CS5800 Algorithms

Module 2. Sorting Algorithms and Divide and Conquer

Basic Sorting Algorithms

- How do you sort a deck of cards?
- Most people use **Insertion sort**.
- Time complexity: $O(n^2)$
- Space complexity: $O(1)$ additional memory
- Stable
  - Does not change the order of the same valued cards
- Online
  - You can sort cards as you receive cards
- http://visualgo.net/sorting

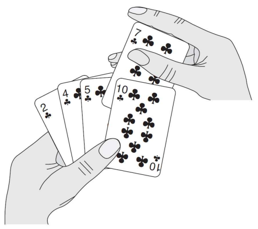


Figure 2.1 Sorting a hand of cards using insertion sort.

Insertion Sort Algorithm

```
INSERTION-SORT(A)
1  for $j = 2$ to $A.length$
2      $key = A[j]$
3      // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
4      $i = j - 1$
5      while $i > 0$ and $A[i] > key$
6          $A[i+1] = A[i]$
7          $i = i - 1$
8      $A[i+1] = key$
```

Let’s prove the correctness of the algorithm!

- Proof technique: Loop Invariant
- Formal statement: At the start of each iteration of the for-loop of lines 1-8 (that is, after line 1 but before line 2), the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.
- Why is the algorithm true? Show 3 things about the loop invariant:
  - Initialization: It is true prior to the first iteration of the loop.
    - Like the base case of a mathematical induction
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
    - Like the inductive step of a mathematical induction
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Correctness Proof

- Initialization: After line 1, before line 2, when $j = 2$: Induction base
  - Does $A[1..j-1] = A[1..1] = A[1]$ consists of $A[1..j-1] = A[1..1] = A[1]$ and in sorted order? Trivially yes (a singleton subarray).

  (Similar to the base case of an induction proof)
- Maintenance
  - Assume the invariant is true at $j = k$: Induction hypothesis
    - That is, $A[1..k-1]$ consists of original $A[1..k-1]$, but in sorted order, at the beginning of this iteration (after line 1, before line 2, when $j = k$).
  - Show the invariant holds true at $j = k + 1$: Induction step.
    - Does $A[1..k]$ consists of original $A[1..k]$ and in sorted order at the beginning of the next iteration (after line 1, before line 2, when $j = k + 1$).
    - Yes. At the iteration, $A[k]$ is placed at its right place in the sorted order of $A[1..k]$.
      - Some hand-waving on the inner loop, which can be formalized using another invariant (skipped)

5

Correctness Proof (Continue)

- Termination
  - When the loop terminates, the invariant should give a useful property about the correctness.
  - When does the loop terminate?
    - j is incremented by 1, so the loop terminates when $j = A.length + 1$.
  - What is the loop invariant then?
    - $A[1..j-1] = A[1..A.length]$ consists of the elements originally in $A[1..A.length]$, but in sorted order.
    - This is exactly the statement of the sorting problem/algorithm's expected/correct output!

6

Selection Sort

- Note subproblem decomposition:
  - For array A[i,n]:
    - Find index of minimum of array values in index i to n. Call it j.
    - Swap A[i] and A[j].
    - Repeat the above for subarray A[i+1,n].
    - Of course if i=n, nothing to do, so just stop.

FIND_MIN_INDEX(A, s)

- Input: Array A[1,n], starting index s.
- Output: Index m (between s and n) of the minimum of A[s,n]
- E.g., if A={55,88,33,44,99} and s=2, return 3 (the index of 33, which is the minimum of A[2,5])
- Steps:

```
• indexOfMinimumSoFar = s
• for i = s to n:
    • if A[i] < A[indexOfMinimumSoFar]:
        • indexOfMinimumSoFar = i
• return indexOfMinimumSoFar
```

SEL_SORT(A)

