From practice to theory and back again

In theory there is no difference between theory and practice, but not in practice

- We’ve studied binary search, that requires a sorted vector
  - Much faster than sequential search (how much)
  - Add elements in sorted order or sort vector after adding

- Many sorting algorithms have been well-studied
  - Slower ones are often “good enough” simple to implement
  - Some fast algorithms are better than others
    - Always fast, fast most-of-the-time
    - Good in practice even if flawed theoretically?

- New algorithms still discovered
  - Quick sort in 1960, revised and updated in 1997

Tools for algorithms and programs

- We can time different methods, but how to compare timings?
  - Different on different machines, what about “workload”?
  - Mathematical tools can help analyze/discuss algorithms

- We often want to sort by different criteria
  - Sort list of stocks by price, shares traded, volume traded
  - Sort directories/files by size, alphabetically, or by date
  - Object-oriented concepts can help in implementing sorts

- We often want to sort different kinds of vectors: string and int
  - Don’t want to duplicate the code, that leads to errors
  - Generic programming helps, in C++ we use templates

Removing some elements from vector

```cpp
void RemoveBozos(tvector<string>& a)
// pre: a contains a.size() entries
// post: all bozos removed from a, order of other elements unchanged, a contains a.size() elements
{
    int k;
    int nonBozoCount = 0;
    // invariant: a[0..nonBozoCount-1] are NOT bozos
    for(k=0; k < a.size(); k++)
    {
        if (! IsBozo(a[k]))
        {
            a[nonBozoCount] = a[k];
            nonBozoCount++;
        }
    }
    a.resize(nonBozoCount);
}
```

- How many elements of a are examined? Moved?
  - 1000 element vector takes 20 secs., how long for 2000 elements?

Another version of removing elements

```cpp
void RemoveBozos(tvector<string>& a)
{
    int j,k;
    for(k=0; k < a.size(); k++)
    {
        if (IsBozo(a[k]))
        {
            for(j=k; j < a.size()-1; j++)
            {
                a[j] = a[j+1];
            }
            a.pop_back();
            k--; // k++ coming, but a[k] not checked
        }
    }
}
```

- Note k--, use a while loop instead (for common in student solutions)
- How many elements of a compared/shifted? Worst case? Best case?
On to sorting: Selection Sort

- Find smallest element, move into first array location
- Find next smallest element, move into second location
  - Generalize and repeat
- How many elements examined to find smallest?
  - How many elements examined to find next smallest?
  - Total number of elements examined? $N + (N-1) + \ldots + 1$
- How many elements swapped?
- Simple to code, reasonable in practice for small vectors
  - What’s small? What’s reasonable? What’s simple?

Selection Sort: The Code (selectsort2.cpp)

```cpp
void SelectSort(tvector<int> & a)
// pre: a contains a.size() elements
// post: elements of a are sorted in non-decreasing order
{
    int j,k,temp,minIndex,numElts = a.size();
    // invariant: a[0]..a[k-1] in final position
    for(k=0; k < numElts - 1; k++)
    {
        minIndex = k;          // minimal element index
        for(j=k+1; j < numElts; j++)
        {   if (a[j] < a[minIndex])
            { minIndex = j;      // new min, store index
              }
        }
        temp = a[k];          // swap min and k-th elements
        a[k] = a[minIndex];
        a[minIndex] = temp;
    }
}
```

Creating a function template

```cpp
template <class Type>
void SelectSort(tvector<Type> & a)
// pre: a contains a.size() elements
// post: elements of a are sorted in non-decreasing order
{
    int j,k,minIndex,numElts = a.size();
    Type temp;
    // invariant: a[0]..a[k-1] in final position
    for(k=0; k < numElts - 1; k++)
    {
        minIndex = k;          // minimal element index
        for(j=k+1; j < numElts; j++)
        {   if (a[j] < a[minIndex])
            { minIndex = j;      // new min, store index
              }
        }
        temp = a[k];          // swap min and k-th elements
        a[k] = a[minIndex];
        a[minIndex] = temp;
    }
}
```

What changes if we sort strings?

- The parameter changes, the definition of `temp` changes
  - Nothing else changes, code independent of type
  - We can use features of language to capture independence
- We can have different versions of function for different array types, with same name but different parameter lists
  - Overloaded function: parameters different so compiler can determine which function to call
  - Still problems, duplicated code, new algorithm means ...
- With function templates we replace duplicated code maintained by programmer with compiler generated code

- When the user calls this code, different versions are compiled
Some template details

- Function templates permit us to write once, use several times for several different types of vector
  - Template function “stamps out” real function
  - Maintenance is saved, code still large (why?)

