Informatics Notes

Albert Smith

Contents

Chapter 1

General notes

- When for loops are used, they are exclusive of the upper bound / follow the C behaviour unless otherwise noted.
- Complexities of algorithms are given in the following format: $O(a, b)$, where a is the time complexity and b the space complexity. (Please see the Mathematics section for an explanation of Big-O notation)
- Complexities refer to the complexity of the solution given here, not necessarily the best possible complexity of any algorithm found to date.
- *log* is log_2 unless otherwise noted.

Chapter 2

Mathematics

2.1 Set / Group theory

2.1.1 Unary / binary operators

A unary operator is an operation that is performed on a single element of a set S. For example, negation can be a unary operator: $-x$.

Binary operators, on the other hand, are performed on pairs of elements. Examples include: $a + b$, $a - b$, $a \times b$, a/b, $gcd(a, b)$, $a * b$, where $*$ represents an arbitrary operation.

We say that a set S is *closed* under an operation $*$ if $a * b \in S \; \forall a, b \in S$.

2.1.2 Properties of binary operators

Some properties that binary operators can have are: - *Associativity*: $(a + b) + c = a + (b + c)$ (by extension this is equal to $(a + b + c)$) - *Commutativity*: $(a + b) = (b + a)$ - *Idempotence*: $a + a = a$. Note that the 'equals' operator as we use it normally does not have this property; an example is the *min* and *max* functions. - *Distributivity*: For distributivity we need to define another operator, $*$. We say that $*$ distributes over $+$ iff $a + (b * c) = (a * b) + (a * c)$.

2.1.3 Existence of an identity

$e \in S$ is an identity of $*$ iff $\forall a \in S$: $a * e = e * a = a$. Examples include 0 for the *gcd* function and 1 for the *lcm* function.

2.1.4 Semigroups

$(S, \cdot)$ is a semigroup if: - S is closed under $\cdot$ - $\cdot$ is associative

We can use this to perform an optimized query to find $A_l \cdot A_{l+1} \cdot A_{l+2} \cdot \ldots \cdot A_r$ in a range $[L, R]$ and an array A.

If $(S, \cdot)$ is associative we can bracket it in any way and thus split it up, which is how a range tree works. Therefore range trees work on all associative operators.

2.1.5 Groups

Suppose every element of S has an inverse and $\cdot$ has an identity e. If $a \in S$ has an inverse $b \in S$ s.t. $a \cdot b = e$ and b is unique, $(S, \cdot)$ is a group.

With a group, we can use a prefix sum / cumulative sum array to calculate what we wanted to find above; namely $A_l \cdot A_{l+1} \cdot A_{l+2} \cdot \ldots \cdot A_r$ in a range $[L, R]$ and an array A. A prefix sum data structure has only an $O(N)$ preprocessing time and $O(inverse)$ query, where $inverse$ is the inverse of $\cdot$.

2.2 Amortized complexity a.k.a. Big O notation

The amortized complexity of an algorithm can be seen as an average runtime of a function, slightly different from the asymptotic complexity. In particular, amortized complexity should not be greatly impacted by a single case which has an extremely long runtime.

One definition is that, if $f(x)$ has an amortized complexity of $O(1)$, then over N operations the total time divided by N is approximately 1.

One example would be an STL vector in C++. It takes constant time to push some number of elements onto the vector, but when the vector's size is doubled to fit more elements the push operation can take up to $O(N)$ time. However, the amortized complexity of the push function remains $O(1)$, as these doubling slowdowns only occur very rarely.

2.2.1 Amortized analysis

There are a number of methods for amortized analysis, such as aggregate analysis (essentially finding the upper bound on N operations and dividing by N), the accounting method (taking into account the costs, execution time and influence on future operations' run time) and the potential method, also called the physicist's method. The potential method is what will be described here.

The greek capital letter phi (Φ) is typically used to represent the cost after i operations.

Φ_i is the cost after i operations, $\Phi_0 = 0$ is the cost after 0 operations, and $\Phi_i > \Phi_0$ for all i. We will also be using c_i to represent the cost of the ith operation, $\hat{c}_i$ to represent the amortized cost of the ith operation, and N to represent the total number of operations.

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + ((\Phi_n - \Phi_{n-1}) + (\Phi_{n-1} - \Phi_{n-2}) + \ldots + (\Phi_1 - \Phi_0)) \\ &= \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0\end{aligned}$$

The amortized time calculated overestimates by Φ_n, which means it is always an upper bound on the actual time.