- Let m= FIND_MIN_INDEX(A,1). Then swap A[1] and A[m].
- Let m= FIND_MIN_INDEX(A,2). Then swap A[2] and A[m].
- ...
- Steps:
  - for s = 1 to n:
    - m = FIND_MIN_INDEX(A,s)
    - tmp = A[m]; A[m] = A[s]; A[s] = tmp;   // Swap A[s] and A[m]

SEL_SORT(A) Running Time Analysis

- Count number of operations executed
  - for s = 1 to n: // n times of 1 comparison and 1 increment, plus:
    - m = FIND_MIN_INDEX(A, s)   // # FindMin(A,1) + # FindMin(A,2) + ... (not n times)
    - tmp = A[m]; A[m] = A[s]; A[s] = tmp;   // 3
- Therefore, T_SelSort(n)=5n + $\sum_{s=1}^{n}$ #FIND_MIN_INDEX(A, s)
- # FIND_MIN_INDEX(A,s):
  - indexOfMinimumSoFar = s   // 1
  - for i = s to n: // n-s+1 times of 1 comparison and 1 increment, plus:
    - if A[i] < A[indexOfMinimumSoFar]: // 1
      - indexOfMinimumSoFar = i   // 0 ~ 1 (1 if condition is true, 0 otherwise)
  - return indexOfMinimumSoFar   // 1
- # FIND_MIN_INDEX(A,s) = 2(n-s+1)+2+(0~1)*(n-s+1) $\geq 2(n-s)+4 \ and \leq 3(n-s)+5 \Rightarrow \Theta(n-s)$

Final T_SelSort(n): $\Theta(n^2)$

- T_SelSort(n)=5n + $\sum_{s=1}^{n}$ #FIND_MIN_INDEX(A, s)
  $\geq 5n + 2\{(n-1) + (n-2) + \cdots + 1\} + 4n$
  $= 2\frac{n(n-1)}{2} + 5\text{n} + 4\text{n}$
  $= n^2 + 8n$
- T_SelSort(n)=5n + $\sum_{s=1}^{n}$ #FIND_MIN_INDEX(A, s)
  $\leq 5n + 3\{(n-1) + (n-2) + \cdots + 1\} + 5n$
  $= 3\frac{n(n-1)}{2} + 5\text{n} + 5\text{n}$
  $= \frac{3}{2}n^2 + \frac{17}{2}n$
- These 2 inequalities fit the definition of $\Theta(n^2)$!

Algorithm Analysis

- **Selection sort** in two steps:
  - Find the index of the minimum of array values
  - Place the minimum in the correct position
    - And repeat this process on the smaller array
  - $O(n^2)$ time, $O(1)$ additional space, Not stable, Not online
- Analysis of selection sort:
  - Time for FIND_MIN_INDEX: $T_1(n) = \Theta(n)$
  - Time for SEL_SORT: $T_2(n) = T_1(n) + c + T_2(n-1)$
    - Recurrence!
    - Solving recurrence, get $T_2(n) = \Theta(n^2)$

Time Complexities of Quadratic Sorting Algorithms

Selection, Insertion, Bubble Sorts

VisuAlgo.Net/sorting

- Click the link above and do the following experiments:
- For each 'BUBBLE', 'SELECT', 'INSERT', do the following:
  - Click Create. Then for each 'Random', 'Sorted-Increasing', 'Sorted-Decreasing':
    - Click 'Sort-Go'.
- See how each algorithm works visually
- Think about time complexity of each case
  - "Complexity": Since we are not using the raw running time, the word "complexity" is used to refer to the performant nature of an algorithm (its difficulty)
  - Complexity is always stated in asymptotic notation (big-O, Omega, Theta)

Q: Given the original array of 22, 33, 11, 88, 66, what is the resulting array after the first pass of selection sort is completed? Assume we are sorting in ascending order.

A: 11, 33, 22, 88, 66

Q: Given the original array of 22, 33, 11, 88, 66, what is the resulting array after the first pass of bubble sort is completed? Assume we are sorting in ascending order.