- What properties must hold for vector elements?
  - Comparable using < operator
  - Elements can be assigned to each other

- Template functions capture property requirements in code
  - Part of generic programming
  - Some languages support this better than others

From practical to theoretical

- We want a notation for discussing differences between algorithms, avoid empirical details at first
  - Empirical studies needed in addition to theoretical studies
  - As we’ll see, theory hides some details, but still works

- Binary search: roughly 10 entries in a 1,000 element vector
  - What is exact relationship? How to capture “roughly”?
  - Compared to sequential/linear search?

- We use O-notation, big-Oh, to capture properties but avoid details
  - \( N^2 \) is the same as \( 13N^2 \) is the same as \( 13N^2 + 23N \)
  - \( O(N^2) \), in the limit everything is the same

Running times @ \( 10^6 \) instructions/sec

<table>
<thead>
<tr>
<th>( N )</th>
<th>( O(\log N) )</th>
<th>( O(N) )</th>
<th>( O(N \log N) )</th>
<th>( O(N^2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003</td>
<td>0.0001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.0010</td>
<td>0.000664</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000010</td>
<td>0.0100</td>
<td>0.010000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.0100</td>
<td>0.132900</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>0.1000</td>
<td>1.661000</td>
<td>2.78 hr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000020</td>
<td>1.0</td>
<td>19.9</td>
<td>11.6 day</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000030</td>
<td>16.7 min</td>
<td>18.3 hr</td>
<td>318 centuries</td>
</tr>
</tbody>
</table>

What does table show? Hide?

- Can we sort a million element vector with selection sort?
  - How can we do this, what’s missing in the table?
  - What are hidden constants, low-order terms?

- Can we sort a billion-element vector? Are there other sorts?
  - We’ll see quicksort, an efficient (most of the time) method
  - \( O(N \log N) \), what does this mean?

- Sorting code for different algorithms in sortall.h/sortall.cpp
  - Template functions, prototypes in .h file, implementations in .cpp file, must have both (template isn't code!!)
Templates and function objects

- In a templated sort function vector elements must have certain properties (as noted previously)
  - Comparable using operator <
  - Assignable using operator =
  - Ok for int, string, what about Date? ClockTime?
- What if we want to sort by a different criteria
  - Sort strings by length instead of lexicographically
  - Sort students by age, grade, name, ...
  - Sort stocks by price, shares traded, profit, ...
- We can’t change how operator < works
  - Alternative: write sort function that does NOT use <
  - Alternative: encapsulate comparison in parameter, pass it

Function object concept

- To encapsulate comparison (like operator <) in a parameter
  - Need convention for parameter: name and behavior
  - Other issues needed in the sort function, concentrate on being clients of the sort function rather than implementors
- Name convention: class/object has a method named compare
  - Two parameters, the vector elements being compared (might not be just vector elements, any two parameters)
- Behavior convention: compare returns an int
  - zero if elements equal
  - +1 (positive) if first > second
  - -1 (negative) if first < second

Function object example

class StrLenComp
{
  public:
    int compare(const string& a, const string& b) const // post: return -1/+1/0 as a.length() < b.length()
    {
      if (a.length() < b.length()) return -1;
      if (a.length() > b.length()) return 1;
      return 0;
    }
};
// to use this:
StrLenComp scomp;
if (scomp.compare("hello", "goodbye") < 0) …
- We can use this to sort, see strlensort.cpp
  - Call of sort: InsertSort(vec, vec.size(), scomp);

Another function object example

- Consider “directory.h” and the class DirEntry
  - DirEntry encapsulates file/directory
  - Methods: Name(), Size(), Path(), GetTime(), …
- To sort using Name() use class below, what about Size() ?
class DirNameComp
{
  public:
    int compare(const DirEntry& a, const DirEntry& b) const // post: return -1/+1/0 as a.length() < b.length()
    {
      if (a.Name() < b.Name()) return -1;
      if (a.Name() > b.Name()) return 1;
      return 0;
    }
}
From smarter code to algorithm

- We’ve seen selection sort, other \( O(N^2) \) sorts include
  - Insertion sort: better on nearly sorted data, fewer comparisons, potentially more data movements (selection)
  - Bubble sort: dog, dog, dog, don’t use it

- Efficient sorts are trickier to code, but not too complicated
  - Often recursive as we’ll see, use divide and conquer
  - Quicksort and Mergesort are two standard examples

- Mergesort divide and conquer
  - Divide vector in two, sort both halves, merge together
  - Merging is easier because subvectors sorted, why?

