CSCA48 Summer 2017

Week 12 - Complexity

Brian Harrington

University of Toronto Scarborough

July 24-26, 2017

University of Toronto Scarborough

Complexity

- First attempts to analyze an algorithm: not great
- Want a way to analyze algorithms that is:
    - Independent of machine/implementation/language
    - Easy to directly compare
    - Focused on “big picture” (large values)

Asymptotic Upper Bound (Big-Oh)

- given $f(n)$ and $g(n)$
- $f(n) \in O(g(n))$ iff:
  - There exists constants c and b such that
  - for n larger than b
  - $f(n) \leq c * g(n)$
- This means:
  - we can find values for c and b, such that
  - we can multiply $g(n)$ by c
  - and eventually (after reaching $n = b$, it will `dominate` $f(n)$

Asymptotic Upper Bound (Big-Oh)

- Back to our selection sort example
- $selection\_sort(n) = 14n^2 + 16n$
- Show that $insertion\_sort(n) \in O(n^2)$
  - Need to find b and c
  - $c = 15$
  - $14n^2 + 16n \leq 15n^2$ for $n \geq ?$
- What if we over-under counted?
- Need to find a different b and c, but will still be $\in O(n^2)$

Asymptotic Upper Bound (Big-Oh)

- In practice:
  - Come up with worst possible input for algorithm
  - Analyze steps for that input
  - Only care about largest term in expression
- Let's practice!

Break


INEFFECTIVE SORTS
DEFINE HALFHEARTEDMERGESORT(LIST):
IF LENGTH(LIST) < 2:
RETURN LIST
PIVOT = INT(LENGTH(LIST) / 2)
A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
// UMMMMM
RETURN [A, B] // HERE. SORRY.
DEFINE FASTBOGOSORT(LIST):
// AN OPTIMIZED BOGOSORT
// RUNS IN O(N LOG N)
FOR N FROM 1 TO LOG(LENGTH(LIST)):
SHUFFLE(LIST):
IF ISSORTED(LIST):
RETURN LIST
RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
DEFINE JOBINTERVIEWQUICKSORT(LIST):
OK SO YOU CHOOSE A PIVOT
THEN DIVIDE THE LIST IN HALF
FOR EACH HALF:
CHECK TO SEE IF IT'S SORTED
NO, WAIT, IT DOESN'T MATTER
COMPARE EACH ELEMENT TO THE PIVOT
THE BIGGER ONES GO IN A NEW LIST
THE EQUAL ONES GO INTO, UH
THE SECOND LIST FROM BEFORE
HANG ON, LET ME NAME THE LISTS
THIS IS LIST A
THE NEW ONE IS LIST B
PUT THE BIG ONES INTO LIST B
NOW TAKE THE SECOND LIST
CALL IT LIST, UH, A2
WHICH ONE WAS THE PIVOT IN?
SCRATCH ALL THAT
IT JUST RECURSIVELY CALLS ITSELF
UNTIL BOTH LISTS ARE EMPTY
RIGHT?
NOT EMPTY, BUT YOU KNOW WHAT I MEAN
AM I ALLOWED TO USE THE STANDARD LIBRARIES?
DEFINE PANICSORT(LIST):
IF ISSORTED(LIST):
RETURN LIST
FOR N FROM 1 TO 10000:
PIVOT = RANDOM(0, LENGTH(LIST))
LIST = LIST[PIVOT:] + LIST[:PIVOT]
IF ISSORTED(LIST):
RETURN LIST
IF ISSORTED(LIST):
RETURN LIST:
IF ISSORTED(LIST): //THIS CAN'T BE HAPPENING
RETURN LIST
IF ISSORTED(LIST): //COME ON COME ON
RETURN LIST
// OH JEEZ
// I'M GONNA BE IN SO MUCH TROUBLE
LIST = [ ]
SYSTEM("SHUTDOWN -H +5")
SYSTEM("RM -RF ./")
SYSTEM("RM -RF ~/*")
SYSTEM("RM -RF /")
SYSTEM("RD /S /Q C:*") //PORTABILITY
RETURN [1, 2, 3, 4, 5]

Selection Sort

- Simple
- In-place
- Idea:
  - ```
    for each index i in L:
          go through L[i:]  to find
          item that belongs at L[i]
    ```
- When is selection sort most efficient?
- When is insertion sort least efficient?
    ```

Insertion Sort

- Simple
- In-place (requires little/no extra memory)
- Idea:
  - ```
    for each index i in L:
            insert the item at L[i] into the
            correct place in L[:i]
    ```
- When is insertion sort most efficient?
- When is insertion sort least efficient?
    ```

Quicksort

- Classic "Divide and conquer" recursion
- Select item from L to be `pivot`
- split L into L1, L2, L3
- L1 - items < pivot
  L2 - items = pivot
  L3 - items > pivot
- S1 = quicksort(L1)
  S3 = quicksort(L3)
- return S1 + L2 + S3

Complexity of Quicksort

- What is worst possible input?
- Depends on how you choose your `pivot`

MergeSort

- split L into L1 and L2
- recursively sort L1 and L2
- merge L1 and L2
- `mergesort(L):`

```
if len(L) < 2, return L
split L into L1 and L2
S1 = mergesort(L1)
S2 = mergesort(L2)
S = merge(S1, S2)
return S
```

- Merging can be very efficient using linked lists

Complexity of Merge Sort

- $log(n)$ "levels"
- each item visited once per level
- Therefore $O(n * log(n))$

HEAPSORT

- put each element in L into a heap
- get them back out one at a time
- heapsort(L):

```
    for next_item in L
        my_heap.insert(next_item)
    S = []
    while(not my_heap.is_empty())
        s.append(my_heap.remove_min())
    return S
```

- Can actually be done in-place

Complexity of HeapSort

- *n* insertions + *n* remove_mins
- each insertion requires (at worst) one swap for each level of the tree
- each remove_min requires (at worst) one swap for each level of the tree
- we have an efficient Heap implementation that results in a complete tree
- a complete tree of *n* nodes has at most $log(n)$ levels
- Therefore $O(n * log(n) + n * log(n)) = O(n * log(n))$