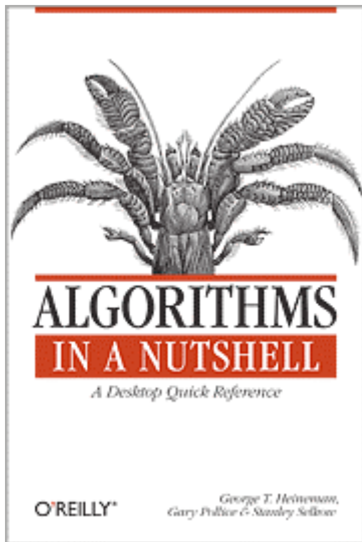
Algorithms in a Nutshell



Session 2

Sorting

9:40 – 10:30

Outline

- Sorting Principles
- Themes
  - Divide and Conquer
  - Space vs. Time
  - Arrays vs. Pointers
  - Comparison vs. non-comparison
- Algorithms
  - QUICKSORT, HEAPSORT, BUCKET SORT
- Domains
  - Integers, Strings, Complex Records

Sorting Principle: Comparison

- Comparing elements e_1 and e_2 only one of the following is true
  1. $e_1 < e_2$
  2. $e_1 = e_2$
  3. $e_1 > e_2$

32-bit int comparison: O(1) constant time operation

n-byte String comparison: O(n)

- Operation <u>may be costly</u> depending upon representation
  - Sort molecules by number of carbon atoms
  - Compare CH_3COCH_2Br with C_2H_8

Sorting Principle: Swapping

- Swap location of two elements
  - Fundamental operation
  - Assumes random access to any individual element

```
tmp = ar[i]
ar[i] = ar[j]
ar[j] = tmp
```

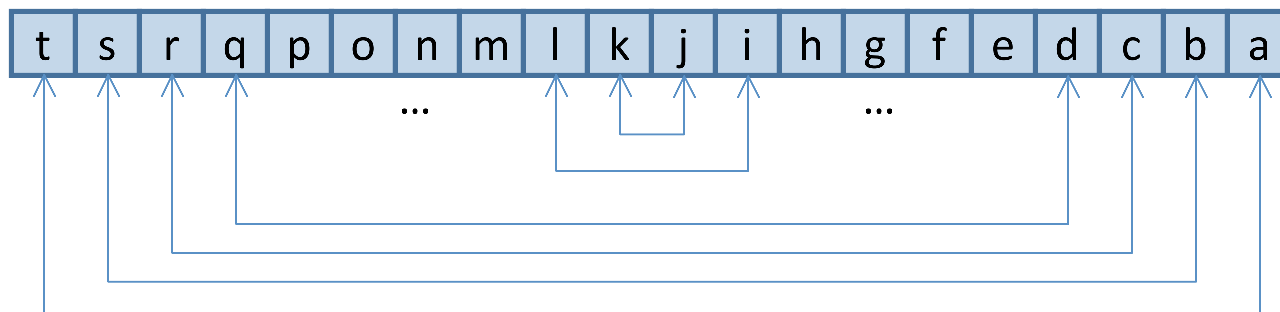
- Shift two or more elements
  - Suitable for arrays

```
void *memmove(&dest, &src, n)
```

- Swapping is often the dominant cost of sorting
  - Algorithms seek to reduce wasted swaps

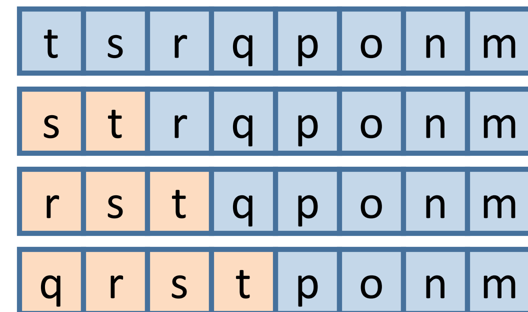
Swapping Example

- INSERTION SORT Worst Case



Only 10 swaps are really needed!

- Every element swapped maximum # of times
  - n(n-1)/2 = 19*20/2 = 190
  - O(n^2) number of swaps
- Can we avoid such situations?



Divide and Conquer

- Common computer science technique
- Break up a problem into smaller parts
  - Solve each independently

INSERTION SORT

t	s	r	q	p	o	n	m
s	t	r	q	p	o	n	m
r	s	t	q	p	o	n	m
q	r	s	t	p	o	n	m
p	q	r	s	t	o	n	m
o	p	q	r	s	t	n	m
n	o	p	q	r	s	t	m
m	n	o	p	q	r	s	t

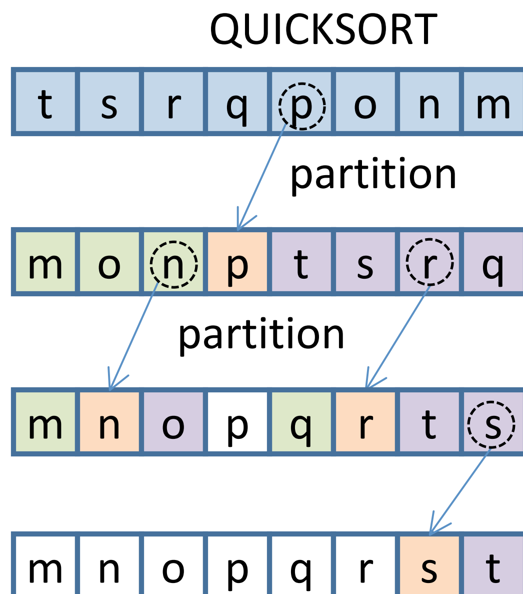
Note how each successive pass through INSERTION SORT actually solves larger problems

Not much dividing!

