Algolab 2017 – STL Week 2

A couple of general remarks:

- Public test sets are different than the test sets on the judge.

A couple of general remarks:

- Public test sets are different than the test sets on the judge.
- In particular, judge test sets may (or may not) contain special edge cases not existing in the public ones.

A couple of general remarks:

- Public test sets are different than the test sets on the judge.
- In particular, judge test sets may (or may not) contain special edge cases not existing in the public ones.
- Think about test cases for which your algorithm might fail or possibly behave strangely

A couple of general remarks:

- Public test sets are different than the test sets on the judge.
- In particular, judge test sets may (or may not) contain special edge cases not existing in the public ones.
- Think about test cases for which your algorithm might fail or possibly behave strangely
- To be on the safe side, always use

  ```
  std::ios_base::sync_with_stdio(false);
  ```

  as the first line of the main procedure for faster I/O.

Problem: Barber shop

A barber takes L minutes to cut a customer's hair.

- n customers arrive at times $t_0, \ldots, t_{n-1}$.
- The last customer leaves at time T.
- What is L?

Problem: Barber shop

A barber takes L minutes to cut a customer's hair.

- n customers arrive at times $t_0, \ldots, t_{n-1}$.
- The last customer leaves at time T.
- What is L?
- The barber starts cutting the next customer's hair as soon as he is finished with his current customer.
- But maybe he has to wait for the next customer to arrive.

Reformulation

Let us say that L is too small if the last customer will leave before time T, assuming the barber needs L minutes for a haircut.

We need to find the smallest L that is not too small.

Reformulation

Let us say that L is too small if the last customer will leave before time T, assuming the barber needs L minutes for a haircut.

We need to find the smallest L that is not too small.

Simpler problem: check whether a **fixed** L is too small.

To check if a fixed L is too small:

To check if a fixed L is too small:

- Precomputation: sort the arrival times in increasing order.

```
sort(t.begin(), t.end());
```

To check if a fixed L is too small:

- Precomputation: sort the arrival times in increasing order.

```
sort(t.begin(), t.end());
```

- Check if L is too small:

```
bool too_small(int L) {
    int time = 0;
    for (int i = 0; i < n; i++) {
        if (t[i] <= time)
            time += L;
        else
            time = t[i] + L;
    }
    return (time < T);
}
```

Trick/technique (Sorting)

Sorting can be a powerful pre-computation step.

To sort a vector, always use the `std::sort` function from the `<algorithm>` library.

- $\mathcal{O}(n \log n)$ time for the pre-computation
- $\mathcal{O}(n)$ time per call of `too_small`

- $\mathcal{O}(n \log n)$ time for the pre-computation
- $\mathcal{O}(n)$ time per call of `too_small`

How can we find the smallest L that is not too small?

Binary search

- The property of not being too small is monotone.

Binary search

- The property of not being too small is monotone.
- So we can use **binary search** to efficiently find the smallest L with this property.

Binary search

- The property of not being too small is monotone.
- So we can use **binary search** to efficiently find the smallest L with this property.
- First, we find an upper bound on the smallest L (**exponential search**):

```
int lmin = 0, lmax = 1;
while (too_small(lmax)) lmax *= 2;
```

Binary search

- The property of not being too small is monotone.
- So we can use **binary search** to efficiently find the smallest L with this property.
- First, we find an upper bound on the smallest L (**exponential search**):

```
int lmin = 0, lmax = 1;
while (too_small(lmax)) lmax *= 2;
```

- Now we do the binary search:

```
while (lmin != lmax) {
    int p = (lmin + lmax)/2;
    if (too_small(p))
        lmin = p + 1;
    else
        lmax = p;
}
L = lmin;
```

Binary search

- The property of not being too small is monotone.
- So we can use **binary search** to efficiently find the smallest L with this property.
- First, we find an upper bound on the smallest L (**exponential search**):

```
int lmin = 0, lmax = 1;
while (too_small(lmax)) lmax *= 2;
```

- Now we do the binary search:

```
while (lmin != lmax) {
    int p = (lmin + lmax)/2;
    if (too_small(p))
        lmin = p + 1;
    else
        lmax = p;
}
L = lmin;
```

- In general, `std::binary_search` from `<algorithm>` can be useful.

