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.
- In some problems where the order of input is not important, it is a good idea to randomise the input. If your solution is timing out but only slightly on some of the larger test cases, it is a good idea to try randomising the input. The function `random_shuffle` (defined in header `<algorithm>`) can shuffle an input array `arr` as follows: `random_shuffle(arr,arr+arr_length);`.
- In the pseudocode, `- ...` represents a comment, unless it is clearly used in the context of a mathematical operation.

Chapter 2

C++ notes

2.1 "The bits trick"

The `bits/stdc++.h` header is a testing header which imports every standard c++ (and c) header, which makes it very useful for programming competitions. You can import `<bits/stdc++.h>` on Linux systems, and with some workarounds on macOS. The issue on some operating systems is that `stdc++.h` is included in GNU G++, which is not what is installed on macOS in particular; by default macOS uses Clang. One way around this is simply to install GNU G++ on your system. It may already be installed - if there is a `g++` command which ends in a `-*`, where `*` represents a version number (e.g. `g++-6`) then this is very likely GNU G++.

One other workaround is as follows.

1. Create a file named `stdc++.h` in a folder named `bits` in the folder that you will be doing all of your programming in (alternatively, you can put it somewhere else and use an absolute pathname in step 3)
2. Fill the file with the contents of the following file: https://gist.github.com/eduarc/6022859 (Please note that this is the header for `gcc 4.8.0`, so is likely outdated. Also, ensure you do this *before* the exam has begun.) This may not be up in future; if not, you will have to do some digging for the `bits/stdc++.h` source code. I recommend searching something like `bits/stdc++.h source file`.
3. In a terminal, use the `export` command to change the value of the `CPLUSPLUS_INCLUDE_DIR` variable to either `.` or the absolute path of the `bits` folder's superfolder. On bash and bourne shells, this is similar to the following:
   ```
   export CPLUSPLUS_INCLUDE_DIR='.'
   ```
   You may want to put this in your `~/.bash_profile` or `~/.profile` files if you are planning on having more than one terminal open. (In particular on macOS `~/.bash_profile` is adviseable as if it exists as *well* as a `~/.profile` file the only one that will be read is `~/.bash_profile`.)

Alternatively, In your code, use `"bits/stdc++.h"` instead of `<bits/stdc++.h>` everywhere (note the double quotes instead of angle brackets.)

Now write a test program which prints `Hello World!` using the `bits/stdc++.h` header. If there are any compilation errors, delete the offending headers in the `stdc++.h` source file.

2.2 `using namespace std`

It is *always* adviseablle in Informatics competitions to add a `using namespace std` line to the top of your code. This allows you to shorten the length of the names of all of the standard library containers you import, for example: `std::cout` becomes just `cout`. Keep in mind that this is barely ever a good idea outside of informatics competitions.

2.3 `cppreference`

The online resource http://cppreference.com/ is very useful for informatics competitions. In particular, it gives you information about the runtime complexities of all the STL data structures. (It also tells you the headers which the classes are located in if you're not using the `bits` trick.) Most competitions will allow you to use any online C++ documentation, in which case `cppreference` is highly recommended.

2.4 Vectors vs. Arrays

The `vector` class (defined in header `<vector>`) is similar to an array, but dynamically resizes, so one can 'push back' and 'pop' from a vector. Unlike an array, one does not need to specify the size of a `vector` beforehand (although you can if you want - see the `resize` and `reserve` functions.)

Vectors are slower than arrays but only slightly. It may be adviseable to use vectors instead of arrays in the general case.

2.5 Set and Unordered Set

The `set` class (`<set>`) is a collection of elements that maintains sorted order, i.e. any new element inserted into the set is automatically placed in the set such that the set is sorted, and popping off the set always returns the lowest element. In this regard it is similar to a `priority_queue`, but you can access elements in the middle of the set (and a set is slower.)

