Computational Geometry

Chapter 4

Linear Programming

On the Agenda

- Linear programming
- Smallest enclosing disk

Linear Programming: Definition

- Define:
  - x_i – the amount of food of type i – variables ($1 \le i \le d$).
  - j – types of vitamins ($1 \le j \le n$).
  - a_{ji} – the amount of vitamin j in one unit of food i.
  - c_i – the number of calories in one unit of food i.

- Constraints (we need to consume some minimal amount of every vitamin):

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1d}x_d \ge b_1$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nd}x_d \ge b_n$$

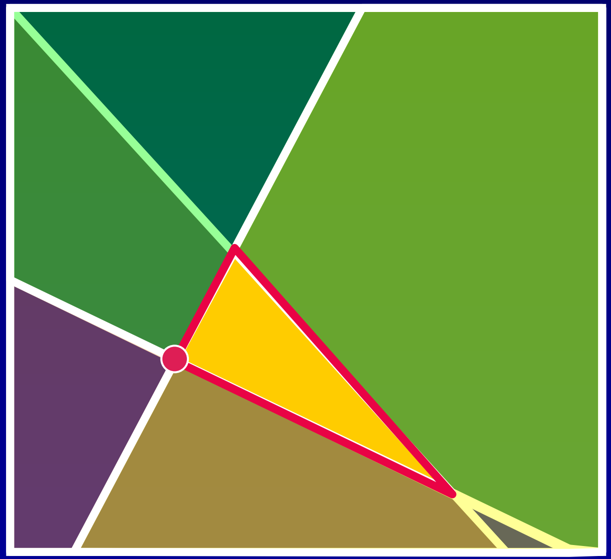
$$\text{Minimize}: \ c^T x$$
$$\text{Subject to}: \ Ax \ge b$$

- Minimize: the total number of calories consumed:

$$C(x) = c_1x_1 + c_2x_2 + \cdots + c_dx_d$$

Linear Programming: Geometry

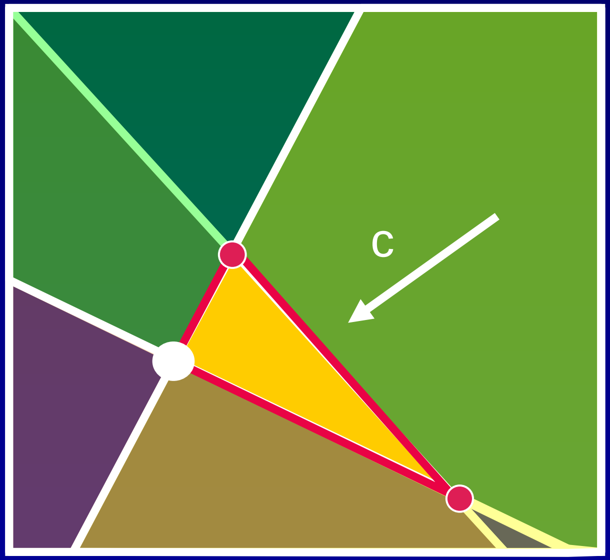
- Each constraint defines a half-space in the *d*-dimensional space.
- The *feasible region* is the (convex) intersection of these half-spaces.
- **Question**: Why is the feasible region convex?
- We will discuss the planar case (*d* = 2), in which each constraint defines a *half-plane*.



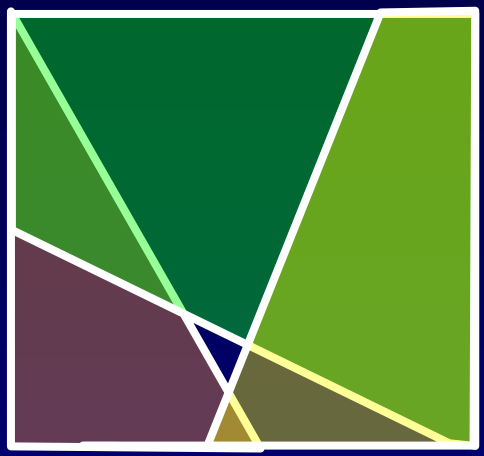
More Geometry

- The solution to the linear program is the (or a) point in the feasible region that is extreme in the direction of the *target function*.