In general

The problem: find the smallest k that is ‘large enough’.

For a **fixed** k, you can check efficiently if it is ‘large enough’.

How to find the smallest k efficiently?

In general

The problem: find the smallest k that is 'large enough'.

For a **fixed** k, you can check efficiently if it is 'large enough'.

How to find the smallest k efficiently?

Trick/technique (Binary search)

In such situations, we can use **binary search** to find the optimal k.

The running time is multiplied only with a factor of $\mathcal{O}(\log K)$, where K is the smallest k that is large enough.

Problem of the Week: Deck of Cards

Problem: Deck of Cards

Given numbers $v_0, \ldots, v_{n-1}$, minimize

$$\left| k - \sum_{\ell=i}^{j} v_\ell \right|$$

over all $0 \leq i \leq j < n$.

It is possible to solve this in time $\mathcal{O}(n)$:

It is possible to solve this in time $\mathcal{O}(n)$:

```
int i = 0, j = 0;
int sum = v[0];
int best = sum;

while (i < n) {
    best = min(best, abs(sum - k));
    if (sum < k && j < n) {
        j++;
        sum += v[j];
    }
    else {
        sum -= v[i];
        i++;
    }
}
```

Why this works (sketch):

- Say an interval is interesting if it has a chance to be optimal (given current knowledge).

Why this works (sketch):

- Say an interval is interesting if it has a chance to be optimal (given current knowledge).
- Suppose the current interval is $[i, j]$ and suppose that all unseen interesting intervals $[i', j']$ are such that $i' \geq i$ and $j' \geq j$. (This is true when $i = j = 0$.)

Why this works (sketch):

- Say an interval is interesting if it has a chance to be optimal (given current knowledge).
- Suppose the current interval is $[i, j]$ and suppose that all unseen interesting intervals $[i', j']$ are such that $i' \geq i$ and $j' \geq j$. (This is true when $i = j = 0$.)
- If $\mathrm{sum}([i, j]) > k$ then all unseen interesting intervals $[i', j']$ have $i' > i$. So we set $i = i + 1$.

Why this works (sketch):

- Say an interval is interesting if it has a chance to be optimal (given current knowledge).
- Suppose the current interval is $[i, j]$ and suppose that all unseen interesting intervals $[i', j']$ are such that $i' \geq i$ and $j' \geq j$. (This is true when $i = j = 0$.)
- If $\mathrm{sum}([i, j]) > k$ then all unseen interesting intervals $[i', j']$ have $i' > i$. So we set $i = i + 1$.
- If $\mathrm{sum}([i, j]) \leq k$ then all unseen interesting intervals $[i', j']$ have $j' > j$. So we set $j = j + 1$.

Trick/technique (Sliding window)

Some problems in which you need to find some optimal interval can be solved in linear time using a similar sliding window approach.

Graph Traversals

Storing graphs

If the graph is given explicitly, it is typically best to store it as an **adjacency list**. For example:

```
vector< vector<int> > adj(n);

for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Storing graphs

If the graph is given explicitly, it is typically best to store it as an adjacency list. For example:

```
vector< vector<int> > adj(n);

for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

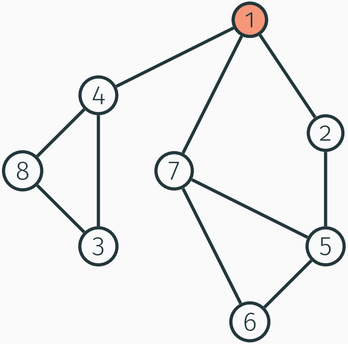
(This will be different when you start using the BGL!)

