Chapter 1

Lecture 1 – Basics and Greedy Algorithms

In this lecture we cover:

- Basics of algorithm analysis: efficiency, asymptotic order of growth, data structures
- Greedy Algorithms

1.1 Algorithms

1.1.1 Designing the algorithm

There are several definitions for algorithms, more or less formal. For us it is enough to say (following Knuth):

Definition. An **algorithm** is a step-by-step problem solving method, that fulfills the following:

- *Finiteness:* Has to stop after a finite number of steps
- *Definiteness:* The sequence of steps has to be uniquely determined.
- *Effectiveness:* Each step has to be sufficiently basic.

Then of course we want the algorithm to be *correct*, that is, it solves the problem at hand.

1.1.2 Efficiency and asymptotic order of growth

In this part we cover

- Definition of efficiency
- Asymptotic order of growth
- Common running times
- Implementation - data structures

Efficiency definitions:
(1) It runs quickly on real inputs...
(2) Achieves qualitatively better worst case performance than brute search ...
What is qualitatively better: if the input increases by a constant factor, the algorithm slows down with a constant factor.

Definition. (3) The algorithm is **efficient** if it has a polynomial running time (polynomial in input size). Formally: There are constants c and d so that on every input size N, the running time is bounded by cN^d. That is: $\exists c, d > 0$, such that, $T(N) \leq cN^d$, $\forall N$.

This definition fulfils our idea above: e.g., $N \to 2N$ then the running time bound increases as $cN^d \to c(2N)^d = cN^d s^d$. This is a slow down of 2^d, which is a constant factor.

How to characterize the running time of an algorithm?

- For increasing input size, express growth rate of *worst case* running times, disregard constant factors and low-order terms. This brings us to the notion of *asymptotic order of growth.*

Definition. Asymptotic upper bound, $O(f(n))$: For $T(n)$ worst case running time $T(n)$ is order of $f(n)$ $T(n) = O(f(n))$: $\exists c > 0$ and $n_0 \geq 0$, such that $T(n) \leq cf(n), \forall n \geq n_0$.

Note, it is somewhat lazy formalism: we mean $T(n) \in O(f(n))$, also n is typically natural number.

Definition. Asymptotic lower bound, $\Omega(f(n))$: $T(n) = \Omega(f(n))$: $\exists c > 0$ and $n_0 \geq 0$, such that $T(n) \geq cf(n), \forall n \geq n_0$..

Definition. Asymptotically tight bound, $\Theta(f(n))$: $T(n) = \Theta(f(n))$: $\exists c_1 > 0, c_2 > 0$ and $n_0 \geq 0$, such that $c_1 f(n) \geq T(n) \geq c_2 f(n)$ $\forall n \geq n_0$.

Corollary 1. *If* $\lim_{n\to\infty} T(n)/f(n) = c > 0$, *then* $T(n) = \Theta(f(n))$.

Proof. Definition of limit: we say that $\lim_{x\to\infty} g(x) = L$ if $\forall \varepsilon > 0$ $\exists M > 0$, such that $|g(x) - L| < \varepsilon$ for $x > M$. ******** ☐

Asymptotic bounds of some common functions

- transitivity: $f = O(g), g = O(h) \Rightarrow f = O(h)$. Same for Ω.
- polynomials: highest-order term matters, e.g., $T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p + q + r)n^2$
- logarithms: $\Theta(\log_a n) = \Theta(\log_b n)$ for $\forall a, b > 0$, that is, we do not need to specify the base
- $\log n = O(n^d)$ for **every** $d > 0$, that is, any logarithmic increase is slower than a polynomial increase
- $n^d = O(r^n)$, that is, an exponential growth is always faster than a polynomial
- Note: $r^n \neq \Theta(s^n)$, that is, for exponential growth, the base matters.

1.1.3 Common Running Times of Algorithms

Linear Time, $O(n)$: Typical: algorithms that process the input in a single pass, spending a constant amount of time on each input item, or other reasons: compute maximum, merge two lists.