A: 22, 11, 33, 66, 88

Q: Among the soring algorithms given below, whose best-case complexity is worse than that of the others?

a) Selection sort

b) Bubble sort

c) Insertion sort

The summary of quadric sorting algorithms

	Worst Case	Best Case	Additional Space	Stable?	Online?
Insertion Sort	$O(n^2)$	$O(n)$	$O(1)$	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	No
Bubble Sort	$O(n^2)$	$O(n)$	$O(1)$	Yes	No

Merge Sort And Intro To Divide-And-Conquer

Improving Sorting Performance From Quadratic To Linear-Logarithmic

Sub-Problem Property

- Recall selection sort:
  - After the first pass, we got a smaller problem of the same type (sorting).
    - Sorting an array with one fewer items, though the indexing structure is different.
  - Or we could say that we deliberately seek to reduce the original problem into a smaller sub-problem and the related reduction process.
    - In this case, the reduction process is a pre-process of finding the smallest and swapping it with the first entry.
  - Very similar nature in bubble sort: Reducing problem size by one every pass
- Can we think of a different sub-problem structure and reduction?
  - If we are reducing the problem size by one, why not reducing bigger?
  - How about *dividing* the original problem by halves?

Merge Sort: Divide-And-Conquer Sorting

- Divide the original array into two halves.
  - Sort each half.
    - Need to use recursion. Recall the recursive algorithm for Fibonacci calculation.
    - For now, do not consider jumping into the recursive calls.
      - Just assume that the recursive call returned with the sorted sub-array (half).
- Then merge the two sorted halves. E.g.:
  - Sorted first half: 11, 33, 44, 66, 88
  - Sorted second half: 22, 35, 40, 77, 80
  - Merging two sorted subarrays:
- CLRS Section 2.3 Designing Algorithms

Go to https://visualgo.net/en/sorting for merge sort example.

Q: The merge sort algorithm is currently merging the following two sorted sub-arrays: 3,5,15,26,36,38,44,47 and 2,4,19,27,46,48,50. What is the merged array after five values are picked and placed in it?

A: 2, 3, 4, 5, 15

Code: Merge Sort

- Top-level sort is easy: (CLRS pp. 34)

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Top-level call is MERGE-SORT(A, 1, A.length) for $A = \langle A[1], A[2], \ldots, A[n] \rangle$ where A.length = n.

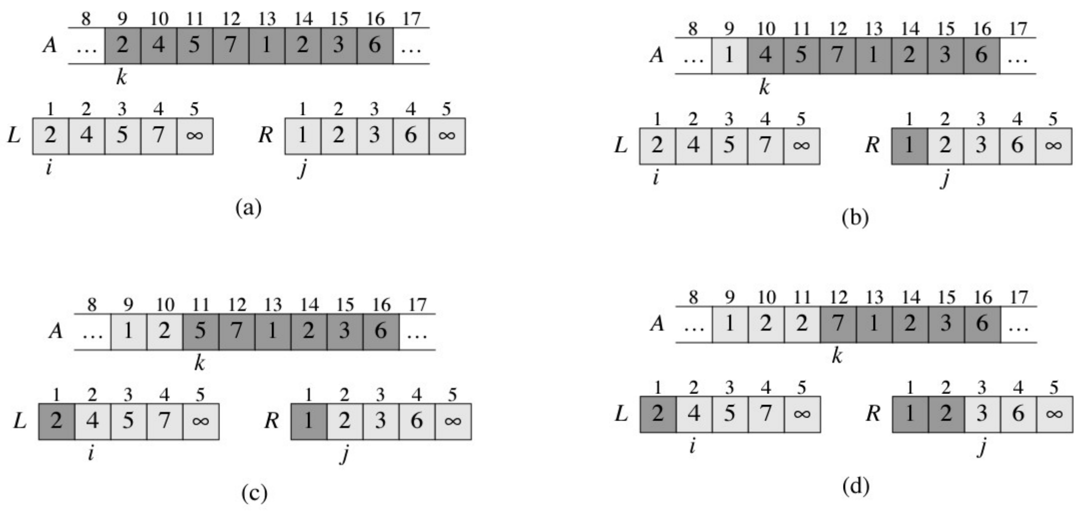
```
MERGE-SORT(A)
1 MERGE-SORT(A, 1, A.length)
```

