CSCI 311

CSUC – Algorithms & Data Structures
04.01 Sorting

Today

- Sorting

Sorting

```
function isSorted(L)
  for i from 1 to length(L)-1
    if L[i-1] > L[i]
      return false
  return true
```

Run time?

Sorting

- Why does sorting matter?
- What does it mean for something to be sorted?

```
function isSorted(L)
   if length(L) <= 1
     return true
   if L[0] > L[1]
          return false
   return isSorted(rest(L))
```

Kinds of Sorts

- In-place
  - Use less than $\theta(n)$ extra space
  - The original ordering is overwritten
  - A sort algorithm in which the sorted items occupy (almost) the same storage as the original ones.

The algorithm's spatial (memory) requirement must be $<= \theta(n)$

Kinds of Sorts

- Stable vs unstable
  - Relative order is preserved when there are ties or repeats in a stable sort

$$[3, 2, {\color{red}5}, 1, {\color{blue}5}, 4] \Rightarrow [1, 2, 3, 4, {\color{red}5}, {\color{blue}5}]$$

$$[3, 2, {\color{red}5}, 1, {\color{blue}5}, 4] \Rightarrow [1, 2, 3, 4, {\color{blue}5}, {\color{red}5}]$$

Kinds of Sorts

- Comparison
  - Does the sort use a comparison operator like $>$?
  - Radix sort, for example, does not (Non-Comparison Based)

Shuffle Sort

```
Function shuffleSort(L)
  while not isSorted(L)
     L = shuffle(L)
  return L
```

- The asymptotic run time of this function is something like $O((2n)!)$
- Impractical for most inputs

Bubble Sort

```
function bubbleSort(L)
  sorted = false
  while not sorted
    sorted = true
    for i from 1 to length(L)-1
      if L[i-1] > L[i]
        temp = L[i-1]
        L[i-1] = L[i]
        L[i] = temp
        sorted = false
  return L
```

[2, 5, 4, 3, 1]

best case run time?

worth case run time?

Bubble Sort (bad)

```
function bubbleSort(L)
  sorted = false
  while not sorted
    sorted = true
    for i from 1 to length(L)-1
      if L[i-1] > L[i]
        temp = L[i-1]
        L[i-1] = L[i]
        L[i] = temp
        sorted = false
  return L
```

- On what kinds of lists is this slow? What is the run time?
- On what kinds of lists is this fast? What is the run time?
- How well might we expect this to perform on a random list?

(Let us use only Big-Oh)

ALGORITHM	BEST CASE	AVERAGE CASE (Random Inputs)	WORST CASE
Bubble Sort	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)

Insertion Sort

```
function insertionSort(L)
  i = 1
  while i < length(L)
   j = i
    while j > 0 and L[j] < L[j-1]
      temp = L[j-1]
      L[j-1] = L[j]
      L[j] = temp
      j = j - 1
    i = i + 1
  return L
```

[2, 5, 4, 3, 1]

best case run time?

worth case run time?

Insertion Sort

```
function insertionSort(L)
  i = 1
  while i < length(L)
   j = i
    while j > 0 and L[j] < L[j-1]
      temp = L[j-1]
      L[j-1] = L[j]
      L[j] = temp
      j = j - 1
    i = i + 1
  return L
```

- On what kinds of lists is this slow? What is the run time?
- On what kinds of lists is this fast? What is the run time?
- How well might we expect this to perform on a random list?

ALGORITHM	BEST CASE	AVERAGE CASE (Random Inputs)	WORST CASE
Bubble Sort	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)
Insertion Sort	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)

Selection Sort

```
function selectionSort(L)
  for i from 0 to length(L)-2
    uMin = i
    for j from i+1 to length(L)-1
      if L[j] < L[uMin]
        uMin = j
    temp = L[i]
    L[i] = L[uMin]
    L[uMin] = temp return L
```

[2, 5, 4, 3, 1]

best case run time?

worth case run time?

