STL::sort and Shellsort

qsort

- qsort is a C-style QuickSort implementation that is ignostic about data and uses a user supplied function

- Prototype is:
  - void qsort (void* base, size_t num, size_t width, int compareFunc(const void *, const void *)

- You have two options for Project #2:
  - Add in two C-style comparison functions, one for strings and one of numbers that look like above
  - Do pointer magic (see Piazza) to convert the C++ style ones into C style
  - Purpose: to compare STL and more C-like sorting empirically

```c
/* qsort int comparison function */
int int_cmp(const void *a, const void *b)
{
    const int *ia = (const int *)a; // casting pointer types
    const int *ib = (const int *)b;
    return *ia  - *ib;
        /* integer comparison: returns negative if b > a
        and positive if a > b */
}
```

```c
/* qsort C-string comparison function */
int cstring_cmp(const void *a, const void *b)
{
    const char **ia = (const char **)a;
    const char **ib = (const char **)b;
    return strcmp(*ia, *ib);
        /* strcmp functions works exactly as expected from
        comparison function */
}
```

http://www.anyexample.com/programming/c/qsort__sorting_array_of_strings__integers_and_structs.xml

Sorting example from same ref

```c
/* sorting integers using qsort() example */
void sort_integers_example()
{
    int numbers[] = { 7, 3, 4, 1, -1, 23, 12, 43, 2, -4, 5 };
    size_t numbers_len = sizeof(numbers)/sizeof(int);

    puts("*** Integer sorting...");

    /* print original integer array */
    print_int_array(numbers, numbers_len);

    /* sort array using qsort functions */
    qsort(numbers, numbers_len, sizeof(int), int_cmp);

    /* print sorted integer array */
    print_int_array(numbers, numbers_len);
}
```

C++ standard as a guide

- The standard for C++ states that sorting should be O(n log n)

- Caveats:
  - Stability matters so merge sort and/or a linked list implementation (Project 2) of quick sort may be in play

  - Most implementations use something called “intro-sort,” which is a hybrid between Quick sort and Heap Sort (spoiler for next week)

  - Std::stable_sort exists to guarantee stability, list::sort exists to sort lists in the STL (sorry, can’t use it for Project 2)

One more thing on Quick sort

- A question asked at the end of last class was, “Dr. Scott, what if we are unlucky and the first element is the minimum relative to the list?”

- There are two solutions:
  - Dr. Plank takes the median of the first, middle, and last number. Does better and is closest to STL::sort since the median is “in place”

  - It is also possible to shuffle the numbers in O(n) time to make the worst case for QuickSort, which is O(n^2), unlikely. Fisher-Yate’s that is also known as Knuth’s shuffle is the simplest thing to implement

List::sort

- Actually uses merge sort since merging is pretty easy in a linked list; you just maintain two pointers (vs. indices), a ”front” pointer, and move nodes around to put each subproblem in order

- Also relatively low overhead in most implementations

Merge pseudo code

link merge (link a, link b) {

- Node head;
- Link c = &head;   // link is a pointer, get address of node
- While (a != null) && (b != NULL).   // merge until one list is empty
  - if less (a->item, b-> item)
    - {c->next = a; c = a; a = a->next;}
  - else
    - {c->next = a; c = a; a = a->next;}
- c->next = (a==NULL) ? b : a; // tack on the remaining list onto new list c
- Return head.next;

}

https://www.geeksforgeeks.org/bubble-sort-on-doubly-linked-list/

Bubble sort alternative

```
/* Bubble sort the given linked list */
void bubbleSort(struct Node *start)
{
    int swapped, i;
    struct Node *ptr1;
    struct Node *lptr = NULL;

    /* Checking for empty list */
    if (start == NULL)
        return;

    do
    {
        swapped = 0;
        ptr1 = start;

        while (ptr1->next != lptr)
        {
            if (ptr1->data > ptr1->next->data)
            {
                swap(ptr1->data, ptr1->next->data);
                swapped = 1;
            }
            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    }
    while (swapped);
}
```

Stability

- A sorting algorithm is considered **stable** if it maintains the relative order of equivalent elements

- For example, consider this array:
  - 4 6^1 6^2 3 7

- A stable sort would guarantee 6^1 comes before 6^2

Why does stability matter?

- There are a few times when you want to preserve order. The most obvious is sorting on multiple factors (Last Name, First Name)

- In general, we have two options for multi-factor sorting
  - Use a stable sorting method (see reading assignment)
  - Use a custom comparison function, similar to the current Project #2

Example

- Josh N.
- Josh A.
- Josh H.
- Josh B.
- Andrew A.
- Andrew B.
- Alex T.
- Alex S.
- Alex L.

