CSC148 Week 11

Larry Zhang

Sorting Algorithms

Selection Sort

```
def selection_sort(lst):
    for each index i in the list lst:
        swap element at index i with the smallest element to the right of i
```

Image source: https://medium.com/@notestomyself/how-to-implement-selection-sort-in-swift-c3c981c6c7b3

Selection Sort: Code

```
 1 def find_min(L, i):
 2     '''(list, int) -> int
 3     Return the index of the smallest item in L[i:].
 4     '''
 5     smallest_index = i
 6     for j in range(i + 1, len(L)):
 7         if L[j] < L[smallest_index]:
 8             smallest_index = j
 9     return smallest_index
10
11 def selection_sort(L):
12     '''(list) -> NoneType
13     Sort the elements of L in non-descending order.
14     '''
15     for i in range(len(L) - 1):
16         smallest_index = find_min(L, i)
17         L[smallest_index], L[i] = L[i], L[smallest_index]
```

Worst-case runtime:

$O(n^2)$

Insertion Sort

```
def insertion_sort(lst):
    for each index from 2 to the end of the list lst
        insert element with index i in the proper place in lst[0..i]
```

image source: http://piratelearner.com/en/C/course/computer-science/algorithms/trivial-sorting-algorithms-insertion-sort/29/

Insertion Sort: Code

```
 1 def insert(L, i):
 2     '''(list, int) -> NoneType
 3     Move L[i] to where it belongs in L[:i].
 4     '''
 5     v = L[i]
 6     while i > 0 and L[i - 1] > v:
 7         L[i] = L[i - 1]
 8         i = i - 1
 9     L[i] = v
10
11 def insertion_sort(L):
12     '''(list) -> NoneType
13     Sort the elements of L in non-descending order.
14     '''
15     for i in range(1, len(L)):
16         insert(L, i)
```

Worst-case runtime:

$O(n^2)$

Bubble Sort

swap adjacent elements if they are “out of order”

go through the list over and over again, keep swapping until all elements are sorted

Worst-case runtime:

$O(n^2)$

Image source: http://piratelearner.com/en/C/course/computer-science/algorithms/trivial-sorting-algorithms-bubble-sort/27/

Summary

$O(n^2)$ sorting algorithms are considered slow.

We can do better than this, like O(n log n).

We will discuss a recursive fast sorting algorithms is called **Quicksort**.

- Its worst-case runtime is still $O(n^2)$
- But “on average”, it is O(n log n)