Selection Sort

```
function selectionSort(L)
  for i from 0 to length(L)-2
    uMin = i
    for j from i+1 to length(L)-1
      if L[j] < L[uMin]
        uMin = j
    temp = L[i]
    L[i] = L[uMin]
    L[uMin] = temp return L
```

- On what kinds of lists is this slow? What is the run time?
- On what kinds of lists is this fast? What is the run time?
- How well might we expect this to perform on a random list?

ALGORITHM	BEST CASE	AVERAGE CASE (Random Inputs)	WORST CASE
Bubble Sort (bad)	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)
Insertion Sort	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)
Selection Sort	O(n^2) (sorted, you still get less memory accesses by data movements)	O(n^2)	O(n^2) (special, like 2,4,5,3,1)

Quicksort

```
function quicksort(L)
  if length(L) <= 1
    return
  Lpivot = L[0]
  A = []
  B = []
  for e in rest(L) if e <= pivot
      append(A, e)
    else
      append(B, e)
  return quicksort(A) + [pivot] + quicksort(B)
```

Come up with an intuitive description of quicksort (data movement must use least number of steps)

Show how quicksort works on the list $[8, 6, 3, 10, 9, 1, 14, 15, 7]$

What is the asymptotic run time of quicksort? (best case vs worst case)

Quicksort

- On what kinds of lists is this slow? What is the run time?
- On what kinds of lists is this fast? What is the run time?
- How well might we expect this to perform on a random list?

ALGORITHM	BEST CASE	AVERAGE CASE (Random Inputs)	WORST CASE
Bubble Sort (bad)	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)
Insertion Sort	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)
Selection Sort	O(n^2) (sorted, you still get less memory accesses by data movements)	O(n^2)	O(n^2) (special, like 2,4,5,3,1)
Quick Sort	O(nlog(n)) (every time the pivot value cut the original array into two almost equal sized subarray)	O(n log(n))	O(n^2) (the pivot value cut the array into two arrays, and one of them is of size 1, and the other is of size n-1)

Merge Sort

```
function mergeSort(L)
  if length(L) <= 1
    return L
  A = mergeSort(L[:length(L)/2])
  B = mergeSort(L[length(L)/2:]) return
  merge(A, B)
```

[8, 6, 3, 10, 9, 1, 14, 15, 7]

Merge Sort

```
function merge(A, B)
  C = {}
  while length(A)>0 and length(B)>0
    if A[0] <= B[0]
      append(C, A[0])
      A = rest(A)
    else
      append(C, B[0])
      B = rest(B)
  if length(A)>0
    C = C + A
  else if length(B)>0
    C = C + B
  return C
```

Merge Sort

- On what kinds of lists is this slow? What is the run time?
- On what kinds of lists is this fast? What is the run time?
- How well might we expect this to perform on a random list?

ALGORITHM	BEST CASE	AVERAGE CASE (Random Inputs)	WORST CASE
Bubble Sort(bad)	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)
Insertion Sort	O(n) (sorted)	O(n^2)	O(n^2) (reversely sorted)
Selection Sort	O(n^2) (sorted, you still get less memory accesses by data movements, still worse than bubble sort)	O(n^2)	O(n^2) (special, like 2,4,5,3,1)
Quick Sort	O(nlog(n))(every time the pivot value cut the original array into two almost equal sized subarray)	O(n log(n))	O(n^2) (the pivot value cut the array into two arrays, and one of them is of size 1, and the other is of size n-1)
Merge Sort	O(nlog(n)) (sorted, you still get less memory accesses by data movements when merging)	O(n log(n))	O(n log(n)) (reversely sorted)

The Sorts

Stable Sorts: Bubble Sort, Insertion Sort, Merge Sort
Unstable Sorts: Selection Sort and Quick Sort

Can prove it yourself as a practice and for fun! ☺