Alex Clanton

10/2/2021

Prof.Long

CSE13S

Pseudocode/DESIGN.pdf Initial

Rundown:

This program implements Insertion Sort, Shell Sort, Heapsort, and recursive Quicksort. The program is run via command line inputs (use -h for more information). The program takes 'random' numbers and implements them in a dynamically allocated array. The selected sorts then sort the array and print out the efficiency statistics.

3 given parameters are Stats *stats, uint32_t *A, uint32_t n

Outlined structure:

sorting.c links to header files of:

insert.h-implemented in insert.c

heap.h-implemented in heap.c

quick.h-implemented in quick.c

set.h

stats.h-implemented in stats.c

shell.h-implemented in shell.c

sorting.c:

Create a stat struct that was defined in stats.c/stats.h

Main receives int argc, char **argv

Define 4 uint32_t pointers for the dynamic arrays

Define an 3 sized array for -n -p -r inputs

Define options should be “aeisqrnph ”

Use getopt(argc,argv, OPTIONS)

Depending on the input via a switch case, keep track of the user inputted values by set, set.h.

In cases of ‘r’,’n’,’p’. Use atoi() and take the variable argv where the element optind is converted. This value is now an int

Insert a value corresponding to a selected choice into the set. Such as insert 0 to mean that the heapsort was selected by the user. Increment an int track variable to know how many inputs the user entered

Pass user or default seed if the user did not enter -r to srandom() and create a dynamic array of random numbers. Use malloc for a sized 32-bit int and size of the user array value. Create 4

-user default seed of 13371453

-user defaulted size of 100

Test if the user array is too big for the size of memory on the computer

Test if user entered no values for -r -n -p

Give default

For loop for the number of entries

If a value of 8 is in the set, a help prompt was selected. Print and exit program

if/else if the set has the value for an input

Call the sort user entered, in the arguments pass the address of stat struct, dynamic array, and size of the array

Use stats. moves and stats. comparison values and print those results to the user

For loop the size of the array

And print a 5 column table, spaced 13 widths apart. Every 5th increment print a new line

Reset the stats struct

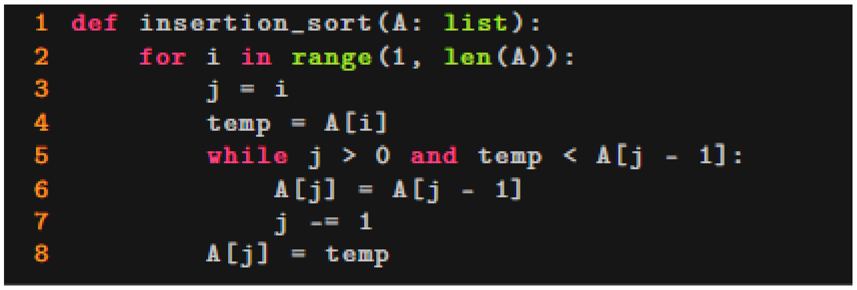
Delete the value in the set

Continue this pattern using track for total amount of sorts selected

Free the dynamic array:

Do the same for structure every sort, but call a different sorting algorithm if the value is in the set