The `unordered_set` class (`<unordered_set>`) is a collecion of elements where order is not maintained, and popping could return any element from the set. (Technically popping actually involves checking the element at `set.begin()`; erasing is a different method.) This is *always* better than `set` if the order of elements is not important, i.e. you are storing which of a group of people are currently in a room. One alternative is, if the identifiers of the people (or whatever it may be) are integers or can be transformed into integers, to create an array of booleans, corresponding to whether element `i` is in the set. (Also look into the `bitset` class.)

2.6 Range based for loops, auto specifier

C++11 introduces both range-based for loops and the `auto` specifier. To compile your code using C++11, insert the flag `std=c++11` into your compilation command.

The `auto` specifier works as follows:

```
auto x = 1;
```

This creates a new variable `x` whose type is inferred to be `int`.

Range-based for loops work as follows:

```
vector<vector<pair<int,int>>> graph;

... populate graph ...

for (auto n : graph[1]) {
  ... do things ...
}

for (int n : {1,3,5,6}) {
  cout << n << ”\n”;
}
```

The `auto` specifier is quite often useful in conjuction with a range-based for loop, as in the example above.

2.7 Out of memory?

Firstly, check that you actually shouldn’t be out of memory.

Also note that huge arrays in `main` or another function will probably crash your program, as it is being initialised on the *stack* (where currently run functions are stored in the memory.)

```
int main(int argc, char* argv[]) {
  int huge[134217728]; // 128 MiB - probably cause a crash

  return 0;
}
```

Such arrays should be placed in global scope, outside of any functions, so they are allocated on the *heap* (which is much larger than the stack, although slower):

```
int huge[134217728];

int main(int argc, char* argv[]) {
  return 0;
}
```

2.8 Array initialisation

You can initialise an n-dimensional array to all 0s as follows: (this only works at initialisation)

```
int graph[1000][1000][2] = {0};
```

(If a variable is initialised outside of a function i.e. in global scope, on the heap, it will be initialised by default to 0. You can of course add in the `= {0}` anyway.)

Note this does not work with other integers such as `1`; doing this will only initialise the first value in the array to the value in braces, and everything else will be given a 'default value' of sorts, which happens to be `0`. This is fine if this is what you want to happen.

```
int graph[1000] = {1};
cout << graph[0] << graph[1] << graph[2] << endl; // 100
```

2.9 `#define` and `typedef`

C++ includes a `#define` preprocessor macro.

```
#include <bits/stdc++.h>

using namespace std;

#define INFINITY (2 << 31)
#define max(a,b) ((a)>(b) ? (a) : (b))

int main(int argc, char* argv[]) {
  cout << INFINITY << endl; // 4294967296
  cout << max(5,10) << endl; // 10
  return 0;
}
```

This substitutes the strings at runtime. The brackets in the defined function are quite significant. In the below example:

```
#include <bits/stdc++.h>

using namespace std;

#define VARIABLE 50+10
#define ANOTHER (VARIABLE*2)

int main(int argc, char* argv[]) {
  cout << VARIABLE << endl; // 60
  cout << ANOTHER << endl; // 70

  return 0;
}
```

We see that because we have left the brackets out around the definition of `VARIABLE` and in the `ANOTHER` function, the arithmetic rules take over. In the above example, line 10 is substituted to:

```
  cout << (50+10*2) << endl;
```

which of course evaulates to 70.

C++ also has a `typedef` keyword.

```
#include <bits/stdc++.h>

using namespace std;

typedef pair<int,int> pi;

int main(int argc, char* argv[]) {
  pi a;

  cout << a.first << a.second << endl; // 00

  return 0;
}
```

Chapter 3

Mathematics

3.1 Set / Group theory

3.1.1 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 $a*b \in S\ \forall a,b \in S$.

3.1.2 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 $a+(b*c) = (a*b)+(a*c)$.

3.1.3 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.

3.1.4 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.

3.1.5 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 $\cdot$.

3.2 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.

3.2.1 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.

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.

