Admin

- Boggle due
  - Joy poll
  - Little assign break -- use weekend to prep for midterm!
- Today's topics
  - Quicksort, generic sorting template, client callbacks
- Reading
  - Ch 7, Ch. 11.4
- Midterm next Tuesday evening
  - Terman Aud 7-9pm
- Cafe today in Terman after class

Quadratic vs linearithmic

- Compare SelectionSort to MergeSort

10,000	3 sec	.05 sec
20,000	13 sec	.15 sec
50,000	78 sec	.38 sec
100,000	5 min	.81 sec
200,000	20 min	1.7 sec
1,000,000	8 hrs (est)	9 sec

- O(NlogN) is pretty good, can we do better?
  - Theoretical result (beyond scope of 106B) no general sort algorithm better than NlogN
  - But a better NlogN in practice?

Quicksort idea

- "Divide and conquer" algorithm
  - Divide input into low half and high half
  - Recursively sort each half
  - Join two halves together
- "Hard-split easy-join"
  - Each element examined and placed in correct half
  - Join step is trivial

Partitioning for quicksort

- Partition step uses "pivot" value
  - All elems less than pivot in one half, all elems greater in other
- How to choose pivot to get even split?
  - How to know range for values in the input at all?
- Simple choice: use first elem as pivot
  - Known to be in range at least
  - We'll examine this choice later

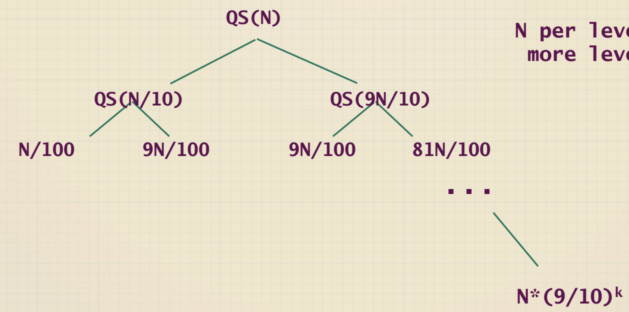
Quicksort code

```
void Quicksort(Vector<int> &v, int start, int stop)
{
   if (stop > start) {
     int pivot = Partition(v, start, stop);     } O(N)
     Quicksort(v, start, pivot-1);
     Quicksort(v, pivot+1, stop);
   }

}
```

- Assume ideal 50/50 split
  - T(N) = N + 2T(N/2)   => O(NlogN)

Assuming bad 90/10 split

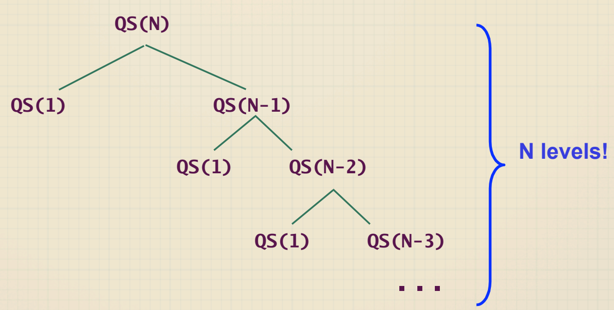


N per level, but
more levels...

Solve $N*(9/10)^k = 1$
$N = (10/9)^k$
$\log_{10/9}N = K$
$C\log_2 N = K$

Still O(NlogN)

Assuming worst N-1/1 split



Ack! $O(N^2)$

What input has worst split?

- If pivot is smallest in input
  - Input already in sorted order!
- If pivot is largest in input
  - Input in reverse sorted order
- Others not so likely
  - Smallest, then largest, etc
- "Degenerate" case
  - May tolerate poor worst-case outcome if input is unlikely
  - Does that apply here?

What to do?

- Choose pivot differently
- Compute actual median
  - O(N) algorithm exists for this
  - Guarantee 50/50 split
- "Median of three"
  - Approximate median
  - Choose middle from first, last, mid
- Random
  - Choose random element
  - Worst-case still possible, but unlikely

In clock time

- Compare MergeSort to Quicksort
  - 10,000 .05 sec .008 sec
  - 20,000 .15 sec .01 sec
  - 50,000 .38 sec .11 sec
  - 100,000 .81 sec .21 sec
  - 200,000 1.7 sec .45 sec
  - 1,000,000 9 sec 2.6 sec
- Both O(NlogN) but Quicksort's advantage
  - No secondary storage
  - Moves elements more quickly to correct position