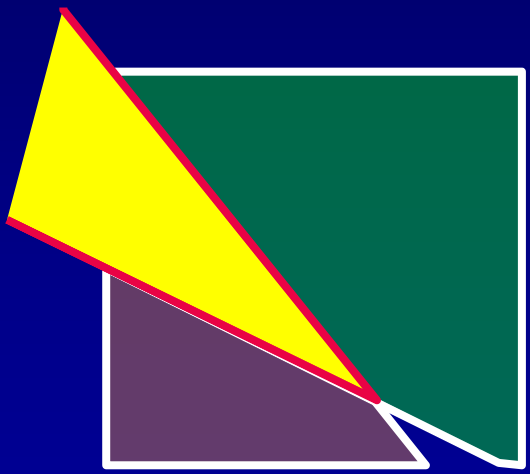
- **Observation:** Any bounded linear program that is feasible either has
  - A unique solution, which is a *vertex* of the feasible region; or
  - Infinitely-many solutions that are a face of the feasible region which is perpendicular to the target function.
- **Proof:** By convexity.



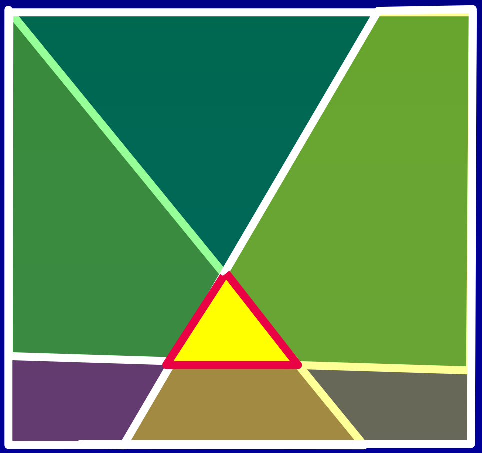
Degenerate Cases



- The feasible region may be:
  - Empty
  - Unbounded
  - A line/ray/line-segment
  - A point



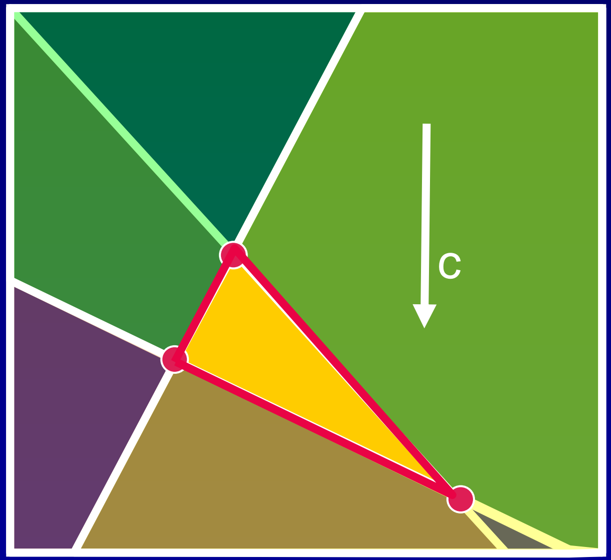
- The solution may be:
  - Not unique



The Simplex Algorithm

- Assume, without loss of generality, that the target function points “downwards”.
- Construct (some of) the vertices of the feasible region.
- Walk edge by edge downwards until reaching a local minimum (which is also a global minimum).

- In R^d, the number of vertices might be $\Theta(n^{\lfloor d/2 \rfloor})$, and the algorithm may traverse $\Theta(n^{\lfloor d/2 \rfloor})$ of them.



History of Linear Programming

- Mid 20th century: Simplex algorithm, time complexity $\Theta(n^{\lfloor d/2 \rfloor})$ in the **worst case**. Practically, this algorithm is commonly used due to its efficient **expected** running time (linear in *n*).
- Early 1980's: Khachiyan's ellipsoid algorithm with time complexity poly(*n*,*d*).
- Early 1980's: Karmakar's interior-point algorithm with time complexity poly(*n*,*d*).
- 1984: Nimrod Megiddo's parametric-search algorithm:
  - Time complexity $O(C_d n)$ (linear in *n*), where C_d is a constant dependent only on *d*.
  - His initial constant was as high as 2^{2^d}.
  - Later the constant was improved to 3^{d^2}.
  - There were further improvements of C_d.
  - This is optimal when *d* is constant.