- Makes n–1 iterations

Divide and Conquer

- Common computer science technique
- Break up a problem into smaller parts



Note how each successive pass through QUICKSORT divides a problem into two problems that are about half as big

Solve each sub-problem, recursively

- Makes $\log_2(n)$ iterations

Recursion: An Aside

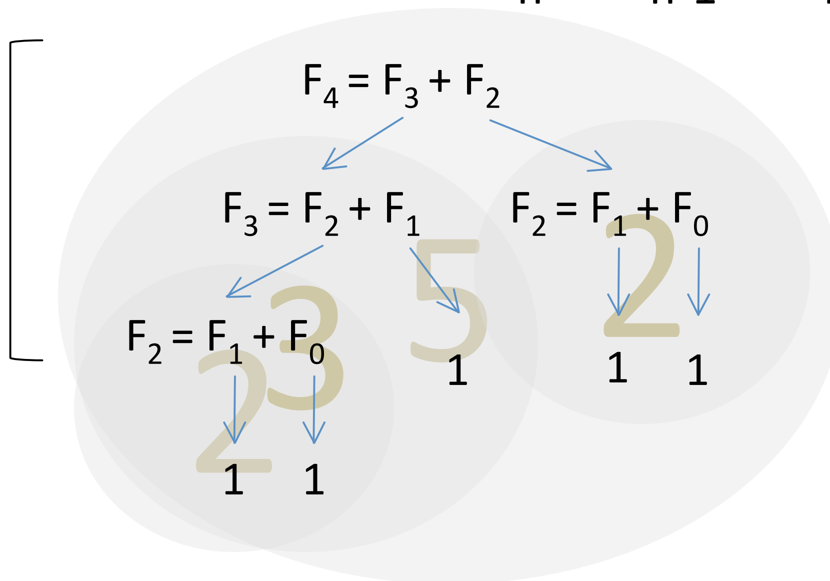
- Define a solution to a problem using that same solution as a sub-step
- Common examples
  - Fibonacci Series: $F_n = F_{n-1} + F_{n-2}$ where

Base Cases

$F_0 = F_1 = 1$

How deep is the recursion?

n–2 levels



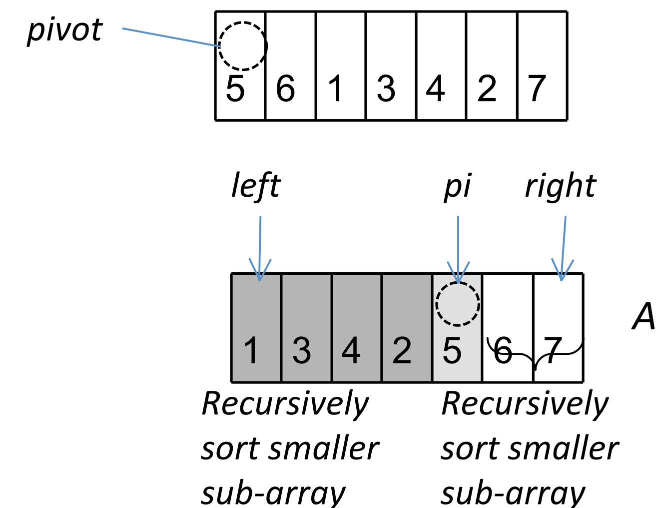

```
int fib(int n) {
  if (n == 0 || n == 1) {
    return 1;
  }

  return fib(n-1) + fib(n-2);
}
```

QUICKSORT

79

```
sort (A)
1.   quickSort (A, 0, n–1)
end

quickSort (A, left, right)
1.   if (left < right) then
2.     pi = partition (A, left, right)
3.     quickSort (A, left, pi–1)
4.     quickSort (A, pi+1, right)
end
```

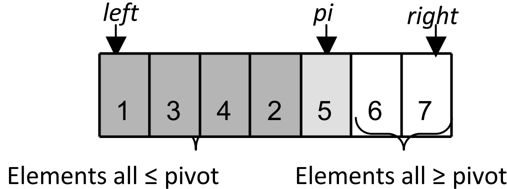
Best case	Average case	Worst case
O(n log n)	O(n log n)	$O(n^2)$



- Partition
  - selects an element to be *pivot*
  - divides array into left and right sub-arrays
- Recursion
  - Base Case: No need to sort sub-array that is either empty or has a single element: left ≥ right
  - How deep: log(*n*) on average, but worst-case *n*–1

QUICKSORT Fact Sheet

- Partition
  - selects an element to be *pivot*
  - divides array into left and right sub-arrays



QUICKSORT

Best case	Average case	Worst case
O(n log n)	O(n log n)	$O(n^2)$

Recursion | Divide and Conquer | Array

```
sort (A)
1.   quickSort (A, 0, n–1)
end

quickSort (A, left, right)
1.   if (left < right) then
2.      pi = partition (A, left, right)
3.      quickSort (A, left, pi–1)
4.      quickSort (A, pi+1, right)
end
```

pivot

A: 5 6 1 3 4 2 7

left, pi, right: 1 3 4 2 5 6 7

Recursively sort smaller sub-array

Recursively sort smaller sub-array

Base Case

No need to sort sub-array that is either empty or has a single element: left≥right

How deep is recursion?