3.3 Graph Theory

3.3.1 Data Structures

3.3.1.1 Adjacency matrix

Stores the distance between any two vertices, i.e.: `graph[i][j]` is the distance between vertices `i` and `j`. If there is no edge between two vertices a value representing inifinity can be used, and there is obviously 0 distance from a vertex to itself.

$O(V^2)$ space.

3.3.1.2 Adjacency list

Stores the distance from every vertex to every vertex it has an edge to, i.e. `graph[i][j]` represents the `j`-th outgoing edge from vertex `i`.

$O(EV)$ space. Typically much less. Normally one would use a list of vectors to represent the outgoing edges, to account for sparse and dense graphs alike.

3.3.1.3 Conversion

```
- The following functions assume adjacency lists
- store edges in the format {dist, node}
- instead of what may seem more logical,
- {node, dist}. This is to allow the lists
- to be easily sorted.

function matrixToList takes array of arrays graph:
  array of vectors of pairs result

  for i in range 0 to size of graph:
    for j in range 0 to size of graph:
      if i != j and graph[i][j] != INFINITY:
        result[i].push_back( {graph[i][j],j} )

  return result

function listToMatrix takes array of vectors of pairs graph:
  array of arrays result with default value INFINITY

  for i in range 0 to size of graph:
    - C++11 range-based for loop.
    - see the C++ tips section
    result[i][i] = 0
    for ( auto j : graph[i] ):
      result[i][j.second] = j.first

  return result
```

3.3.2 Tree

An undirected graph in which any two vertices are connected by exactly one path.

A graph G with n vertices is a tree if: (Technically these are equivalent statements)

- G is connected i.e. no vertices have no edges, but is not connected if any edge is removed
- G has no cycles, and a cycle is formed if any edge is added
- G is connected and has $n - 1$ edges
- G has no cycles and has $n - 1$ edges

3.3.3 Minimum Spanning Tree

The minimum spanning tree of a graph is defined to be the tree with the smallest weight that connects every node in the graph. (It may not be unique.)

- Every MST has $n - 1$ edges, where there are n edges in the graph
- If every edge has a different weight the MST is unique
- The longest edge in any cycle cannot belong to an MST
- If the shortest edge in a graph is unique, that edge is in every MST of the graph

Chapter 4

Data Structures

4.1 Range Tree

4.1.1 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.

4.1.2 Complexity

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

4.1.3 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]
```

4.1.4 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.

4.2 Prefix sum

4.2.1 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.

4.2.2 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.

4.2.3 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]
```

4.2.4 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.

4.2.5 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.

4.3 Sparse Tables

4.3.1 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.

4.3.2 Complexity

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

4.3.3 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)
```

4.3.4 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.)

4.3.5 Tips

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

4.4 Priority Queue

A data structure which allows adding elements and popping the smallest / largest / some other as defined by a comparator function. If implemented as a binary heap, a common implementation, has `O(log N)` update and pop.

You can provide comparator functions as per the entry on http://cppreference.com.

```
#include <queue>

...

priority_queue<int> q;
q.push(4);
q.push(8);
q.push(6);

cout << q.top() << endl; // 8
q.pop();
```

4.5 Binary Heap

One implementation of a priority queue. In particular a binary heap consists of a root node with two children, who may have children and so on; the children satisfy the heap invariant, which states that each of the children have a higher value than their parent; in the way it is implemented in the C++ STL, this becomes each of the children having a lower value than their parent. Binary heaps have `O(log N)` update and pop.

This is already implemented in the C++ STL. Details on asymptotic time complexity can be found at http://cppreference.com. Please note the implemented heap is a max heap. It is often adviseable to use a priority queue: `#include <queue> ... priority_queue<int> q;` as custom comparator functions are easier to work with, along with the API being easier in general.