Insert sort:



```
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

The function will need the struct stats passed as a parameter, the pointer to the dynamic array, and the size n of the array. The random value array is a pointer to save memory usage rather than making a copy, and int for the number of selected elements.

Follow a pattern of checking the previous term and if it’s larger or smaller, then move that previous term ‘up’. You can either sink large or sink the smallest. Every term is compared to previous, moved, and continued.

Use 1 for loop for size of the array

Use move() from stats.h/stats.c when moving A[i] to temp to keep track of comparisons

While loop until j, the value of for loop above is greater than 0 and previous is smaller than current. Use cmp() for comparison to the array A[j-1], to keep track of comparisons

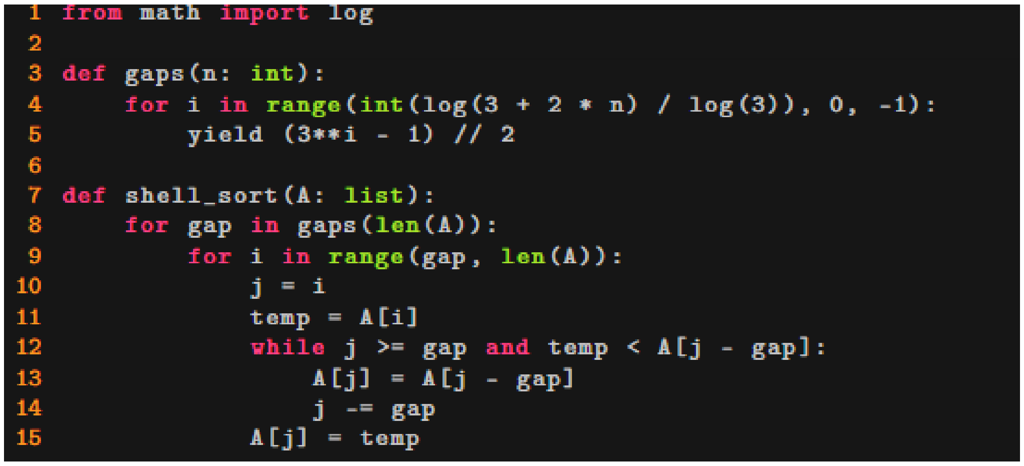
Move values out of temp

Use move() function for stats.c to keep track of moves

n^2 efficiency

Nothing is returned:

Shell sort:



```python
from math import log

def gaps(n: int):
    for i in range(int(log(3 + 2 * n) / log(3)), 0, -1):
        yield (3**i - 1) // 2

def shell_sort(A: list):
    for gap in gaps(len(A)):
        for i in range(gap, len(A)):
            j = i
            temp = A[i]
            while j >= gap and temp < A[j - gap]:
                A[j] = A[j - gap]
                j -= gap
            A[j] = temp
```

The function will need the struct stats passed as a parameter, pointer to the dynamic array, and int of the length of the array. The random value array is a pointer to save memory usage rather than making a copy, and int for the number of selected elements.

Follow the pattern to sort, sort a pair of elements that are far apart in memory, continue until the gap is 1. Your values will be sorted, the shell is closely associated with the insert.

shell_sort() it goes through the gaps which are called in function gaps()

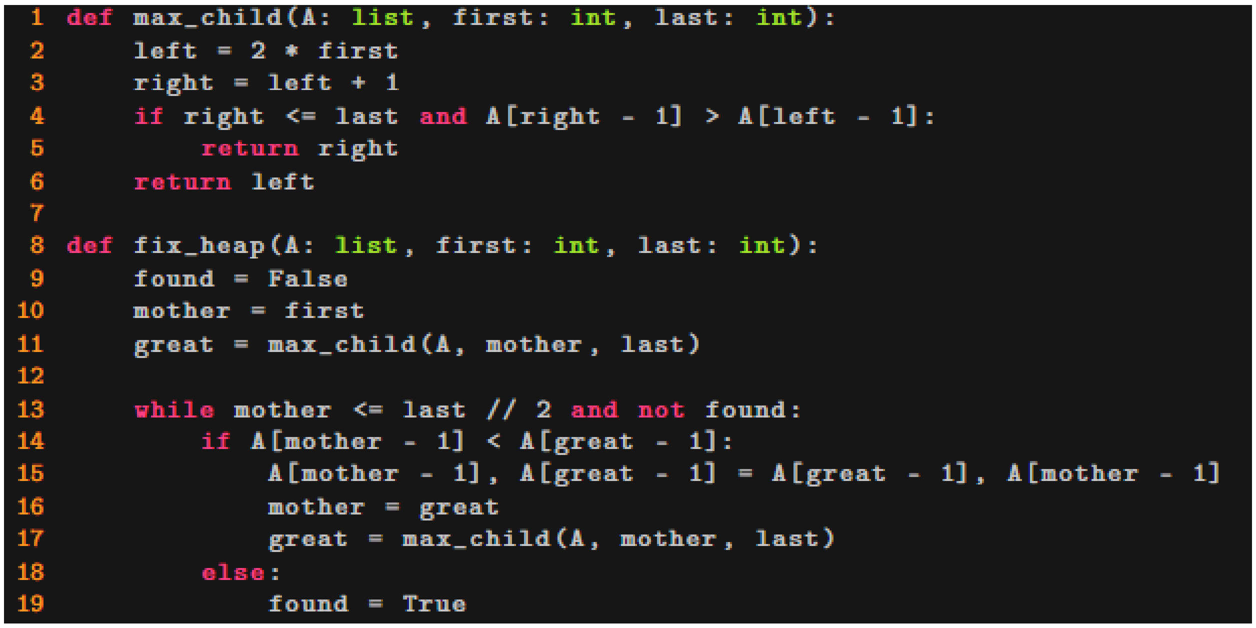
Case of 2:

If theirs 2 elements then cmp() from stats.c them and swap if necessary

while loop the gap is greater than 0