Best case: log (n)

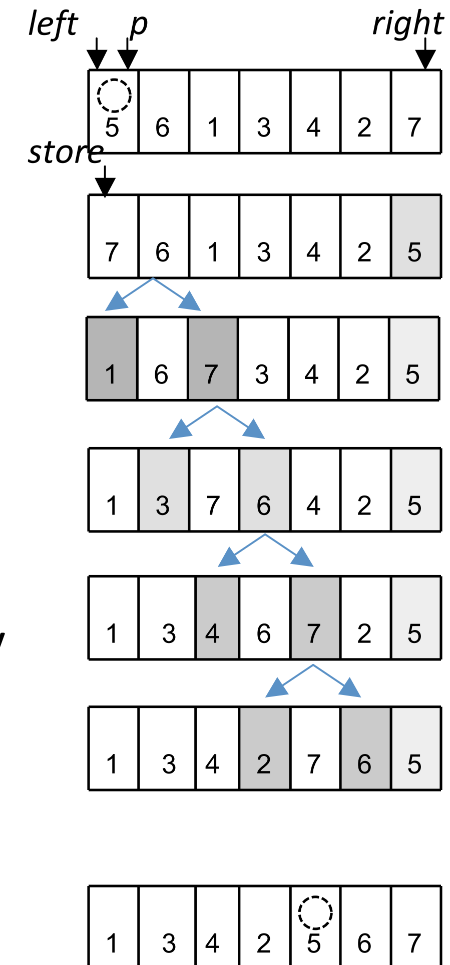
Worst case: n–1

partition (A, left, right)

```
1.  p = select pivot in A[left, right]
2.  swap A[p] and A[right]
3.  store = left
4.  for i = left to right–1 do
5.     if (A[i] ≤ A[right]) then
6.         swap A[i] and A[store]
7.         store++
8.  swap A[store] and A[right]
9.  return store
end
```

Partition

Best case	Average case	Worst case
O(n)	O(n)	O(n)



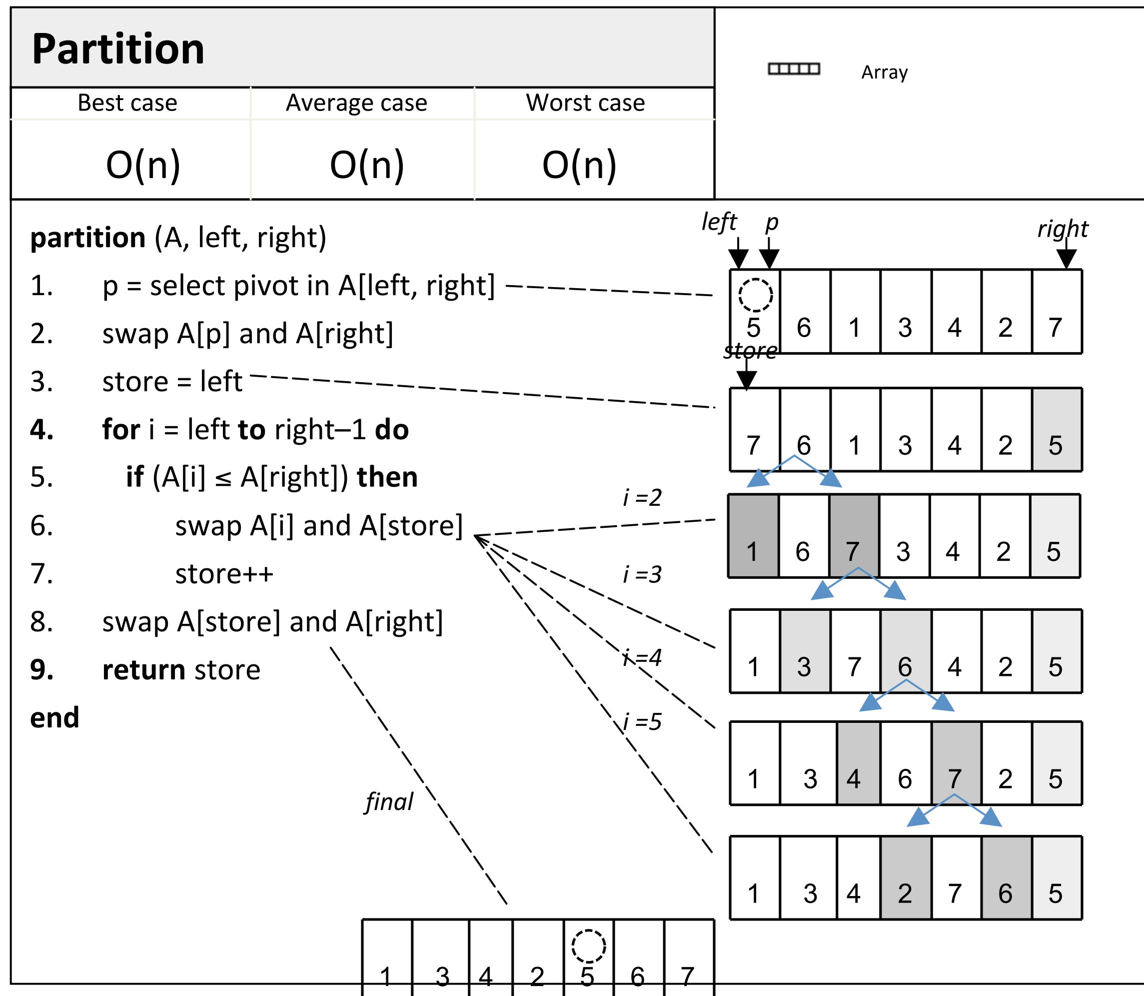
Select a "pivot" value

- Any value in array will do
- Best case is when the pivot value evenly splits the array

