# *Algorithms Lab*

*Cristian Perez Jensen*

*January 31, 2024*

*Contents*

## 1 Overview

In the Algorithms Lab course, tasks entail the following,

1. Find an appropriate model for the problem;
2. Design a suitable algorithm to solve it *efficiently*;
3. Implement (in C++) and test the algorithm on the given data.

In general, problems should take 2 hours to solve.

Problems are formatted such that they consist of a story, a precise definition of the input and output, and then a point distribution. Make sure to read the point distribution, because they often contain simplified versions of the task, which serve as a *roadmap* toward an efficient algorithm to solve the problem.

## 2 Binary search

Whenever there is a monotonic relationship between values, *i.e.* if $x_1 \geq x_2$, then always $f(x_1) \geq f(x_2)$ or always $f(x_2) \leq f(x_1)$, binary search can find a given element in $\mathcal{O}(\log n)$.[1]

Before implementing binary search, first see whether a simple for-loop would suffice. This saves time.

[1] Instead of $\mathcal{O}(n)$ if you would iterate over all possible inputs.

```
1:  function BinarySearch(a, n, T)
2:      ℓ ← 0
3:      r ← n − 1
4:      while ℓ ≤ r do
5:          m ← ⌊(ℓ+r)/2⌋
6:          if a_m < T then
7:              ℓ ← m + 1                    ▷ Search right side of m
8:          else if a_m > T then
9:              r ← m − 1                    ▷ Search left side of m
10:         else
11:             return m                     ▷ a_m = T
12:         end if
13:     end while
14:     return −1
15: end function
```

**Algorithm 1.** Binary search algorithm for finding index containing value $T$ in a sorted array $\boldsymbol{a}$ of length $n$.

## 3 Dynamic programming

Dynamic programming solves a problem by reducing it to smaller subproblems of the same type. These subproblems are described by a *state* $s \in S$. To get an answer to the problem, the same subproblems may need to be solved many times. Thus, in dynamic programming, the

At worst, the table must be filled out fully, thus the worst-time complexity of a dynamic programming algorithm is $\mathcal{O}(st)$, where $s$ is the size of the table, and $t$ the time complexity of computing a single subsolution.

```
1: function BinarySearchLeftMost(a, n, T)
2:     ℓ ← 0
3:     r ← n
4:     while ℓ < r do
5:         m ← ⌊(ℓ+r)/2⌋
6:         if a_m < T then
7:             ℓ ← m + 1                ▷ Search right side of m
8:         else
9:             r ← m                    ▷ Search left side of m
10:        end if
11:    end while
12:    return ℓ
13: end function
```

**Algorithm 2.** Binary search can also be used for finding the minimum input value for which a constraint is still satisfied.

solutions to all subproblems are stored, in a table that maps state to solution value, to be used later.[2]

[2] Dynamic programming is essentially recursion with memoization.

To apply dynamic programming, we need to find three things,

- A state space $S$ that describes subproblems as succinctly as possible;
- A recurrence relationship $r(s) = f(s, r(s_1), \ldots, r(s_n))$, where $\{s_1, \ldots, s_n\} \subseteq S$ are the necessary subproblem states to compute $s$;
- Base cases $B \subseteq S$ with constant values that do not need to be computed, *i.e.* $r(b) = c(b)$ for all $b \in B$, where $c(b)$ maps base cases to their solution value.

Once we have those, we can easily construct a polynomial algorithm.

If possible, always use `std::vector` if the state space can be described by integers, since it has constant insert/access time complexity. Remember that `std::map` has $\mathcal{O}(\log n)$ insert/find/access time complexity.

Dynamic programming differentiates between a top-down and a bottom-up approach. In the top-down approach, the values are computed recursively, each time checking the table for a precomputed value and base case condition before computing the state solution. In bottom-up, the smaller problems are explicitly computed before computing the next state. In general, the top-down approach is easier to implement. However, the bottom-up approach is more memory efficient, because it does not need to keep track of function calls on the stack. Also, using the bottom-up approach, it is easier to reason about the time complexity.