```
#include <algorithm>

...

vector<int> vec;
```

```
make_heap(vec.begin(), vec.end());

for ( auto i : {1,5,2,6,3,4} ) {
  vec.push(i);
  push_heap(vec.begin(), vec.end());
}

cout << vec.front() << endl; // 6
pop_heap(vec.begin(), vec.end());
```

4.6 Disjoint set

4.6.1 Summary

An efficient data structure for storing the connected components of a graph.

4.6.2 Complexity

Find	Union	Space
$O(1)$	$O(1)$	$O(N)$

(Assuming both path compression and union by rank are implemented.) Technically the functions Find and Union are inverse Ackermann complexity; however, for all practical values, this is less than 5.

4.6.3 Pseudocode

```
int parent[MAX_N+1]
int rank[MAX_N+1]
int N

function create takes number of nodes n as input:
  N = n
  for i in range 0 to n inclusive:
    rank[i] = 0
    parent[i] = i

function find takes integer x as input:
  - this is path compression
  if x != parent[x]:
    parent[x] = find(parent[x])

  return parent[x]

- union by rank
function union takes two integers x and y as input:
  x = find(x)
  y = find(y)

  if rank[x] > rank[y]:
    parent[y] = x
  else:
    parent[x] = y

  if rank[x] == rank[y]:
    rank[y]++
```

This implements union by rank and path compression.

Chapter 5

Algorithms

5.1 Dynamic Programming

Dynamic programming is not an algorithm; rather, it is a method of approaching a problem. Dynamic programming can be used whenever a problem exhibits *overlapping subproblems*, or *optimal substructure*, where to calculate the answer to a problem often in a solution you would have to calculate the same number twice.

One example is the following recursive definition of the nth Fibonacci number:

```
func fibonacci takes an integer n:
  if n < 0:
    throw BAD_VALUE
  if n < 2:
    return 1
  return fibonacci(n-1) + fibonacci(n-2)
```

When $n = 2$, this only has to call fibonacci(0) and fibonacci(1):

```
      2
     / \
    0   1
```

However, when $n = 4$, this has to call the following:

```
         4
        / \
       /   \
      2     3
     / \   / \
    0   1 1   2
             / \
            0   1
```

As should be obvious from the diagram, many functions are called more times than necessary. While in this small case it does not matter much, in larger algorithms it can be extremely significant if the same solution is calculated multiple times.

The way to solve this is through *caching*.

```
array cache of length MAX_N default value 0

func fibonacci takes an integer n:
  if n < 0:
    throw BAD_VALUE
  if cache[n] != 0:
    return cache[n]
  if n < 2:
    cache[n] = 1
    return 1
  cache[n] = fibonacci(n-1) + fibonacci(n-2)
  return cache[n]
```

This means we only have to call the following:

```
         4
        / \
       /   \
      2     3
     / \   / \
    0   1 1   2
```

`fibonacci(1)` and `fibonacci(2)` take very minimal time to compute the second time, simply returning elements of an array. Therefore we have cut the runtime in half, and the improvement is much more dramatic for larger values of n.

5.2 Shortest path

The shortest path problem is used a lot in informatics competitions. It involves finding the shortest path between two nodes on a graph; a number of variants exist such as restricting the graph to being directed and acyclic, or finding the shortest path between all nodes on a graph. The most common algorithm is Dijkstra's algorithm, which finds the shortest path between two nodes on a directed graph with no negative edge weights (it can easily be extended to undirected graphs.)

5.2.1 Dijkstra's Algorithm

5.2.1.1 Summary

Finds the shortest path between two nodes on a graph.

5.2.1.2 Complexity

$O(ElogV)$ where E is the number of edges and V is the number of vertices.

5.2.1.3 Pseudocode

```
Use a C++ pair type

- graph represents the outgoing edges from each vertex i.e. graph[1] is outgoing
- edges from vertex 1
- graph[1][0] is, say, {2,3}, representing an edge to node 2 with length 3
function dijkstra takes an array of vectors of pairs graph and two integers from and to:
  vector of integers dist of length graph size and with default value INFINITY
  set of pairs active

  - for simplicity active contains pairs of form {dist, node} for easy sorting.