Scan left to right to find values less than pivot

- Swap values to ensure that all elements to the left of "pivot" are ≤ to its value

Partition Fact Sheet



Select a "pivot" value

- Any value will do
- Best case is when the pivot value evenly splits the array

Scan left to right to find values less than pivot

- Swap values to ensure that all elements to the left of "pivot" are ≤ to its value

Code Check

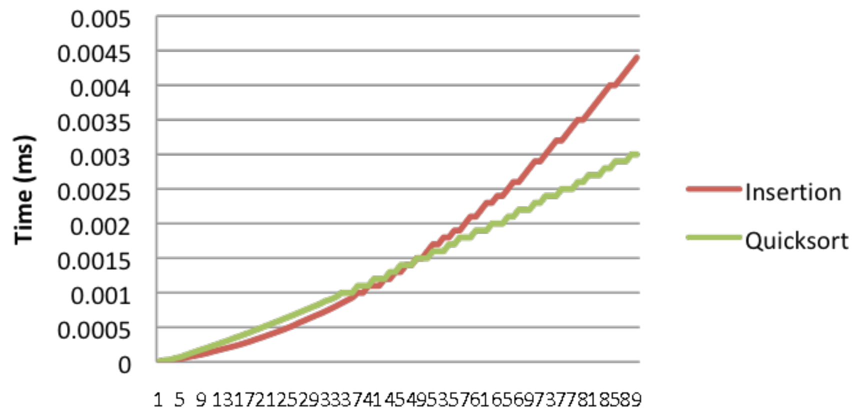
- Show actual running code
  - Handout
  - Debug example

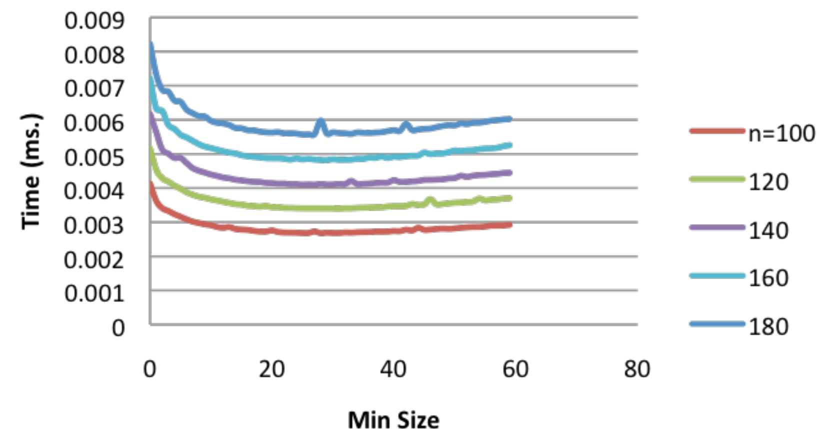
QUICKSORT Optimizations

- Performance, on average, will be O(*n log n*)
  - Can still secure some efficiencies
- Select Pivot
  - First or last
  - Random element
  - Median-of-k (select median of *k* elements)
- Use INSERTION SORT for small sub-arrays
  - Improves base case performance

INSERTION SORT vs. QUICKSORT

- INSERTION SORT outperforms on small arrays
- QUICKSORT benefits from using INSERTION SORT on small sub-arrays



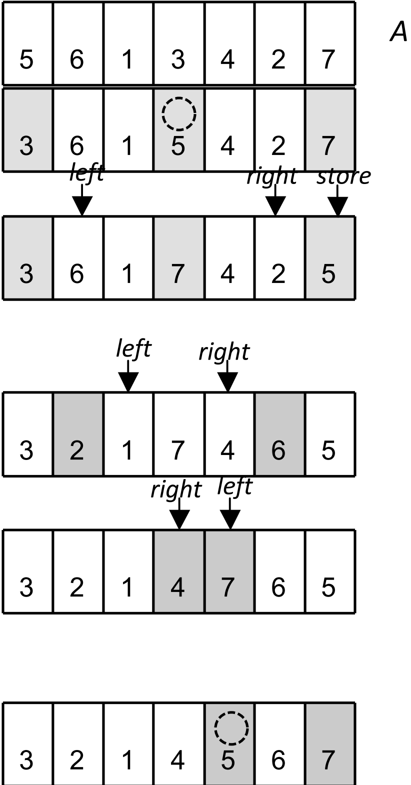


79

Partition Schemes

- Option P1: Shown earlier [p. 79]
- Option P2: "Collapsing Walls"
  - When selecting pivot, order median of three elements
  - Use partition code below

```
partition (A, left, right)
1.   store = right
2.   properly order A[left], A[mid] and A[right], using A[mid] as pivot
3.   swap A[mid] and A[right]
4.   left++ and right--
5.   do
6.     while (A[left] < pivot) { left++ }
7.     while (pivot < A[right]) { right-- }
8.     if (left < right) then
9.        swap A[left] and A[right]
10.       left++ and right--
11.    else if (left == right)  { break }
12.  while (left ≤ right)
13.  swap A[store] and A[left]
14.  return left
end
```