Quicksort

Background

Invented by **Tony Hoare** in 1960

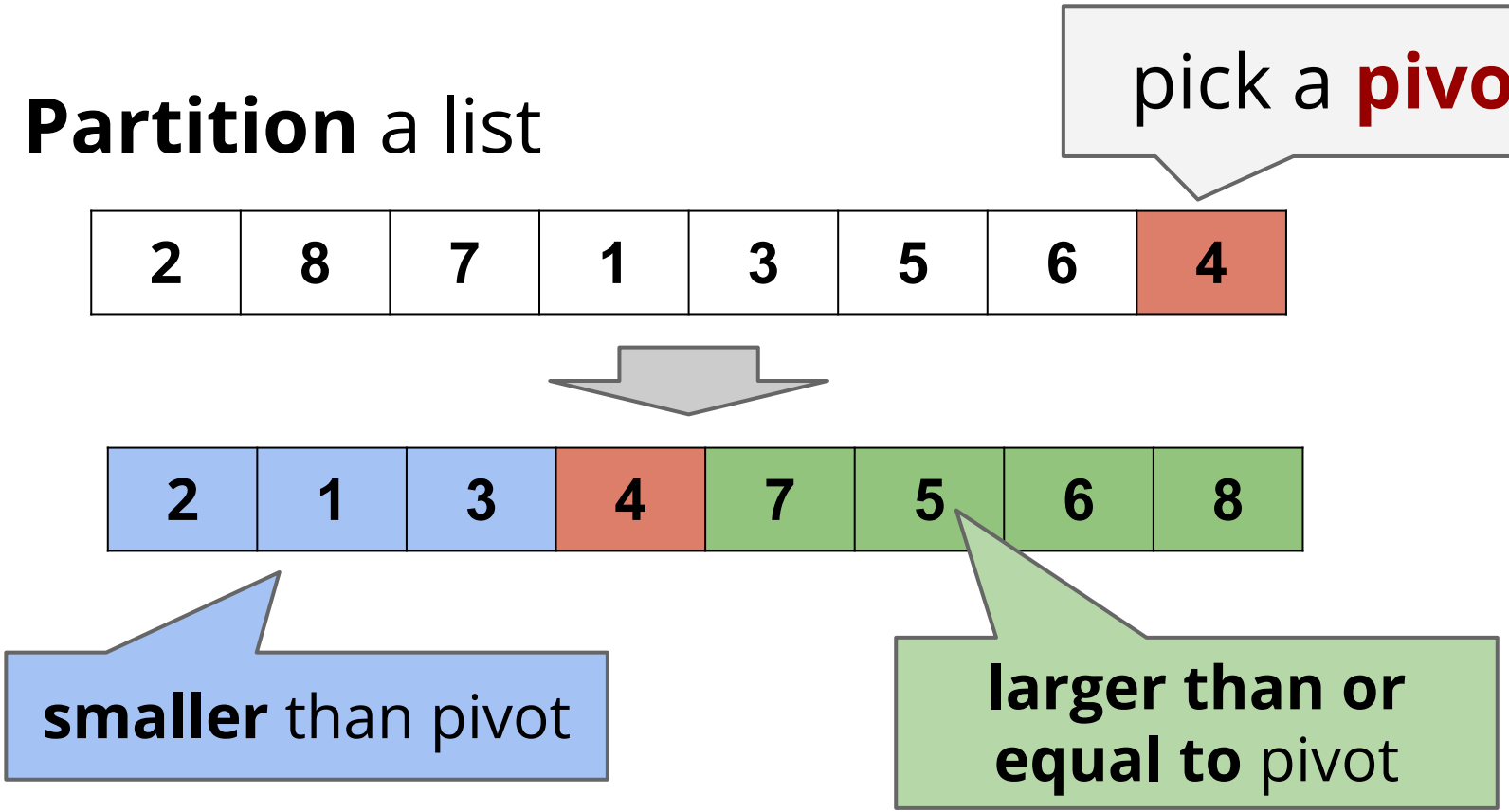
Very commonly used sorting algorithm. When **implemented well**, can be about 2-3 times faster than **merge sort** and **heapsort.**

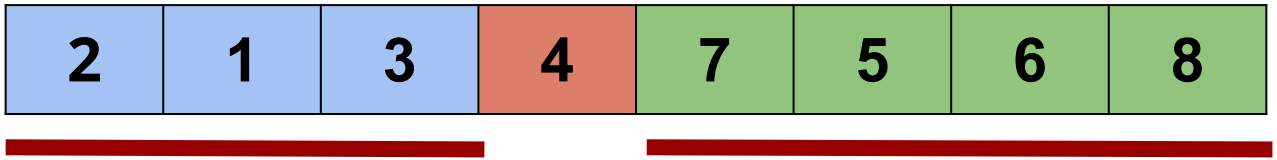
Invented **NULL reference** in 1965. Apologized for it in 2009

http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Quicksort: the idea

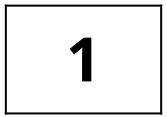
➔ **Partition** a list





Recursively partition the sub-lists **before** and **after** the pivot.

Base case:



sorted

Partition

```
def partition(lst, left, right, pivot)
    '''(list, int, int, int) -> int
       Rearrange lst[left:right+1] so that elements >= pivot
       come after elements < pivot;
       return index of first element >= pivot
    '''
```

```
After partition([6, 2, 12, 6, 10, 15, 2, 13], 0, 8, 5)
lst becomes something like [2, 2, 6, 12, 6, 10, 15, 13]
return value is 2
```

Partition

```
1 def partition(lst, left, right, pivot):
2     '''(list, int, int, int) -> int
3
4     Rearrange lst[left:right+1] so that elements >= pivot
5     come after elements < pivot;
6     return index of first element >= pivot
7     '''
8     i = left
9     j = left
10    while j <= right:
11        if lst[j] < pivot:
12            lst[i], lst[j] = lst[j], lst[i]
13            i += 1
14        j += 1
15    return i
```

Trace:

```
partition([6, 2, 12, 6, 10, 15, 2, 13], 0, 8, 5)
```

See partition.py

Quicksort Procedure

Quicksort(lst, left, right) would sort the segment lst[left:right+1]

1. pick lst[right] as the pivot
2. Partition the list around the pivot
3. Recursive Quicksort the partitions on the left and right of pivot.

