of a *Random Variable*, X, taking a specific value x is found by

$$\mathbb{P}(X = x) = \sum_{\{a \in \Omega : X(a) = x\}} \mathbb{P}(a)$$

Definition 0.12 - *Indicator Random Variable*
An *Indicator Random Variable* is a *Random Variable* which only ever takes 0 or 1 and is used to indicate whether a particular event has happened (1), or not (0).

$$\mathbb{E}(I) = \mathbb{P}(I = 1)$$

Definition 0.13 - *Expected Value,* $\mathbb{E}$
The *Expected Value* of a *Random Variable* is the mean value of said *Random Variable*

$$\mathbb{E}(X) := \sum_{x} x\mathbb{P}(X = x)$$

Theorem 0.1 - *Linearity of Expected Value*
Let $X_1, \dots, X_n$ be random variables. Then

$$\mathbb{E}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathbb{E}(X_i)$$

Theorem 0.2 - *Markov's Inequality*
Let X be a non-negative random variable. Then

$$\mathbb{P}(X \geq a) \leq \frac{1}{a}\mathbb{E}(X) \quad \forall\ a > 0$$

Theorem 0.3 - *Union Bound*
Let $A_1, \dots, A_n$ be *Events*. Then

$$\mathbb{P}\left(\bigcup_{i=1}^{n} A_i)\right) \leq \sum_{i=1}^{n} \mathbb{P}(A_i)$$

N.B. This in an equality if the events are disjoint.

Proof 0.1 - *Union Bound*
Define *Indicator RV* I_i st

$$I_i := \begin{cases} 1 & A_i \text{ happened} \\ 0 & \text{otherwise} \end{cases}$$

Define *Random Variable* $X := \sum_{i=1}^{n} I_i$ (the number of events that happened).
Then

$$\begin{aligned}
\mathbb{P}\left(\bigcup_{i=1}^{n} A_i)\right) &= \mathbb{P}(X > 0) \\
&\leq \mathbb{E}(X) \text{ by Markov's Inequality} \\
&= \mathbb{E}\left[\sum_{i=1}^{n} I_i\right] \\
&= \sum_{i=1}^{n} \mathbb{E}[I_i] \\
&= \sum_{i=1}^{n} \mathbb{P}(I_i = 1) \\
&= \sum_{i=1}^{n} \mathbb{P}(A_i 1)
\end{aligned}$$

□

0.3 NP-Completeness

Definition 0.14 - *NP*
NP is the set of *Decision Problems* which we can validate the answer of in polynomial time.

Definition 0.15 - *NP-Hard*
Let A be a decision problem.
A is *NP-Hard* if $\forall\ B \in NP$, B can be reduced to A in polynomial time.
N.B. This means we can solve B using A.

Definition 0.16 - *NP-Complete*
Let A be a decision problem.
A is *NP-Complete* if

- A is in *NP*.
- A is *NP-Hard*.

Remark 0.1 - *If we can solve an NP-Complete problem we can solve all NP problems*

Definition 0.17 - *Weakly NP-Complete*
We say an *NP-Complete* problem is *Weakly NP-Complete* if there is a *Pseudo-Polynomial Time* algorithm for it.

Definition 0.18 - *Strongly NP-Complete*
We say an *NP-Complete* problem is *Strongly NP-Complete* if it is still *NP-Complete* when all numbers are integers $\leq n^c$ for some constant c.