1 Stable Matching

The stable matching solves the problem of assigning one element $x \in X$ to another element $y \in Y$(for example: men to wimen, intern to hospitals, ecc...).

- Unstable Pairs
  A pair p(x,y) is defined as unstable if $\exists y^* \in Y$ *such that* $p(x, y^*) > p(x, y)$ *and vice* $-$ *versa*.
- Stable Matching
  A matching containing no unstable pairs.

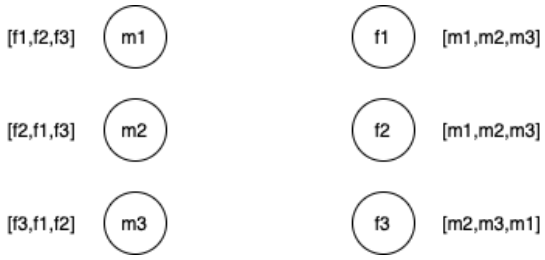


Figure 1: An instance of the problem containing the M and W sets and $\forall m \in M$, $\forall w \in W$ a list of preferences for the elements in the opposite set, in increasing order.

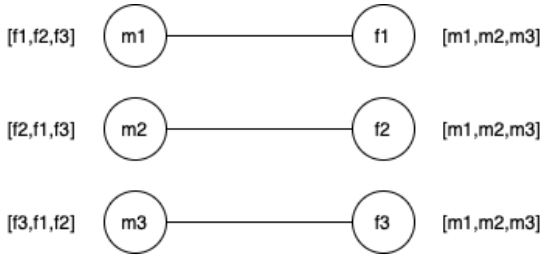


Figure 2: A solution for the instance of the problem, containing only stable pairs.

1.1 The Men-Wimen problem

The problem about finding the stable matching between a set W and M. Each $w \in W$ rank every $m \in M$ from best to worst, and every $m \in M$ do the same.

Algorithm 1: proposeAndReject(M, W) :

Input: M set of men, W set of women
Output: S the set containing all the stable pairs between M and W

$S \leftarrow$ set containing all the pairs between M and W.

while $\exists m \in M$ *that is free and haven't proposed yet* **do**
  $w_i = m[0] \leftarrow$ first woman in m's list of preference to whom he is not yet proposed ;
  if w_i *is free* **then**
    $S = S \cup p(m, w_i) \leftarrow$ create a pair between m and w_i;
  end
  if w_i *prefers* m *to her current partner* m_i **then**
    $S = S - p(m_i, w_i)$;
    $S = S \cup p(m, w_i)$;
  end
end
return S;

- Observation 1
  Every men propose to a women decreasing order of preference, for every women is the opposite.
- Observation 2
  Once a women is matched she never become un-matched, she just change her partner, in increasing order of preference.
- Observation 3
  Assuming $|M| = |W|$ then the algorithm has $\mathcal{O}(n^2)$ complexity.
- Observation 4
  The algorithm is very resistant to modifications: example new conditions on engagements.

2 Asymptotic Order of Growth

2.1 Brute Force

For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution, usually taking 2^n.

2.2 Polynomial Running Time

This is the most desirable running time of any algorithm because it scales with ease at the increment of the input size.
An algorithm can be defined as poly-time if the following property holds:

$$\exists c > 0, \exists d > 0, \textit{ such that } \forall n \in INPUT \textit{ the running time is bounded by } cn^d$$

We can now give the following definition of efficiency:

$$\textit{An algorithm is efficient if it is poly} - \textit{time.}$$

When discussing the running time of a deterministic algorithm is important to take into account the worst-case scenario because, in doing that, the running time is guaranteed $\forall n \in INPUT$. There are some exception to this rule, when the worst-case scenario is very rare.

When discussing of probabilistic algorithm we need, instead, to reason upon the expected running time.

