CS 151

Sorting, reviewed and expanded
Announcements

• prelab 10’s remaining parts are due now
Recall the **sorting** problem:

- **input:** a list \( L \) of \( n \) integers
- **goal:** return \( L \) sorted in increasing (or, non-decreasing) order

We’ve seen a variety of sorting algorithms:

- selection sort
- insertion sort
- mergesort
- heap sort
- bucket sort
- radix sort

We also know that any comparison-based sort is at best \( O(n \log n) \).

Still, some sorts are better than others in certain situations.

And some sorts require extra space while others can be done “in-place”.

Today we’ll review the sorts so far, and present two more sorting algorithms.
Selection Sort

idea: keep the p smallest elements at front, in order
Start with p=0 (the smallest element may not be at the front)
On each iteration, find the next smallest element (i.e. increment p),
and move it to the end of the sorted portion

Worst-case running time: $O(n^2)$
Best-case running time: $O(n^2)$ because finding next smallest elt is $O(n)$
This is in-place because we only use the original space used by the input list.
Insertion Sort

idea: keep a growing “sorted list so far” in front portion of array
   Initially, the sorted portion contains only the first element (it’s sorted)
   On each iteration, take first element of unsorted portion, and
       insert it into the sorted portion at the correct position

Worst-case running time: $O(n^2)$
Best-case running time: $O(n)$ if list is already sorted
This is in-place because we only use the original space used by the input list.
Mergesort

idea: if the given list has 0 or 1 elements, return it as sorted
otherwise, split the list into two equal-ish halves L and R
recursively sort L and R using mergesort
merge the sorted L and R:
  initialize a new sorted list S to the empty list
  repeatedly remove the min remaining element from L and R
  combined, and add it to the end of our sorted list S
return S

Worst-case running time: $O(n \log n)$
Best-case running time: $O(n \log n)$ because always do same amount of work.
This is not in-place because we make use of a new sorted list S on each call.
(But we use $O(n)$ extra space for S.)
Heap sort

idea: insert each of the n elements into a heap
initialize a new sorted list S to the empty list
while the heap is not empty
    repeatedly remove the min remaining element from the heap, and
    add it to the end of our sorted list S
return S

Worst-case running time: $O(n \log n)$
Best-case running time: $O(n \log n)$ b/c every removal could cause a percDown.
This is not in-place because we make use of a new sorted list S on each call.
(But we use $O(n)$ extra space for S.)
Bucket sort

idea: if we know the elements are all within the range 0...R for some int R
initialize an array A[0..R] (one bucket per range value)
for each element e, place it at A[e]
initialize a new sorted list S to the empty list
for i=0 to R, if A[i] is non-null, add A[i] to end of S
return S

Worst-case running time: O(R)
Best-case running time: O(R) b/c the O(R) for loop is run regardless of values.
This is not in-place because we make a possibly LARGE array A.
(And this array A is O(R) amount of space, and possibly R >> n.)
Radix sort

idea: if we know the elements are each a sequence of radii / bases
   insert each element into a Trie T
initialize a new sorted list S to the empty list
loop through the Trie in order, adding isWord elements to end of S
return S

Worst-case running time: $O(\text{?})$ where $e^*$ is the max-length element
Best-case running time: $O(\text{?})$ b/c we have to insert and remove each elt.
This is not in-place because we make a possibly LARGE Trie T.
(And this Trie T is $O(\ (#\text{bases})^{\mid e^* \mid})$ amount of space, and likely $>> n.$)
Quicksort

idea: if the given list has 0 or 1 elements, return it as sorted
pick any value v in list as a “pivot” value
partition list-v into L, R s.t. elts of L are < pivot, and elts of R are >= pivot
recursively quicksort L and R
return concatenation(L, pivot, R)

RT: Let T(n) denote the running time of quicksort on a list of n elements.
Then T(n) = T(|L|) + T(|R|) + O(n) and T(0)=T(1) ∈ O(1)
When pivot is “bad”, could have |L|=n-1 every time. If “good”, |L| ∈ O(n/2)
Worst-case running time: O(n^2) but AVERAGE is O(n log n)
Best-case running time: O(n log n) b/c same RT as mergesort.

This can be done in-place if the partition is done carefully in-place (and it can.)
(move pivot to end. find l-most elt > pivot, r-most elt < pivot, and swap them...)
Q: how do we pick good pivots, i.e. a pivot that causes \[ \frac{1}{4} n \leq |L| \leq \frac{3}{4} n \]
  • first element? last element? middle element? any one element could be bad.
  • pick 3 random elts, take the median one. not perfect but ok in practice.
Shellsort

idea: a multi-pass generalization of insertion sort
on each pass with some gap g, perform g insertion sorts on elts g apart
sequence of gaps is given to you, always ends with gap 1 (insertion sort)
idea is that successive sorts are more and more sorted... and insertion
sort behaves really well on almost-sorted lists

A nifty idea. Depending on choice of gaps, it can improve things. But the
analysis is quite tricky.

Worst-case running time: $O(n^2)$ for bad gaps, $O(n^{(4/3)})$ for good gaps.
Best-case running time: $O(n)$ if everything is sorted and make $O(1)$ passes.
This can be done in-place, like insertion sort.
Theorem: every (comparison-based) algorithm for sorting is at least $n \log n$.

Proof: Sorting can be thought of as determining one permutation of its input. (The values themselves don’t matter, just what their relative order is.) Since the input can be in any order, the number of possible inputs is the number of permutations on $n$ elements, of which there are $n!$. We model the decisions of the algorithm with a binary tree. At the root of our tree, we have all possible permutations, because we haven’t made any comparisons and have no information as to which permutation is the real sorted one.

After each comparison of some element $a$ to some element $b$, we either know we are in a permutation where $a$ is before $b$ (because $a < b$), or $b$ is before $a$ (because $b \leq a$). That is, each comparison allows us to branch into one of two groups, each of half the size. We are narrowing down our search space by a half. The algorithm ends when our group is of size 1.
Theorem: every (comparison-based) algorithm for sorting is at least $n \log n$.

Proof: So, we start with $n!$ possible orderings / permutations. With each comparison, we divide the number of possibilities in two. The algorithm ends when we are down to one possibility. The number of comparisons is thus the number of times we can divide $n!$ by two before we get down to 1. I.e. we need at least $\log(n!)$ comparisons. Using Stirling’s approximation (don’t worry ‘bout it), we know that $\log(n!)$ is $(n \log n)$. 