Things to be careful about:

1. Is the subproblem getting smaller?
2. Are the subproblems containing the correct values?

Quicksort Attempt #1


```
 1 from partition import partition
 2
 3 def quicksort(lst, left, right):
 4     '''(list, int, int) -> NoneType
 5     Sort lst[left:right+1] in nondecreasing order.
 6     '''
 7     if left < right:
 8         pivot = lst[right]
 9         i = partition(lst, left, right - 1, pivot)
10         quicksort(lst, left, i - 1)
11         quicksort(lst, i, right)
```

Recursive call not guaranteed to run on a smaller subproblem, e.g., i=0.
Pivot should be excluded.

Quicksort Attempt #2

```
 1 from partition import partition
 2
 3 def quicksort(lst, left, right):
 4     '''(list, int, int) -> NoneType
 5     Sort lst[left:right+1] in nondecreasing order.
 6     '''
 7     if left < right:
 8         pivot = lst[right]
 9         i = partition(lst, left, right, pivot)
10         quicksort(lst, left, i - 1)
11         quicksort(lst, i + 1, right)
```

We are excluding the element at index i, this element is supposed to be the pivot, but it may not be true here. How to fix?

Quicksort Correct Attempt

```
 1 from partition import partition
 2
 3 def quicksort(lst, left, right):
 4     '''(list, int, int) -> NoneType
 5     Sort lst[left:right+1] in nondecreasing order.
 6     '''
 7     if left < right:
 8         pivot = lst[right]
 9         i = partition(lst, left, right - 1, pivot)
10         lst[i], lst[right] = lst[right], lst[i]
11         quicksort(lst, left, i - 1)
12         quicksort(lst, i + 1, right)
```

pivot will be at index "right" after the partition.

This swap moves the pivot to index i

Exclude index i (the pivot) when making recursive call. Subproblem guaranteed to be smaller!

DEMO

Trace an example in quicksort.py

Compare the runtimes: time_sorting.py

Think

What kind of input list is **“good”** for Quicksort?

What kind of input list is **“bad”** for Quicksort?

Runtime for Partition

```
1 def partition(lst, left, right, pivot):
2     '''(list, int, int, int) -> int
3
4     Rearrange lst[left:right+1] so that elements >= pivot
5     come after elements < pivot;
6     return index of first element >= pivot
7     '''
8     i = left
9     j = left
10    while j <= right:
11        if lst[j] < pivot:
12            lst[i], lst[j] = lst[j], lst[i]
13            i += 1
14        j += 1
15    return i
```

O(n)

