

Tabla comparativa

Lenguaje	Capacidad inicial	Factor de crecimiento	Características únicas
Java – ArrayList	10 (aunque en Java 8 se inicializa con array vacío y se expande a 10 al primer uso)	~1.5x (crece en 50% cuando se llena)	Permite null sin problema; optimización: usa un array compartido vacío hasta que se agrega el primer elemento.
Python – list	No fija capacidad inicial; empieza vacío	Estrategia compleja: sobreesigna más espacio (aprox. 1.125x–1.5x según tamaño)	Soporta tipos mixtos; “over-allocation” reduce la frecuencia de realocaciones y mejora rendimiento.
C++ – std::vector	No definida (puede ser 0)	No garantiza factor exacto; solo asegura crecimiento amortizado constante	Move semantics: al crecer puede mover objetos en vez de copiarlos; reserve() preasigna capacidad, shrink_to_fit() intenta liberar memoria extra
JavaScript – Array (V8)	Flexible, sin capacidad fija	No aplica: arrays son objetos dinámicos	“Packed” arrays (índices contiguos, mismos tipos) son rápidos; “holey” arrays (con huecos o tipos mezclados) degradan rendimiento

Decisiones de diseño notables y trade-offs

- **Java ArrayList**
 - Decisión: capacidad inicial de 10 y crecimiento 1.5x.
 - Trade-off: balance entre memoria desperdiciada y costo de realocación.
 - Sorpresa: en Java 8 el constructor usa un array vacío y solo asigna capacidad al primer add().
- **Python list**
 - Decisión: sobreesignación adaptativa.
 - Trade-off: más memoria usada, pero menos realocaciones.
 - Sorpresa: el factor de crecimiento no es fijo, sino que depende del tamaño actual (ej. crece más agresivamente al inicio, menos después).
- **C++ std::vector**

- Decisión: no fija factor de crecimiento, solo garantiza complejidad amortizada $O(1)$.
- Trade-off: deja libertad a cada implementación (libstdc++, MSVC, etc.) para optimizar.
- Sorpresa: con move semantics, crecer puede ser muy barato si los objetos son movibles.
- **JavaScript Array (V8)**
 - Decisión: arrays son objetos dinámicos, optimizados internamente según “element kinds”.
 - Trade-off: máxima flexibilidad para el programador, pero riesgo de perder optimización si se mezclan tipos o índices.
 - Sorpresa: el motor clasifica arrays como “packed” o “holey”, y esa diferencia puede multiplicar el rendimiento por 10x.

La decisión más interesante

La más relevante es la de Python list con sobreasignación adaptativa.

- No usa un factor fijo como 2x, sino que ajusta dinámicamente cuánto crecer según el tamaño actual.
- Esto es sorprendente porque muestra un diseño pragmático: optimizar para cargas reales y evitar picos de memoria, en lugar de seguir una regla matemática rígida.

Cómo cambiaría mi implementación

Si tuviera que rehacer mi propio arreglo dinámico:

- Adoptaría la estrategia de sobreasignación adaptativa de Python, porque equilibra bien entre memoria y rendimiento.
- Incorporaría move semantics de C++ para minimizar copias costosas.
- Añadiría un sistema de clasificación interna como V8 para optimizar casos comunes (arrays homogéneos y contiguos).
- Mantendría la simplicidad de Java (factor fijo) como fallback, pero con mejoras para inicialización perezosa.