Recurrences and Sorting

Data Structures and Algorithms (094224)

Tutorial 8

Winter 2022/23

Recurrences

- Sometimes, we can find *explicit* ⟨**מפורשים**⟩ (tight) bounds on the run-time of a recursive algorithm $\mathcal{A}$ via combinatorial arguments
  - E.g., `DFS`, algorithms presented in tutorial 3
- The more common process:
  1. Express the run-time of $\mathcal{A}$ as a *recurrence* ⟨**משוואת נסיגה**⟩
  2. Solve the recurrence and provide explicit bounds on the run-time
- Several techniques for solving recurrences
  - Substitution method
  - Recursion-tree method
  - Master theorem ⟨**שיטת האב**⟩

Question – 1

Give, as tight as you can, asymptotic upper bound on the running time of the following algorithm.

`Hanoi`(n, A, C, B)

1: **if** n $==$ 1 **then**
2: Print: ‘move plate from A to C
3: **else**
4: `Hanoi`$(n-1, A, B, C)$
5: Print: ‘move plate from A to C
6: `Hanoi`$(n-1, B, C, A)$

Where A is the source tower, B is the auxiliary tower and C the destination

Solution

- If $n = 1$, then the algorithm takes $T(1) = O(1)$ time
- For $n > 1$
  - Lines 1,2 and 5 takes $O(1)$ time
  - Line 4 takes $T(n-1)$ time
  - Line 6 takes $T(n-1)$ time
- The run-time function $T(n)$ of *Hanoi* satisfies

$$T(n) \leq \begin{cases} O(1), & n = 1 \\ 2T(n-1) + O(1), & n > 1 \end{cases}$$

- Recurrence with explicit constants

$$T(n) \leq \begin{cases} c_1, & n = 1 \\ 2T(n-1) + c_2, & n > 1 \end{cases}$$

Solution — cont.

- Since we are interested in upper bound take $c = max\{c_1, c_2\}$

$$T(n) \leq \begin{cases} c, & n = 1 \\ 2T(n-1) + c, & n > 1 \end{cases}$$

- Develop

$$\begin{aligned} T(n) &\leq 2T(n-1) + c \leq 2(2T(n-2) + c) + c = 4T(n-2) + 3c \\ &\leq 4(2T(n-3) + c) + 3c = 8T(n-3) + 7c \\ &\vdots \\ &= 2^j T(n-j) + c(2^j - 1) \end{aligned}$$

- Setting $j = n - 1$, the halt condition is met

$$\begin{aligned} T(n) &\leq 2^{n-1} T(n - (n-1)) + c(2^{n-1} - 1) \\ &\leq 2^{n-1} c + c(2^{n-1} - 1) = c(2^n - 1) \end{aligned}$$

Solution — cont.

- Therefore $T(n) \leq c(2^n - 1)$
- Verify by induction:
- Base: $T(1) \leq c = c(2^1 - 1)$
- Hypothesis: for all positive integer $m < n$, $T(m) \leq c(2^m - 1)$
- Step:

$$T(n) \leq 2T(n-1) + c \underbrace{\leq}_{i.h} 2c(2^{n-1} - 1) + c$$
$$= c2^n - 2c + c = c(2^n - 1)$$

Question — 2

Give, as tight as you can, asymptotic upper bound on the running time of the `Binary_Search`(A, x, ℓ, r) algorithm.

Recall:

- *Binary_Search*(A, x, ℓ, r)
  - A – an array
  - x – an element to search for in array A
  - ℓ (left) – index in array A
  - r (right) – index in array A
  - Assumption: array A is sorted between index ℓ and r
  - Searches for element x in the sorted array A. Returns *NIL* if x is not in A; otherwise, returns the position of x in A.
- The high-level idea:
  1. Search for x in index $m = \lfloor(\ell + r)/2\rfloor$
  2. If $A[m] = x$ return m
  3. If $A[m] < x$ then, since A is non decreasing, x cannot be found in $A[\ell...m]$
  4. Otherwise, x cannot be found in $A[m...r]$

Question — 2

```
Binary_Search(A, x, ℓ, r)
 1: if ℓ > r then
 2:     return NIL
 3: m = ⌊(ℓ + r)/2⌋
 4: if A[m] = x then
 5:     return m
 6: else
 7:     if x < A[m] then
 8:         return Binary_Search(A, x, ℓ, m − 1)
 9:     else
10:         return Binary_Search(A, x, m + 1, r)
```