Quick exercise: how do we prove that merging in $O(n)$? The algorithm gets back to the same item several times, we could guess, in the worst case n elements are considered n times. However, the output increases by 1 in each iteration!

$O(n \log n)$ Time: Algorithms that split the input into two, and process the parts (by splitting into two...). For many algorithms the sorting is the most expensive step. E.g., sorting.

Quadratic time $O(n^2)$, Polynomial Time $O(n^k)$: Nested loops, e.g., closets points on a plane.

Beyond Polynomial Time $O(k^n), O(n!)$: Exponential time, when all subsets need to be evaluated, factorial when all permulations need to be considered. E.g., brute force for matching, traveling salesman.

Sublinear Time: when the input can be queried (do not need to be read), and in each steps a part of the input is discarded. E.g., binary search.

1.1.4 Data structures

Arrays: query in $O(1)$, find in $O(n)$, or in $O(\log n)$ if already ordered. Cumbersome to add and remove.

Linked lists: Easy to add and remove element. To query i-th element takes $O(i)$ time.

Queue: Doubly linked list, remove first element, add as last element.

Stack: Doubly linked list, remove first element, add as first element.

Union-Find: to maintain connected components, implemented with array or with pointers. Union $O(1)$, find $O(\log n)$, set up $O(n)$.

Heap: To combine benefits of arrays and linked list, for data structures where elements have some kind of priority, *key(v)*. Heap is a *balanced* binary tree, where each node's key is at least as large as its parent's. (See figure 2.3) Note, the same set of key values may be stored in very different heaps.

We can store a heap in a nicely designed array: for any node in position i, the children nodes are in $2i$ and $2i+1$. For a node in position i, the parent is in position $\lfloor i/2 \rfloor$. Note the depth of a heap with n elements in $\log_2 n$.

- To add an element to the heap, attach to the bottom, and "switch it up" until a good position. It takes $O(\log n)$ steps.
- To delete an element from the heap, remove the element, move up the last element to the "hole", and then switch up or switch down the out of order element. It takes $O(\log n)$ steps.
- To find the minimum (but not remove it), take $O(1)$ time, it is always the first element.

1.2 Greedy algorithm

Outline:

- Greedy stays ahead - the interval scheduling example
- Exchange argument - job scheduling
- Greedy graph algorithms: shortest path, spanning tree and arborescence
- When greedy works - matroids

Greedy algorithms: there is no exact definition. Algorithms where the solution is found through a sequence of locally optimal steps.

1.2.1 Greedy Stays Ahead - Interval Scheduling

Problem. Interval scheduling. We have a set R of requests i, given by starting and finishing times $(s(i), f(i))$. A subsets of requests is compatible, if no two of them overlap in time. The goal is to find a maximum size compatible subset A.

We are looking for a simple rule to select the next interval to add to the set A. Ideas that do not work: earliest start time, shortest interval, fewest conflict.

Algorithm. Earliest Finish-time First algorithm (EFTF)

1. Sort intervals according to increasing finishing time

2. Select interval with smallest f_i^*, add to A

3. Remove from R all intervals that are not compatible with i^*

4. If R is nonempty, go back to step 3.

See algorithm in the book.

The simple description of the algorithm looks like requiring n^2 steps, but with smart data structures we only need to sort, and then go through the list only once, which gives running time of $O(n \log n)$.

Now we will prove the correctness of the algorithm (that is, that is gives a largest independent set) by the greedy stays ahead argument. The idea is to show that for every partial solution the greedy algorithm is doing the same or better than an optimal solution $\mathcal{O}$.

Lemma 1. *The algorithm provides a compatible set of intervals.*

Let us denote the intervals in A by $i_1, ...i_k$ and in $\mathcal{O}$ by $j_1...j_m$, ordered according to start and finish points, Our objective is to show that $k = m$.

Lemma 2. *For all $r \leq k$ we have $f(i_r) \leq f(j_r)$*

Note, that this means exactly, that the greedy stays ahead after each interval selection, compared to any optimal solution.