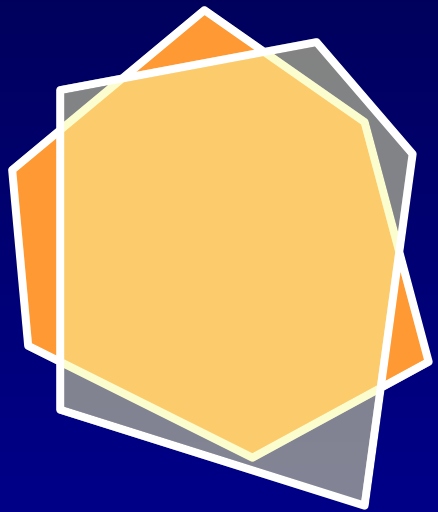
O(n log n)-Time D&C 2D-LP Algorithm

- Input:
  - *n* half-planes.
  - A target function that (w.l.o.g.) points down.

- Algorithm:
  1. Construct the feasible region of the whole problem:
     a. Partition the *n* half-planes into two sets of size *n*/2.
     b. Compute recursively the feasible region for each group.
     c. Compute the intersection of the two feasible regions.
  2. Check the target function on the vertices of the feasible region.

D&C: Time-Complexity Analysis

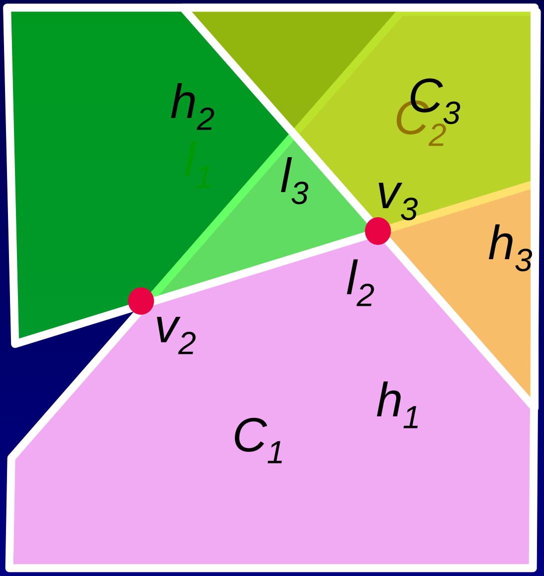
- The complexity of the intersection of two convex n-gons is $O(n)$. Why?
- Stage 1.c:
  - Intersection of two convex polygons (of $\leq n$ vertices): solved by a plane-sweep algorithm.
  - No more than *four* segments are simultaneously in the SLS, and there are $O(n)$ events (vertices and intersections) in the EQ. Total time: $O(n)$; Worst case: $\Theta(n)$ time.
- Stage 2:
  - Time of finding the vertex minimizing the target function: $O(\log n)$.
- The total time is the solution of the recursive equation $T(n) = 2T(n/2) + O(n)$, which is $T(n) = O(n \log n)$.



O(n^2)-Time Incremental Algorithm

- Start by intersecting two halfplanes.
- Add halfplanes one by one, and update the optimum vertex by solving a 1-D linear-programming problem on the new line.
- We will handle first the addition of a halfplane when the feasible region is already bounded; then we will handle the unbounded case.

Incremental Algorithm: Notation



Definitions:

h_i : the i^{th} halfplane

l_i : the line that defines h_i

C_i : the feasible region after i constraints

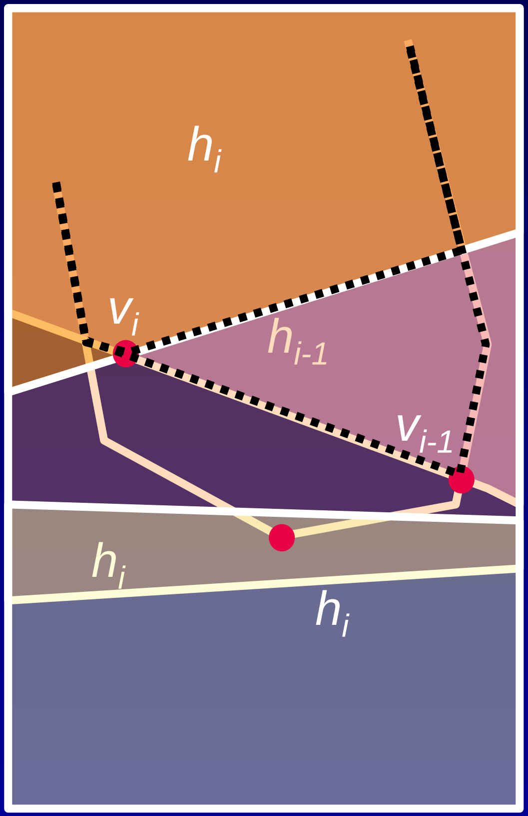
V_i : the optimum vertex of C_i

Incremental Algorithm: Basic Theorem

- Theorem:
  1. If $v_{i-1} \in h_i$, then $v_i = v_{i-1}$.
  2. If $v_{i-1} \notin h_i$, then either
     a. $C_i = \emptyset$