$\hat{c}_i$ represents the total runtime; we then divide this by N to get the amortized asymptotic complexity.

2.2.1.1 Example

We have an integer x. x is initially set to 0. x is incremented N times. The cost of incrementing x is the number of bit flips performed, i.e. the cost of incrementing 2 to 3 is 1 and the cost of incrementing 7 to 8 is 4.

Let Φ be the number of ones. $\Phi_0 = 0$

For the ith operation:

$$\begin{aligned}c_i &= t_i + 1 \\ \Delta\Phi &= \Phi_i - \Phi_{i-1} \\ &= 1 - t_i\end{aligned}$$

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi \\ &= t_i + 1 + 1 - t_i \\ &= 2\end{aligned}$$

So $\sum_{i=1}^{n} \hat{c}_i = 2n$. This is the total runtime and we want to find the average. Therefore, we divide by n.

$\frac{2n}{n} = 2 = O(1)$

So this operation is amortized $O(1)$ complexity.

Chapter 3

Data Structures

3.1 Range Tree

3.1.1 Summary

A range tree allows querying the minimum / sum / product / other operations (see Requirements) of the range $[L, R]$ in an array in $O(logN)$ time. It is typically the best option for range queries, and its fast updates normally make it the best option for querying a range. There are a few exceptions; for example, when there are no updates and the operation you are using has a unique inverse, prefix tables are a better (and much faster) option; sparse tables can also be used in some cases.

A range tree essentially allows reduction of a query to a lowest common ancestor problem.

3.1.2 Complexity

Preprocessing	Query	Update	Space
$O(N)$	$O(logN)$	$O(logN)$	$O(N)$

3.1.3 Pseudocode

```
array tree of length at least 2*MAX_N-1

func mid takes a, b as input:
   - remember outer brackets - order of ops
```

```
  return a + ((b-a)>>1)

func preprocess takes an array arr, l, r, pos as input:
  - We can store the children of the current node
  - as indices 2i+1 and 2i+2.
  - The maximum depth of a range tree is log N.
  - In this example we are using the operation
  - min; range trees work on any associative operators.
  - l and r are the range of the current
  - node in the array;
  - pos is the index in the tree.
  if l == r:
    tree[pos] = arr[l]
    return arr[l]

  middle = mid(l,r)
  - if we were using, say, addition, this would be +
  tree[pos] = min(preprocess(arr, l, middle, 2*pos+1),
                  preprocess(arr, middle+1, r, 2*pos+2))
  return tree[pos]


func query takes pos, l, r, s, e as input:
  - pos represents the current node i.e. where to
  - start searching from; when query is called this
  - should almost always be set to 0.
  - l and r are the edges of the query range.
  - s and e should be 0 and n-1 respectively,
  - or otherwise the range of start.
  if l <= s and r >= e:
    return tree[pos]

  if e < l or s > r:
    return INFINITY - or whatever the identity is;
                    - for addition this would be 0

  middle = mid(s,e)
  - since we are using min in this example
  - if we were using addition we would use +
  return min(query(2*pos + 1, l, r, s, middle),
             query(2*pos + 2, l, r, middle+1, e))


func update takes pos, s, e, new, ind as input:
  - s and e are the bounds of the current node.
  - pos is the same as query.
```

```
- new is the value of the new element.
- ind is the position of the new element.

- if the destination index is outside our range
if ind < s or ind > e:
  return

if s != e:
  middle = mid(s,e)
  update(2*pos+1, s, middle, new, ind)
  update(2*pos+2, middle+1, new, ind)

  - Another way of doing this is
  - minning each element of the tree
  - with new as we go along;
  - I find this clearer.
  - we use min as, again, it is the operator
  - we have chosen.
  tree[pos] = min(tree[2*pos+1],
                  tree[2*pos+2])
else:
  arr[ind] = new
  tree[pos] = arr[ind]
```

3.1.4 Requirements

A range tree works on any associative operators.

Identities are not necessary if you implement a range tree differently, that is by returning a placeholder value if a node is out of range in the query function and checking for that placeholder being returned. There *are* actually associative operators without identities; for example, strictly upper triangular matrix multiplication, an example from here. Although of course it is very unlikely these will turn up in an informatics competition.

3.2 Prefix sum

3.2.1 Summary

A prefix sum array is a data structure which stores the 'sum' of all of the operations up to an index. For example, if you wished to construct a prefix sum array on the array $[1, 2, 3]$ using addition as your operation, the prefix sum array would be $[1, 1 + 2, 1 + 2 + 3]$.

