CSC148H Week 10

Sadia Sharmin

Selection Sort: (selection_sort.py)

```
def find_min(L, i):
  '''(list, int) -> int
  Return the index of the smallest item in L[i:].
  '''
  smallest_index = i
  for j in range(i + 1, len(L)):
      if L[j] < L[smallest_index]:
        smallest_index = j
  return smallest_index

def selection_sort(L):
  '''(list) -> NoneType
  Sort the elements of L in non-descending order.
  '''
  for i in range(len(L) - 1):
    smallest_index = find_min(L, i)
    L[smallest_index], L[i] = L[i], L[smallest_index]
```

Insertion Sort: (insertion_sort.py)

```
def insert(L, i):
  '''(list, int) -> NoneType
  Move L[i] to where it belongs in L[:i].
  '''
  v = L[i]
  while i > 0 and L[i - 1] > v:
    L[i] = L[i - 1]
    i -= 1
  L[i] = v

def insertion_sort(L):
  '''(list) -> NoneType
  Sort the elements of L in non-descending order.
  '''
  for i in range(1, len(L)):
    insert(L, i)
```

The Slow Sorts

- Selection sort and insertion sort are both $O(n^2)$
  - They have an outer loop that runs n times
  - On each iteration, a function is called that takes at most n steps
- We'll discuss a much faster recursive sorting method called quicksort
- Interestingly, the worst-case running time of quicksort is still $O(n^2)$, though on average it is $O(n \lg n)$

Properties of Quicksort

- Unlike the iterative sorts, it is not an in-place sorting method
  - We use at least $\lg n$ additional stack space to carry out the recursion
- The algorithm works by choosing a pivot element, and partitioning the list so that elements smaller than the pivot are to its left and elements bigger than the pivot are to its right
- If we could then sort these two sublists, the original list would be entirely sorted
- We sort these sublists recursively

Partition Procedure

- We will write a function to partition list `lst[left..right]` around pivot `pivot`
- As we proceed through the list, we will maintain three consecutive slices
  - Elements $<$ `pivot`
  - Elements $\geq$ `pivot`
  - Unprocessed elements

Partition Procedure...

- We're going to keep indices `i` and `j`
- Stuff to the left of `i` is less than the pivot
- Stuff from `i` up to but not including `j` is greater or equal to the pivot
- Stuff from `j` to the right is unprocessed

Partition Code (partition.py)

```
def partition(lst, left, right, pivot):
  '''(list, int, int, int) -> int

  Rearrange lst so that elements >= pivot follow
  elements < pivot; return index of first element >= pivot
  '''
  i = left
  j = left
  while j <= right:
    if lst[j] < pivot:
      lst[i], lst[j] = lst[j], lst[i]
      i += 1
    j += 1
  return i
```

Partition Practice

`[6, 2, 12, 6, 10, 15, 2, 13]`

Partition this list around pivot 5.

Use the code on the previous slide:

- Start `i` and `j` at 0
- If `lst[j] >= 5`, increment `j`
- Otherwise, swap `lst[i]` and `lst[j]`, and increment both `i` and `j`

Quicksort Procedure

- As a choice for the pivot, we will choose the rightmost element in the list being sorted on each recursive call
- Once we get the pivot value, we might try to
  - Partition the list around this pivot value
  - Recursively sort the elements less than the pivot
  - Recursively sort the elements greater than or equal to the pivot
- ... will this work?

Quicksort Attempt 1

```
from partition import partition

def quicksort(lst, left, right):
  '''(list, int, int) -> NoneType
  Sort lst[left..right] in nondecreasing order.
  '''
  if left < right:
    pivot = lst[right]
    i = partition(lst, left, right - 1, pivot)
    quicksort(lst, left, i - 1)
    quicksort(lst, i, right)
```

Quicksort Attempt 2

```
from partition import partition

def quicksort(lst, left, right):
  '''(list, int, int) -> NoneType
  Sort lst[left..right] in nondecreasing order.
  '''
  if left < right:
    pivot = lst[right]
    i = partition(lst, left, right, pivot)
    quicksort(lst, left, i - 1)
    quicksort(lst, i + 1, right)
```

What Went Wrong?