Storing graphs

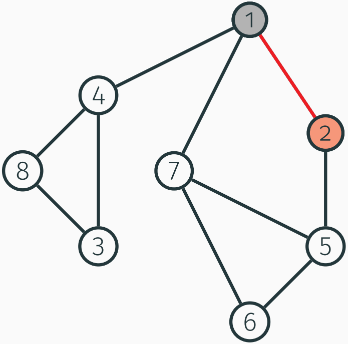
Often the graph is given **implicitly**, meaning that you can efficiently compute the neighbourhood of a given vertex.

In this case, it is probably best **not** to store the graph in memory.

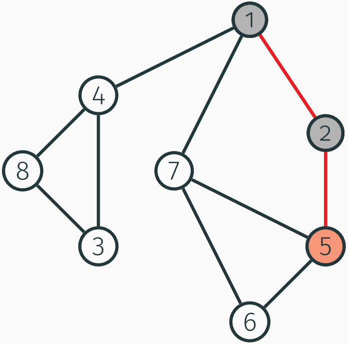
Depth-first search



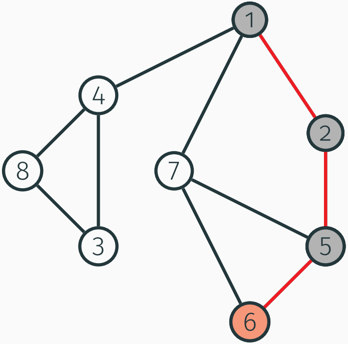
Depth-first search



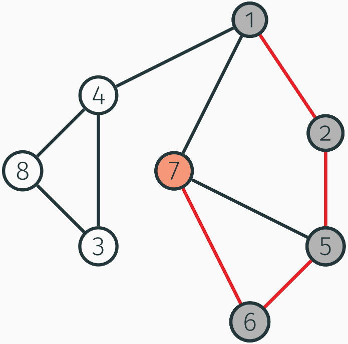
Depth-first search



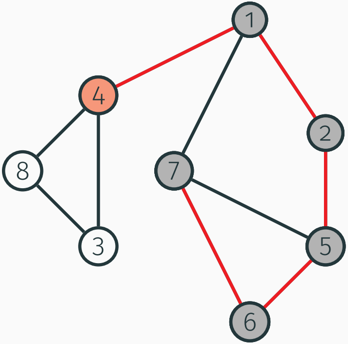
Depth-first search



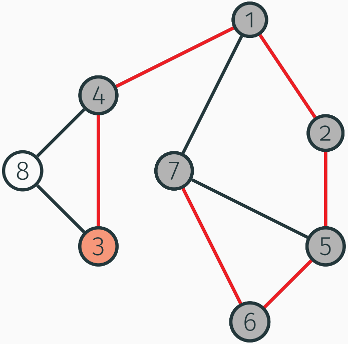
Depth-first search



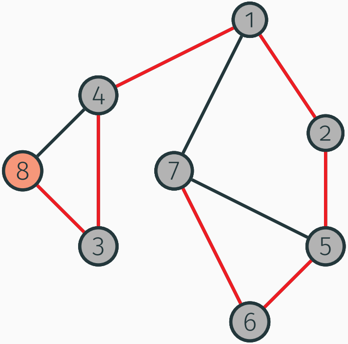
Depth-first search



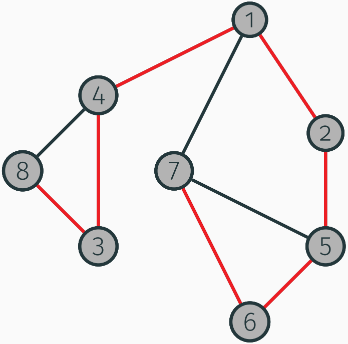
Depth-first search



Depth-first search

