COT 5407: Introduction to Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/5407S19.html

Solving Recurrences using Master Theorem

Master Theorem:

Let a,b >= 1 be constants, let f(n) be a function, and let

$$T(n) = aT(n/b) + f(n)$$

1. If $f(n) = O(n^{\log_b a-e})$ for some constant e>0, then
   - $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then
   - $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a+e})$ for some constant e>0, then
   - $T(n) = \Theta(f(n))$

Solving Recurrences by Substitution

- Guess the form of the solution
- (Using mathematical induction) find the constants and show that the solution works

Example

$T(n) = 2T(n/2) + n$

Guess (#1) $T(n) = O(n)$

Need $T(n) <= cn$ for some constant $c>0$

Assume $T(n/2) <= cn/2$ Inductive hypothesis

Thus $T(n) <= 2cn/2 + n = (c+1) n$

Our guess was wrong!!

Solving Recurrences by Substitution: 2

$$T(n) = 2T(n/2) + n$$

Guess (#2) $T(n) = O(n^2)$

Need $T(n) <= cn^2$ for some constant $c>0$

Assume $T(n/2) <= cn^2/4$ Inductive hypothesis

Thus $T(n) <= 2cn^2/4 + n = cn^2/2+ n$

Works for all n as long as c>=2 !!

But there is a lot of "slack"

Solving Recurrences by Substitution: 3

$T(n) = 2T(n/2) + n$

Guess (#3) $T(n) = O(n\log n)$

Need $T(n) <= cn\log n$ for some constant $c>0$

Assume $T(n/2) <= c(n/2)(\log(n/2))$ Inductive hypothesis

Thus $T(n) <= 2\,c(n/2)(\log(n/2)) + n$

$<= cn\log n - cn + n <= cn\log n$

Works for all n as long as c>=1 !!

This is the correct guess. WHY?

Show $T(n) >= c'n\log n$ for some constant $c'>0$

Solving Recurrence Relations

Recurrence; Cond	Solution
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-c) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-c) + O(n)$	$T(n) = O(n^2)$
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a = b$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a < b$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$ $af(n/b) \leq cf(n)$	$T(n) = \Omega(n^{\log_b a} \log n)$