## 4 *Boost Graph Library*

The Boost Graph Library (BGL) is a library of common graph algorithms. In BGL, graphs are represented as adjacency lists, *i.e.* a vector of vectors.

```cpp
#include <boost/graph/adjacency_list.hpp>

typedef boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,  // Or boost::directedS
  boost::no_property,  // Vertex property
  boost::property<boost::edge_weight_t, int>
> graph;
```

**Listing 1.** Basic type for a weighted graph.

A list of BGL functions can be found in Table 1. See the tutorial slides for how to use them. Also, the BGL documentation contains detailed descriptions of how these algorithms work.

In the exam-like problems, Dijkstra's is the only function actually used commonly. Dijkstra's algorithm computes the distance, in a weighted graph, from a source node to all other nodes. A modified Kruskal's algorithm is often implemented with proximity structures to find whether two vertices are reachable under some constraints.

Table 1 can also be found in the tutorial slides.[3] So, in the exam, if there is no way to solve a graph problem with the other methods, then take a look at this table and figure out how to potentially apply these algorithms.

[3] Also, the table contains links to the documentation of the functions, which contains good descriptions of the algorithms.

| Algorithm | Runtime |
|---|---|
| `boost::breadth_first_search` | $\mathcal{O}(n+m)$ |
| `boost::depth_first_search` | $\mathcal{O}(n+m)$ |
| `boost::dijkstra_shortest_path` | $\mathcal{O}(n \log n + m)$ |
| `boost::kruskal_minimum_spanning_tree` | $\mathcal{O}(m \log m)$ |
| `boost::edmonds_maximum_cardinality_matching` | $\mathcal{O}(mn \cdot \alpha(m,n))$ |
| `boost::strong_components` | $\mathcal{O}(n+m)$ |
| `boost::connected_components` | $\mathcal{O}(n+m)$ |
| `boost::biconnected_components` | $\mathcal{O}(n+m)$ |
| `boost::articulation_points` | $\mathcal{O}(n+m)$ |
| `boost::is_bipartite` | $\mathcal{O}(n+m)$ |

**Table 1.** Common graph algorithms that appear throughout the course.

## 5 *Computational Geometry Algorithms Library*

*Infinite precision and range.* CGAL is a library that provides infinite precision and range types.[4] However, large numbers also have a higher computational cost in CGAL, so only use exactly as much algebra as

[4] As opposed to the default C++ types in Table 2.

needed in problems that require infinite precision.

| Type | Bits (specific to this course) | Minimum | Maximum |
|---|---|---|---|
| `int` | 32 | $-2^{31}$ | $2^{31}-1$ |
| `long` | 64 | $-2^{63}$ | $2^{63}-1$ |
| `double` | 64 | $-2^{53}$ | $2^{53}-1$ |

**Table 2.** C++ types with their representational range.

In general, the following guidelines should be followed to avoid unnecessary computational cost,

1. Avoid square roots where possible, which is often possible, because the function is monotonic, *i.e.*,[5]
$$\forall x, y \geq 0 : \sqrt{x} < \sqrt{y} \iff x < y;$$
2. Avoid divisions, which is often possible in comparisons, *i.e.*,[6]
$$\forall x, y > 0 : \frac{a}{b} < \frac{c}{d} \iff ad < bc;$$
3. Estimate to check if loss of precision may occur. *I.e.*, first check whether the values will fit within one of the default C++ types (Table 2). If we need to multiply two values with $a$ and $b$ bits, respectively, we will need a type with $a + b$ bits. If we need to add two values, we need $\max\{a, b\} + 1$ bits.

[5] This is especially useful when working with distances,
$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$,
because we only need squared distances if we only need them for comparison.

[6] To keep the code clean, you can also use the `CGAL::Gmpq` type, which represents divisions by the numerator and denominator, and does this under the hood.

*Geometretic computing.* CGAL also provides predicates and constructions for geometry. The library provides three kernels, shown in Table 3.

| Kernel | Feature |
|---|---|
| `CGAL::Exact_predicates_inexact_constructions_kernel` | Constructions use `double`, so not exact |
| `CGAL::Exact_predicates_exact_constructions_kernel` | Exact constructions, supporting $+, \times, \div$. |
| `CGAL::Exact_predicates_inexact_constructions_kernel_with_sqrt` | Exact constructions, supporting $+, \times, \div, \sqrt{\cdot}$. |

**Table 3.** CGAL kernels, ordered by increasing computational cost.

CGAL has many different geometries that it can represent, *e.g.*, `K::Point_2`, `K::Line_2`, `K::Ray_2`, `K::Segment_2`. However, they are only necessary if constructions are absolutely necessary.[7] Usually, there is a more efficient method that requires no constructions. See the CGAL documentation for provided predicates and constructions.