     or

     b. $C_i = C_{i-1} \cap h_i$ and v_i lies on l_i.



- Proof:
  1. Trivial. Otherwise v_i would not have been optimum before.

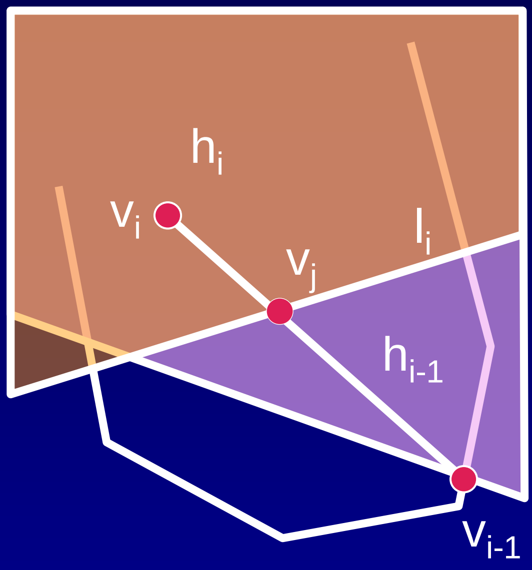
  2a. Also trivial.

Basic Theorem (cont.)

2b. Assume on the contrary that v_i is not on l_i. v_i must be in C_{i-1}. By convexity, the entire line segment $v_i v_{i-1}$ is in C_{i-1} .

Consider v_j, the intersection point of the segment $v_i v_{i-1}$ with l_i. By definition, v_j is in C_i, and by linearity it is better than v_i.

This is a **contradiction**.



Incremental Step: Given v_{i-1} & h_i, Find v_i

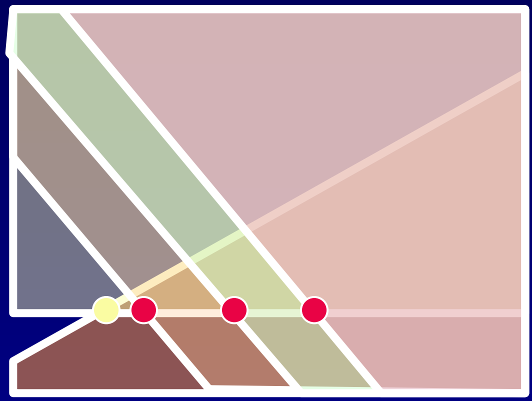
- If $v_{i-1} \in h_i$ (can be checked in O(1) time), then don't do anything ($v_i = v_{i-1}$).
- Intersect all h_j ($j<i$) with l_i, generating i-1 rays representing feasible half-unbounded intervals (in the direction of the target function).
- If l_j and l_i are parallel, then the entire line is either good (so ignore it), or bad (so report "no solution").
- Intersect the i-1 rays in $\Theta(i)$ time. How?
- If the intersection is empty, then report "no solution", else report the lowest point. How?

Complexity Analysis

- Time:

$$T(n) = \sum_{i=3}^{n} O(i) = O(n^2)$$

(Summation starts from 3 since two halfplanes that certify that the problem is bounded are found in the initialization step.)



$\Theta(n^2)$ in the worst case.

- Space: $\Theta(n)$.

Unbounded LP

- Input: The entire LP program.
- Output: An indication that the feasible region is either
  - A. Unbounded (+ a ray completely contained in it); or
  - B. Bounded (+ two of the halfplanes that make it so).
- Algorithm: See in [BKOS, §4].
- Time: $\Theta(n)$.
- Space: $\Theta(n)$.
- The time & space of the entire algorithm remain the same.
- Comments:
  - The procedure *may* detect that the problem is infeasible.
  - When we are not interested in infinite solutions, we can arbitrarily define a target function, based on the first two halfplanes, that makes the problem bounded.

