Acer Cristea
Cruz-id: acristea
Winter 2023 - CSE13S

Assignment 3 - Sorting: Putting your affairs in order

Overview

Implement Shell Sort, Batcher Sort (Batcher's method), Heapsort, and recursive Quicksort based on the provided Python pseudocode in C. The interface for these sorts will be given as the header files shell.h, batcher.h, heap.h, and quick.h. You are not allowed to modify these files for any reason. Implement a test harness for your implemented sorting algorithms. In your test harness, you will create an array of pseudorandom elements and test each of the sorts. Your test harness must be in the file sorting.c. Gather statistics about each sort and its performance. The statistics you will gather are the size of the array, the number of moves required, and the number of comparisons required. Note: a comparison is counted only when two array elements are compared.

The Deliverables
(files that need to be included in the directory "asgn3")

1. batcher.c: This file contains batcher_sort(). This Merge Exchange Sort is a sorting network and will sort a randomized array via Batcher's method.

2. batcher.h: This file is the header file for batcher.c and specifies its interface.

3. shell.c: This file contains shell_sort(). This file will sort a randomized array via a Shell Sort method. REMEMBER to use gaps.h in this file.

4. shell.h: This file is the header file for shell.c and specifies its interface.

5. gaps.h: This file contains an array of gaps that will be used to sort shell.c

6. heap.c: This file contains heap_sort(). This implements Heapsort.

7. heap.h: This file is the header file for heap.c and specifies its interface.

8. quick.c: This file contains quick_sort(). This implements Quicksort.

9. quick.h: This file is the header file for quick.c and specifies its interface.

10. set.c: This file implements bit-wise Set operations.

11. set.h: This file specifies the interface to set.c.

12. stats.c: Implements the statistics module. Provides a swap mechanic and compare mechanic. Also used to track moves and reset them.

13. stats.h: This file specifies the interface to the statistics module.

14. sorting.c: This file contains main() and may contain any other functions necessary to complete the assignment.
    - -a: Employs all sorting algorithms.
    - -h: Enables Heap Sort.
    - -b: Enables Batcher Sort.
    - -s: Enables Shell Sort.
    - -q: Enables Quicksort.
    - -r seed: Set the random seed to seed. The default should be 13371453.
    - -n size: Set the array size to size. The default should be 100.
    - -p elements: Print out elements number of elements from the array. The default number of elements to print out should be 100. If the size of the array is less than the specified number of elements to print, print out the entire array and nothing more.
    - -H: Prints out program usage.

15. Makefile: Not provided, and must include:
    - CC = clang must be specified
    - CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified
    - make must build the mathlib-test executable, as should make all and make mathlib-test.
    - make clean must remove all files that are compiler generated.
    - make format should format all your source code, including the header files.

16. README.md: This must use proper Markdown syntax. It must describe how to use your script and Makefile

17. DESIGN.pdf: What I am currently writing, a description of the programs and the design process including supporting pseudocode.

18. WRITEUP.pdf: This document must be a proper PDF. This writeup must include what you learned from the different sorting algorithms. Under what conditions do sorts

perform well? Under what conditions do sorts perform poorly? What conclusions can you make from your findings? Must also include graphs explaining the performance of the sorts on a variety of inputs, such as arrays in reverse order, arrays with a small number of elements, and arrays with a large number of elements. You should use LATEX to produce this document.

Sets

Functions given:

- set_empty(void) → returns an empty set
- set_universal(void) →returns a set in which every possible member is part of the set.
- set_insert(Set s, uint8_t x) → inserts x into s
- set_remove(Set s, uint8_t x) → removes x from s
- bool set_member(Set s, uint8_t x) → returns a bool indicating the presence of the given value x in the set s
- set_union(Set s, Set t) → returns bits of members that are equal to 1 in either s or t (OR)
- set_intersect(Set s, Set t) → returns bits of members that are equal to 1 in both s and t (AND)
- set_difference(Set s, Set t) → returns set of elements in s and not in t
- set_complement(Set s) → return the complement of a given set

Pseudocode / Structure

*Code will be spread out across multiple files

What I know from asgn3 doc:	**Pseudo-code:**

Shell Sort (shell.c/h)
 	
Shell Sort in Python
`def shell_sort(arr):`
`  for gap in gaps:`
`  for i in range(gap, len(arr)):`
`  j = i`
`  temp = arr[i]`
`  while j >= gap and temp < arr[j - gap]:`
`  arr[j] = arr[j - gap]`
`  j -= gap`
`  arr[j] = temp`
	For this algorithm we need to use gaps.h, a file we can get when we run gaps.h.py gotten from resources, which gives an array we will use for our gaps.
	
	The shellsort() function uses two nested for loops: for each step in the gap of the array, for some i in the range of array considering the steps, and a while loop, while j is greater than the gap and our temporary number is less than the next number after applying the gap.
	
	Using these loops, we can reorder using the i's and j's as the index of some element in the array inputted.