Solution

- Let $n = r - \ell + 1$
- If $n \leq 1$, then the algorithm takes $O(1)$ time
- If $n > 1$, then $T(n) \leq T(\lfloor n/2 \rfloor) + O(1)$
- The run-time function $T(n)$ of *Binary_Search* satisfies

$$T(n) \leq \begin{cases} O(1), & n \leq 1 \\ T(\lfloor n/2 \rfloor) + O(1), & n > 1 \end{cases}$$

- Using explicit constants

$$T(n) \leq \begin{cases} c, & n \leq 1 \\ T(\lfloor n/2 \rfloor) + c, & n > 1 \end{cases}$$

Solution — cont.

- Notice:
  - $\left\lfloor \frac{\lfloor \frac{n}{2^i} \rfloor}{2} \right\rfloor = \lfloor \frac{n}{2^{i+1}} \rfloor$
  - for $j = \lfloor \lg n \rfloor$, $\lfloor \frac{n}{2^j} \rfloor = 1$. Since $\lfloor \frac{n}{2^j} \rfloor = \left\lfloor \frac{2^{\lg n}}{2^{\lfloor \lg n \rfloor}} \right\rfloor = \lfloor 2^{\lg n - \lfloor \lg n \rfloor} \rfloor$, and $\lg n - \lfloor \lg n \rfloor < 1$
- Develop

$$\begin{aligned} T(n) &\leq T(\lfloor n/2 \rfloor) + c \leq T(\lfloor n/4 \rfloor) + 2c \leq T(\lfloor n/8 \rfloor) + 3c \\ &\vdots \\ &= T(\lfloor n/2^j \rfloor) + jc \end{aligned}$$

- Setting $j = \lfloor \lg n \rfloor$, the halt condition is met

$$T(n) \leq T(\lfloor n/2^{\lfloor \lg n \rfloor} \rfloor) + c\lfloor \lg n \rfloor \leq c + c \lg n = c(1 + \lg n)$$

Solution — cont.

- Therefore $T(n) \leq c(1 + \lg n)$
- Verify by induction:
- Base: $T(0) \leq T(1) \leq c \leq c(\lg 1 + 1) = c$
- Hypothesis: for all positive integer $m < n$, $T(m) \leq c(\lg m + 1)$
- Step:

$$T(n) \leq T(\lfloor n/2 \rfloor) + c \underbrace{\leq}_{i.h} c(\lg(\lfloor n/2 \rfloor) + 1) + c \leq c(\lg(n/2) + 1) + c$$

$$\leq c(\lg n - 1 + 1) + c = c(\lg n + 1) \quad \blacksquare$$

- $T(n) = \Omega(\log(n))$, if x is not in array A

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence
$T(n) = aT(n/b) + f(n)$.
Where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Master Theorem – Examples

1. $T(n) = 9T(n/3) + 1 = \Theta(n^2)$ – fits case 1
   - $b = 3$, $a = 9$, $f(n) = 1 = O(n^{\log_3 9 - 1})$, for $\epsilon = 1$
2. $T(n) = T(2n/3) + 1 = \Theta(\lg n)$ – fits case 2
   - $b = 3/2$, $a = 1$, $f(n) = 1 = \Theta(n^{\log_{3/2} 1})$
3. $T(n) = 3T(n/4) + n \lg n = \Theta(n \log n)$ – fits case 3
   - $b = 4$, $a = 3$, $f(n) = n \lg n = \Omega(n)$, for $\epsilon = 1 - \log_4 3$
   - Also, $3\frac{n}{4} \lg(n/4) = \frac{3}{4} n \lg n - \frac{3}{2} n \leq cn \lg n$, for $c = 3/4$ and for all $n \geq 1$

1 Recurrences

2 Sorting

Question 3

Observe that the while loop of lines 4 to 7 of the *Insertion_Sort* procedure uses a linear search to scan (backward) through the sorted subarray $A[1...j-1]$.
Can we use a binary search instead to improve the overall running time of insertion sort to $\Theta(n \log(n))$?