Quicksort, an efficient sorting algorithm

- Step one, partition the vector, moving smaller elements left, larger elements right
  - Formally: choose a pivot element, all elements less than pivot moved to the left (of pivot), greater moved right
  - After partition/pivot, sort left half and sort right half

Quicksort details

```c
void Quick(tvector<string> & a, int first, int last)
// pre: first <= last
// piv: a[first] <= ... <= a[last]
{
    int piv;
    if (first < last)
    {
        piv = Pivot(a, first, last);
        Quick(a, first, piv-1);
        Quick(a, piv+1, last);
    }
    // original call is Quick(a, 0, a.size()-1);

    // How do we make progress towards basecase? What’s a good pivot versus a bad pivot? What changes?
    // What about the code for Pivot?
    // What about type of element in vector?
}
```

How is Pivot similar to RemoveBozos?

```c
int Pivot(tvector<string> & a, int first, int last)
// pre: first <= last
// post: returns piv so: k in [first..piv],  a[k] <= a[piv]
// k in (piv,last]  piv, a[piv] < a[k]
//
int k,p=first;
string piv = a[first];
for(k=first+1,k<=last,k++)
{
    if (a[k] <= piv) <= 0)
        p++;
        Swap(a[k],a[p]);
}
Swap(a[p],a[first]);
return p;
```

- Partition around a[first], can change this later, why is p initially first?
- What is invariant?
What is complexity?

- We've used O-notation, (big-Oh) to describe algorithms
  - Binary search is $O(\log n)$
  - Sequential search is $O(n)$
  - Selection sort is $O(n^2)$
  - Quicksort is $O(n \log n)$

- What do these measures tell us about “real” performance?
  - When is selection sort better than quicksort?
  - What are the advantages of sequential search?

- Describing the complexity of algorithms rather than implementations is important and essential
  - Empirical validation of theory is important too

Do it fast, do it slow, can we do it at all?

- Some problems can be solved quickly using a computer
  - Searching a sorted list
- Some problems can be solved, but it takes a long time
  - Towers of Hanoi
- Some problems can be solved, we don’t know how quickly
  - Traveling salesperson, optimal class scheduling
- Some problems can’t be solved at all using a computer
  - The halting problem, first shown by Alan Turing

- The halting problem: can we write one program used to determine if an arbitrary program (any program) stops?
  - One program that reads other programs, must work for every program being checked, computability

The halting problem: writing \texttt{DoesHalt}

```cpp
bool DoesHalt(const string& progname, const string& s)
// post: returns true if progname halts given s as input, false otherwise
int main()
{
  string f = PromptString("enter filename ");
  string s = PromptString("input for "+filename);
  if (DoesHalt(f,s)) cout << "does halt" << endl;
  else               cout << "does not halt" << endl;
}
```

- A compiler is a program that reads other programs as input
  - Can a word counting program count its own words?
- The \texttt{DoesHalt} function might simulate, analyze, ...
  - One program/function that works for any program/input

Consider the program \texttt{confuse.cpp}

```cpp
#include "halt.h"
int main()
{
  string f = PromptString("enter filename ");
  if (DoesHalt(f,f))
  {
    while (true)
    {  // do nothing forever
      //
    }
  }
  return 0;
}
```

- We want to show writing \texttt{DoesHalt} is impossible
  - Proof by contradiction:
  - Assume possible, show impossible situation results
What is computer science?

- **What is a computation?**
  - Can formulate this precisely using mathematics
  - Can say “anything a computer can compute”
  - Study both theoretical and empirical formulations, build machines as well as theoretical models
- **How do we build machines and the software that runs them?**
  - Hardware: gates, circuits, chips, cache, memory, disk, …
  - Software: operating systems, applications, programs
- **Art, Science, Engineering**
  - How do we get better at programming and dealing with abstractions
  - What is hard about programming?

---

Shafi Goldwasser

- RCS professor of computer science at MIT
  - Co-inventor of zero-knowledge proof protocols
  - How do you convince someone that you know something without revealing “something”
- Consider card readers for dorms
  - Access without tracking

*Work on what you like, what feels right, I now of no other way to end up doing creative work*