[7] An edge case is `K::Point_2`, which does not require constructions and are often useful.

If you need to take the floor of an infinite precision type, such as `K::FT`, use the function in Listing 2.

```
typedef CGAL::
    Exact_predicates_inexact_constructions_kernel K;

double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}
```

**Listing 2.** Floor of infinite precision type. If the ceiling must be computed, the following identity can be used,

$$\lceil x \rceil = -\lfloor -x \rfloor.$$

## 6 Greedy algorithms

The greedy approach rarely yields optimal solutions, but it is easy to convince yourself that the greedy approach "works". This is why the proof step is important, but, in this course, there is no time to construct a proof. Thus, first exhaust all other options before resorting to a greedy approach.

A greedy algorithm can be applied if *locally optimal choices* result in a *globally optimal solution*. Usually, these are tasks that require the construction of a set that is in some sense globally optimal. In general, a greedy approach has the following steps,

1. *Greedy choice*: Given already chosen elements $c_1, \ldots, c_{k-1}$, decide how to choose $c_k$, based on some local optimality criterion;
2. *Proof*: Prove that the elements obtained in this way result in a globally optimal set;[8]
3. *Implementation*: Implement the greedy choice to be as efficient as possible.

[8] We can prove that a greedy solution works using an *exchange argument* or a *staying ahead* argument. We can disprove one via a counterexample.

The hard part lies in the first step, where we need to figure out how to pick the next element of the set, given already chosen elements.

## 7 Split and list

Only use split and list of $n \leq 50$, where $n$ is the amount of elements in the set. It is often necessary for $\mathcal{NP}$-hard problems to get full points.

For some problems, we need to consider every possible "configuration" to solve it, resulting in $\mathcal{O}(c \cdot 2^n)$ time complexity, which is okay for $n \approx 25$ in this course, where $\mathcal{O}(c)$ is the runtime of checking whether a configuration satisfies some condition. In some cases, using split and list, we can get it down to $\mathcal{O}(c \cdot 2^{n/2})$, which is okay for $n \approx 50$ in this course.

Split and list can be used if the elements $S$ can be split into $S_1$ and $S_2$ such that the results of the configurations of $S_1$ and $S_2$ make up the result of a full configuration of $S$.

We iterate over all configurations of $S_1$ and $S_2$ and compute their results, stored in $L_1$ and $L_2$, respectively. Sort $L_2$. Then, for each

$k_1 \in L_1$, check if there is a $k_2 \in L_2$ (using binary search) such that their combination make up the target.

## 8 Maximum flow

*Tip.* The max flow from $u$ to $v$ is the same as from $v$ to $u$ in the reversed graph.

In maximum flow problems, we have a graph where the edges are given flow capacity, which is how much can flow through an edge. Then, the question becomes how much flow can go from a source vertex to a sink vertex in such a graph. Using BGL, we compute the maximum flow of a graph with the push-relabel algorithm ($\mathcal{O}(V^3)$).[9]

[9] You also need to add residual connections to all edges, but this is done with the `edge_adder` struct that is given in the maximum flow example of the course documentation.

Common techniques that are very useful in such problems are the following,

- *Multiple sources/sinks*: If you need multiple sources (or sinks), you can simply add a supernode that has infinite capacity to all the sources (or sinks);
- *Vertex capacities*: If vertices should have a certain capacity that is allowed to flow through it, use two vertices to represent it. The input vertex should take all inputs of the vertex and the output vertex should take all outputs. Then, add an edge from input to output with the vertex capacity;
- *Undirected edges*: If you need an undirected edge between $a$ and $b$ with capacity $c$, just add directed edges from $a$ to $b$ and $b$ to $a$, both with capacity $c$;[10]
- *Minimum edge constraint*: If we need the following constraint on an edge $e = (u, v)$,
$$c_{\min}(e) \leq f(e) \leq c_{\max}(e),$$
where $f(e)$ is the flow through edge $e$, we need to adjust the demand, supply, and capacity as follows,
$$\begin{aligned} d_u &\leftarrow d_u + c_{\min} \\ s_v &\leftarrow s_v + c_{\min} \\ c(e) &\leftarrow c_{\max} - c_{\min}, \end{aligned}$$
where $d_u$ is the demand of $u$, *i.e.*, the amount of capacity to the target, $s_v$ is the supply of $v$, *i.e.*, the amount of capacity from the source, and $c(e)$ is the capacity of $e$.