2.3 Big-O Notation

Assuming we have an algorithm rappresented by the function:

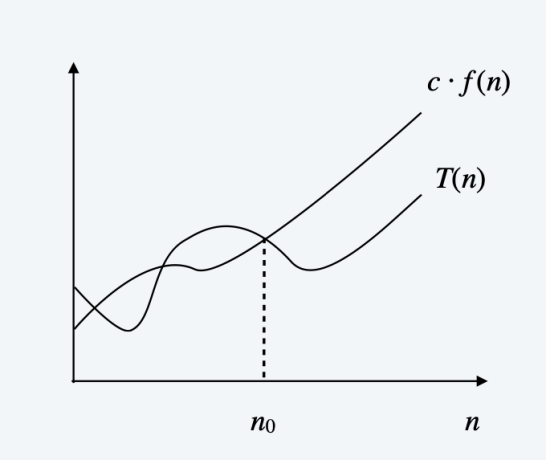
$$T(n) = 1.6n^2 + 3.5n + 8$$

We want a way to express the growing rate in a way that is insensitive to constant factors and low-ordered terms, in fact It can be said that T(n) grows like $\mathcal{O}(n^2)$.

- The lower bound $\mathcal{O}$
  We can say that T(n) is $\mathcal{O}(f(n))$ if $\exists c > 0, \exists n_0 \geq 0 / \ T(n) \leq cf(n), \ \forall n \geq n_0$
  We can give an alternative definition as follow:

$$\mathcal{O}(f(n)) = \lim_{n \to \infty} \frac{T(n)}{F(n)} < \infty$$

  Graphically:



Again, going back to our function: $T(n) = 1.6n^2 + 3.5n + 8$,

$$T(n) = n^3$$

$$T(n) = n^2$$

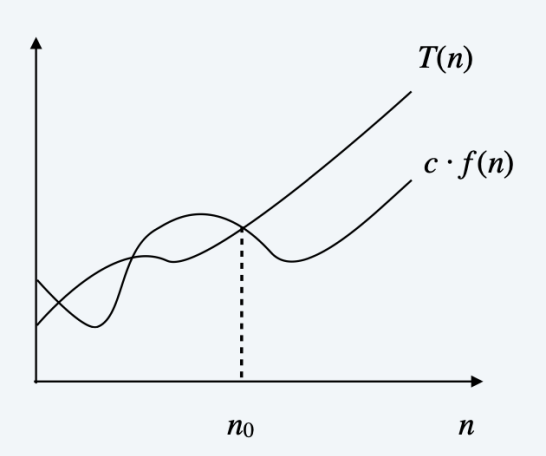
$$T(n) \neq n$$

- The upper bound Ω
  We can say that T(n) is $\Omega(f(n))$ $if \exists c > 0, \exists n_0 \geq 0 / \ T(n) \geq cf(n), \ \forall n \geq n_0$
  Again, going back to our function: $T(n) = 1.6n^2 + 3.5n + 8$,

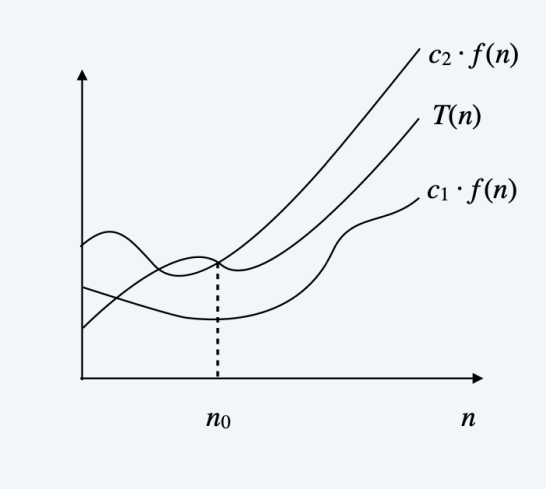
$$T(n) = \Omega(n)$$

since $T(n) \leq pn^2 \leq pn$.

Graphically:



- The tight bound Θ
  T(n) is $\Theta(f(n))$ if $\exists c_1 > 0, c_2 > 0$ *and* $\nexists$ n_0 *such that* $c_1 f(n) \leq T(n) \leq c_2 f(n)$ $\forall n \geq n_0$
  Graphically:



- Usefull Properties

  Transitivity:

  $$if\ f = \mathcal{O}(g)\ and\ g = \mathcal{O}(h),\ then\ f = \mathcal{O}(h)$$
  $$if\ f = \Omega(g)\ and\ g = \Omega(h),\ then\ f = \Omega(h)$$

  Sum of functions: Supposing that f and g are two functions such that, for some other function h, we have $f = \mathcal{O}(h)\ and\ g = \mathcal{O}(h),\ then\ f + g = \mathcal{O}(h)$.