pivot = 5

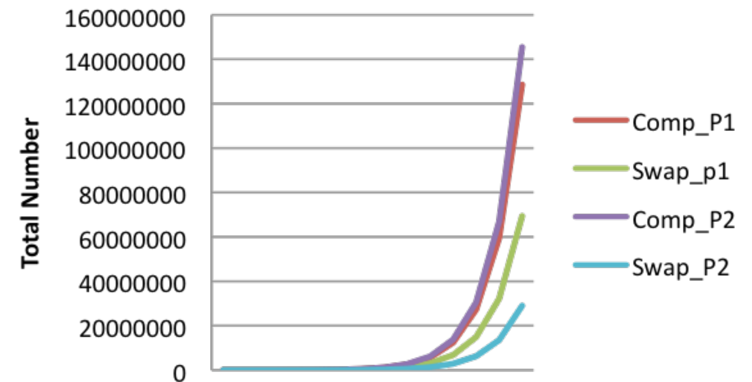
*First time through the **do** loop, we locate and swap {6, 2}*

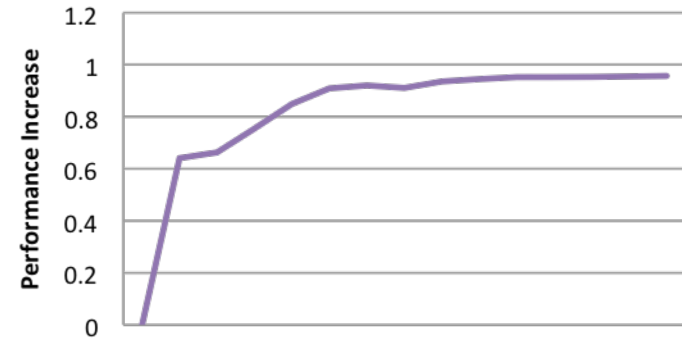
*Second time through the **do** loop, we locate and swap {7, 4}*

Compare different partition methods

- Option P1
  - More Swaps, Fewer Comparisons
- Option P2
  - More Comparisons, Fewer Swaps

n	ratio
1	0
2	0.641026
4	0.662921
8	0.754545
16	0.849095
32	0.909091
64	0.92
128	0.910714
256	0.935484
512	0.944649
1024	0.952055
2048	0.952191
4096	0.952735
8192	0.954768
16384	0.956729



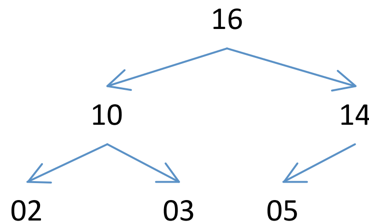


Aside

- What is the best performance for a sorting algorithm using comparison-based sorting?
  - Turns out to be O(n log n)
  - Assuming fixed number of processors and no restrictions on the size or composition of input set
- Implementation Issues
  - In practice, two algorithms that are classified as the same O (n log n) can have different performance

HEAPSORT

- Let's design a sorting algorithm
  - O (n log n) is best we can do with comparison-based sorting
- Can a *heap* be a useful structure?



Heap Property: Each node is greater than either child

Shape Property: Fill Tree by level, left to right

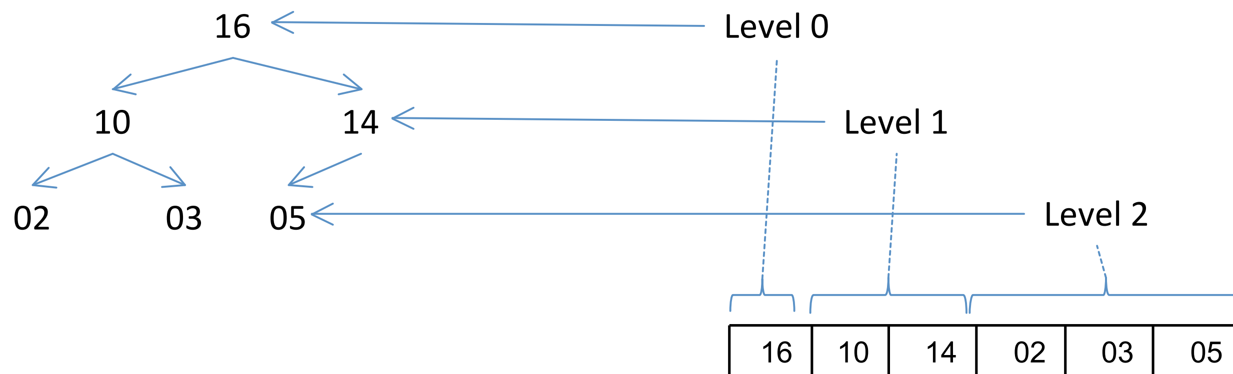
- Note that largest element is root of heap
  - Thus a *findMax* operation for a heap is O(1)

HEAPSORT

- Given a Heap *H*, the following process outputs the content of a heap in descending order

```
while (H has elements)
   remove max and output value
   rebuild heap H
end while
```

- A *heap* can be stored in an array (shape property)