QuickSort

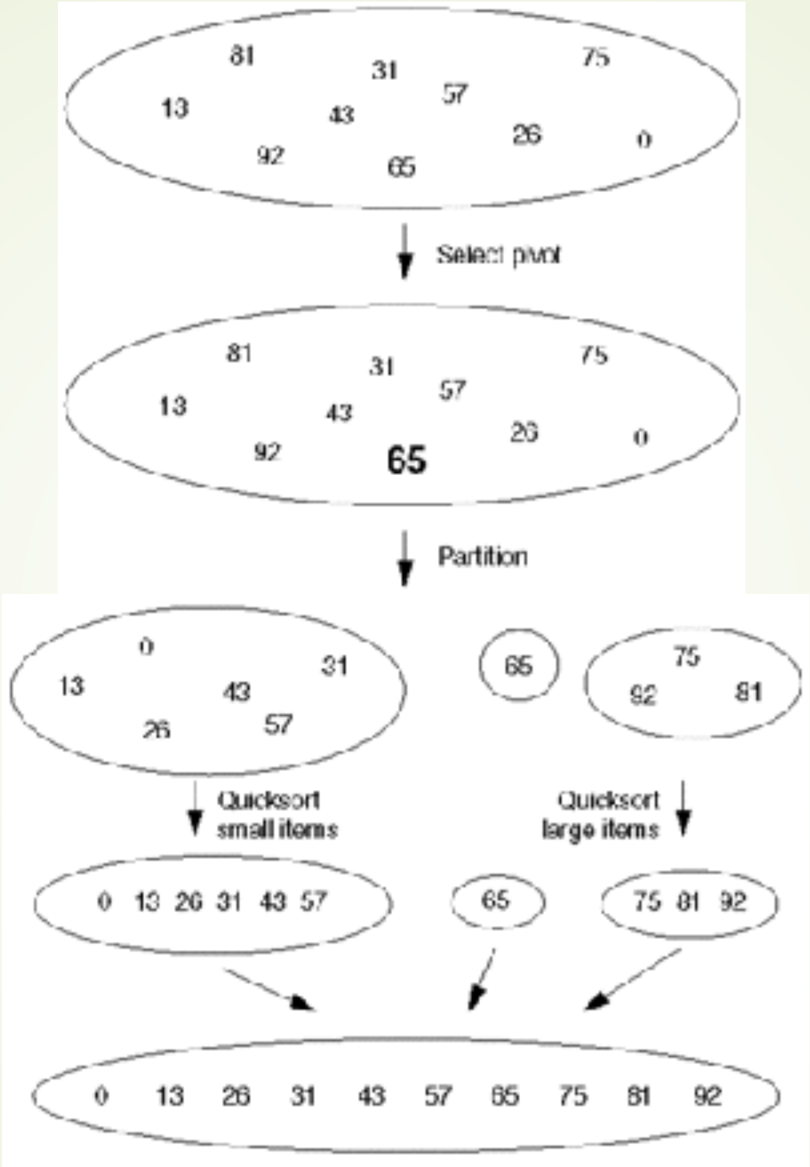
MergeSort

- Divide into 2 equal sublists
- Sort each sublist “recursively”
- Merge 2 sorted sublists
  - Assumption: Merging is faster than sorting from fresh
- Most of work is done in merging

QuickSort

- Partition into 2 sublists using a pivot
- Sort each sublist “recursively”
- Concatenate 2 sorted sublists
  - Assumption: Partition is faster than sorting
- Most of work is done in partition
- Process described using a tree
  - Top-down process: Partition each list into 2 sublists
  - Bottom-up process: Concatenate two sorted sublists into one sorted sublist

Figure 8.10 Quicksort



Partition Algorithm

O(N) time

- Pick a pivot
- Compare each item to a pivot and create two lists:
  - L = list of all items smaller than the pivot
  - R = list of all items larger than the pivot
- One scan through the list is enough, but seems to need extra space
- How to design an in-place partition algorithm!

Partition

Figure A If 6 is used as pivot, the end result after partitioning is as shown in the Figure B.

2	1	4	5	0	3	9	8	7	6

Figure B Result after Partitioning

2	1	4	5	0	3	6	8	7	9

QuickSort

```
QUICKSORT(array A, int p, int r)
if (p < r)
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p, q − 1)
         QUICKSORT(A, q + 1, r)
```