A proliferation of Swap

```
void SwapChars(char & ch1, char & ch2)
{
   char tmp = ch1;
   ch1 = ch2;
   ch2 = tmp;
}
void SwapInts(int & num1, int & num2)
{
   int tmp = num1;
   num1 = num2;
   num2 = tmp;
}
void SwapStrings(string & str1, string & str2)
{
   string tmp = str1;
   str1 = str2;
   str2 = tmp;
}
                          // and so on ...
```

Function template

- Same general idea as class template
  - Generic function uses same algorithm for any type
  - Write/test/debug once, use in many situations
- Template written using one or more placeholders
  - e.g. swap exchanges two values of any type
- Using function instantiates specific version
  - Call to swap passing two doubles uses a different version than a call passing two strings
- Compiler infers placeholder type if possible
  - So may not need explicit Swap<double>
  - (Unlike classes where <> always required)

Swap function template

```
template <typename Type>
  void Swap(Type & one, Type & two)
  {
     Type tmp = one;
     one = two;
     two = tmp;
  }
```

- Template from which to create many Swap functions
  - Can create Swap for ints, chars, strings, etc. from same pattern

Using function template

```
int main()
{
   int a = 12, b = 45;
   string str1 = "apple", str2 = "orange"};

   Swap(a, b);         // infers Swap<int>
   Swap(str1, str2);  // infers Swap<string>
```

- Compiler infers placeholder type if possible
  - Can explicitly call Swap<int>(a, b) but usually isn't necessary

Template instantiation

```
template <typename Type>
  void Swap(Type & one, Type & two)
  {
     Type tmp = one;
     one = two;
     two = tmp;
  }
```

- What happens on call to Swap?

```
int a = 4, b = 19;
Swap(a, b);
```

```
void Swap<int>(int & one, int & two)
  {
     int tmp = one;
     one = two;
     two = tmp;
  }
```

  - Template instantiated with Type => int
  - Compiler internally names this version Swap<int>
  - Code is then compiled

Instantation errors

```
template <typename T>
  void PrintVector(Vector<T> &v)
  {
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
  }
```

← This line is trying to use << on a coordT

- Try to instantiate PrintVector for non-primitive type

```
struct coordT {
   double x, y;
};

 Vector<coordT> c;
 PrintVector(c);
```

Instantation errors

- Compiler's response:

```
main.cpp:16: error: no match for 'operator<<' in 'std::cout
  Vector<ElemType>::operator[] [with ElemType = coordT](i)'
```

- Template error reporting
  - Template itself is largely ignored by compiler
  - When called, version is created with placeholder filled in, and only then is compiled
  - Errors within template now reported, triggered by client's instantiation
- Template may have hidden requirements on type
  - e.g. Uses << to output or compares using ==
  - Code instantiated won't compile if type doesn't support needed ops
  - Most common operators to watch for: output, assignment, comparison/relational

Sort template

```
template <typename Type>
  void Sort(Vector<Type> &v)
  {
    for (int i = 0; i < v.size() - 1; i++) {
      int minIndex = i;
      for (int j = i+1; j < v.size(); j++) {
         if (v[j] < v[minIndex])
             minIndex = j;
      }
      Swap(v[i], v[minIndex]);
    }
  }
```

- Template functions awesome for algorithms
  - Searching (linear/binary), sorting (all varieties), median, mode, permute, summarize, remove duplicates, etc.

Client use of Sort template

```
int main()
{
    Vector<int> nums = ...;
    Sort(nums);

    Vector<string> strs = ...;
    Sort(strs);
```

- What must be true about the element type?
  - Will <u>every</u> type work?
  - Consider:

```
struct coordT {
  double x, y;
};

Vector<coordT> pts;
Sort(pts);
```

Fully generic sort

- Sort function template uses < to compare elements
  - This works for some types, but not all
- Division between client/implementor
  - Client knows how data is to be compared
  - Implementor is the one doing the actual comparing
- Need client/implementor cooperation
  - Client tells implementor how to appropriately compare elements
- Add function parameter
  - Client knows how to compare elements, it supplies this knowledge in the form of a function pointer
  - Callback function —implementation "calls back" to client

Sort template with callback fn

```
template <typename Type>
  void Sort(Vector<Type> &v, int (cmp)(Type, Type))
  {
    for (int i = 0; i < v.size() - 1; i++) {
      int minIndex = i;
      for (int j = i+1; j < v.size(); j++) {
         if (cmp(v[j], v[minIndex]) < 0)
            minIndex = j;
      }
      Swap(v[i], v[minIndex]);
    }
  }
```

- Now can truly work for all types!
  - Client supplies function pointer to handle compare