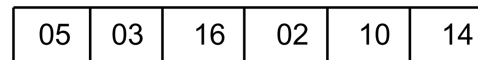


HEAPSORT

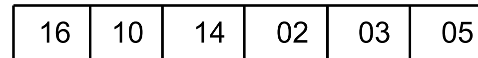
87

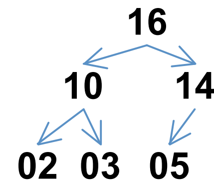
Best case	Average case	Worst case
O(n log n)	O(n log n)	O(n log n)

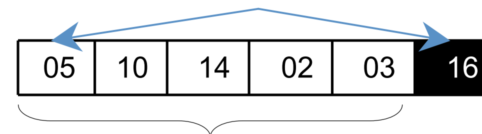
```
sort (A)
1.  buildHeap (A,n)
2.    for i = n – 1 downto 1
3.      swap A[0] with A[i]
4.      heapify (A, 0, i)
5.  end

6.  buildHeap (A, n)
7.    for i = ⌊n/2⌋ downto 0
8.      heapify (A, i)
9.  end

10. heapify (A, idx, max)
11.   left = 2*idx + 1
12.   right = 2*idx + 2
13.   if (left < max and A[left] > a[idx]) then
14.     largest = left
15.   else largest = idx
16.   if (right < max and A[right] > a[largest]) then
17.     largest = right
18.   if (largest ≠ idx) then
19.     swap A[idx] and A[largest]
20.     heapify (A, largest, max)
end
```