Proof. Proof by induction. The statement is true for $r = 1$, due to the definition of the algorithm.

Now consider step r. We know that $f(j_{r-1}) < s(j_r)$. Also $f(i_{r-1}) \leq f(j_{r-1})$, that is $f(i_{r-1}) < s(j_r)$. Since j_r is one of the available intervals, $f(i_r) \leq f(j_r)$ □

Theorem 1. *The EFTF algorithm returns an optimal set A.*

Proof. By contradiction. Assume that $m > k$. According to the lemma, $f(i_k) \leq f(j_k)$. It means that j_{k+1} is compatible with A, but the greedy algorithm stops, which is contradiction. □

Lots of extensions, e.g., Interval Partitioning Problem, we want to schedule all requests on as few resources as possible. Greedy algorithm exists to find the solution. Weighted interval scheduling, at the same time, can not be solved by simple greedy reasoning and will be addressed by dynamic programming.

1.2.2 Exchange argument – minimum lateness scheduling

This is a simple example for an optimality proof with exchange argument. Since Earliest Deadline First is a known scheduling concept, it may be interesting for networking research.

Problem. Minimum lateness scheduling: Given a set of requests with processing time $t(i)$ and deadline $d(i)$. Schedule the requests such that the maximum lateness $f(i) - d(i)$ is minimized.

Algorithm. Earliest Deadline First Scheduling (EDF)

1. Order the jobs in increasing order of deadlines.

2. Schedule them in this order.

The running time is $O(n \log n)$ because of the sorting. We prove the correctness of the algorithm, that is, that is finds the minimum lateness scheduling, by first stating a couple of simple observations about any correct (optimal) solution.

Lemma 3. *There is an optimal schedule with no idle time.*

Note, that there can be optimal solutions whit idle time. Once the job with the longest lateness is served, the others can be served with some time lag.

Let us call inversion where a job i is served before job j, while $d(j) < d(i)$.

Lemma 4. *All schedules with no inversion and no idle time have the same maximum lateness.*

Proof. Such schedules can differ only in the order of jobs with the same deadline. □

Note that EDF produces such a schedule, with no inversion and with no idle time. That is, if we can prove that there is an optimal solution with these properties, our solution A is optimal too.

Lemma 5. *There is an optimal schedule that has no inversions and no idle time.*

Proof. We prove the lemma through three statements.

1. If an optimal schedule $\mathcal{O}$, with no idle time, has inversion, then there is a pair of inverted jobs i, j right after each other.

2. After swapping these, we get a schedule with less inversion. (No new inversion is created.)

3. The new swapped schedule has a maximum lateness no larger than that of $\mathcal{O}$. This also means that the new schedule is optimal as well.

Then the lemma follows. □

Note that this part corresponded to the exchange argument. We step by step transformed an optimal solution to the structure that the EDF algorithm constructs.

Theorem 2. *The schedule A produced by EDF has optimal maximum lateness.*

Proof. Follows from the lemmas above. □

Reading: similar proof for optimal caching.

1.2.3 Shortest path in weighted graphs – greedy stays ahead

Problem. Shortest path: Consider a (directed) graph $G = (V, E)$, with length of edges $\{l_e\}, l_e \geq 0$. $|V| = n, |E| = m$. Determine the shortest path from a source node s to all other nodes.

Algorithm. The Dijksta algorithm

```
Let $S$ be the set of explored nodes, for each $u \in S$, we have distance $d(u)$.
Initially $S = \{s\}$ and $d(s) = 0$.
While $S \neq V$
Select a $v \notin S$, that has the lowest $d'(v) = \min_{e=(u,v),u \in S} d(u) + l_e$
$S = S \cup v$, $d(v) = d'(v)$
EndWhile
```

Dijsktra algorithm is greedy in the sense that in each step it fixes the distance to a node that can be reached with the lowest distance. We prove its correctness using the *greedy stays ahead* argument.