2.4 Common Running Times

- Linear $\mathcal{O}(n)$

  The running time is proportional to the input size.

  Example: finding the maximum into an array.

- Linearithmic time $\mathcal{O}(nlogn)$

  Very common in divide-and-conquer algorithms, is also the running time of most of the sorting algorithms (ex: MergeSort or HeapSort).

  Note: if an algorithm A execute n times a sorting operation, let's say n=3, and the sorting operation is still the most computational costly operation in the whole algorithm: then A is still $\mathcal{O}(nlogn)$ because $3\mathcal{O}(nlogn) \simeq \mathcal{O}(nlogn)$.

- Quadratic time $\mathcal{O}(n^2)$

  Common when enumerating all pairs of elements, ex: given a list of n points in the plane (x_1, y_1), ..., (x_n, y_n), find the pair that is closest to each other.

- Cubic time $\mathcal{O}(n^3)$

  Example:Set disjointness. Given n sets: $S_1, ..., S_n$ each of which is a subset of 1, 2, ..., n, is there some pair of these which are disjoint?

- Ploynomial time $\mathcal{O}(n^k)$

  Example:Independent set of size k: Given a graph, are there k nodes such that no two are joined by an edge? The solution is to enumerate all subsets of k nodes.

- Exponential time

  Given a graph, what is maximum cardinality of an independent set? Enumerate all the possible subsets $\Rightarrow \mathcal{O}(n^2 2n)$

- Sublinear time

  Search in a sorted array. Given a sorted array A of n numbers, is a given number x in the array? $\Rightarrow \mathcal{O}(logn)$ Binary search.

2.5 Why it Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Figure 3: Comparison of the running times of different complexity algorithms.

Notice that, as we distance from the linear complexity, for bigger value of $n \in INPUT$, the execution time goes very high, this is bad: let's just say we want to deploy our non-linear algorithm on a distributed system dealing with a lots of records, as they grows, the system could become unable to process them in a feasible time.

2.6 The Heap Data Structure

Is a balanced binary tree T(V,E) that satisfy the following property, defining with C the set containing all the children of a node w_i

$$\forall w_i \in V, \forall v_i \in C,\ key(w_i) \leq key(v_i)$$

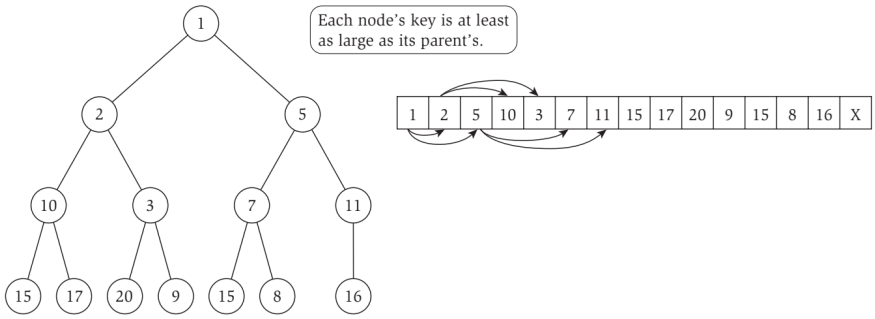


Figure 4: An example of heap.

This data structure is very helpfull, instead of using an array or a list, here are the complexity of some operations involving the heap structure:

$$create\ heap = \mathcal{O}(n)$$
$$insert\ heap = \mathcal{O}(logn)$$
$$find\ min = \mathcal{O}(1)$$
$$delete = \mathcal{O}(logn)$$

3 Greedy Algorithms

Formal definition: the greedy algorithm "stays ahead" locally: meaning that is better than any other solutions, alternatively we can prove that a greedy algorithm's solution is optimal by transforming the know optimal solution for the problem, into our solution, this method is called "argument exchange".