[10] This works because if flow goes both ways, they might as well both stay on their original side (which the algorithm will do). So, the maximum flow through this undirected edge is achieved if $c$ goes from $a$ to $b$ and 0 goes from $b$ to $a$ (or vice versa).

### 8.1 Minimum cut

The maximum flow between two vertices can also be seen as the bottleneck between them. Thus, we can also see this as the minimum cut,

where we need to cut/block the botlteneck to disconnect the two vertices. The actual vertices that are on the two sides of the cut can then be found by breadth-first search on the edges with non-zero residual capacity (see slides).

### *8.2 Bipartite matching*

In a bipartite matching problem, we want to compute to take as many non-adjacent edges as possible, *i.e.*, make as many assignments as possible. This can also be computed by maximum flow. First, we need to construct the bipartite graph. Assign the source to one side with 1 flow, and the target to the other side with 1 flow. Then, connect all pairs with 1 flow. The maximum flow is then the maximum amount of matchings that can be made.

**Theorem 1** (König)**.** *In a bipartite graph, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover.*

The maximum independent set $I \subseteq V$ is the largest set of vertices, such that none of them are connected by an edge,

$$\nexists\, u, v \in I : (u, v) \in E.$$

The minimum vertex cover $C \subseteq V$ is the smallest set of vertices, such that every edge is connected to one of the vertices in this set,

$$\forall (u, v) \in E : u \in C \vee v \in C.$$

Using Theorem 1, we can compute the size of the minimum vertex cover $|C|$ of a bipartite graph by computing the maximum flow. Then, we can compute the size of the maximum independent set by $|I| = |V| - |C|$.

### *8.3 Minimum cost maximum flow*

Only use minimum cost maximum flow if number $n \leq 1000$, where $n$ is the amount of vertices. If there are negative costs, only use it if $n \leq 600$.

We can also associate cost with flow on the edges. In minimum cost maximum flow problems, we then want to first maximize the flow, and then, as a second priority, compute the minimum cost. *I.e.*, we want to find the cheapest among all maximum matchings.

We could also maximize the cost by making them negative. However, the non-negative solver is much faster than the one that allows negative costs. Thus, if we eed a negative cost, we should compute some upper-bound $U$, and give $B - c$ cost, such that the costs become positive. Then, afterward, we need to remove the added $B$ in some way. This can usually be done as a function of the maximum flow, since that is how many times $B$ was added to the cost.

## 9 *Proximity structures*

In this course, we often have geometry problems with many points on an $x, y$ coordinate system. In these cases, we also want to be efficient. In a problem, we might have points with some radius, where we need to find the minimum radius such that some condition is satisfied. Or, we might need to find the maximum distance a point can remain from all points when moving out of the convex hull. Or, we might need to find the closest point for $m$ points in a large list of $n$ points. In this case, naively computing the closest point would result in $\mathcal{O}(mn)$ complexity, but we can do better. If we precompute a triangulation in $\mathcal{O}(n \log n)$, we can find closest points in $\mathcal{O}(\log n)$. This results in a $\mathcal{O}((n+m) \log n)$ time complexity, which is much better than the naive version.

### 9.1 *Delaunay triangulation*

```
typedef CGAL::
    Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;

std::vector<K::Point_2> points(n);
// Read in points...

Triangulation t;
t.insert(points.begin(), points.end());
```

**Listing 3.** Delaunay triangulation in C++. The exact constructions kernel is necessary if access to the Voronoi diagram is needed.

Delaunay triangulation for a set of discrete points is a triangulation such that no point is inside the circumcircle of any triangle in the triangulation. Let a disk be of maximal radius if it passes through three points, its center is inside the conex hull of the points, and the disk does not contain any other points. The maximal empty disks of the graph make up a Delaunay triangulation.

Delaunay triangulation has the following properties,

- It contains the Euclidean minimum spanning tree;
- Each point has an edge to all closest other points;
- It can be constructed efficiently in $\mathcal{O}(n \log n)$.