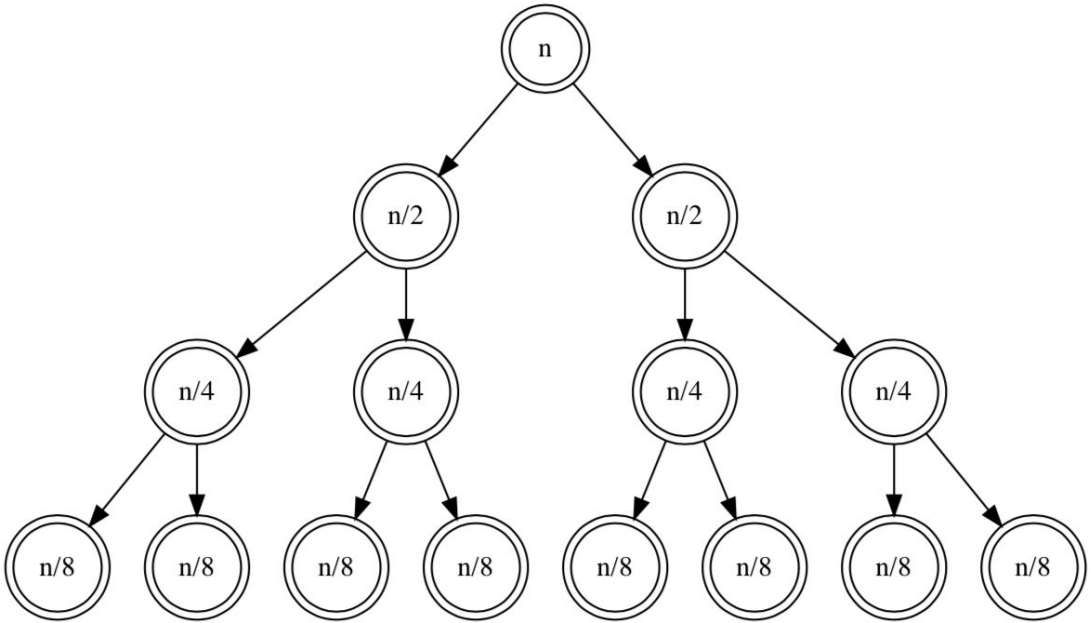
more precisely,

`right - left`, i.e., the length of the range to sort

(the value of j goes from left to right)

Best-case runtime of Quicksort

If every time, the pivot happens to be the **median** of the range to sort.



Total amount of work at each level

- n

How many levels?

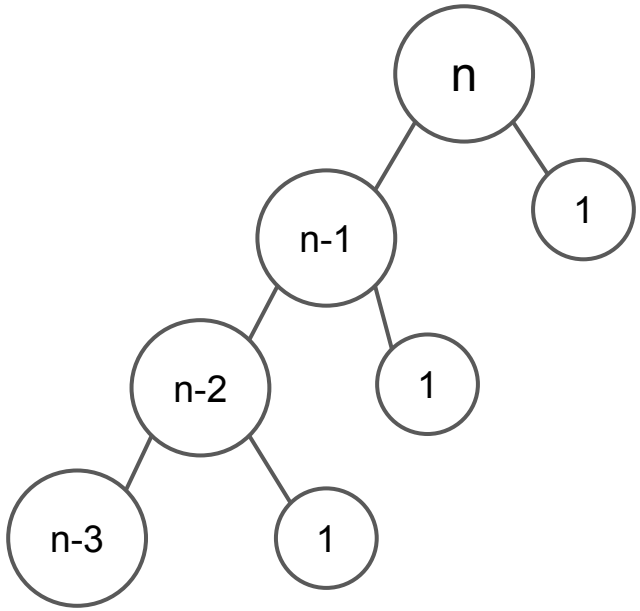
- log(n)

Total runtime is

- O(n log(n))

Worst-case runtime of Quicksort

If every time, the pivot is the **maximum/minimum** of the range to sort



Total amount of work at each level

- n, n-1, n-2, n-3, …, 1

How many levels?

- n

Total runtime is

- 1+2+3+4+...+n
- $O(n^2)$

Example of a worst-case input?

- [0,1,2,3,4,5,6,7,8]
- an already-sorted list!

Summary of Quicksort

Quicksort isn't always "quick", theoretically.

- In the worst case, it's still quite slow ($O(n^2)$)

However, in practice, Quicksort performs very well.

- The worst case is rare, and on average the performance is good (O(n logn)).

What if someone maliciously keep passing the worst-case input to Quicksort?

- We could shuffle the input before calling Quicksort; or randomly pick the pivot rather than picking a fixed position.
- This "randomization" guarantees the average-case performance. It is also a major topic in algorithm design and analysis.

Think

What if …

the pivot is not the median (dividing the range into half-half),

but divides the range by 90%-10% ratio.

What's the big-Oh runtime for this case?

- The depth of recursion would be log base 10/9 rather base 2
- Overall, still O(n log n)

Merge Sort

recursive, worst-case O(n logn)

Merge Sort

It is a recursive algorithm that works as follows:

- divide the input list into two halves
- recursively sort left half and the right half, then we obtain two sorted lists for the two halves.
- **merge these two sorted lists into one big sorted list**, that's the answer
- Base case: size of the input list 1, already sorted.