- What's difficult is MERGE(A, p, q, r)

Code: Merge (CLRS pp. 31)

```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
**for** $i = 1$ **to** $n_1$
    $L[i] = A[p + i - 1]$
**for** $j = 1$ **to** $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
**for** $k = p$ **to** $r$
    **if** $L[i] \le R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    **else** $A[k] = R[j]$
        $j = j + 1$
```

Merge Operations (CLRS Fig. 2.3 in pp. 32)



Sidebar: MERGE() Implementation

- What if we can't depend on the availability of ∞?
- Creating "new arrays" in Line 3 of MERGE(A,p,q,r) is computationally expensive. How can we avoid creating new arrays every time MERGE() is called?
- Can you re-implement MERGE(A,p,q,r) with the two constraints above?
    - This is a homework problem, and may be an exam problem!

Analysis of Merge Sort

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Straightforward top-level recurrence for $T_{MergeSort}(n)$:

$$T_{MergeSort}(n) = 2T_{MergeSort}\left(\frac{n}{2}\right) + T_{Merge}(n) + c_1$$

    - Of course $T_{MergeSort}(0) = T_{Merge}(0) = c_2$ (all c_i's are some constants)
- What is $T_{Merge}(n)$?
    - Counting the number of steps executed in MERGE(A,p,q,r) when r-p+1=n, we get:

$$T_{Merge}(n) = c_3 n + c_4$$

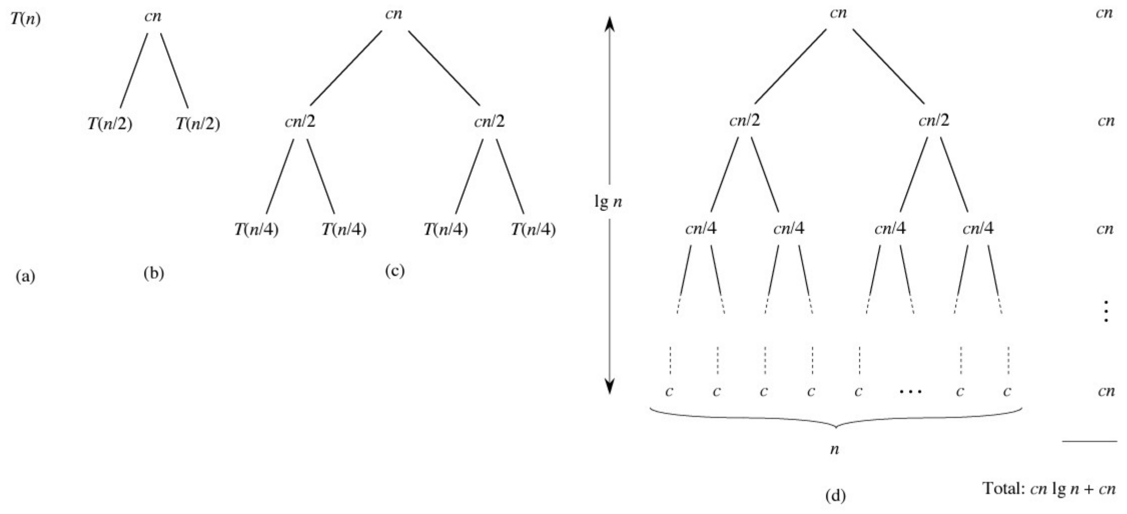
- Therefore, $T_{MergeSort}(n) = 2T_{MergeSort}\left(\frac{n}{2}\right) + cn + c'$

Solving $T(n) = 2T\left(\frac{n}{2}\right) + cn$

- Expansion method:

- Solution: $T(n) =$

Recursion Tree Method for Solving Recurrence (CLRS Fig. 2.5 in pp. 38)