Theorem 3. *Consider the set S at any point of the algorithm's execution. For each $u \in S$, the path P_u is the shortest path.*

Proof. Proof by induction. Also refer to figure 5.8 (4.8). Correctness for $|S| = 1$ follows from the algorithm definition. Assume, the theorem holds for $|S| = k$, and let the new node we add be v, and the final edge of P_v be (u, v).

We consider any other path P to v, and show that it is at least as long as P_v. The last node on P still in S is x. The subpath $s - x$ is P'. The next node to x is y.

$$l(P') \geq l(P_x) = d(x),$$

$$l(P) \geq l(P') + l(x, y) \geq d(x) + l(x, y) \geq d'(y).$$

But since Dijkstra selected v in this iteration

$$d'(y) \geq d'(v) = l(P_v)$$

That is,

$$l(P) \geq l(P_v).$$

□

Implementation and running time: in a simple implementation we add nodes one by one to S, and in each step we need to consider all edges from S, which leads to a running time of $O(nm)$. With smart implementation $O(m \log n)$ is possible.

Reading: implementation with priority queue (that is, heap).

Comment: note that we need the positive weight property. Check, where is the point when the proof breaks.

1.2.4 Minimum spanning three – exchange argument

Problem. Minimum spanning tree: Consider an connected undirected graph $G = (V, E)$, with cost of edges $\{c_e\}, c_e \geq 0$. $|V| = n, |E| = m$. Find $T \subseteq E$, such that graph (V, T) is connected, and the total cost $\sum_T c_e$ is minimized.

Lemma 6. *If T is a minimum cost solution, then (V, T) is a tree.*

There are many greedy algorithms that work:

Kruskal's Algorithm: start from empty set, and add minimum cost edge that does not produce cycle.

Reverse-Delete Algorith: start from the complete graph, and delete most expensive edge, that does not make the graph disconnected.

Prim's Algorithm: like Dijkstra, but add the node with minimum c_e.

Here we consider Kruskal's algortihm. The other proofs are similar.

Lemma 7. *Cut property. Assume all edge costs are distinct. Let $\emptyset \subset S \subset V$. Let $e = (v, w)$ be the minimum cost edge with $v \in S, w \in V - S$. Then every minimum cost spanning tree contains e.*

Proof. We prove this using the exchange argument. Let T be a spanning tree, $e \notin T$. We show that T is not minimum cost. We do this by finding an $e' \in T$ that can be exchanged with e: this is the edge e' that is on the path from v to w in T, connecting S and $V - S$. Since $c_e < c_{e'}$, exchanging them we get a cheaper spanning tree (need to prove that we get a tree, and that it is cheaper). □

Theorem 4. *Kruskal's algorithm produces a minimum cost spanning tree of G.*

Proof. We need to prove, that output of the Algorithm is a spanning tree.
i) It does not contain cycles, due to the algorithm design.
ii) It is connected, since otherwise the algorithm would not terminate.

Now we prove that the algorithm is correct - that is, adds the right edegs. Let's consider an edge $e = (v, w)$, when added to the spanning tree. Let's S be the set of nodes v had a path to before e is added. Clearly, $v \in S$, $w \notin S$. Also, there is no edge from S to $V - S$ with lower cost than e, since that would have been added already. Thus e is the cheapest edge, and has to be part of each spanning tree, according to the Lemma. □

Reading: the implementation of Kruskal's algorithm. Note, that it, in each step, requires the evaluation that no cycle is formed, which is costly. Union-Find data structure is needed to reach a running time of $O(mlogn)$.

Not clear: What happens when edge costs are not distinct?

1.2.5 Greedy Algorithms and Matroids

Definition. *Subset system.* Consider a set E and set of its subsets I. In a subset system I is closed under inclusion, that is if $X \subseteq Y$ and $Y \in I$ then $X \in I$.

We also refer to I as independent sets, and sometimes we refer to subset systems as independence systems.

