Measuring Performance

- **Use analysis, e.g., big-Oh, to reason about performance**
  - Good in practice, what about in theory?
  - The proof of a pudding is in the eating...

- **Use coarse-granularity timing**
  - `/usr/bin/time` or shell time functions
  - Tapestry CTimer functions

- **Use a profiling tool like gprof**
  - See resources page

- **What about our friend Heisenberg?**
Does performance matter?

- **Obviously it depends**
  - Run it once, throw it away: probably don’t care
  - Route TCP/IP packets: millions/second: do care

- **Performance is one aspect of a system**
  - Correctness
  - Time to develop/deploy
  - If it’s not a bottleneck, don’t optimize

- **Premature optimization is the root of all evil (or similar): Knuth**
  - Remember, the compiler is your friend: try –O2
Time and Space trade-offs

- Two principal performance resources: time and space
  - Time is CPU time, wall-clock time isn’t always the same
  - Space is memory resources used by program

- Often we can trade-off time for space and *vice versa*
  - Consider the O(1) solution to itoa
    - $2^{32}$ values x 8 bytes/value = 32 Gigabytes (conservative)
  - In KWIC, we can use more memory for faster/easier code

- There are sometimes subtle effects due to cache
  - What’s on disk, what’s in memory?
  - Memory is “different”, cache, main memory, etc.
Modules, design, coding, refactor, XP

- **Make it run, make it right, make it fast, make it small**
- **Do the simplest thing that can possibly work (XP)**
  - Design so that refactoring is possible
  - Don’t lose sight of where you’re going, keep change in mind, but not as the driving force [it will evolve]

- **Refactor: functionality doesn’t change, code does**
  - Should mean that new tests aren’t written, just re-run
  - Depends on modularity of code, testing in pieces

- **What’s a module in C++**
  - Could be a class, a file, a directory, a library, a namespace
  - We should, at least, use classes, files, directories
STL overview

- **STL implements generic programming in C++**
  - Container classes, e.g., vector, stack, deque, set, map
  - Algorithms, e.g., search, sort, find, unique, match, ...
  - Iterators: pointers to beginning and one past the end
  - Function objects: less, greater, comparators

- **Algorithms and containers decoupled, connected by iterators**
  - Why is decoupling good?
  - Extensible: create new algorithms, new containers, new iterators, etc.
  - Syntax of iterators reflects array-pointer origins, an array can be used as an iterator
STL examples: wordlines.cpp, kwic.cpp

- How does an iterator work?
  - Start at beginning, iterate until end: use [first..last) interval
  - Pointer syntax to access element and make progress

```cpp
vector<int> v; // push elements
vector<int>::iterator first = v.begin();
vector<int>::iterator last = v.end();
while (first < last) {
    cout << *first << endl;
    ++first;
}
```

- Will the while loop work with an array/pointer?

- In practice, iterators aren’t always explicitly defined, but passed as arguments to other STL functions
Review: what’s a map, STL/Tapestry

- **Maps keys to values**
  - Insert key/value pair
  - Extract value given a key
  - STL uses red-black tree, Tapestry uses bst or hashtable
    - STL unofficially has a hash_map, see SGI website
  - Performance and other trade-offs?

- **In Tapestry, there’s an inheritance hierarchy of tmap, BSTMap, HMap**
  - The hash-table requires map of string->value
    - Makes programming simpler, too restrictive in practice
  - See tapwordlines.cpp
Toward a KWIC program

- Do we need all the words in memory?
  - All words in the file?
  - All unique words?

- Do we need the words in the same order they appear in file?
  - Alternatives and tradeoffs
  - How can we access these words

- What about the I/O bottleneck
  - Reading words in C++ is expensive
  - Reading characters in C++ is cheap
  - How can we leverage the latter to help the former