Merge Sort vs. Quadratic Sorts

- $\Theta(n \log_2 n)$ merge sort vs. $O(n^2)$ quadratic (selection, insertion, bubble) sorts
- Assuming each execution step takes $1\mu s$ (microsecond = 10^{-6}s, frequently typed as 'us'), $T_{quad}(n) = n^2$ and $T_{merge}(n) = 10000 n \log_2 n$ (some big coeff),

n	10	100	1000	10000	100000	10^6	10^7	10^8
n^2	0.1ms	0.01s	1s	100s	~2.8h	~11.6 days	~3.17 years	~317 years
$10000n\log_2 n$	~0.33s	~6.6s	~100s	22m	~4.6h	~2.3 days	~26.7 days	~307.6 days

- Which algorithm would you use? Can we do better?

Q: Which of the following sorting algorithms is asymptotically fastest in the worst case?

- A) Insertion Sort
- B) Selection Sort
- C) Bubble Sort
- D) Merge Sort

Q: Which of the following sorting algorithms is asymptotically fastest in the best case?

- A) Insertion Sort
- B) Selection Sort
- C) Bubble Sort
- D) Merge Sort

Q: Which of the following sorting algorithms show the same asymptotic time complexity for both the best case and the worst case?

- A) Insertion Sort
- B) Selection Sort
- C) Bubble Sort
- D) Merge Sort

Methods For Solving Recurrences

Tools To Analyze Divide-And-Conquer Algorithms

Substitution Method For Solving Recurrences (CLRS 4.3)

- Given a recurrence,
  - Take a guess of the solution
    - Not easy, requiring intuitions, experiences
  - Then prove it by induction
    - Not easy either, but could be routine

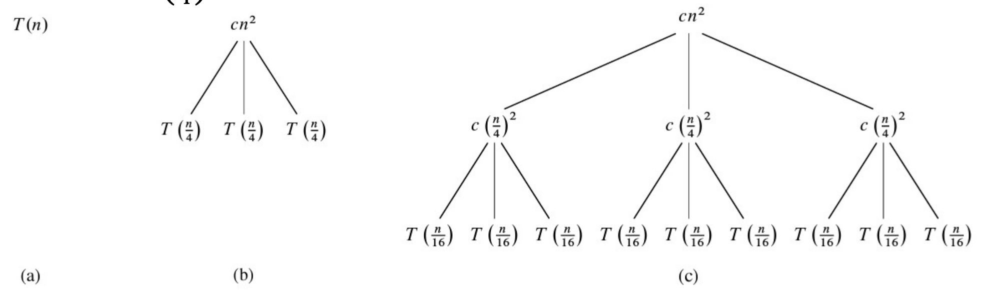
- E.g, $T(n) = 2T(\lfloor n/2 \rfloor) + n, T(1) = 1$
  - Guess that $T(n) = O(n \log_2 n)$
    - How? … From previous experiences?
  - Need to prove $T(n) \le cn \log_2 n$ for some c (we get to choose) and for all $n \ge n_0$ (we get to choose n_0 as well).
    - Above statement is just the definition of $T(n) = O(n \log_2 n)$
    - We prove this by mathematical induction, especially strong induction

