```
# lightweight poly for abstract cost estimation
abstract type AbstractPoly{D,T<:Number} end
Base.one(x::AbstractPoly) = one(typeof(x))
Base.zero(x::AbstractPoly) = zero(typeof(x))

function Base.show(io::IO, p::AbstractPoly{D,T}) where {D,T<:Real}
    N = degree(p)
    for i = N:-1:0
        if i > 0
            print(io, "$(abs(p[i]))*")
            print(io, "$D")
            i > 1 && print(io, "^$i")
            print(io, p[i-1] < 0 ? " - " : " + ")
        else
            print(io, "$(abs(p[i]))")
        end
    end
end
function Base.show(io::IO, p::AbstractPoly{D,T}) where {D,T<:Complex}
    N = degree(p)
    for i = N:-1:0
        if i > 0
            print(io, "($(p[i]))*")
            print(io, "$D")
            i > 1 && print(io, "^$i")
            print(io, " + ")
        else
            print(io, "($(p[i]))")
        end
    end
end
struct Power{D,T} <: AbstractPoly{D,T}
    coeff::T
    N::Int
end
degree(p::Power) = p.N
Base.getindex(p::Power{D,T}, i::Int) where {D,T} = (i == p.N ? p.coeff :
zero(p.coeff))
Power{D}(coeff::T, N::Int = 0) where {D,T} = Power{D,T}(coeff, N)

Base.one(::Type{Power{D,T}}) where {D,T} = Power{D,T}(one(T), 0)
Base.zero(::Type{Power{D,T}}) where {D,T} = Power{D,T}(zero(T), 0)

Base.convert(::Type{Power{D}}, coeff::Number) where {D} = Power{D}(coeff, 0)
Base.convert(::Type{Power{D,T}}, coeff::Number) where {D,T} = Power{D,T}(coeff, 0)
Base.convert(::Type{Power{D,T}}, p::Power{D}) where {D,T} = Power{D,T}(p.coeff, p.N)

function Base.show(io::IO,p::Power{D,T}) where {D,T}
    if p.coeff == 1
    elseif p.coeff == -1
        print(io, "-")
    elseif isa(p.coeff, Complex)
        print(io, "($(p.coeff))")
    else
```

```
        print(io, "$(p.coeff)")
    end
    p.coeff == 1 || p.coeff == -1 || p.N == 0 || print(io, "*")
    p.N == 0 && (p.coeff == 1 || p.coeff ==- 1) && print(io, "1")
    p.N > 0 && print(io, "$D")
    p.N > 1 && print(io, "^$(p.N)")
end

Base.:*(p1::Power{D}, p2::Power{D}) where {D} = Power{D}(p1.coeff*p2.coeff,
degree(p1)+degree(p2))
Base.:*(p::Power{D}, s::Number) where {D} = Power{D}(p.coeff*s, degree(p))
Base.:*(s::Number, p::Power) = *(p,s)
Base.:/(p::Power{D}, s::Number) where {D} = Power{D}(p.coeff/s, degree(p))
Base.:\(s::Number, p::Power) = /(p,s)
Base.:^(p::Power{D}, n::Int) where {D} = Power{D}(p.coeff^n, n*degree(p))

struct Poly{D,T} <: AbstractPoly{D,T}
    coeffs::Vector{T}
    function Poly{D,T}(coeffs::Vector{T}) where {D,T}
        if length(coeffs) == 0 || coeffs[end] == 0
            i = findlast(!iszero, coeffs)
            return i === nothing ? new{D,T}(T[0]) : new{D,T}(coeffs[1:i])
        else
            return new{D,T}(coeffs)
        end
    end
end
degree(p::Poly) = max(0,length(p.coeffs)-1)
Base.getindex(p::Poly{D,T}, i::Int) where {D,T} = (0<=i<=degree(p) ? p.coeffs[i+1]
: zero(p[0]))
Poly{D}(coeffs::Vector{T}) where {D,T} = Poly{D,T}(coeffs)
Poly{D}(c0::T) where {D,T} = Poly{D,T}([c0])
Poly{D}(p::Power{D,T}) where {D,T} = Poly{D,T}(vcat(zeros(T, p.N), p.coeff))
Poly{D,T}(c0::Number) where {D,T} = Poly{D,T}([T(c0)])
Poly{D,T1}(p::Power{D,T2}) where {D,T1,T2} =
Poly{D,T1}(vcat(zeros(T1,p.N),T1(p.coeff)))