Batcher's Sort (batcher.c/h)	Batcher's Sort acts as a parallel to Shell Sort. Instead of sorting pairs of elements that are a set gap apart, like in Shell Sort, it ksorts the even and odd subsequences of the array,

Merge Exchange Sort (Batcher's Method) in Python

```python
def comparator(A: list, x: int, y: int):
    if A[x] > A[y]:
        A[x], A[y] = A[y], A[x]

def batcher_sort(A: list):
    if len(A) == 0:
        return

    n = len(A)
    t = n.bit_length()
    p = 1 << (t - 1)

    while p > 0:
        q = 1 << (t - 1)
        r = 0
        d = p

        while d > 0:
            for i in range(0, n - d):
                if (i & p) == r:
                    comparator(A, i, i + d)
            d = q - p
            q >>= 1
            r = p

        p >>= 1
```

where k is some power of 2.
There's an even sequence and odd sequence.

We have a comparator function, which simply swaps value x if it's greater than y. That k sorts it. When k-sorting a subsequence do not appear as indices in another comparison during the same round. A clever use of the bitwise AND operator, &, guarantees this property.

The variable p tracks the current round of k-sorting. The variable r effectively represents which partitions can be considered for comparison.

Quick Sort (quick.c/h)

Partition in Python

```python
def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
    A[i], A[hi - 1] = A[hi - 1], A[i]
    return i + 1
```

** partition() places elements less than the pivot into the left side of the array and elements greater than or equal to the pivot into the right side and returns the index that indicates the division between the partitioned parts of the array.*

Quick Sort partitions arrays into two sub-arrays by selecting an element from the array and designating it as a pivot. Elements in the array that are less than the pivot go to the left sub-array, and elements in the array that are greater than or equal to the pivot go to the right sub-array.

Have to know how to navigate around range using a for loop in C

Recursive Quicksort in Python

```python
# A recursive helper function for Quicksort.
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

Quicksort is then applied recursively on the partitioned parts of the array, thereby sorting each array partition containing at least one element.

Heapsort (heap.c/h)

The first step to Heap Sort is building a heap. They can be built in multiple different ways

Heap maintenance in Python

```python
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left

def fix_heap(A: list, first: int, last: int):
    found = False
    mother = first
    great = max_child(A, mother, last)

    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
```

Routines to sort elements using Heapsort:

1) *Building a heap (ordering the array elements → obey Max heap constraints for this assignment)*
2) *Fixing a heap (sorting the array)*

Heapsort in Python

```python
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
        fix_heap(A, father, last)

def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, first, leaf - 1)
```

focussing on different things, in our case we want to build based of Max Heap, meaning the highest number is the root, and it slowly goes down the more children it has. max_child orders the array and fix_heap sorts the array.

build_heap() → using our sorted array, we then build it into a heap giving a starting root and assigning children to parents. No more than 2 children per parent.

heapsort() → this goes through the heap checking leaf by leaf if they are in the correct order. If a child is bigger than a parent, they are swapped. That -1 swaps their positions in the tree.

Test Harness (sorting.c)

```
$ ./sorting -q -n 1000 -p 0
Quick Sort, 1000 elements, 18642 moves, 10531 compares
$ ./sorting -hb -n 15 -p 0
Bubble Sort, 15 elements, 144 moves, 70 compares
Heap Sort, 15 elements, 82 moves, 65 compares
$ ./sorting -a -n 15
Bubble Sort, 15 elements, 90 moves, 59 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Heap Sort, 15 elements, 144 moves, 70 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Quick Sort, 15 elements, 135 moves, 51 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
```

```c
printf("%13" PRIu32); // Include <inttypes.h> for PRIu32.
```

Similar to that of assignment 2, we will be supporting a variety of command-line options using getopt().

The following must be supported:

- a (all sorting algorithms)
- s (shell sort)
- b (batcher's sort)
- q (quick sort)
- h (heap sort)
- r (seed) → needs : for optarg
- n (size) → needs : for optarg
- p (prints elements)
- H (prints program usage)

For the sorting algorithms, we will be using a similar implementation to that of the mathematical functions in the assignment 2. We should utilize printf() statements to create the formatting of the data as shown to the left.

We will be using pointers to assign the values into their respective variables. We will be

	using the random generator with the seed given to us by the output, else use the default seed. Each case for the sorting functions has to have a respective array of length size that the user inputs, else use the default, as well as each element being a 30 bit number generated by the random generator. We will need to use sets for masking the 64 unsigned integer generated by the random number generator, into the 30 bit unsigned integer than we needed. And the malloc of calloc function to create these arrays.
WRITEUP.pdf *Create graphs explaining the performance of the sorts for the following inputs:* - *Arrays in reverse* - *Arrays with small number of elements* - *Arrays with a large number of elements* ** use awk for parsing the output* **axes are log scaled*	Use a bash script that will compute the data we want to graph. To compute the reverse of the data, we can use the "sort" command and the "-nr" to imply that we want the data sorted in reverse order before redirecting it into the file. For all of the plots, we can use a for loop to iterate over some numbers (this will change based on which graph, i.e. graph 2 only needs a small number of elements, etc.). We will also need to use the command "awk" to parse the output correctly and get the data we want to plot. Next step is to use gnuplot and its attributes to create the graphs to visualize our data.