Practical definition: A lot of the time a greedy algorithms involves scanning all the $n \in INPUT$ of the problem, if the current element respect some conditions, then it can be added to the optimal solution returned by the algorithm.

3.1 The Cashier Problem / Coin changing

Given a currency with the following values: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Example: 34 dollars, how many coins?

Solution: at each iteration, add coin of the largest value that does not take us past the amount to be paid.

Algorithm 2: cashierAlgorithm(amount,C):

Input: *amount* containing the amount to give back to the client, C set containing all the coins of the current currency.
Output: S the set containing all the coins to give back to the client

$S \leftarrow$ set containing all the coins to give back to the client;
$total = 0 \leftarrow$ total expressed in the current currency system;
$Cs \leftarrow$ set containing all the coins in the currency sorted by increasing values;

```
while total < amount do
    for i=Cs.length to 0 do
        if total + Cs[i] < amount then
            total = total + Cs[i];
            S = S ∪ Cs[i];
            break;
        else if total + Cs[i] = amount then
            S = S ∪ Cs[i];
            return S;
    end
end
return S;
```

Claim The cashier algorithm always returns an optimal solution.

Proof The aim of the algorithm is to give as little change as possible (in terms of number of coins), if this is not true, then it must exist a solution $|S^*| < |S|$, but the algorithm always give highest compatible coin in the for loop, therefore a contradiction. □

3.2 Interval Scheduling

We have a set J of jobs to execute on a machine and $\forall j \in J\ j = (s_j, f_j)$ where s and f are start and finish.Two jobs are compatible if the don't overlap: $\forall j_i, j_j \in C\ f_i \geq s_j\ or\ vice - versa$.
Once we defined the problem, is just the matter of identifying the best strategy to sort all the jobs in J. Turns out, the best strategy is to sort all the jobs by finishing time, from first to last (for a former proof please check the professor's slide, but the main idea is to prioritize the jobs that release the machine as soon

as possible).

Algorithm 3: earliestFinishingTime(J):

Input: J set containing all the jobs
Output: S the set containing all the compatible jobs

$S \leftarrow$ set containing all the compatible jobs;
$Js \leftarrow$ set containing all the sorted jobs in increasing order of finishing time $f1 < ....f_j$

for *i=0 to Js.lenght* **do**
  if $J[i].start \geq S.last.finish$ **then**
    $S = S \cup Js[i]$
  end
end

return S;

Claim The earliestFinishingTime always return an optimal solution.
Proof Assume this is not true, then it must exist a solution $|S^*| > |S|$,containing at least one more compatible job, but the algorithm check every job in J, therefore a contradiction. □

Claim The complexity of the algorithm is $\mathcal{O}(nlogn)$
Proof The for loop costs $\mathcal{O}(n)$, while, as stated before, the sorting costs $\mathcal{O}(nlogn)$, since $\mathcal{O}(nlogn) > \mathcal{O}(n)$, the whole algorithm is $\mathcal{O}(nlogn)$. □

3.3 Interval Partitioning

Assuming we have a set of lectures L and, $\forall l \in L,\ l = (s, f)$ where s and f are start time and finishing time, find the minimum amount of classroom tho schedule all the lectures, so that no two lectures overlap in the same classroom (the definition of overlap is the same as 3.2).

Algorithm 4: intervalPartitioning(L):

Input: L set containing all the lectures
Output: S set containing the minimum amount of classroom needed

$S \leftarrow$ set containing the minimum amount of classroom needed;
$Ls \leftarrow$ set containing all the sorted lectures in increasing order of starting time $s1 < ....s_j$

for *i=0 to Ls.lenght* **do**
  $C = \varnothing$
  if $J[i].start \geq C.last.finish$ **then**
    $C = C \cup Ls[i]$;
  else
    $S = S \cup C$;
    $C = \varnothing$;
    $C = C \cup Ls[i]$;
  end
end

return S;

The number of classrooms needed is more or equal to the depth of $|S|$, that is also the maximum amount of items that contains at any given time.