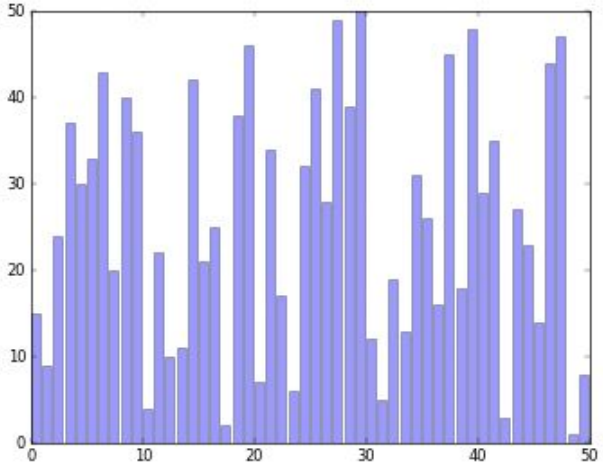
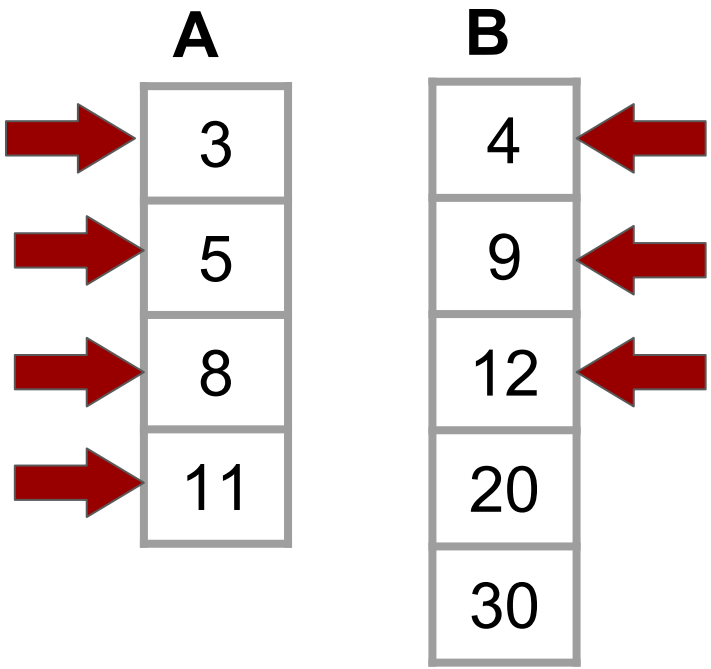


Image source: https://github.com/heray1990/AlgorithmVisualization

Merge two sorted lists



The procedure

- compare the two pointed by arrows
- add the smaller one to the output
- advance the arrow of the smaller one
- When reaching the end of one list, append everything left in the other

Worst-case runtime:
O(len(A) + len(B))
or O(n)

Merge two sorted lists

```
1 def merge(list1: list, list2: list) -> list:
2     '''Return merge of sorted list1 and list2.'''
3     lst = []
4     i = 0
5     j = 0
6     while i < len(list1) and j < len(list2):
7         if list1[i] < list2[j]:
8             lst.append(list1[i])
9             i = i + 1
10        else:
11            lst.append(list2[j])
12            j = j + 1
13
14    lst.extend(list1[i:])
15    lst.extend(list2[j:])
16
17    return lst
```

merge.py

Merge Sort (returning a sorted copy of lst)

```
 1 from merge import merge
 2
 3 def mergesort(lst: list) -> list:
 4     '''Return a sorted copy of lst.'''
 5     if len(lst) <= 1:
 6         return lst[:]
 7     mid = len(lst) // 2
 8     left = lst[:mid]
 9     right = lst[mid:]
10     left_s = mergesort(left)
11     right_s = mergesort(right)
12     return merge(left_s, right_s)
```