```
    For loop is the gap is less than the number of elements
        Use a temp value and us move() from stats.c to keep track of moves
    Store a temporary variable of the current position in A
    While that gap is greater than the old gap, and temp value and the array value is less than
        Swap from stats function to keep track of moves
        Decrement gap
    Move() from stats to assign temp to the array to keep track of moves
Gaps return smallest and largest gap found by:
0 is greater than total elements /3
    Then increment(step) is multiplied by 3:w
    Then 1 is added to increment
    Return increment
```

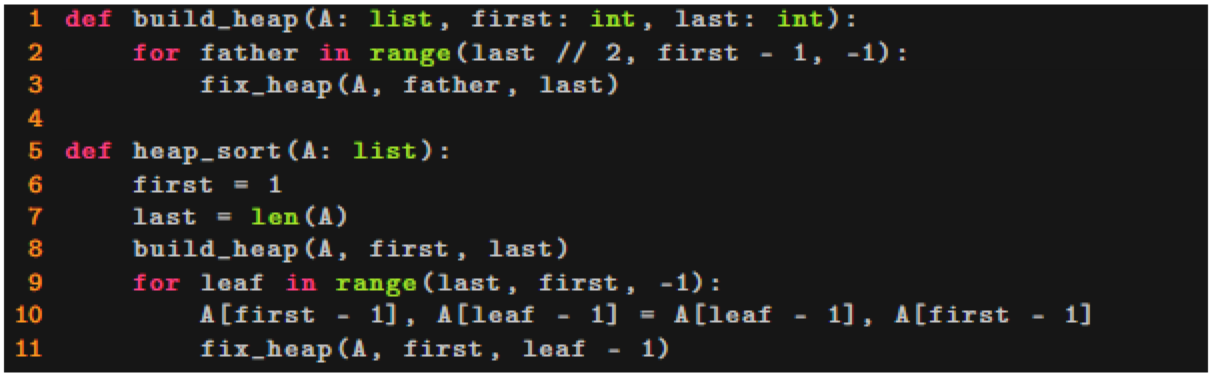
Heapsort:



```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
```



```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)
```

The heap_sort() will need the struct stats passed as a parameter, a pointer to the dynamic array A, and the size of the int. The random value array is a pointer to save memory usage rather than making a copy, and int for the number of selected elements. The other functions will be passed these values as needed. heap_sort() creates two variables first which is given 1, and last which is the size of the array.

heap_sort() calls bulid_heap which passes the 3 given parameters and first and last value.

heap_sort() then iterates through last until its less than first where last decrements by 1

> You then swap first-1 and last-1 values. Use swap() from stats.c to keep track of moves
>
> Fix_heap() which takes 3 parameters and first, and last-1

bulid_heap() takes 5 parameters above, and creates the parent child tree, but finding the father= last/2, and test= first-1

For loop, father is greater than first-1 decrement father by 1

> Call fix_heap()

fix_heap():

Found is false

Create 2 int variables where mother=false and greatest value is, call max_child() where 3 given parameters and first and last are passed

While the mother is greater/equal to last/2 and found is false

    Use cmp() if mother-1 is less than greatest-1 to keep track of comparisons

Compare mother-1 and greatest-1

    If true you use swap() to keep track of moves, those and mother gets greatest, and max_child is called to make a new heap

largest child

max_child():

Left child is first*2

The right child is left+1

If the right is less than the last

    Compare right-1 and left-1, use cmp() to keep track of comparisons

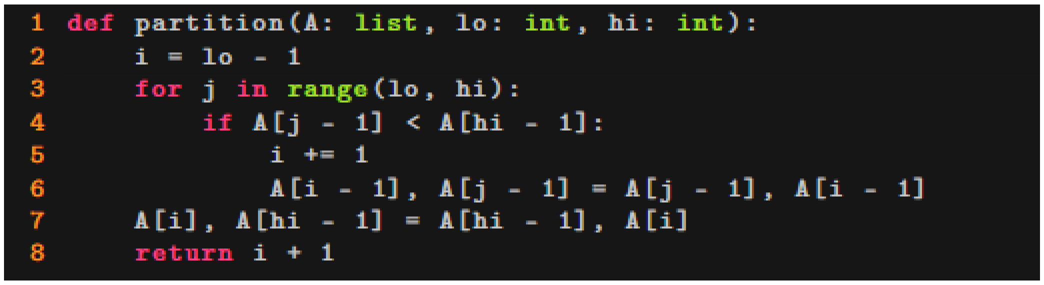
    If true then: return right

Otherwise, return left

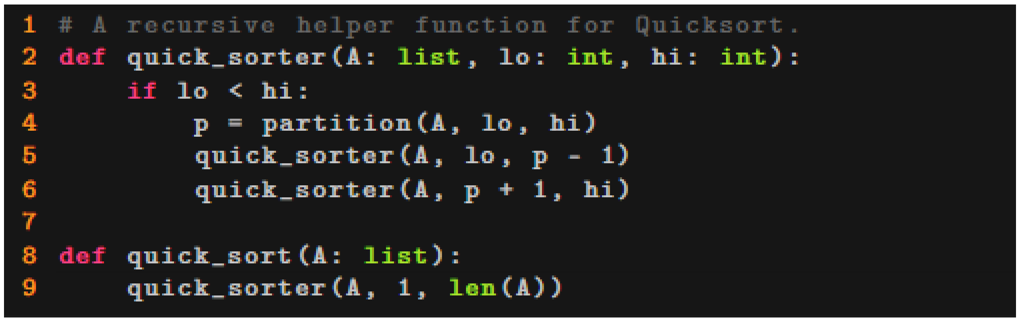
Heapsort is similar to binary trees where a value is chosen as the parent, the children nodes must be smaller or equal to the parent. The left child value is 2k, and the right is 2k+1

Construct your heap as a max heap, find the largest value and make it the first element of the array. Where the largest found values are at the end of the array.

Quicksort:



```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
```



```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))
```

The function quick_sorter() will need the struct stats passed as a parameter, the random value array as a pointer to save memory usage, an int for the number of selected elements. The other functions will be passed these values as needed.

Call quick_sort() which calls quick_sorter() where 3 parameters are passed, and low int is passed as 0, and high as n-1

quick_sorter(), compares if high is greater than low

> If true call partition() pass 3 parameters and high and low
>
> Store the return as p, then recursively call itself where high is p-1
>
> Recursively call itself p+1 as low

partition()

Sets pivot to the high value of the array A

A low as low-1

For loop, if low is less than high, increment low

> Compare A of low to pivot value, use cmp() from stats.c to keep track of comparisons

If true then swap those values, use swap() from stats.c to keep track of moves

Swap low-1 and high, use swap() from stats.c to keep track of moves

Return low+1

Quicksort, taking an element makes it a pivot. Move smaller elements to the left side and larger elements to the right. Then partition() returns pivot point and recursively sorts the left and right

Statistics:

Cmp() takes x and y element, stats struct as pointer to save memory

Increment compares

Return -1 if x is less than y, or 0 if x is equal and 1 if x is greater

move() in insertion and shell by adding one to moves and return x

Increment moves

swap() swap address in the pointers, of the 2 variables

Increment moves by 3

Set move to and comparison to 0