Base.one(::Type{Poly{D,T}}) where {D,T} = Poly{D,T}([one(T)])
Base.zero(::Type{Poly{D,T}}) where {D,T} = Poly{D,T}([zero(T)])

Base.convert(::Type{Poly{D}}, x::Number) where {D} = Poly{D}([x])
Base.convert(::Type{Poly{D,T}}, x::Number) where {D,T} = Poly{D,T}(T[x])
Base.convert(::Type{Poly{D}}, p::Power{D}) where {D} =
Poly{D}(vcat(fill(zero(p.coeff), p.N), p.coeff))
Base.convert(::Type{Poly{D,T}}, p::Power{D}) where {D,T} =
Poly{D,T}(vcat(fill(zero(T), p.N), convert(T, p.coeff)))
Base.convert(::Type{Poly{D,T}}, p::Poly{D}) where {D,T} =
Poly{D,T}(convert(Vector{T}, p.coeffs))

Base.:+(p::AbstractPoly{D},s::Number) where {D} = Poly{D}([p[i]+ifelse(i==0, s,
zero(s)) for i=0:degree(p)])
Base.:+(s::Number,p::AbstractPoly) = +(p,s)
Base.:+(p1::AbstractPoly{D}, p2::AbstractPoly{D}) where {D} = Poly{D}([p1[i]+p2[i]
for i=0:max(degree(p1),degree(p2))])
```

```
Base.:-(p::Poly{D}) where {D} = Poly{D}(-p.coeffs)
Base.:-(p::AbstractPoly{D},s::Number) where {D} = Poly{D}([p[i]-ifelse(i==0, s,
zero(s)) for i=0:degree(p)])
Base.:-(s::Number,p::AbstractPoly{D}) where {D} = Poly{D}([-p[i]+ifelse(i==0, s,
zero(s)) for i=0:degree(p)])
Base.:-(p1::AbstractPoly{D}, p2::AbstractPoly{D}) where {D} = Poly{D}([p1[i]-p2[i]
for i=0:max(degree(p1),degree(p2))])

Base.:*(p1::Power{D}, p2::Poly{D}) where {D} = Poly{D}([p1.coeff*p2[n-degree(p1)]
for n=0:degree(p1)+degree(p2)])
Base.:*(p1::Poly{D}, p2::Power{D}) where {D} = *(p2,p1)
Base.:*(p::Poly{D}, s::Number) where {D} = Poly{D}(s*p.coeffs)
Base.:*(s::Number, p::Poly) = *(p,s)
Base.:/(p::Poly{D}, s::Number) where {D} = Poly{D}(p.coeffs/s)
Base.:\(s::Number, p::Poly) = /(p,s)
function Base.:*(p1::Poly{D}, p2::Poly{D}) where {D}
    N = degree(p1)+degree(p2)
    s = p1[0]*p2[0]
    coeffs = zeros(typeof(s), N+1)
    for i = 0:degree(p1)
        for j = 0:degree(p2)
            coeffs[i+j+1] += p1[i]*p2[j]
        end
    end
    return Poly{D}(coeffs)
end

Base.promote_rule(::Type{Power{D,T1}}, ::Type{Power{D,T2}}) where
{D,T1<:Number,T2<:Number} = Power{D,promote_type(T1,T2)}
Base.promote_rule(::Type{Power{D,T1}}, ::Type{T2}) where {D,T1<:Number,T2<:Number}
= Power{D,promote_type(T1,T2)}
Base.promote_rule(::Type{Poly{D,T1}}, ::Type{Poly{D,T2}}) where
{D,T1<:Number,T2<:Number} = Poly{D,promote_type(T1,T2)}
Base.promote_rule(::Type{Poly{D,T1}}, ::Type{Power{D,T2}}) where
{D,T1<:Number,T2<:Number} = Poly{D,promote_type(T1,T2)}
Base.promote_rule(::Type{Poly{D,T1}}, ::Type{T2}) where {D,T1<:Number,T2<:Number} =
Poly{D,promote_type(T1,T2)}

function Base.:(==)(p1::AbstractPoly{D}, p2::AbstractPoly{D}) where {D}
    for i = max(degree(p1),degree(p2)):-1:0
        p1[i] == p2[i] || return false
    end
    return true
end
Base.:(==)(p1::AbstractPoly, p2::Number) = degree(p1) == 0 && p1[0] == p2
Base.:(==)(p1::Number, p2::AbstractPoly) = degree(p2) == 0 && p2[0] == p1
function Base.:<(p1::AbstractPoly{D}, p2::AbstractPoly{D}) where {D}
    for i = max(degree(p1),degree(p2)):-1:0
        p1[i] < p2[i] && return true
        p1[i] > p2[i] && return false
    end
    return false
end
Base.isless(p1::AbstractPoly{D}, p2::AbstractPoly{D}) where {D} = p1 < p2
```