Claim The intervalPartitioning always return an optimal solution.
Proof Same as 3.2, if it exist a better solution S^* we would have $|S^*| > |S|$ but that is not possible, since the algorithm checks every lecture in the input set. □

Claim The complexity of the algorithm is $\mathcal{O}(nlogn)$

Proof Identical as 3.2. □

3.4 Minimizing The Lateness

Assuming now we have just one resource that can process just one job at a time, we want to process all possible jobs. For the notation: $\forall j \in J$, j requires t_j amount of time to be executed, with a finishing time $f_j = s_j + t_j$, we have also to define a due date d_j and finally the lateness of a job as: $max(0, d_j - f_j)$

Algorithm 5: earliestDeadLineFirst(J):

Input: L set containing all the lectures
Output: S set containing the minimum amount of classroom needed

$S \leftarrow$ set containing all the compatible jobs;
$Js \leftarrow$ set of all jobs sorted by increasing deadline $s1 < ....d_j$

$t = 0$;
for *i=0 to Js.lenght* **do**
    $s_i = t$;
    $f_i = s_i + Js[i].finish$;
    $S = S \cup (s_i, f_i)$;
    $t = t + Js[i].time$;
end
return S;

Claim EarliestDeadLineFirst algorithm return an optimal solution.
Proof Identical as 3.2 / 3.3. □

Claim The complexity of the algorithm is $\mathcal{O}(nlogn)$
Proof Identical as 3.2. □

3.5 Dikstra Algorithm

Def: A path is a sequence of edges which connects a sequence of nodes.

Def: A cycle is a path with no repeated nodes or edges other than the starting and ending nodes.

Shortest Path problem: Given a digraph G = (V, E), edge lengths $le \geq 0$, source $s \in V$, and destination $t \in V$, find the shortest directed path from s to t.

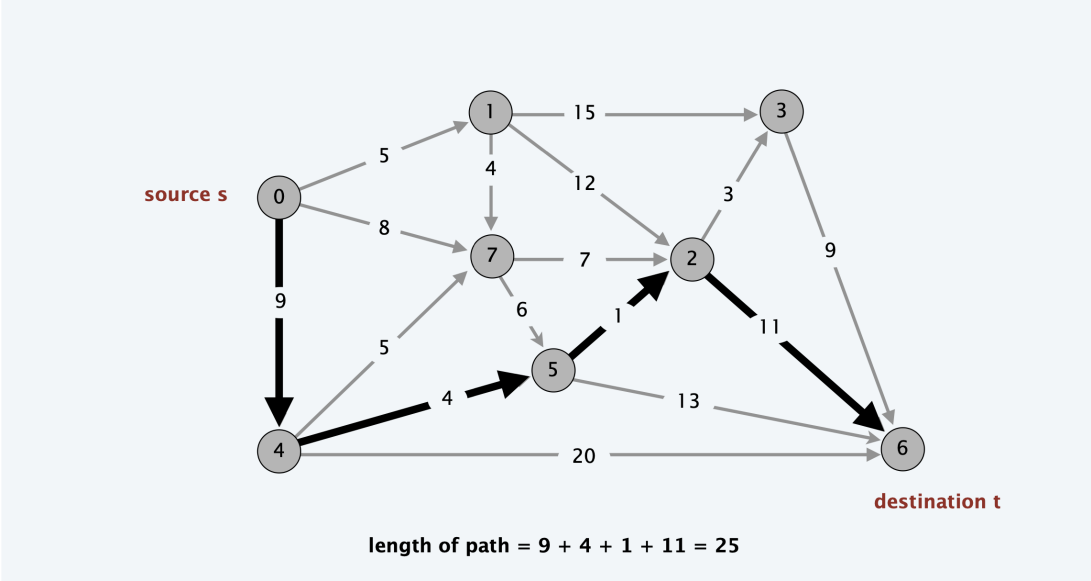


Figure 5: A solution for the Shortest path problem.