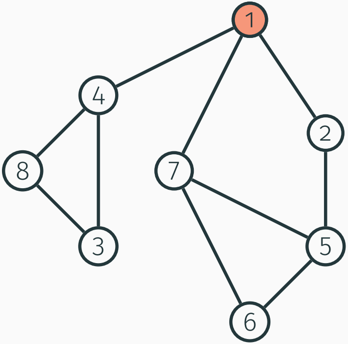


Depth-first search

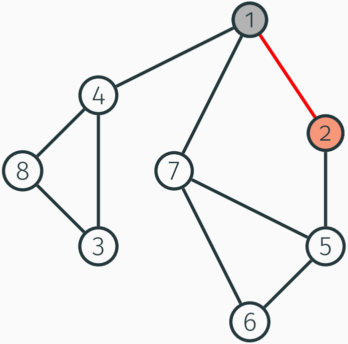


Order in which the vertices are visited: 1, 2, 5, 6, 7, 4, 3, 8.

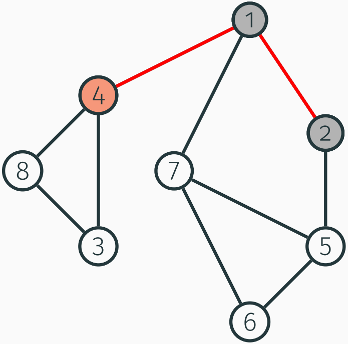
Breadth-first search



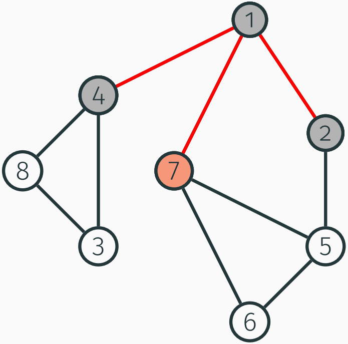
Breadth-first search



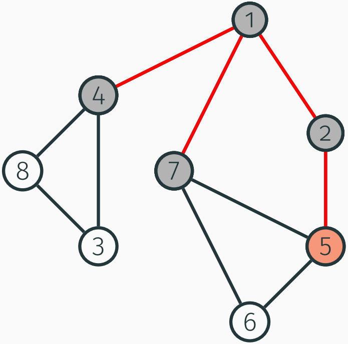
Breadth-first search



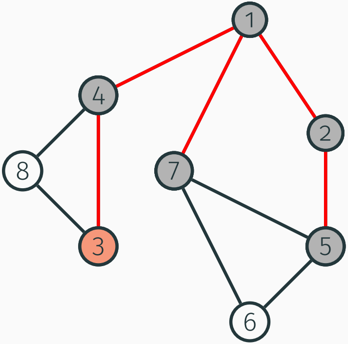
Breadth-first search



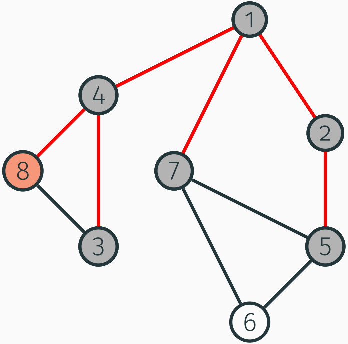
Breadth-first search



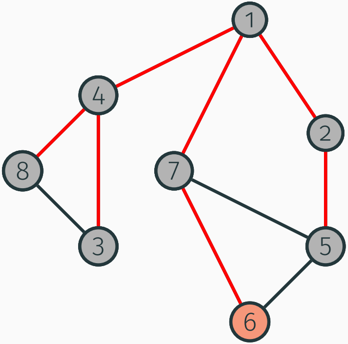
Breadth-first search



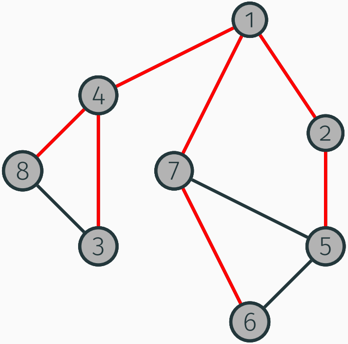
Breadth-first search



Breadth-first search



Breadth-first search



Order in which the vertices are visited: 1, 2, 4, 7, 5, 3, 8, 6.

