Tu-An Nguyen
tunhnguy@ucsc.edu
02/15/2021

CSE 13S Winter 2021
Assignment 5: Sorting
Design Document

Pre-lab

Part 1

1. How many rounds of swapping do you think you will need to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using Bubble Sort?

```
8 7 9 22 31 5    initial
8 7 9 22 5 31    1 swap
8 7 9 5 22 31    1 swap
8 7 5 9 22 31    1 swap
8 5 7 9 22 31    1 swap
8 5 7 9 22 31    1 swap
8 5 7 9 22 31    sorted
```

It will take 5 rounds of swapping.

2. How many comparisons can we expect to see in the worse case scenario for Bubble Sort? Hint: make a list of numbers and attempt to sort them using Bubble Sort.

```
6 5 4 3 2 1    initial
5 4 3 2 1 6    5 comparisons
4 3 2 1 5 6    4 comparisons
3 2 1 4 5 6    3 comparisons
2 1 3 4 5 6    2 comparisons
1 2 3 4 5 6    1 comparison
1 2 3 4 5 6    sorted
```

It will take `n + (n - 1) + (n - 2) + ... + 1 = (n (n+1))/2` comparisons in the worst case scenario.

3. How would you revise the algorithm so the smallest element floats to the top instead?

At each comparison, swap only if the second element is larger than the first.

Part 2

1. The worst time complexity for Shell Sort depends on the sequence of gaps. Investigate why this is the case. How can you improve the time complexity of this sort by changing the gapsize? Cite any sources you used.

The point of the gaps is to quickly move the smaller numbers to the left and the larger numbers to the right so that on the last pass, there will be less comparisons. If the gaps are too big or small, there won't be much moving around, and the efficiency will be more similar to insertion sort. I believe the way to improve time complexity is to choose a gapsize that isn't too big or small. One way to do this is through the Knuth Sequence where we pick a gap of size `h` with this formula: `f(h) = 3h + 1`. Source: https://www.codesdope.com/blog/article/shell-sort/

Part 3

1. Quicksort, with a worst case time complexity of $O(n^2)$, doesn't seem to live up to its name.Investigate and explain why Quicksort isn't doomed by its worst case scenario. Make sure to cite any sources you use.

Part 4

2. Explain how you plan on keeping track of the number of moves and comparisons since each sort will reside within its own file.

Purpose

The purpose of this Lab is to implement Bubble Sort, Shell Sort, Quicksort, and Heapsort. There will be a test harness for the implemented sorting algorithms. The test harness will create an array of random elements and test each of the sorts. Lastly, the program will gather statistics about each sort and its performance. This includes the size of the array, the number of "moves" required, and the number of "comparisons" required.

Pseudocode

```
sorting.c

int main(int argc, char **argv):
    Set opts;
```

```
    while there are options:
        put options in Set opts

    dynamically allocate memory for array of integers, arr
    int[] arr;
    put random numbers into arr

    for each sort in opts
        sort arr
        collect size of array, number of moves required, and number of
comparisons required
        print sorted array elements

    free arr
```

set.c

```
Set set_empty(void):
    return 0 in uint32 as a Set

bool set_member(Set s, uint8 x):
    uint32 mask = 1 << x

    s = s & mask

    if s:
        return true
    return false

Set set_insert(Set s, uint8 x):
    uint32 mask = 1 << x

    s = s | mask

    return s

Set set_remove(Set s uint8 x):
    uint32 mask = ~(1 << x)

    s = s & mask

    return s

Set set_intersect(Set s, Set t):
    return s & t
```

```
Set set_union(Set s, Set t):
    return s | t

Set set_complement(Set s):
    return ~s

Set set_difference(Set s, Set t):
    return s & ~t
```

bubble.c

```
void bubble_sort(uint32 *A, uint32 n):
    swapped = true

    while swapped:
        swapped = false

        for i in [1, n):
            if A[i] < A[i - 1]:
                swap A[i] and A[i - 1]
                swapped = true

        n -= 1
```

shell.c

```
void shell_sort(uint32 *A, uint32 n):
    for int g in [0, GAPS):
        for int i in [gaps[g], n):
            uint32 j = i

            uint32 temp = A[i]

            while j >= gaps[g] && temp < A[j - gaps[g]]:
                swap A[j] and A[j - gaps[g]]

                j -= gaps[g]

            A[j] = temp
```

quick.c

```
int64 partition(uint32 *A, uint32 lo, uint32 hi):
    uint32 pivot = A[lo + ((hi - lo) // 2)];
    uint32 i = lo - 1
```

```
    uint32 j = hi + 1

    while i < j:
        do:
            i += 1
        while A[i] < pivot

        do:
            j -= 1
        while A[j] > pivot

        if i < j:
            swap A[i] and A[j]

    return j

void quick_sort(uint32 *A, uint32 n):
    uint32 left = 0
    uint32 right = n - 1

    Stack s = stack_create()
    stack_push(s, left)
    stack_push(s, right)

    while !stack_empty(s):
        int64 hi, lo, p
        stack_pop(s, hi)
        stack_pop(s, lo)
        p = partition(A, lo, hi)

        if p + 1 < hi:
            stack_push(s, p + 1)
            stack_push(s, hi)

        if lo < p:
            stack_push(s, lo)
            stack_push(s, p)
```

stack.c

```
struct Stack:
    uint32 top
    uint32 capacity
    int64 *items

Stack *stack_create(void):
    allocate memory for stack w/ MIN_CAPACITY
```

```
    top = 0
    capacity = MIN_CAPACITY

void stack_delete(Stack **s):
    free allocated memory in s

bool stack_empty(Stack *s):
    return top == 0

bool stack_push(Stack *s, int64_t x):
    if stack is full reallocate memory w/ double the capacity
        return false if reallocation fails

    s.items[top] = x
    s.top += 1

    return true

bool stack_pop(Stack *s, int64_t *x):
    if stack is empty:
        return false

    top -= 1
    x = s.items[top]

    return true

void stack_print(Stack *s):
    for i in [0, capacity):
        print s.items[i]
```

heap.c

```
uint32 max_child(uint32 *A, uint32 first, uint32 last):
    uint32 left = 2 * first
    uint32 right = left + 1

    if right <= last && A[right - 1] > A[left - 1]:
        return right
    return left

void fix_heap(uint32 *A, uint32 first, uint32 last):
    bool found = false
    uint32 parent = first
    uint32 great = max_child(A, parent, last)

    while parent <= last / 2 && !found:
        if A[parent - 1] < A[great - 1]:
```

```
            swap A[parent - 1] and A[great - 1]
            parent = great
            great = max_child(A, parent, last)
        else:
            found = true

void build_heap(uint32 *A, uint32 first, uint32 last):
    for int parent in [last / 2, first - 1) w/ step = -1:
        fix_heap(A, parent, last)

void heap_sort(uint32 *A, uint32 n):
    uint32 first = 1
    uint32 last = n

    build_heap(A, first, last)

    for leaf in [last, first) w/ step = -1:
        swap A[first - 1] and A[leaf - 1]
        fix_heap(A, first, leaf - 1)
```