  dist[from] = 0
  active.push( {0, from} )

  while (!active.empty()):
    curr = active.begin()->second
    if curr == target: return dist[curr]
    active.erase(active.begin())
    for nxt in graph[curr]:
      if dist[nxt.first] > dist[curr] + graph[nxt.second]:
        active.erase( {dist[nxt.first], nxt.first})
        dist[nxt.first] = dist[curr] + graph[nxt.second]
        active.insert( {dist[nxt.first], nxt.first})

  return INFINITY
```

5.2.1.4 Requirements

A graph with no negative cycles (i.e. cycles with a negative length.) The algorithm assumes a directed graph, and an undirected graph can easily be used as well by changing each edge of the graph to two directed edges with the same weight and opposite directions, i.e. `1 - 2` becomes `1 -> 2` and `2 -> 1`.

5.2.1.5 Tips

The pseudocode assumes a `set` type with the same API as the C++ one, and assumes that the graph is in adjacency list format.

5.2.2 Floyd-Warshall

5.2.2.1 Summary

The Floyd-Warshall algorithm is an algorithm which finds the shortest path between every two vertices in a graph.

5.2.2.2 Complexity

$O(V^3)$.

5.2.2.3 Pseudocode

```
function floydwarshall takes an array of arrays graph as input:
  array best of size graph size by graph size

  for i in range 0 to graph size:
    for j in range 0 to graph size:
      best[i][j] = graph[i][j]

  for k in range 0 to graph size:
    for i in range 0 to graph size:
      for j in range 0 to graph size:
        best[i][j] = min(best[i][j], best[i][k] + best[k][j])

  return best
```

5.2.2.4 Requirements

Same as Dijkstra.

5.2.2.5 Tips

The pseudocode assumes the graph is an adjacency matrix. See the section on graph theory for information on this.

5.3 Minimum spanning tree

See the section under Mathematics on Graph Theory.

5.3.1 Prim's Algorithm

5.3.1.1 Summary

Prim's Algorithm is a greedy algorithm which finds the minimum spanning tree of a weighted undirected graph by choosing the lowest-cost edge from any vertex currently in the minimum spanning tree at any point.

5.3.1.2 Complexity

$O(ElogV)$.

5.3.1.3 Pseudocode

```
- Need algorithm header for greater<>

- graph[i][j] represents the j-th outgoing edge from the i-th node
- represented in format {node, dist} i.e. if there was an edge from 0
- to 1 with length 3, and this was the only edge, graph[0][0] = {1,3}.
function prim takes an array of vectors of pairs graph as input:
  priority queue of pairs of integers with underlying storage vector of pairs
  and comparison function greater of pairs queue
  - priority_queue<pair<int,int>, vector<pair<int,int>>,
                                  greater<pair<int,int>>> queue;

  // It doesn't matter which vertex the source is.
  integer source = 0

  vector of integers weights of size graph size defaulting to INFINITY
  - vector<int> weights(graph size, INFINITY);

  vector of integers parent of size graph size defaulting to -1, representing a
  - nonexistent node
  - vector<int> parent(graph size, -1);

  vector of booleans visited of size graph size
  - vector<bool> visited(graph size, false);

  queue.push( {0, source} )
```

```
weights[source] = 0

while (!queue.empty()):
  integer vert = queue.top().second
  queue.pop()

  visited[vert] = true

  for i in graph[vert]:
    integer next = i.first
    integer dist = i.second

    if (!visited[next] && weights[next] > dist):
      weights[next] = dist
      queue.push( {weights[next], next} )
      parent[next] = vert

for i in range 1 to the size of graph:
  - This prints every edge of the MST
  print parent[i], i
```

5.3.1.4 Requirements

A weighted undirected graph.
In its most basic form, shown here, it only finds the MST of a connected graph.
The algorithm here assumes the input is given as an adjacency list; this is more efficient than the implementation for an adjacency matrix.

5.3.1.5 Tips

The vectors can be replaced quite simply with arrays, assuming you know the maximum size of the graph. Use a for loop to initialise every value to infinity if using this method.

While the above function prints every edge of the MST, it can easily be modified to store these in a list of pairs of integers. Keep in mind that by definition an MST has $V - 1$ edges, where V is the number of vertices in the graph.

5.3.2 Kruskal's Algorithm

5.3.2.1 Summary

A greedy algorithm for finding the minimum spanning tree of a graph. Essentially works by continually adding the smallest edge to the MST, and then removing it again if it would result in a cycle.

5.3.2.2 Complexity

$O(ElogV)$.

5.3.2.3 Pseudocode