Recurive DFS implementation

```
vector<int> visited(n, 0);

void dfs(int v) {
    // do something for v
    for (int u : adj[v]) {
        if (not visited[u]) {
            visited[u] = 1;
            dfs(u);
        }
    }

    // maybe do something else for v
}
```

Correct BFS implementation

```
queue<int> q;
vector<int> visited(n, 0);

q.push(v0);
visited[v0] = 1;

while (not q.empty()) {
    int v = q.front();
    // do something for v
    q.pop();
    for (int u : adj[v]) {
        if (visited[u] == 0) {
            q.push_back(u);
            visited[u] = 1;
        }
    }
}
```

Greedy Algorithms

- Often choices that seem best in a particular moment turn out not to be optimal in the long run (e.g., in chess, life, etc.).
- But sometimes locally optimal choices result in a globally optimal solution.
- This is when we can apply greedy algorithms.

A greedy approach typically has the following steps:

1. **Modelling**: realise that your task requires you to construct a set that is in some sense globally optimal.

A greedy approach typically has the following steps:

1. **Modelling**: realise that your task requires you to construct a set that is in some sense globally optimal.
2. **Greedy choice:** given already chosen elements $c_1, \ldots, c_{k-1}$, decide how choose c_k, based on some local optimality criterion.

A greedy approach typically has the following steps:

1. **Modelling**: realise that your task requires you to construct a set that is in some sense globally optimal.
2. **Greedy choice:** given already chosen elements $c_1, \ldots, c_{k-1}$, decide how choose c_k, based on some local optimality criterion.
3. **Prove** that elements obtained in this way result in a globally optimal set.

A greedy approach typically has the following steps:

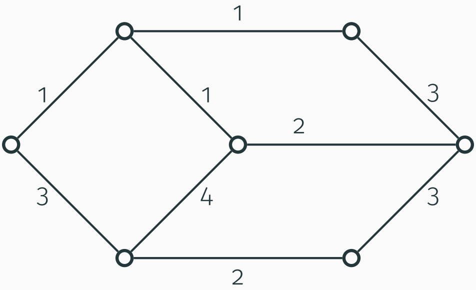
1. **Modelling**: realise that your task requires you to construct a set that is in some sense globally optimal.
2. **Greedy choice:** given already chosen elements $c_1, \ldots, c_{k-1}$, decide how choose c_k, based on some local optimality criterion.
3. **Prove** that elements obtained in this way result in a globally optimal set.
4. **Implement** the greedy choice to be as efficient as possible.

Example: MST

In a graph G with non-negative edge weights, find a minimum weight spanning tree.

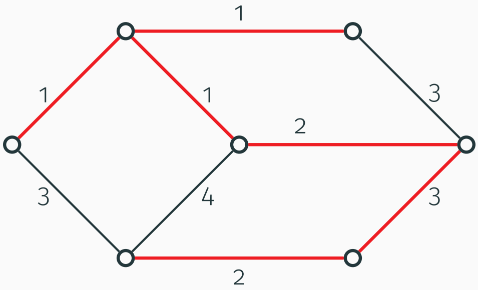
Example: MST

In a graph G with non-negative edge weights, find a minimum weight spanning tree.



Example: MST

In a graph G with non-negative edge weights, find a minimum weight spanning tree.



Example: MST

First step: model as an optimization problem over sets.

In this case, we want to find a set of edges with minimum weight that forms a spanning tree.

Example: MST

Second step: how to make the greedy choice.

Idea:

- suppose we already have edges $e_1, \ldots, e_{k-1}$

Example: MST

Second step: how to make the greedy choice.

Idea:

- suppose we already have edges $e_1, \ldots, e_{k-1}$
- choose e_k so that

Example: MST

Second step: how to make the greedy choice.