`Insertion_Sort`(A)

1: **for** $j = 2$ to $A.length$ **do**
2: $\quad key = A[j]$
3: $\quad i = j - 1$ $\quad \triangleright$ insert key into the sorted sequence $A[1 \ldots j-1]$
4: $\quad$ **while** $i > 0$ and $A[i] > key$ **do**
5: $\quad\quad A[i+1] = A[i]$
6: $\quad\quad i = i - 1$
7: $\quad A[i+1] = key$

Solution

- What is accomplished by the linear search?
- In addition to finding the correct position of the element - it prepares the position in which the element is to be inserted
- If we use a binary search to find the position, we may still need to move elements to maintain a sorted subarray
- If array A is ordered in decreasing order, in iteration j of the for loop
  - $\Omega(1)$ time for fixed time operations
  - $\Omega(\log(j-1))$ time for binary search
  - $\Omega(j-1)$ time to move $A[j]$ to $A[1]$
  - A total of $\Omega(j-1)$

Solution — cont.

- The running time of *Insertion_Sort* with binary sort on an ordered in decreasing order array A of size n

$$\sum_{j=2}^{n} \Omega(j-1) = \sum_{j=1}^{n-1} \Omega(j) = \Omega(n^2)$$

- Since there exists an instance I, $|I| = n$ for all n such that the running time of `Insertion_Sort` with binary search is $\Omega(n^2)$ on I it cannot be that the running time function of `Insertion_Sort` with binary search is $O(n \log(n))$

Question – 4

Describe a $\Theta(n \log(n))$-time algorithm that, given an array A of n numbers and another number x, determines whether or not there exist two elements (not necessarily distinct) in A whose sum is exactly x.

Solution — First attempt

The high-level idea:

- For each element in A go over every element in A and compare the sum to x

`FindSumPair1`(A, x)

```
1: for i = 1 to n do
2:     for j = 1 to n do
3:         if A[i] + A[j] == x then
4:             return true
5: return false
```

Solution — First attempt — cont.

Complexity Analysis:

- Each iteration of the for loop in lines 2–4 takes $\Theta(n)$ time
- The for loop of line 1 iterate n times
- A total of $\Theta(n^2)$

NOT good enough. A different approach is needed.

Solution

The high-level idea:

1. Sort elements in A
2. For each element in A use `Binary_Search` to find another element such that their sum is x

`FindSumPair2`(A, x)

```
1: Merge_Sort(A, 1, n)
2: for i = 1 to n do
3:     index = Binary_Search(A, x − A[i], 1, n)
4:     if index ! = NIL then
5:         return true
6: return false
```

Solution — cont.

Complexity Analysis:

- Line 1 takes $\Theta(n \log n)$
- Each iteration of the for loop takes $\Theta(\log n)$
- The for loop iterate n times. A total of $\Theta(n \log n)$
- A total of – $\Theta(n \log n) + \Theta(n \log n) = \Theta(n \log n)$

Exactly what we needed.

Solution — Another approach

`FindSumPair3`(A, x)

```
1:  Merge_Sort(A, 1, n)
2:  left = 1
3:  right = n
4:  while left ≤ right do
5:      if A[left] + A[right] == x then
6:          return true
7:      else
8:          if A[left] + A[right] > x then
9:              right = right − 1
10:         else
11:             left = left + 1
12: return false
```

Solution — Another approach — cont.

Complexity Analysis:

- Line 1 takes $\Theta(n \log n)$ time
- While loop takes $\Theta(n)$ time
- A total of $\Theta(n \log n)$

Solution — Another approach — cont.

Invariant

At the beginning of each iteration of the while loop in line 4, if their exist two elements in A whose sum is x, then they can be only found in indices i, j such that $left \leq i \leq j \leq right$

- Initialization:
  - Since $left = 1$ and $right = n$ it holds
- Maintenance:
  - Assume $A[left] + A[right] < x$ (the case for $A[left] + A[right] > x$ is analogous)
  - The sum must be incremented and since A is sorted it can only be done by incrementing index $left$
  - This is exactly the case in line 11
  - Since $left$ is only incremented by 1 the invariant holds at the beginning of the next while iteration
    - If two elements whose sum is x exist they can only be found in indexes i, j such that $left \leq i \leq j \leq right$

Solution — Another approach — cont.