```
- Add all the Disjoint set implementation here


- graph is an edge list:
- {weight, {from, to}}
function kruskal takes vector of pairs of (int, pair of ints) graph as input:
  integer weight = 0       - of MST
  sort(graph.begin(), graph.end())

  - Using the function names from the Disjoint set pseudocode above
  create(graph size)

  for i in graph:
    integer groupa = find(i.second.first)
    integer groupb = find(i.second.second)

    if groupa != groupb:   - they do not create a cycle
      print u, v

      weight += i.first    - the weight of the edge

      union(groupa, groupb)


  return weight
```

5.3.2.4 Requirements

A weighted undirected graph.
This algorithm can find the MST of a disconnected graph. This assumes input is given in *edge list* format. This is similar to an adjacency list but every edge is stored separately, and it is trivial to convert between the two. See the pseudocode for the format it is stored in.

5.3.3 Tips

While the above function prints every edge of the MST, it can easily be modified to store these in a list of pairs of integers. Keep in mind that by definition an MST has $V - 1$ edges, where V is the number of vertices in the graph.

5.4 Convex Hull Trick

This could also be classified as a data structure. This essentially involves finding the minimum (sometimes maximum) y-value of a group of linear functions at any x value. The example given here (http://wcipeg.com/wiki/Convex_hull_trick) is of four functions: $y = 4, y = 4/3 + 2/3x, y = 12 - 3x, y = 3 - 1/2x$. If the query was $x = 1$, the answer would be the value 2, with the function $y = 4/3 + 2/3x$.

The naive approach is $O(MQ)$, with M lines and Q queries. The trick enables us to increase this to a speed of $O((Q + M)logM)$. The space required is $O(M)$, and the construction time is $O(MlogM)$.

In the example given above, we first note that $y = 4$ will never be the lowest. The trick is to find the intervals at which each line is the minimum and binary search these intervals to answer each query, of course after removing irrelevant lines such as $y = 4$.

It is significant to note that each line that we add must only be compared with the two lines before it.

The below code assumes all the lines are added initially; there are variants which allow for logarithmic arbitrary line insertion.

```
long M[MAX_LINES]
long C[MAX_LINES]
integer length
integer pointer

- This function assumes that m descends. If all the input is given prior to
- runtime this is fine - just sort the input descending.
function addline takes integers m, c as input:
  - The intersection of the line represented by M[length-2], C[length-2] and
  - M[length-1], C[length-1] must be to the left of the intersection of M[length-1],
  - C[length-1] with m, c.
  while length >= 2 and (C[length-2]-C[length-1])*(m-M[length-1]) >=
                        (C[length-1]-c)*(M[length-1]-M[length-2]):
    length--

  M[length] = m
  C[length] = c
  length++
function minValue takes integer x as input:
  - Given ascending values of x
  pointer = min(pointer, length-1)

  while (pointer+1) < length and M[pointer+1]*x+C[pointer+1] <=
                                 M[pointer]*x+C[pointer]:
    pointer++

  return M[pointer]*x + C[pointer]
```

5.5 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 by dividing the array into $\sqrt{n}$ blocks of size $\sqrt{n}$ each. Square Root Decompositions work for any associative operators.

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.

5.5.1 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.

5.6 Knapsack

5.6.1 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.

5.6.2 Unbounded

There are no constraints; 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]
        )
```

5.6.3 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]
        )
```