Furthermore, Delaunay triangulation is the straight-line dual of the Voronoi diagram. The Voronoi diagram for a set of points $\mathcal{P}$ partitions the plane into regions for which the closest point from $\mathcal{P}$ is the same. If

you move along these edges, you will always be as far away as possible from the points $\mathcal{P}$.

The dual of a face, can be found with `t.dual(f)`. However, this is inefficient, because it uses constructions. Often, we do not need to explicitly compute the dual vertex. We could often simply use `t.nearest_vertex(p)`, which only uses predicates. In the exam-like problems, we only needed the dual vertex for a problem where we needed to compute the maximal radius of a face. We can compute this by computing the distance between the dual and any vertex of the face, `CGAL::squared_distance(t.dual(f), f.vertex(0))`.

## 10 *Linear programming*

Only use linear programming if $\min\{n, m\} \leq 200$, where $n$ is the amount of variables, and $m$ the amount of constraints.

Linear programming is a linear optimization problem subject to $m$ linear constraints on $n$ variables. In general, a linear program looks like the following,

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{c}^\top \boldsymbol{x} + c_0 \\ \text{subject to} \quad & \boldsymbol{a}_1 \boldsymbol{x} \leq b_1 \\ & \vdots \\ & \boldsymbol{a}_m \boldsymbol{x} \leq b_m. \end{aligned}$$

Note that we can also form constraints of the form $\boldsymbol{a}_j \boldsymbol{x} \geq b_j$ by adding the following constraint,

$$-\boldsymbol{a}_j \boldsymbol{x} \leq -b_j.$$

Furthermore, we can also specify a maximization objective by minimizing the following,

$$-(\boldsymbol{c}^\top \boldsymbol{x} + c_0),$$

and then flipping the sign after solving the problem.

Geometrically, a linear program defines an $n$-dimensional convex polyhedron with $m$ faces. The optimum objective value is found at one of the vertices of this polyhedron. There are an exponential amount of vertices, thus the worst-case time complexity is exponential in $n$ and $m$, but for small $\min\{n, m\}$, the complexity is $\mathcal{O}(\max\{n, m\})$.

The following are tips when working with linear programming,

- If you want to maximize a minimum value that is a function $f(x_i)$ of the linear programming variables $x_i$, you need to add a variable, $f_{\min}$, and the following constraints,

$$\forall x_i : f_{\min} \leq f(x_i).$$

```
// Let 0 be the infinite vertex
auto f = t.incident_faces(t.infinite_vertex());
do {
  f->info() = 0;
} while (++f != t.incident_faces(t.infinite_vertex()));

// Give each face a unique index
size_t num_faces = 1;
for (auto f = t.finite_faces_begin(); f != t.
    finite_faces_end(); ++f)
  f->info() = num_faces++;

std::vector<std::vector<std::pair<size_t, K::FT>>> adj(
    num_faces);

// Iterate over faces
for (
  auto f = t.finite_faces_begin();
  f != t.finite_faces_end();
  ++f
)
{
  size_t index = f->info();

  for (int i = 0; i < 3; i++)
  {
    size_t n_index = f->neighbor(i)->info();
    K::FT length = t.segment(f, i).squared_length();

    // Construct graph outside the DT
    adj[index].push_back({ n_index, length });

    // Infinite vertices
    if (n_index == 0)
      adj[0].push_back({ index, length });
  }
}

// Locate face in which point ‘p‘ is by ‘t.locate(p)‘
```

**Listing 4.** If we need the graph where the faces are vertices and the minimum radius that can go between faces as edge weights, we can use this snippet. This is needed when doing motion planning between points, where we want to know how to move a disk without colliding with any points.

Then maximize this variable. Ditto if you want to minimize the maximum value;

- If any of the constraints contain fractional coefficients, multiply the constraints such that all coefficients are whole numbers;
- The signed distance from a point $\boldsymbol{x}$ to a hyperplane $\boldsymbol{a}^\top \boldsymbol{x} + b = 0$ is computed as follows,

$$\frac{\boldsymbol{a}^\top \boldsymbol{x} + b}{\|\boldsymbol{a}\|_2^2}.$$

This is not specific to linear programming, but often appears in linear programming problems in this course.