Idea:

- suppose we already have edges $e_1, \ldots, e_{k-1}$
- choose e_k so that
    1. adding e_k to $e_1, \ldots, e_{k-1}$ does not close a cycle (**compatibility**)

Example: MST

Second step: how to make the greedy choice.

Idea:

- suppose we already have edges $e_1, \ldots, e_{k-1}$
- choose e_k so that
  1. adding e_k to $e_1, \ldots, e_{k-1}$ does not close a cycle (**compatibility**)
  2. e_k has minimum weight among all compatible edges (**local optimality**)

Example: MST

Third step: prove that this is correct.

General method: Exchange Argument

Example: MST

Third step: prove that this is correct.

General method: Exchange Argument

- Let $e_1, \ldots, e_{n-1}$ be the choices made by the greedy algorithm.

Example: MST

Third step: prove that this is correct.

General method: Exchange Argument

- Let $e_1, \ldots, e_{n-1}$ be the choices made by the greedy algorithm.
- Let T be an optimal solution.

Example: MST

Third step: prove that this is correct.

General method: Exchange Argument

- Let $e_1, \ldots, e_{n-1}$ be the choices made by the greedy algorithm.
- Let T be an optimal solution.
- If all edges $e_1, \ldots, e_{n-1}$ belong to T, we are done (why?).

Example: MST

Third step: prove that this is correct.

General method: Exchange Argument

- Let $e_1, \ldots, e_{n-1}$ be the choices made by the greedy algorithm.
- Let T be an optimal solution.
- If all edges $e_1, \ldots, e_{n-1}$ belong to T, we are done (why?).
- Otherwise, suppose T contains $e_1, \ldots, e_i$, but not e_{i+1}.

Example: MST

Third step: prove that this is correct.

General method: Exchange Argument

- Let $e_1, \ldots, e_{n-1}$ be the choices made by the greedy algorithm.
- Let T be an optimal solution.
- If all edges $e_1, \ldots, e_{n-1}$ belong to T, we are done (why?).
- Otherwise, suppose T contains $e_1, \ldots, e_i$, but not e_{i+1}.
- Modify T to obtain an optimal solution containing $e_1, \ldots, e_{i+1}$.

Example: MST

Final step: implement the algorithm efficiently.

Example: MST

Final step: implement the algorithm efficiently.

1. Sort the edges according to increasing weight.

Example: MST

Final step: implement the algorithm efficiently.

1. Sort the edges according to increasing weight.
2. Iterate over the edges in this order.

Example: MST

Final step: implement the algorithm efficiently.

1. Sort the edges according to increasing weight.
2. Iterate over the edges in this order.
3. For each edge $\{u, v\}$, if u and v are in the different components formed by the previous edges, add the edge to the MST.

Example: MST

Final step: implement the algorithm efficiently.

1. Sort the edges according to increasing weight.
2. Iterate over the edges in this order.
3. For each edge $\{u, v\}$, if u and v are in the different components formed by the previous edges, add the edge to the MST.

To keep track of the components, use a union find data structure.

This takes time $\mathcal{O}(m \log m)$.

This is Kruskal's algorithm for MST.

Example: Interval scheduling

- Your CPU needs to execute n jobs described by time intervals $[s_i, f_i]$.
- Job i starts at time s_i and ends at time f_i.
- Two jobs are compatible if their intervals are disjoint.
- Goal: find the maximum number of mutually compatible jobs.

Example: Interval scheduling

Modelling: done for us in the problem description — find the maximum set of compatible jobs.

Example: Interval scheduling

Greedy choice: decide how to choose the job j_k given already chosen jobs $j_1, \ldots, j_{k-1}$.

Natural candidates:

Example: Interval scheduling

Greedy choice: decide how to choose the job j_k given already chosen jobs $j_1, \ldots, j_{k-1}$.

Natural candidates:

- **Earliest start time** – among compatible jobs, take the one with smallest s_k.

