Sequences

• Simple operations are $O(1)$
• Question: What’s simple? What’s not?
  • Example of complex from last time: `sorted()`
  • Language/compiler docs are a required reference, e.g. Python: [https://wiki.python.org/moin/TimeComplexity](https://wiki.python.org/moin/TimeComplexity)
  • Same applies for `std::vector`, `ArrayList`, etc. Know where the documentation/specifications are.
Low Level Arrays

- **byte** - smallest addressable quantity
  - 8 bits unless ancient or unusual architecture
- **word** - # of bytes CPU likes to work with
  - 32-bit CPU – 4 byte word
  - 64-bit CPU – 8 byte word
- conceptually memory is one huge array
- byte-by-byte memory addressing
- O(1) lookup cost -> Random Access Memory (RAM)
Low Level Arrays 2

- Array: contiguous sequence of memory
- Language/compiler handles size of element access vs direct address accessing
  - E.g. in C `short a[10]; long double b [10];` have different # of bytes for each element, but programmers just have to worry about `a[5]` or `b[5].`
- Python uses for `str` (since immutable)
  - space depends on character encoding (bit representation of characters)
  - Python 3 uses utf-8
Referential Arrays

• Ah, but everything in Python is an object, different sizes
  • no way to align arbitrary arrays...

• Pointers/references are variables of type address
  • same size for all types...

• Solution! Use array of references
  • Size of object at other end doesn’t affect alignment

• Side effect: Slicing is trivial
  • just make new reference array
Compact Arrays

- `str` by default
- `array` module available for numerical work
- `numpy` module for serious numerical work
  - Uses C under the hood, very fast
- `(btw, std::valarray)`
Dynamic Arrays

• Python lists expand/shrink automatically
  • expanding/shrinking have a cost
  • experiment_list_size.py

• Process
  • array size change triggered
  • allocate new memory
  • copy references – $O(n)$
  • (mark old memory for garbage collection)
  • (garbage collection)

• Copying references is the expensive part
Implementing an Array

• `dynamic_array.py`
Amortization

• Big $\Omega$ ("worst best case") because we want to know what hidden cost to account for

• Resize expense is copying

• If and only if (iff) cost can be spread out over time it is amortized (effectively ignorable (usually...))

• Trick is to make sure resizes don’t happen too often for price to be spread out

• Solution: Grow arrays exponentially under assumption resize requests are less than exponential
Amortization II

• Will be using this type of analysis frequently

• Web notes version:
  • A cost is incurred
  • Lingering effects diminish as program runs

• Textbook version:
  • Program has some reserve of expected runtime
  • Costs taken out of this reserve
Python Sequence Efficiency

• Non-mutating operation costs apply to all sequence types, mutating operation costs apply only to lists

• (See charts in web notes/textbook)
  • When there’s an i, j, k it means “still linear but often enough less than n to make a difference”
  • When there’s an m it means “also linear, and no guarantee this size is related to n”
  • Amortized Big O – operation may trigger a resize

• Fundamental search operation, a linear search, O(n)
  • have to look at elements one-by-one until found