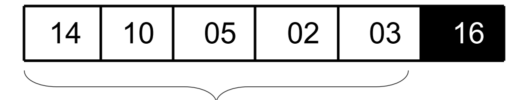
buildHeap



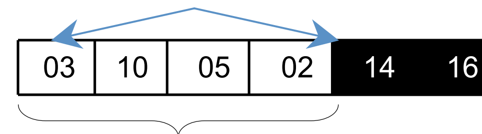




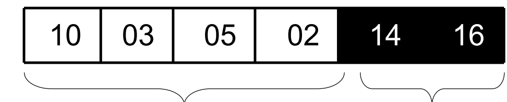
Might no longer be a heap



A heap again

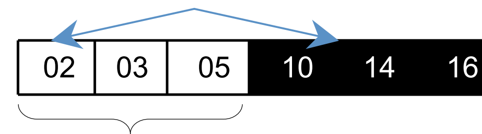


Might no longer be a heap

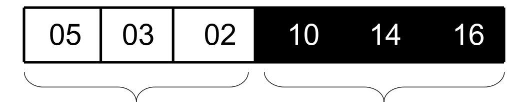


A heap again

Sorted sub-array



Might no longer be a heap

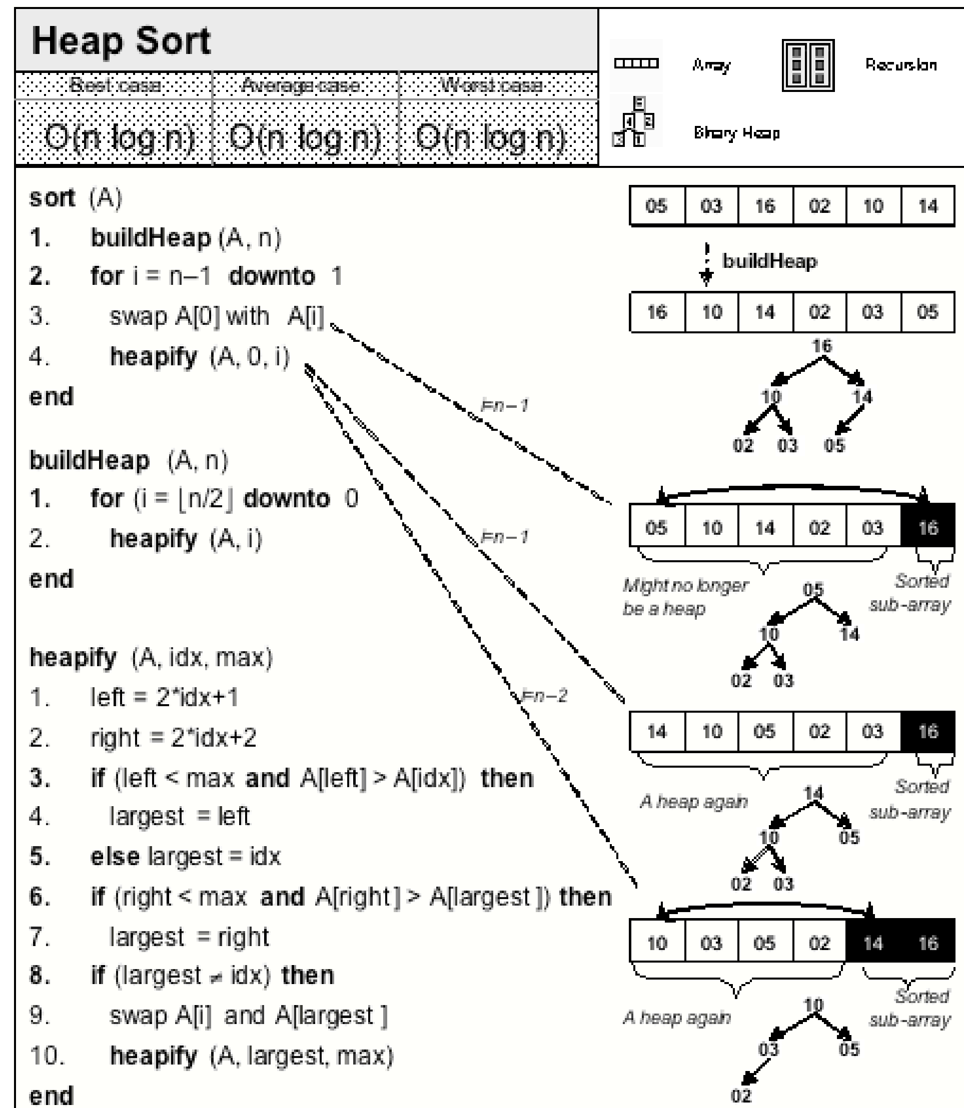


A heap again

Sorted sub-array

HEAPSORT final pieces

- Store binary heap in an array
  - Sort "in place" by swapping maximum element with proper place in array
  - Rebuild Heap after each swap
- Will need *n*–1 iterations
  - heapify takes O(log n)
- Achieves O(n log n)
  - (n–1) * log n
- Fixed Worst Case
  - Also O(n log n)



Code Check

- Show actual running code
  - Handout
  - Debug example

Why discuss HEAPSORT

- Introduce heap structure
  - Useful to understand
- Algorithm shows “tight” bounds
  - Average, Worst cases are similar

How to sort without comparing

- Aggressive Divide and Conquer strategy
  - Divides one problem of size *n* into *n* problems whose average size is 1
- Given *n* elements to sort, create an array of *n* buckets B[]
  - Assign each element from input to a bucket
  - some buckets may be empty or contain (a few) elements
  - Overwhelm the problem with extra space

Importance of *hash* function

- Construct special hash function *hash* (a_i)
  - input data must be uniformly distributed
  - *hash* (a_i) is ordered; if $a_i < a_j$ then *hash*(a_i) < *hash*(a_j)
- Because data is uniformly distributed…
  - A small constant number of elements per bucket
  - Which means total sort time for all buckets is O(n)
- Because hash function is ordered…
  - Can retrieve sorted elements by processing buckets in order, once their contents are sorted

(c) 2009, George Heineman

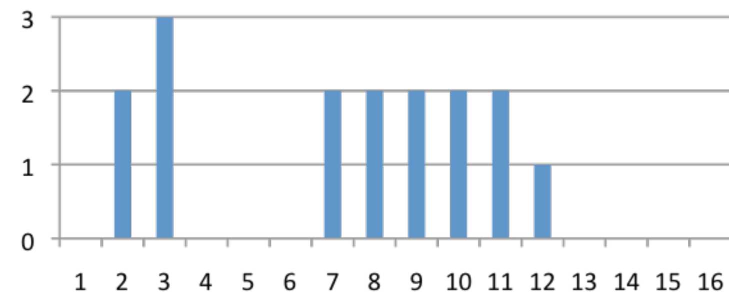
Uniform Distribution Example

- n=16 floating point values from the set [0, 1)
  - $b_i = [\frac{i-1}{16}, \frac{i}{16})$

0.183…
0.544…
0.113…
0.444…
0.102…
0.619…
0.435
0.433
0.141…
0.163…
0.606…
0.437…
0.654…
0.720…
0.685…
0.500…

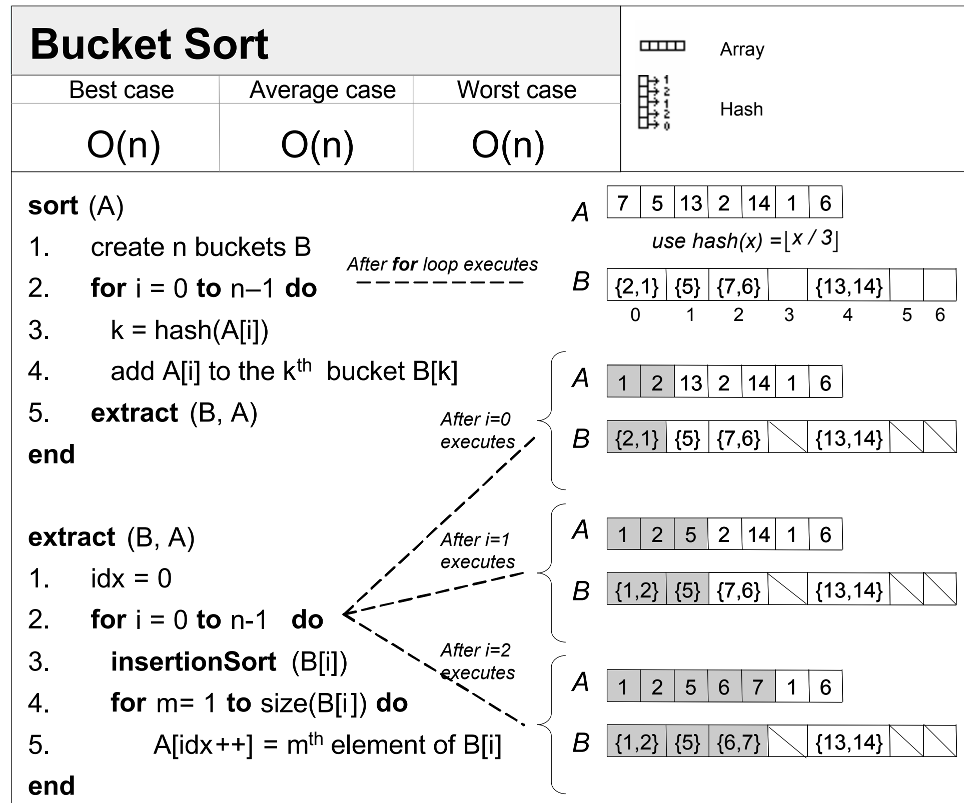
b_2 = {0.102, 0.113}
b_3 = {0.141, 0.163, 0.183}
b_7 = {0.433, 0.435}
b_8 = {0.437, 0.444}
b_9 = {0.500, 0.544}
b_{10} = {0.606, 0.619}
b_{11} = {0.654, 0.685}
b_{12} = {0.720}