Example: Interval scheduling

Greedy choice: decide how to choose the job j_k given already chosen jobs $j_1, \ldots, j_{k-1}$.

Natural candidates:

- **Earliest start time** – among compatible jobs, take the one with smallest s_k.
- **Earliest finish time** – among compatible jobs, take the one with smallest f_k.

Example: Interval scheduling

Greedy choice: decide how to choose the job j_k given already chosen jobs $j_1, \ldots, j_{k-1}$.

Natural candidates:

- **Earliest start time** – among compatible jobs, take the one with smallest s_k.
- **Earliest finish time** – among compatible jobs, take the one with smallest f_k.
- **Shortest length** – among compatible jobs, take the one with smallest $f_k - s_k$.

Example: Interval scheduling

Greedy choice: decide how to choose the job j_k given already chosen jobs $j_1, \ldots, j_{k-1}$.

Natural candidates:

- Earliest start time – among compatible jobs, take the one with smallest s_k.
- Earliest finish time – among compatible jobs, take the one with smallest f_k.
- Shortest length – among compatible jobs, take the one with smallest $f_k - s_k$.
- Fewest conflicts – among compatible jobs, take the one which conflicts with the least amount of other compatible jobs.

Example: Interval scheduling

Earliest start time

Earliest finish time

Shortest length

Fewest conflicts

Which one do you think will work?

Example: Interval scheduling

Earliest start time

Example: Interval scheduling

Earliest start time

WRONG

Example: Interval scheduling

Shortest length

Example: Interval scheduling

Shortest length

WRONG

Example: Interval scheduling

Fewest conflicts

Example: Interval scheduling

Fewest conflicts

WRONG

Example: Interval scheduling

Earliest finish time

Example: Interval scheduling

Earliest finish time

Maybe???

Example: Interval scheduling

Prove that earliest finish time is correct.

General method: Exchange Argument

Example: Interval scheduling

Prove that earliest finish time is correct.

General method: Exchange Argument

- Let $j_1, \ldots, j_N$ be the jobs chosen according to earliest finish time.

Example: Interval scheduling

Prove that earliest finish time is correct.

General method: Exchange Argument

- Let $j_1, \ldots, j_N$ be the jobs chosen according to earliest finish time.
- Let J be a maximum set of jobs.

Example: Interval scheduling

Prove that earliest finish time is correct.

General method: Exchange Argument

- Let $j_1, \ldots, j_N$ be the jobs chosen according to earliest finish time.
- Let J be a maximum set of jobs.
- If all jobs $j_1, \ldots, j_N$ belong to J, we are done (why?).

Example: Interval scheduling

Prove that earliest finish time is correct.

General method: Exchange Argument

- Let $j_1, \ldots, j_N$ be the jobs chosen according to earliest finish time.
- Let J be a maximum set of jobs.
- If all jobs $j_1, \ldots, j_N$ belong to J, we are done (why?).
- Otherwise, suppose J contains $j_1, \ldots, j_i$, but not j_{i+1}.

Example: Interval scheduling

Prove that earliest finish time is correct.

General method: Exchange Argument

- Let $j_1, \ldots, j_N$ be the jobs chosen according to earliest finish time.
- Let J be a maximum set of jobs.
- If all jobs $j_1, \ldots, j_N$ belong to J, we are done (why?).
- Otherwise, suppose J contains $j_1, \ldots, j_i$, but not j_{i+1}.
- Modify J to obtain an optimal solution containing $j_1, \ldots, j_{i+1}$.

Example: Interval scheduling

Final step: implement the algorithm efficiently.

1. Sort the jobs according to increasing finish time.
2. Iterate over the jobs in this order.
3. For each job with interval $[s_i, f_i]$, add the job if s_i is greater than the finish time of the last job that was added.

This takes time $\mathcal{O}(n \log n)$.

Conclusion:

- Some (but not all!) problems can be solved with a greedy approach.
- Deciding how to make the greedy choice can be non-obvious.
- We can check whether the greedy solution works using an **exchange argument**.