To sort array call QUICKSORT$(A, 1, length[A])$.

```
PARTITION(array A, int p, int r)
x ← A[r]                          ▷ Choose pivot
i ← p − 1
for j ← p to r − 1
    do if (A[j] ≤ x)
        then i ← i + 1
             exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

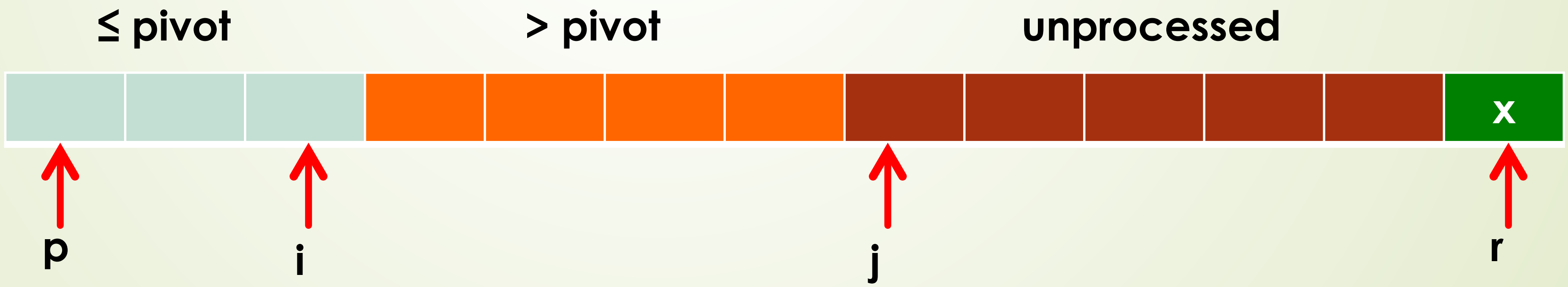
Page 146, CLRS

Time Complexity

- $T(N) = O(N) + T(N_1) + T(N_2)$
- On the average, $N_1 = N_2 = N/2$
- Thus, average-case complexity = $O(N \log N)$
- Worst-case: Either N_1 or $N_2 = 0$
  - Thus, $T(N) = O(N) + T(N - 1)$
  - $T(N) = O(N^2)$

Invariant for Partition

- At the start of iteration j,
  - A[1..i] has elements that are smaller than or equal to pivot x
  - A[i+1..j-1] has elements that are larger than pivot x
  - A[j..r-1] have not yet been processed
  - A[r] has the pivot x
- Try to prove this invariant!



Time Complexity

Recurrence Relaton

- $T(N) = O(N) + T(N_1) + T(N_2)$

Average-Case Complexity

- On average, $N_1 = N_2 = N/2$
- $T(N) = O(N) + 2T(N/2)$
- Thus, average-case complexity = O(N log N)

Worst-Case Complexity

- Worst-case: Either N_1 or $N_2 = 0$
  - Thus, $T(N) = O(N) + T(N - 1)$
  - $T(N) = O(N^2)$

• **Warning**: Quicksort cannot be used if a sorting algorithm is needed that runs in time O(n log n) in the worst case.

Variants of QuickSort

- Choice of Pivot
  - Random choice
  - Median of 3
  - Median
- Avoiding recursion on small subarrays
  - Invoking InsertionSort for small arrays

Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- MergeSort
- HeapSort
- QuickSort
- Bucket & Radix Sort
- Counting Sort

Algorithm Analysis

- Worst-case time complexity*
  - Worst possible time of all input instances of length N
- (Worst-case) space complexity
  - Worst possible spaceof all input instances of length N
- Average-case time complexity
  - Average time of all input instances of length N

Upper and Lower Bounds

- Time Complexity of a Problem
  - **Difficulty**: Since there can be many algorithms that solve a problem, what time complexity should we pick?
  - **Solution**: Define upper bounds and lower bounds within which the time complexity lies.
- What is the **upper** bound on time complexity of sorting?
  - **Answer**: Since SelectionSort runs in worst-case $O(N^2)$ and MergeSort runs in $O(N \log N)$, either one works as an upper bound.
  - **Critical Point**: Among all upper bounds, the best is the lowest possible upper bound, i.e., time complexity of the best algorithm.
- What is the **lower** bound on time complexity of sorting?
  - **Difficulty**: If we claim that lower bound is $O(f(N))$, then we have to prove that no algorithm that sorts N items can run in worst-case time $o(f(N))$.

Lower Bounds

- It's possible to prove lower bounds for many comparison-based problems.
- For comparison-based problems, for inputs of length N, if there are P(N) possible solutions, then
  - any algorithm needs $\boxed{\log_2(P(N))}$ to solve the problem.
- Binary Search on a list of N items has at least N + 1 possible solutions. Hence lower bound is
  - $\log_2(N+1)$.
- Sorting a list of N items has at least N! possible solutions. Hence lower bound is
  - $\log_2(N!) = O(N \log N)$
- Thus, MergeSort is an optimal algorithm.
  - Because its worst-case time complexity equals lower bound!

Beating the Lower Bound

- Bucket Sort
  - Runs in time O(N+K) given N integers in range [a+1, a+K]
  - If K = O(N), we are able to sort in O(N)
  - How is it possible to beat the lower bound?
  - Only because we know more about the data.
  - If nothing is know about the data, the lower bound holds.
- Radix Sort
  - Runs in time O(d(N+K)) given N items with d digits each in range [1,K]
- Counting Sort
  - Runs in time O(N+K) given N items in range [a+1, a+K]

Stable Sort

- A sort is **stable** if equal elements appear in the same order in both the input and the output.
- Which sorts are stable? Homework!