The name carries connotations of addition but a prefix sum array works on any operation that forms a group with its set S; see the Requirements section.

3.2.2 Complexity

Preprocessing	Query	Update	Space
$O(N)$	$O(inverse)$*	$O(N)$	$O(N)$

*$inverse$ is the cost of performing the inverse of whatever function you have chosen to perform. For example, if you have a prefix sum array using multiplication, the cost of the inverse would be the cost of division.

3.2.3 Pseudocode

```
func preprocess takes an array arr as input:
  Create a new array pre with the same
  length as the original

  pre[0] = arr[0]
  For i in 1 to the length of arr:
    - In this example we are using
    - addition. As discussed in the
    - Requirements section, a prefix
    - sum array works on any operation
    - forming a group, so the addition below
    - can be replaced with, for example, multiplication.
    pre[i] = pre[i-1] + arr[i]

  return pre

func query takes two integers start and end as input:
  - We assume that start and end are zero-indexed.
  - We also assume that we want to query inclusive.
  - Since we used addition in preprocessing, we will
  - use the inverse of addition here, subtraction.
  if start == 0:
    return pre[end]
  else:
    return pre[end] - pre[start-1]

func update takes two integers newval and index as input:
  - We assume that we are replacing values
```

```
- already in the array, rather than inserting
- any.
arr[index] = newval

if index == 0:
  pre[0] = arr[i]
  For i in 1 to the length of arr:
    - Again, the statements about changing this
    - if you use a different operator than addition
    - hold.
    pre[i] = pre[i-1] + arr[i]
else:
  For i in index to the length of arr:
    pre[i] = pre[i-1] + arr[i]
```

3.2.4 Requirements

A prefix sum array is only guaranteed to work on a group. (See the section on Group theory.) For example, prefix sum arrays do not work with operations such as *min*. In part this is because the operation needs a (unique) inverse. For example, the min function does not work with prefix sum tables.

3.2.5 Tips

A prefix sum array is not well suited to cases where updates are required due to its $O(N)$ update time complexity; in these cases it is overwhelmingly likely that a range tree will be a better choice.

3.3 Sparse Tables

3.3.1 Summary

A sparse table is represented as a 2D array of size $logN$ by N. `Table[i][j]` represents the result of the 'product' of the elements j through $2^i + j$. (Non-inclusive in this case.) The solution is then found by finding the 'product' of the 2-power 'bucket' starting at the left edge of the query range and that ending at the right. Since these may overlap Sparse Tables only work for idempotent operators such as *min*, *max* and *lcm*.

We can also represent a sparse table as a 1D array, meaning we only use N space, rather than $NlogN$. This can be done by representing each row of the original table as a consecutive segment of the 1D array. The example given here is $NlogN$, which is good enough for any practical example: N up to 67 108 864

works within 64 MiB and up to 268 435 436 within 256 MiB, which tends to be the upper bound in programming contests.

A sparse table allows us to do queries in $O(1)$ time, although updates are $O(N)$ - if there are any updates a range tree is a better option.

3.3.2 Complexity

Preprocessing	Query	Update	Space
$O(N)$	$O(1)$	$O(N)$	$O(NlogN)$

3.3.3 Pseudocode

```
array table of size [log MAX_N][MAX_N]

func preprocess takes array arr as input:
  - We are using min as an example here.
  - Keep in mind operators need to be idempotent
  - so operations such as addition do not work.
  For i in 0 to length of arr:
    table[0][i] = arr[i]

  for p in 0 to the floor of log2(length of arr):
    for i in 0 to length of arr:
      table[p+1][i] = min(table[p][i],
                          table[p][i+(1<<p)])

func query takes two integers l, r as input:
  - l and r represent the query range.
  if l == r:
    return table[0][l]

  p = floor of log2(r-l + 1)

  return min(table[p][l], table[p][b-(1<<p)+1])

func update takes pos, new, array arr as input:
  - You can actually implement an update
  - for sparse tables, but the $O(N)$ speed means
  - you might as well just preprocess.
  arr[pos] = new
  preprocess(arr)
```

3.3.4 Requirements

A sparse table only works on a set S and operator $*$ if $*$ is associative, commutative and idempotent. Examples include the *min* function. (No inverse is required.)

3.3.5 Tips

Don't bother using the inclusion-exclusion principle on sparse tables as you could equally well have used a prefix sum array.