Algorithm 6: Dikstra(G,s):

Input: G set containing all the edges of G, s source of the path
Output: S set containing all the edges of a shortest path

$S \leftarrow$ set containing all the compatible jobs;
$Q \leftarrow$ priority queue for nodes contained in the unexplored part of G ;
$D \leftarrow$ set of distances from s $\forall\ e \in G$

$d(s, s) = 0$;
$D = D \cup d(s, s))$;

for *i=0 to G.lenght* **do**
    $e_i = G[i] \leftarrow$ current edge to check;
    if $e_i \neq s$ **then**
        $d(s, e_i) = \infty$;
        $D = D \cup d(s, e_i))$;
        $Q = Q \cup (e_i, d(s, e_i)) \leftarrow$ Insert in queue the current node with key$(e_i) = d(s, e_i)$;
end

while $Q \neq \emptyset$ **do**
    $u = Q.getMin$;
    $Q = Q - \{u\}$;
    foreach $(u, v) \in G$, *leaving* u **do**
        if $d(s, v) > d(s, u) + l(u, v)$ **then**
            $S = S \cup d(s, u) + l(u, v)$;
            decrease-key of v to d(u) + l(u, v) in Q.;
    end
end
return S;

Claim The Dikstra Algorithm always return an optimal solution ($\forall u \in S$, $d(u)$ is the length of the shortest $s \rightarrow u$ path).

Proof Base case: $|S| = 1$ is easy since S = s and d(s) = 0, assume that's true for $|S| > 1$, let v be next node added to S, and let (u, v) be the final edge: the shortest $s \rightarrow u$ path plus (u, v) is an $s \rightarrow v$ path of length $\pi(v)$. Now consider any $s \rightarrow v$ path P. We show that it is no shorter than $\pi(v)$. Let (x, y) be the first edge in P that leaves S,and let P' be the subpath to x.
P is already too long as soon as it reaches y. □

3.6 Kruskal Algorithm

Given an undirected, weighted, connected graph G=(V,E) we want to find the all the edges that connects all the nodes with a minimum cost, alternatively we can say that the algorithm finds the minimum spanning tree (MST) of G.

Formal definition: Given an undirected, weighted, connected graph G = (V, E) with edge costs $c(e)$, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge costs is minimized.

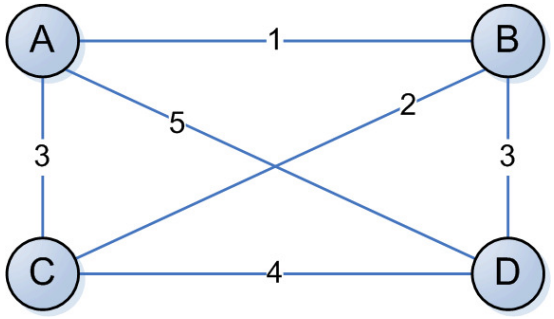


Figure 6: An instance of the MST problem.

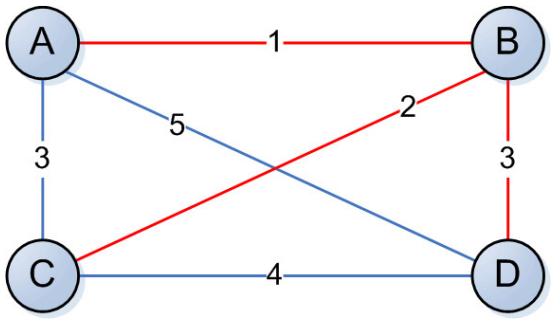


Figure 7: A solution for the MST problem.

Algorithm 7: kruskal(G):

Input: G set containing all the edges of G
Output: S set containing all the edges of MST

$S \leftarrow$ set containing all the compatible jobs;
$E \leftarrow$ set of all the edges sorted by increasing weight $c(e)_1 < ....c(e)_j$

for *i=0 to G.lenght* **do**
  $e_i = G[i] \leftarrow$ current edge to check;
  if $S \cup e_i$ *is not a loop* **then**
    $S = S \cup e_i$;
end

return S;

Claim Kruskal algorithm returns an optimal solution.
Proof Identical as 3.2 / 3.3. □

Claim The complexity of the algorithm is $\mathcal{O}(nlogn)$
Proof Identical as 3.2. □