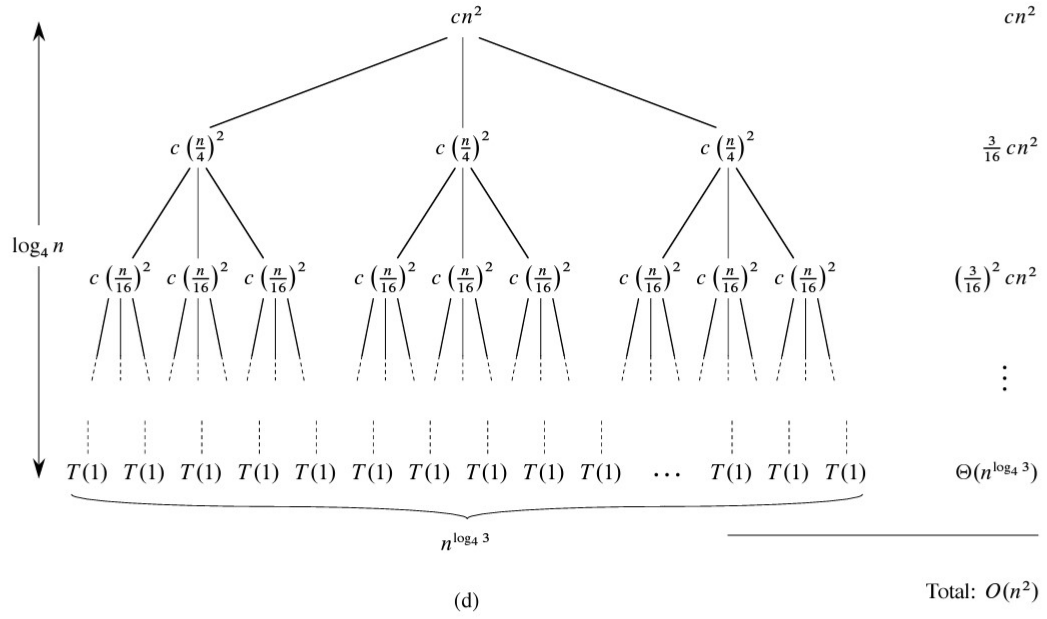
Proof By Induction Example

- Induction step
  - Hypothesis: Assume $T(m) \leq cm \log_2 m$ for all $m < n$
  - Prove: $T(n) \leq cn \log_2 n$
    - Again, don't forget that we get to choose c!
- Induction base
  - May need to choose a different starting value of n
    - Because $T(1) = 1$ may not meet the inequality
    - Remember we get to choose n_0, so don't be limited by the given base case.

Recursion Tree Method For Solving Recurrences (CLRS 4.4)

- Making a good guess for substitution method feels like a black magic!
- Visually expand original $T(n)$, using tree structure all the way down to base case, and reason about the outcome
- E.g., $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$,


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)



39

Master Method For Solving Recurrences (CLRS 4.5)

- "Cookbook" method for solving recurrences of the form
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
- Proof is presented in CLRS 4.6, left as optional (not covered in class)
- Intuitive understanding: We compare $f(n)$ with $n^{\log_b a}$
  - If $f(n)$ is smaller (polynomially & asymptotically), then $T(n) = \Theta(n^{\log_b a})$.
  - If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
  - If $f(n)$ is bigger with some more conditions (see Theorem 4.1), then $T(n) = \Theta(f(n))$

40

Master Method Examples

- $T(n) = 2T\left(\frac{n}{2}\right) + cn$ (merge sort)
  - $f(n) = cn$ and $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
    - Which makes $f(n) = \Theta(n^{\log_b a})$, thus second case, and we get $T(n) = \Theta(n \log n)$.
- $T(n) = 8T\left(\frac{n}{2}\right) + cn^2$ (ordinary div.-and-conquer matrix mult.)
  - $f(n) = cn^2$ and $n^{\log_b a} = n^{\log_2 8} = n^3$
    - Which makes $f(n)$ smaller than $n^{\log_b a}$, thus first case, and we get $T(n) = \Theta(n^3)$.
- $T(n) = 7T\left(\frac{n}{2}\right) + cn^2$ (Strassen's div.-and-conquer matrix mult.)
  - $f(n) = cn^2$ and $n^{\log_b a} = n^{\log_2 7}$
    - Which makes $f(n)$ smaller than $n^{\log_b a}$, thus first case, and we get $T(n) = \Theta(n^{\log_2 7})$.

41

Q: For the recurrence $T(n) = 9T\left(\frac{n}{3}\right) + n$. What is the correct asymptotic notation for $T(n)$?

a) $\theta(n \log n)$

b) $\theta(n)$

c) $\theta(n^2 \log n)$

d) $\theta(n^2)$

42