Asgn4: Final Design Doc
Mika Peer Shalem
Mpeersha

General Idea

The program would have four different sorting algorithms, each with its own faults and benefits. The sorts are Shell sort, Bubble sort, Quick sort, and Heap sort. It will compute statistics about each sorting function to help figure out in which situations each function is preferable. It collects the number of moves and comparisons used in each algorithm. There would be command line options so users can decide how to run the sorts.

Shell.c

Shell sort modifies pairs of elements that are far from each other and uses the gap function to compute the pairs.

next_gap(gap)

If (gap > 1):
    If gap is smaller or equal to 2:
        gap = 1
    Else:
        gap = 5 * gap / 11
Return gap

shellsort()

Reset stats
Gap = 0
Check = 1
for gap=next_gap(n_elements), stops when check = 1, gap = next_gap (gap):
    For int i starts with gap, until i is smaller than the length of the array, i += 1:
        Set j to i
        Set temp to the value of the array at index i
        While j is bigger than or equal to gap AND temp is smaller than a[j-gap]
            Set a[j] to a[j-gap]
            Subtract gap from j
        Set a[j] to temp

If gap is equal to 1:

Set check to 0

Print the modified array

Example

Unsorted list = 33 31 40 8 12 17 25 42

First 4 groups: 33 31 40 8 | 12 17 25 42 → (33,12) (31,17) (40,25) (8,42)

First swap: (12, 33) (17,31) (25,40) (8,42) → 12 17 25 8 33 31 40, 42

Second 2 group: (12, 25, 33, 40) (17, 8, 31, 42) → sort them 12 8 25 17 33 31 40 42

...

Continue until completely sorted

bubble.c

Reset stats

for int i is equal to 0, i is smaller than the length of the given array-1, i +=1:

Swap_status = 1;

for int j is equal to the length of array -1, j is not equal to 1, j -=1:

If arr[j] < arr[j-1]:

Swap arr[j] and arr[j-1]

Swap_status = 0

If swap_status = 1:

Stop the loop

Print the modified array

Quick.c

quicksort()

Set left to 0

Set right to number of elements - 1

Set middle to (right+left) / 2

Set pivot to a[middle]

If number of elements is less than 8:

Call shell sort

Else:

While left is smaller than right:

While arr[left] is smaller than or equal to pivot: // find the first element that is bigger than pivot on left side

If left+1 is smaller than the number of elements

Left += 1

Else

Stop the loop

While arr[right] is bigger than pivot: // find the first element smaller/equal to pivot on right side

If right-1 is bigger than or equal to 0

Right -= 1

Else

Stop the loop

If left is smaller than right

Swap arr[left] and arr[right]

Call quick sort with the array and left as the number of elements

Call quick sort with the array from index right and number of elements - right as the number of elements

Example:

Arr = 2,7,1,9,3,6,8,5,10

left=0, right=8

middle=(0+8)/2=4

pivot=3

Left is smaller than right, so doing the first loop:

Arr[left] <= 3 // 2<=3

Left += 1

When left=2, arr[left] is bigger than pivot

Arr[right] >3 // 10>3

Right += 1

When right equals 4, arr[right] is smaller or equal to the pivot

Swap them → arr is now: 2,3,1,9,7,6,8,5,10

Call quicksort with the args (stats, arr, left)

Call quicksort with the args (stats, arr from position right, number of elements - right)

...

Keep going until the array is sorted

Heap.c

l_child(n)

Return 2*n+1

r_child(n)

Return 2*n+2

parent(n)

If (n <= 1):

Return 0

Return (n-1)/2

up_heap(a,n)

While n is smaller than number of elements -1 AND a[n] is smaller than the value of its parent

Swap a[n] and a[parent(n)]

n = parent(n)

Example

[4, 2, 6, 9, 20, 1, 3]

n = 7: nothing

n=6: a[n]=3, a[parent(n)]=6. Swap → [4, 2, 3, 9, 20, 1, 6].

n=2: a[n]=3, a[parent(n)]=4. Swap → [3, 2, 4, 9, 20, 1, 6].

...

down_heap(a, heap size)

n = 0

Smaller = 0

While left child is smaller than heap size

    If right child is equal to 0

        Smaller = right child

    Else

        If a[left child] is smaller than a[right child]

            Smaller = left child

        Else

            Smaller = right child

    If a[n] is smaller than a[smaller]

        Stop the loop

    Swap a[n] and a[smaller]

    Set n to smaller

Example:

[3,2,4,9,20,6]

n=0: smaller=0. Arr[1] < arr[2] → smaller = 1. a[0]>a[1] →[2,3,4,9,20,6]

n=1: smaller=0. Arr[3]<arr[4] → smaller=4. a[1]<a[4]

Stop

build_heap(a)

    Create a heap array with the size of a and every spot intialized to 0

    Go through every element of the array

        Set heap[n] to a[n]

        Call up_heap with the variables heap and n

    Return pointer to heap

heap_sort(a)

    Set heap to build_heap (a)

    Create a sorted_list array with the size of a and every spot intialized to 0

    for int n is equal to 0, n is smaller than the length of the given array, n +=1:

Set sorted_list[n] to heap[0]
Set heap[0] to heap[len(a)-n-1]
Call down_heap with the variables heap and len(a)-n
Return the sorted list

Example
Bi = big integer numebrs
Sorted list = [bi, bi, bi, bi, bi, bi]
Arr = [1,2,4,9,20,6,3]
n=0:
Sorted list = [1, bi, bi, bi, bi, bi]
Arr = [3,2,4,9,20,6]
n=1:
Sorted list = [1, 2, bi, bi, bi, bi]
Arr = [6,3,4,9,20]

Sorting. C

Set default seed to 13371453
Set default size to 100
Set default elements to 100
Set with the size of all options
While getopt argument is not -1 (getopt goes through all the given options and returns -1 when it runs out of options):

In case the argument is equal to a:
Set the corresponding bit in the set to 1
Do all of the options described below
In case the argument is equal to s:
Set the corresponding bit in the set to 1
Call shell sort
In case the argument is equal to b:
Set the corresponding bit in the set to 1
Call bubble sort

In case the argument is equal to q:
    Set the corresponding bit in the set to 1
    Call quick sort
In case the argument is equal to h:
    Set the corresponding bit in the set to 1
    Call heap sort
In case the argument is equal to r:
    Set the corresponding bit in the set to 1
    Set random seed
In case the argument is equal to n:
    Set the corresponding bit in the set to 1
    Convert optargs to an int
    If optarg is between 1 and 250,000 (inclusive):
        Set size to optarg
In case the argument is equal to p:
    Set the corresponding bit in the set to 1
    Set the number of elements to be printed
In case the argument is equal to h:
    Set the corresponding bit in the set to 1
    Print program usage

Go through each bit of the set; if it's equal to one, call the corresponding program