mergesort.py

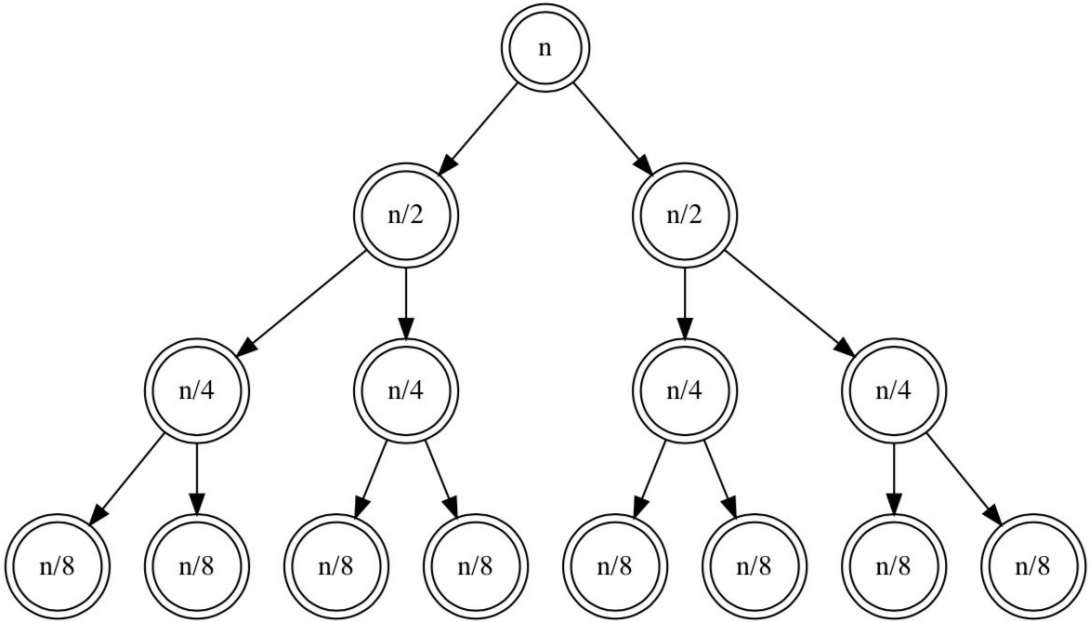
Merge Sort V2 (in-place mutating lst)

```
1 def merge(L, i, mid, j):
2     """Merge the two sorted halves L[i:mid + 1] and L[mid + 1:j + 1] and return
3     them in a new list. Notice that L[mid] belongs in the left half and L[j]
4     belongs in the right half -- the indices are inclusive."""
5
6     result = []
7     left = i
8     right = mid + 1
9
10    while left <= mid and right <= j:
11        if L[left] < L[right]:
12            result.append(L[left])
13            left += 1
14        else:
15            result.append(L[right])
16            right += 1
17
18    return result + L[left:mid + 1] + L[right:j + 1]
19
20 def _mergesort(L, i, j):
21     """Sort the items in L[i] through L[j] in non-decreasing order."""
22
23     if i < j:
24         mid = (i + j) // 2
25         _mergesort(L, i, mid)
26         _mergesort(L, mid + 1, j)
27         L[i:j + 1] = merge(L, i, mid, j)
28
29 def mergesort(L):
30     _mergesort(L, 0, len(L) - 1)
```

mergesort2.py

Runtime of Merge Sort

Every recursive call is working on the subproblem of half the size



Total amount of work at each level

- n

How many levels?

- log(n)

Total runtime is

- O(n log(n))

This is true even in the worst case!

Summary of Merge Sort

Merge Sort is a typical “**divide-and-conquer**” algorithm.

You will learn more about this algorithm design technique in CSC236.

Bogosort

just for fun

Bogosort code

```
 1 import random
 2
 3 def isSorted(data):
 4     for i in range(len(data)-1):
 5         if data[i] > data[i+1]:
 6             return False
 7     else:
 8         return True
 9
10 def bogosort(data):
11     while not isSorted(data):
12         random.shuffle(data)
```

What's the best-case runtime?

What's the worst-case runtime?

What's the average-case runtime?

Bogosort is also known as

- Permutation sort
- Slow Sort
- Stupid Sort
- Shotgun sort
- Monkey sort