Chapter 4

Algorithms

4.1 Range minimum query (and variants)

(Variants include finding the sum of a range, the product of a range, the max of a range etc.)

A range minimum query entails finding the minimum value in an array between indices L and R. The naïve approach is to iterate over every value, which takes $O(N)$ time. Therefore we are only interested in algorithms that take $< O(N)$ time.

A range minimum query is best performed with one of the Range Tree, Prefix Sum and Sparse Table data structures. (Each of these have requirements; see the *Data Structures* section.)

One alternative to Sparse Tables and Prefix Sums are Square Root Decompositions, which are good because they are quick to implement and have reasonably quick $O(\sqrt{n})$ query for $O(\sqrt{n})$ space complexity (as well as $O(\sqrt{n})$ update and $O(n)$ preprocessing.) In most cases a sparse table is a better option however, or, if updates are needed, a range tree is probably best.

A Square Root Decomposition works as follows: - Divide the array into $\sqrt{n}$ blocks of size $\sqrt{n}$ each.

Query: - Calculate the 'product' of each of the buckets in the preprocessed array which are contained entirely within the query range. - Linearly scan elements that are only partially within the buckets.

Square Root Decompositions work for any associative operators.

4.1.1 With range updates

If we need to query ranges and update a range with operations such as 'set everything in this range to 0' or 'multiply everything in this range by a', we can use a modified range tree. For the latter case, you would store a 'multiplication factor' for each node that was entirely contained within the range. This could be done by traversing up through the tree, changing every node's multiplication factor accordingly if they were a part of the range. This is only $O(logN)$.

This solution works with any pair of distributive semigroups. Since multiplication distributes over addition, the above solution works. Another example is that addition distributes over min, so the solution would work if we stored an amount to be added on each node and our queries were of the minimum element in a range.

Chapter 5

Dynamic Programming

5.1 Knapsack

5.1.1 Summary

Refers to any of a group of problems in which a set of items each with weights and values have to be packed into a knapsack such that the sum of their weights is less than or equal to the capacity of the knapsack and their value is as high as possible.

5.1.2 Unbounded

There are no constraints on the knapsack; each item can be taken as many times as desired.

Complexity: $O(NW, W)$

Potential Optimisations: - Divide all of the weights by their greatest common divisor to reduce space / complexity. Should be unnecessary in most cases

Pseudocode:
w: amount of storage in knapsack
n: number of items
$vals$: values of items
$weights$: weights of items

```
Create an array 'dp' of length w+1
- dp[i] represents the highest
- possible score with a weight of
- less than or equal to i.
```

```
Initialise every element of dp to 0

for i in range 0 to w+1:
  for j in range 0 to n:
    - See if this item can be taken
    - to improve on our score
    if i+weights[j] < w+1:
      - We can take this item
      dp[i+vals[j]] = max(
        dp[i+vals[j]],
        dp[i]+vals[i]
        )
```

5.1.3 0/1 (Binary)

Each item can be taken at most once.

Complexity: $O(NW, NW)$

Potential Optimisations: - 'Flatten' the array, storing only one one-dimensional array of length W+1 and rewriting from indexes W to 1 each time, giving the same result for $O(W)$ space. - There is another approach commonly named "meet-in-the-middle" which uses $O(2^{\frac{n}{2}})$ space and $O(n2^{\frac{n}{2}})$ runtime. It may be more optimal for very large values of W. It works as follows:

partition the set of items into two sets of approximately equal size

find the weights and values of the subsets of each set

for each subset of the first set:

find the subset of the second set of greatest value such that their combined weight is less than W

keep track of the greatest value so far

We can optimise this algorithm to the aformentioned $O(n2^{\frac{n}{2}})$ runtime by sorting subsets of the second set by weight, discarding those which weigh more than those of greater / equal value, and using binary search.a

Pseudocode:
w: amount of storage in knapsack
n: number of items
$vals$: values of items
$weights$: weights of items

```
Create an array 'dp' of size n+1 by w+1
- dp[i][j] represents the maximum
- possible value that can be achieved
```

```
- using only the first i items
- up to weight j.

for j in range 0 to w+1:
  dp[0][j] = 0

for i in range 0 to n:
  for j in range 0 to w:
    if weights[i] > j:
      dp[i+1][j] = dp[i][j]
    else:
      dp[i+1][j] = max(
        dp[i][j],
        dp[i][j-weights[i+1]]+vals[i+1]
        )
```