An $\Theta(n)$-Time Randomized Version

- Is there a **good** order that will make the algorithm run in $\Theta(n)$ time? Yes, there is, but unfortunately finding this order requires $O(n^2)$ time. ☹

- The randomized version is exactly like the deterministic one, except that the **order** of the lines is **random**.

- **Theorem:** The **expected** running time of the random incremental algorithm (over all n! permutations of the halfplanes) is $\Theta(n)$.

Complexity Analysis

- There are *n* iterations.
  If $v_i = v_{i-1}$ (no optimum change): O(1) time;
  Otherwise: O(*i*) time.
- Define random variables

$$x_i = \begin{cases} 1 & v_i \neq v_{i-1} \\ 0 & v_i = v_{i-1} \end{cases}$$

- The expected running time is:

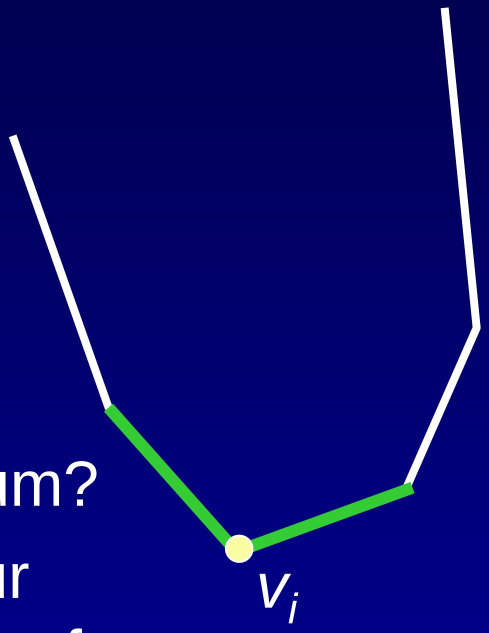
$$\sum_{i=3}^{n} [O(1)(1 - E(x_i)) + O(i)E(x_i)] \le O(n) + \sum_{i=3}^{n} [O(i)E(x_i)]$$

Complexity Analysis (cont.)

Backward analysis:

- Q: What is E[x_i]?

  A: Exactly Pr[$v_{i-1} \notin h_i$].
- Question: So, when given the optimum after *i* halfplanes, what is the probability that the *last* halfplane affected the optimum?
- Answer: 2/*i*, because a change can occur only if the last processed halfplane is one of the two halfplanes that define v_i.
- More precisely:
  - *At most 2/i*, to take into account three lines passing through v_i.
  - It is actually 2/(*i*-2), since the first two halfplanes are fixed.



Complexity Analysis (cont.)

$$E(x_i) = \Pr(v_i \neq v_{i-1}) \approx \frac{2}{i}$$

$$O(n) + \sum_{i=3}^{n} O(i)E(x_i) = O(n) + O\left(\sum_{i=3}^{n} i \cdot \frac{2}{i}\right) = O(n)$$

Expected $\Theta(n)$ Time.

Just to Make Sure…

- False Claim:

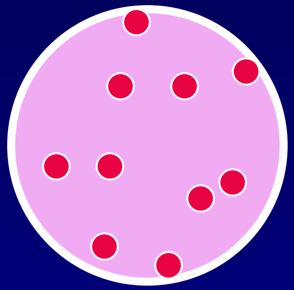
  The probabilistic analysis is for the average **set of halfplanes**. Hence, there exist bad sets of constraints for which the algorithm's expected running time is $\omega(n)$ (*more* than $\Theta(n)$), and there exist good sets of constraints for which the algorithm's expected running time is $o(n)$ (*less* than $\Theta(n)$).

- True Claim:

  The probabilistic analysis is valid for *all* sets of halfplanes. The expected time complexity is over all *permutations* of any set of halfplanes. In this respect all sets are "good".

Smallest Enclosing Disk

- **Input:** *n* points.
- **Output**: The disk of minimum radius that encloses all the points.



- **Theorem:** Let *P* be a finite set of points, and let *D* be its smallest enclosing disk.
  1. The length of an arc of *D* defined by consecutive points is at most π.
  2. If *D* is defined by two points of *P*, then these two points are diametrical on *D*.

- This immediately implies an O(n^4)-time algorithm. (How ?)