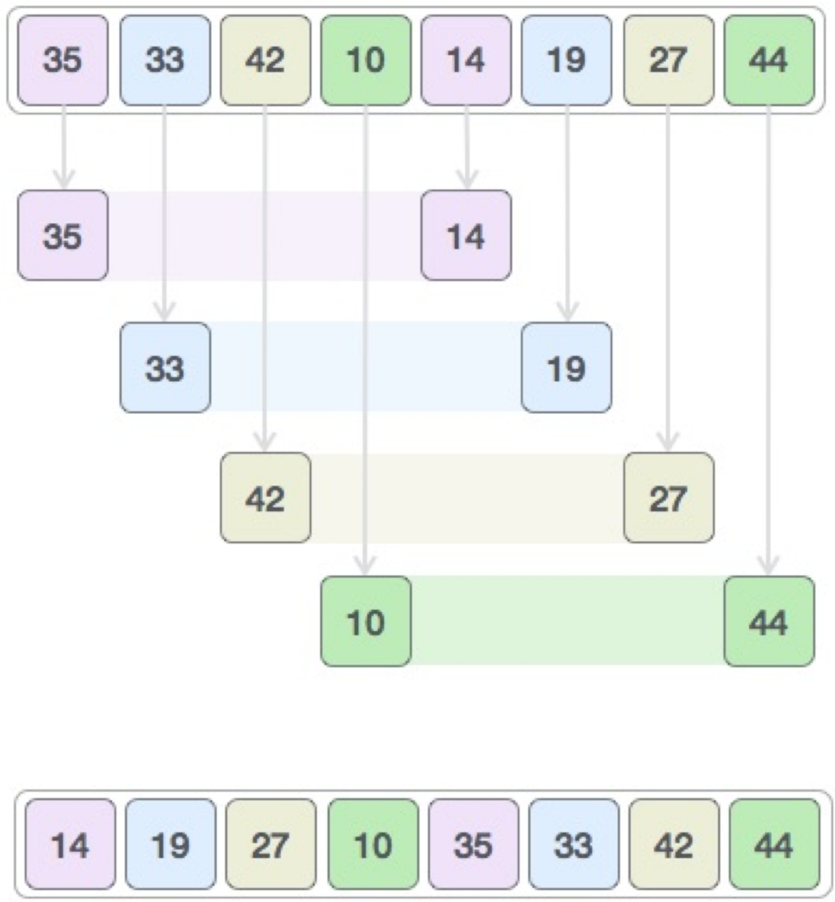
Shell sort

- Algorithm published in 1959 by Shell (thus Shell's sort)

- Key insight is to divide the list into subsets s.t. more distant elements are compared and swapped if necessary

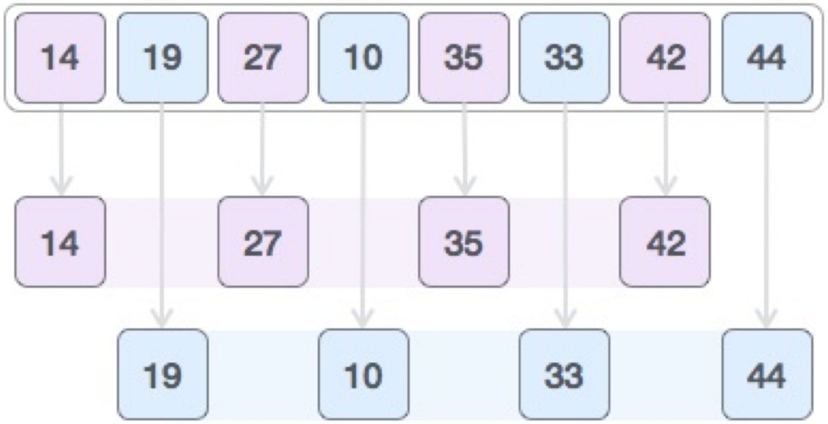
- The distances are reduced per iteration, then insertion sort is run to make sure everything is sorted

Example: Stage 1



https://www.tutorialspoint.com/data_structures_algorithms/shell_sort_algorithm.htm

Example: Stage 2



14. 10. 27. 19. 35. 33. 42. 44

https://www.tutorialspoint.com/data_structures_algorithms/shell_sort_algorithm.htm

Psuedo-code

- Gap = choseInitialGap();   // usually n / 2 per Shell

- While (gap > 0) {
  - For (i = 0; i < gap; i++)

    Insertion sort subsequences starting at I, skipping every gap elements
  - Gap = chooseNextGap (gap);

}

Overview (using dance! And no dance..)

- https://www.youtube.com/watch?v=CmPA7zE8mx0

- https://www.toptal.com/developers/sorting-algorithms/shell-sort

Big picture/take home re: shell sort

- Complexity is hard to analyze, and depends on gap, but in general
  - Average case: O(n^1.25)
  - First known algorithm to be faster than O(n^2), our example is O(n^1.5)
  - Although best case is same as insertion sort (O(n)) in practice usually worse than merge or quick sort
- The basic idea is the final insertion sort will occur on data with fewer large scale inversions, which means smaller distances in the final step

Hybrid sorting methods

- There usually is a trade off between efficiency of divide and conquer (i.e., good theoretical and actual performance) and using the function stack for recursion
- In Dr. Plank's notes, his Quick Sort implementation uses insertion sort with 115 or fewer elements for these advantages:
  - Best case is O(n) if already sorted
  - Easy to implement
- Any other algorithm can be used, though. Some STL implementations use introsort

Big picture thoughts

- In practice, median of 3 quicksort is the best (see Plank's notes); however, an adversary can derive a example when it does not too well

- Worst case scenario is O(n^2), just like other sorts

- Solution: bound the size and use heapsort on subproblems
  - 1/200th the time on 100,000 elements designed to thwart median of 3

Psuedocode form Wikipedia

```
procedure sort(A : array):
    let maxdepth = ⌊log(length(A))⌋ × 2
    introsort(A, maxdepth)

procedure introsort(A, maxdepth):
    n ← length(A)
    p ← partition(A)  // assume this function does pivot selection, p is the final position of the pivot
    if n ≤ 1:
        return  // base case
    else if maxdepth = 0:
        heapsort(A)
    else:
        introsort(A[0:p], maxdepth - 1)
        introsort(A[p+1:n], maxdepth - 1)
```