Some buckets are empty
Some buckets have multiple elements



BUCKET SORT Fact Sheet

- Process all elements
  - insert each into appropriate bucket
- Overwrite original array
  - Extract bucket elements in order



BUCKET SORT

94

Best case	Average case	Worst case
O(n)	O(n)	O(n)

sort (A)
1. create n buckets B
2. **for** i = 0 **to** n– 1 **do**
3. k =hash (A[i])
4. add A[i] to the kth bucket B[k]
5. **extract** (B, A)

end

extract (B, A)
1. idx = 0
2. **for** i = 0 **to** n–1 **do**
3. **insertionSort** (B[i])
4. **for** m = 1 **to** size (B[i]) **do**
5. A[idx++] = m^{th} element of B[i]

end

A: | 7 | 5 | 13 | 2 | 14 | 1 | 6 |

use hash(x) = $\lfloor x/3 \rfloor$

B: | {2,1} | {5} | {7,6} | | {13,14} | | |
indices: 0 1 2 3 4 5 6

i = 0 → A: | 1 | 2 | 13 | 2 | 14 | 1 | 6 |

i = 1 → A: | 1 | 2 | 5 | 2 | 14 | 1 | 6 |

i = 2 → A: | 1 | 2 | 5 | 6 | 7 | 1 | 6 |

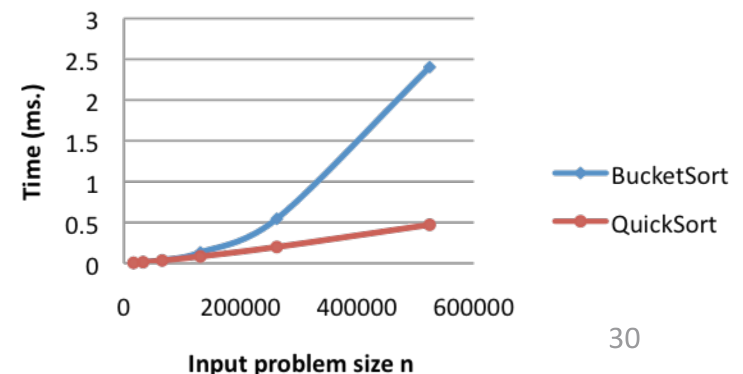
i = 3 → A: | 1 | 2 | 5 | 6 | 7 | 13 | 14 |

BUCKET SORT Summary

- Incredibly effective for uniform data
- With small tweak becomes HASH SORT
  - Surprisingly effective for collections of normal strings if # buckets ≅ 2*n

Consider 26^3 = 17,576 buckets with *hash* function that places a string into bucket based on its first three letters

problem size n	empty buckets	buckets with one	Avg. size of bucket	BUCKET SORT time	QUICKSORT time
16384	7670	5520	1.65495	0.0043	0.0051
32768	4291	3941	2.466	0.0118	0.0132
65536	2390	1281	4.31	0.0368	0.0337
131072	2005	115	8.417	0.1318	0.0833
262144	1977	1	16.805	0.5446	0.1991
524288	1976	0	33.608	2.4036	0.4712



Summary

- Sorting Concepts
  - Comparison and Swapping
- Sorting Algorithms
  - INSERTION SORT [previous session]
  - QUICKSORT [the gold standard]
  - HEAPSORT [interesting data structure at play]
  - BUCKET SORT [how to sort without comparisons]
  - HASH SORT [reduce space needs of Bucket Sort]

QUICKSORT Exercise

1. Can you rewrite to remove **if**?
2. Can you spot the defects here?
   a. What impact does this defect have?
   b. Is it serious?
   c. How would you fix it?

```
/** Sort array ar[left,right] using QuickSort method.
  * The comparison function, cmp, is needed to properly
  * compare elements. */
void do_qsort (void **ar, int(*cmp)(const void *,const void *),
          int left, int right) {
 int pivotIndex;
 if (right <= left) { return; }

 /* partition */
 pivotIndex = selectPivotIndex (ar, left, right);
 pivotIndex = partition (ar, cmp, left, right, pivotIndex);

 if (pivotIndex-1-left <= minSize) {
  insertion (ar, cmp, left, pivotIndex-1);
 } else {
  do_qsort (ar, cmp, left, pivotIndex-1);
 }
 if (right - pivotIndex - 1 <= minSize) {
  insertion (ar, cmp, pivotIndex+1, right);
 } else {
  do_qsort (ar, cmp, pivotIndex+1, right);
 }
}

/**  Qsort straight */
void sortPointers (void **vals, int total_elems,
            int(*cmp)(const void *,const void *)) {
 do_qsort (vals, cmp, 0, total_elems-1);
}
```