Underlying Theorem

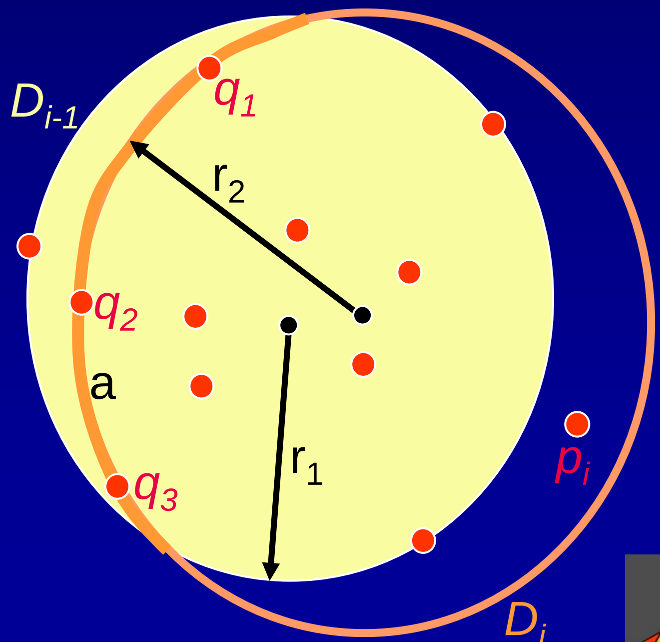
Idea: Use an incremental algorithm, processing one point at a time.

Notation: D_i is the smallest enclosing disk of the first i points.

Theorem: If $p_i \notin D_{i-1}$ then p_i is on the *boundary* of D_i.

Proof:

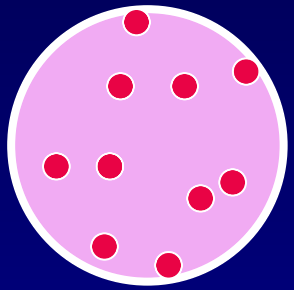
By a continuous deformation between D_{i-1} and D_i.



Expected $\Theta(n)$-Time Incremental Algorithm

Procedures:

- MinDisk(P): Find the smallest enclosing disk of a set of points P.
- MinDisk1(P,q): Find the smallest enclosing disk of a set of points P, *given* that some point q is on its boundary.
- MinDisk2(P,q_1,q_2): Find the smallest enclosing disk of a set of points P, *given* that some points q_1 and q_2 are on its boundary.
- Disk(q_1,q_2,q_3): Find the disk defined by three non-collinear points q_1, q_2, and q_3. (Obvious.)



Incremental Algorithm (cont.)

MinDisk(P):

- D_2 = the minimum disk defined by p_1 and p_2. (That is, the disk whose diameter is p_1p_2.)
- For each point p_i ($3 \le i \le n$):
  - If $p_i \in D_{i-1}$ then $D_i = D_{i-1}$;
  - Else D_i = MinDisk1(P_{i-1}, p_i).
- Return D_n.

Incremental Algorithm (cont.)

MinDisk1(P,q):

- D_1 = the minimum disk defined by q and p_1. (That is, the disk whose diameter is qp_1.)
- For each point p_i ($2 \le i \le |P|$):
  - If $p_i \in D_{i-1}$ then $D_i = D_{i-1}$;
  - Else D_i = MinDisk2(P_{i-1},q,p_i).
- Return D_n.

Incremental Algorithm (cont.)

MinDisk2(P,q_1,q_2):

- D_0 = the minimum disk defined by q_1 and q_2. (That is, the disk whose diameter is q_1q_2.)
- For each point p_i ($1 \le i \le |P|$):
  - If $p_i \in D_{i-1}$ then $D_i = D_{i-1}$;
  - Else D_i = Disk(q_1,q_2,p_i).
- Return D_n.

Time-Complexity Analysis

- Use backward analysis for a random point ordering.
- Total expected time complexity:
  - In the lowest level: $\sum_{i=1}^{|P|} O(1) = O(|P|)$
  - In the middle level: $\sum_{i=2}^{|P|} (O(1) + O(i)\frac{2}{i}) = O(|P|)$
  - In the highest level: $\sum_{i=3}^{n} (O(1) + O(i)\frac{3}{i}) = O(n)$
- Question: Why $2/i$ and $3/i$?
- Linear expected running time.
- Worst case: $\Theta(n^3)$. (When?)