Examples for (E, I) pairs: (edges in a graph G, set of acyclic sets of edges in G), (set of vectors, set of sets of linearly independent vectors)

Definition. *Matroid.* A subset system (E, I) is a matroid if the *exchange property* holds: for $X, Y \in I$, if $|X| < |Y|$, then $\exists e \in Y \setminus X$, such that $X \cup \{e\} \in I$.

Definition. X is a maximal set in I, if there are no set $Y \in I$, such that $X \subseteq Y \subseteq E$. The maximal independent sets are called *bases.*

This leads immediately to an alternative definition. (It is actually not needed to prove our main theorem on the greedy algorithm.)

Definition. *Matroid.* A subset system (E, I) is a matroid if and only if it satisfies the *cardinality property*: for any set $A \subseteq E$, all maximal independent sets in A have the same cardinality (number of elements).

Lemma 8. *Any two maximal sets $X, Y \in I$ has the same cardinality.*

Proof. Follows from the exchange property. Can be easily extended to the second definition. $\square$

Comment: not clear why if in the first definition and iff in the second.

Problem. The optimization problem over subset systems: for a subset system (E, I) with weights given by $w(e) > 0$, find a maximum weight bases.

Algorithm. Greedy algorithm

Input: $(E, I), \{w(e)\}$
Output: maximum weight bases $X\ i \in I$.

```
Sort $e \in E$ by weight, heaviest first.
For each $e$ in this order, add to $X$ if the result is in $I$.
Return $X$.
```

Theorem 5. *For any subset system (E, I) the greedy algorithm solves the optimization problem, if and only if (E, I) is matroid.*

Proof. First we prove that if (E, I) is a matroid, then the greedy algorithm is correct. Let X be the result of the greedy algorithm, and Y an optimal set. We prove that $w(X) \geq w(Y)$.

1) X, Y are both bases (since we have a matroid)

2) By contradiction. Let us order the elements by decreasing weight, $X = \{x_1, x_2 ... x_n\}, Y = \{y_1, y_2 ... y_n\}$. If $w(Y) > w(X)$ then there is a a smallest k, where $w(x_k) < w(y_k)$. Let us denote the first k elements by index k. Let $Z_k = X_{k-1} \cup \{e\}$, $e \in Y_k \setminus X_{k-1}$. Due to the exchange argument $Z \in I$.

Note that each element in Y_k is greater than x_k, and therefore $w(Z) > w(X_k)$, which means that the greedy algorithm does not selected a highest weight elements. Which is a contradiction.

Now we prove that if the greedy algorithm is correct for a subset system (E, I), then it is a matroid. The proof is by example. That is, we give an example when the exchange property does not hold and greedy fails.

Consider X, Y, $m = |X| < |Y|$, and the exchange property does not hold: no elements of Y can be added to X.
Lets set weights of elements in X as $m + 2$
Elements of $Y \setminus X$ as $m + 1$
Other elements weight as 0.

The greedy achieves a weight of $m(m + 2) = m^2 + 2m$. An optimal algorithm can at least find Y with a minimum possible weight $(m + 1)(m + 1) = m^2 + 2m + 1$. $\square$

1.2.6 Some examples and counter examples for matroids

- Matroid: acyclic set of edges, with spanning trees as bases:
  - They are subset systems.
  - All bases have the same size (second definition)
- Not a matroid: matching in G (no common vertex):
  - They are subset systems.
  - The exchange property does not hold (the greedy algorithm does not find the optimum). E.g., 3+3 bipartite graph.

1.2.7 What have we learned?

Algorithms and running times:

- Efficiency of algorithms is measured by the asymptotic growth of the worst case running time.
- Upper, lower and tight bounds.
- Typical running times and how they arise.
- Data structures affect the running time of the algorithms: array, list, heap

Greedy algorithms:

- The definition of greedy algorithms
- Design of greedy algorithms: stay ahead, exchange argument
- Famous greedy algorithms: shortest path, spanning tree
- Matroids - where greedy always works