- Termination:
  - At termination of the while loop either
    - Two elements whose sum is x are found, or
    - $left > right$
  - By the invariant if $left > right$ such elements cannot be found

Question 5

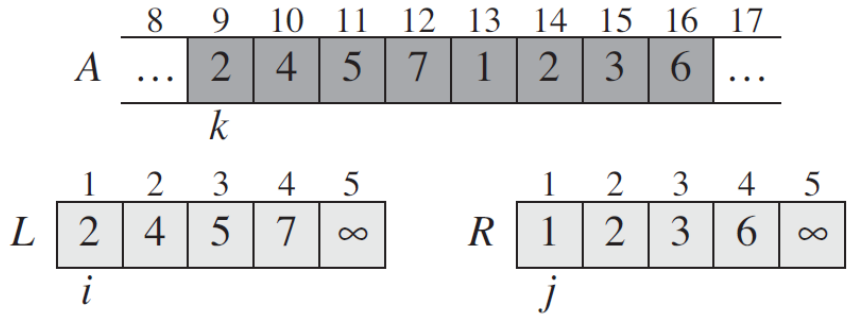
Let A be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair of indices (i, j) is called an inversion of A.
Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \log n)$.

For example:

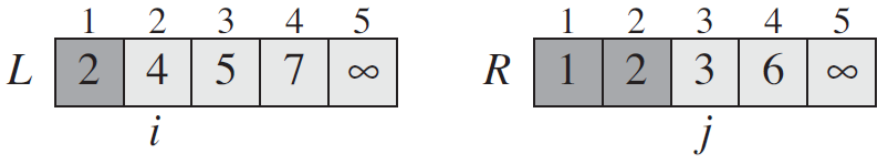
- $A = \langle 2, 3, 8, 6, 1 \rangle$. How many inversions are in A?
- Inversions: $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$.

Solution

- Solution: modify merge sort to count inversions
- In every call to `Merge`, array L elements are in indexes smaller than array R elements in array A



- Let us look at the `Merge` procedure at some iteration of the for loop in line 4



- Since L is sorted, if $L[i] > R[j]$ then an inversion exists in the position of $L[i]$ and $R[j]$ in array A

Solution — cont.

- $L[i']$ for $i < i' \leq n_1$ and $R[j]$ are also inversions in the position of $L[i']$ and $R[j]$ in array A
- $\Longrightarrow$ We can count every inversion
- Won't we count same inversion more than once?
  - `Merge` is invoked "bottom up"
  - After a call to `Merge`(A, p, q, r) all elements $A[q, \cdots, r]$ will appear in the same sorted subarry (L or R) in the next call to `Merge`

Solution — Algorithm

```
Merge_Inversions(A, p, q, r)
 1: (L, R) = Merge_Inv_Init(A, p, q, r)
 2: i = 1
 3: j = 1
 4: inversions = 0
 5: for k = p, ..., r do
 6:     if L[i] ≤ R[j] then
 7:         A[k] = L[i]
 8:         i = i + 1
 9:     else A[k] = R[j]
10:         j = j + 1
11:         inversions = inversions + n_1 − i + 1
12: return inversions
```

```
Merge_Inv_Init(A, p, q, r)
 1: n_1 = q − p + 1
 2: n_2 = r − q
 3: new array L[1 ... n_1 + 1]
 4: new array R[1 ... n_2 + 1]
 5: for i = 1, ..., n_1 do
 6:     L[i] = A[p + i − 1]
 7: for j = 1, ..., n_2 do
 8:     R[j] = A[q + j]
 9: L[n_1 + 1] = ∞
10: R[n_2 + 1] = ∞
11: return (L, R)
```

Solution — Algorithm — cont.

Count inversions in subarray $A[p \ldots r]$

`Count_Inversions`(A, p, r)

1: $inversions = 0$
2: **if** $p < r$ **then**
3: $\quad q = \lfloor (p + r)/2 \rfloor$
4: $\quad inversions = inversions +$ `Count_Inversions`(A, p, q)
5: $\quad inversions = inversions +$ `Count_Inversions`$(A, q + 1, r)$
6: $\quad inversions = inversions +$ `Merge_Inversions`(A, p, q, r)
7: return $inversions$

Complexity Analysis:

- Only added constant-time operations to `Merge_Sort`
- Thus, $T_{\texttt{Count\_Inversions}}(n) = \Theta(n \log n)$