- In attempt 1, we had no guarantee that our subproblem was getting smaller
- In attempt 2, we had no guarantee that the pivot was in the proper place before excluding it from the recursion
- We can modify attempt 2 to swap the pivot value into its correct place

Correct Quicksort (quicksort.py)

```
from partition import partition

def quicksort(lst, left, right):
  '''(list, int, int) -> NoneType
  Sort lst[left..right] in nondecreasing order.
  '''
  if left < right:
    pivot = lst[right]
    i = partition(lst, left, right - 1, pivot)
    lst[i], lst[right] = lst[right], lst[i]
    quicksort(lst, left, i - 1)
    quicksort(lst, i + 1, right)
```

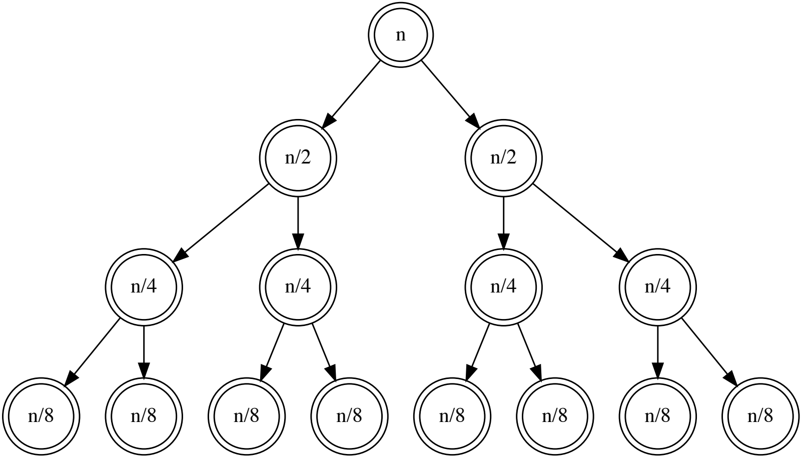
Quicksort Example Execution

- Consider list: $[12, 30, 25, 8, 4, 9, 15, 13]$
- We partition $[12, 30, 25, 8, 4, 9, 15]$ around pivot 13
- This gives $[12, 8, 4, 9, 25, 30, 15, 13]$
- Then swap 13 with 25 to get $[12, 8, 4, 9, 13, 30, 15, 25]$
- . . . and then recursively sort $[12, 8, 4, 9]$ and $[30, 15, 25]$

Best Case for Quicksort

- Assume that we choose a pivot that partitions the list exactly in half on each recursive call
- From binary search, we know that we will recurse on the order of $\lg n$ times
- On the outermost level, the partition step takes n time (one step per element in the list)
- On the two resulting subproblems of size $n/2$, the total partition time is $n/2 + n/2 = n$
- On the resulting four subproblems of size $n/4$ (two from each of the two $n/2$ subproblems), the total partitioning time is still n, and so on
- On each of $\lg n$ levels of recursion, we do a total of n work
- Thus, our running time is $O(n \lg n)$

Best Case for Quicksort: Tree



Worst Case for Quicksort

- It's always possible that we choose a pivot that partitions the list badly
- Consider: we pass a sorted list to quicksort and choose the rightmost element as the pivot
- On each recursive call operating on a list of size n, we may partition it into a list of $n - 1$ elements and a "list" of just 1 element
- Now, we have n levels of recursion, each of whose total partitioning time still takes n time
- We have an $O(n^2)$ algorithm? Did we just waste a lot of time discussing this?

Worst Case for Quicksort...

- Instead of choosing the rightmost element for the pivot, we could choose the middle element
- This fixes the above case, but we can still construct a list to exhibit quicksort's worst-case behavior
- Another popular approach is choosing the median element among the first, middle, and last elements
- Even if there are inputs that will take $O(n^2)$ time, this is rare in practice
- Consider: if we partition so that 90 percent of the elements are in one list and 10 percent are in the other, quicksort is still $O(n \lg n)$
  - The depth of recursion changes to $\log_{10/9} n$, but it is still logarithmic

Merge Sort

- Merge sort is always $O(n \lg n)$, even in the worst case
- It works by
  - Recursively sorting the first half of the list
  - Recursively sorting the second half of the list
  - Merging the two halves into a newly sorted list
- At each level of recursion, the merging takes a total of n steps, and there are $\lg n$ levels before we get to the base case
- The merging requires an auxiliary list (not required in quicksort)

Merging Two Sorted Lists

```
def merge(list1: list, list2: list):
  '''Return merge of sorted list1 and list2.'''
  lst = []
  i = 0
  j = 0
  while i < len(list1) and j < len(list2):
    if list1[i] < list2[j]:
      lst.append(list1[i])
      i = i + 1
    else:
      lst.append(list2[j])
      j = j + 1

  lst.extend(list1[i:])
  lst.extend(list2[j:])

  return lst
```

Mergesort

```
from merge import merge

def mergesort(lst: list) -> list:
  '''Return a sorted copy of lst.'''
  if len(lst) <= 1:
    return lst[:]
  mid = len(lst) // 2
  left = lst[:mid]
  right = lst[mid:]
  left_s = mergesort(left)
  right_s = mergesort(right)
  return merge(left_s, right_s)
```