Exam 1 - Review Session

CSCI570 - Spring 2023

Divide & Conquer

T/F question

(T/F) Suppose that there is an algorithm that merges two sorted arrays in $\sqrt{n}$ then the merge sort algorithm runs in O(n)

T/F question

(T/F) Suppose that there is an algorithm that merges two sorted arrays in $\sqrt{n}$ then the merge sort algorithm runs in O(n)

The time complexity of original merge sort is $T(n) = 2T(n/2) + O(n)$
Using the master theorem:

a = 2, b = 2

c = 1

T/F question

(T/F) Suppose that there is an algorithm that merges two sorted arrays in $\sqrt{n}$ then the merge sort algorithm runs in O(n)

The time complexity of the modified merge sort is $T(n) = 2T(n/2) + \sqrt{n}$.
Using the master theorem:

a = 2, b = 2

c = 1

T/F question

(T/F) Suppose that there is an algorithm that merges two sorted arrays in $\sqrt{n}$ then the merge sort algorithm runs in O(n)

The time complexity of merge sort is $T(n) = 2T(n/2) + \sqrt{n}$.
Using the master theorem:

$a = 2, b = 2$

$c = 1$

$\sqrt{n} = O(n^{(1-e)}) \rightarrow$

$T(n) = O(n)$

MC questions

The quicksort algorithm is a divide and conquer method as follows:
divide step: select a random element *x* from the array. Divide the array to two sub-arrays such that all elements of one of them are less than *x* and all elements of the other are greater than *x*.
Conquer step: attach the two sorted subarrays

What is the worst case complexity of the quicksort for an array of size n:
a) n
b) n log n
c) n^2/3
d) n^2

MC questions

The quicksort algorithm is a divide and conquer method as follows:
divide step: select a random element *x* from the array. Divide the array to two sub-arrays such that all elements of one of them are less than *x* and all elements of the other are greater than *x*.
Conquer step: attach the two sorted subarrays

Divide step iterates through the array once → O(n)

MC questions

The quicksort algorithm is a divide and conquer method as follows:
divide step: select a random element *x* from the array. Divide the array to two sub-arrays such that all elements of one of them are less than *x* and all elements of the other are greater than *x*.
Conquer step: attach the two sorted subarrays

Divide step iterates through the array once → O(n)

Let *i* be the index of *x* in the sorted version of the array then the recurrence equations is:

$T(n) = T(n-i-1) + T(i-1) + O(n)$

MC questions

The quicksort algorithm is a divide and conquer method as follows:
divide step: select a random element *x* from the array. Divide the array to two sub-arrays such that all elements of one of them are less than *x* and all elements of the other are greater than *x*.
Conquer step: attach the two sorted subarrays

Divide step iterates through the array once → O(n)

Let *i* be the index of *x* in the sorted version of the array then the recurrence equations is:

T(n) = T(n-i-1) + T(i-1) + O(n)

Worst case: i = 1 → T(n) = T(n-2)+ O(n) → T(n) = O(n^2)

MC questions

The solution to the recurrence relation $T(n) = 8\ T(n/4) + O(n^{1.5} \log n)$ by the Master theorem is:

a) $O(n^2)$
b) $O(n^2 \log n)$
c) $O(n^{1.5} \log n)$
d) $O(n^{1.5} \log^2 n)$

MC questions

The solution to the recurrence relation T(n) = 8 T(n/4) + O(n^1.5 log n) by the Master theorem is:

a) O(n^2)
b) O(n^2 log n)
c) O(n^1.5 log n)
d) O(n^1.5 log^2 n)

a= 8, b = 4 $\rightarrow$ c = 3/2 = 1.5
(n^ 1.5) ? (n^1.5 log n)

MC questions

The solution to the recurrence relation T(n) = 8 T(n/4) + O(n^1.5 log n) by the Master theorem is:

a) O(n^2)
b) O(n^2 log n)
c) O(n^1.5 log n)
d) O(n^1.5 log^2 n)

a= 8, b = 4 $\rightarrow$ c = 3/2 = 1.5
(n^ 1.5) and (n^1.5 log n) are not polynomially different

MC questions

The solution to the recurrence relation $T(n) = 8\,T(n/4) + O(n^{1.5} \log n)$ by the Master theorem is:

a) $O(n^2)$
b) $O(n^2 \log n)$
c) $O(n^{1.5} \log n)$
d) $O(n^{1.5} \log^2 n)$

$a = 8, b = 4 \rightarrow c = 3/2 = 1.5$
$(n^{1.5})$ and $(n^{1.5} \log n)$ are not polynomially different

Case 2 $\rightarrow$ $T(n) = n^{1.5} \log n ^ 2$

Q1

Emily has received a set of marbles as her birthday gift. She is trying to create a staircase shape with her marbles. A staircase shape contains k marbles in the k th row. Given n as the number of marbles help her to figure out the number of rows of the largest staircase she can make.

A staircase of size 4:

```
*
**
***
****
```

Solution

The size of the maximum staircase is a number in [1, 2, …., n]

Solution

The size of the maximum staircase is a number in [1, 2, …., n]

The number of marbles in a staircase of size m is:

$$f(m) = 1 + 2 + 3 + \ldots + m = m(m+1) / 2$$

Solution

The size of the maximum staircase is a number in [1, 2, …., n]

The number of marbles in a staircase of size m is: $f(m) = m(m+1) / 2$

If k is the size of largest staircase using n marbles:

$$f(k) <= n < f(k+1)$$

Solution

The size of the maximum staircase is a number in [1, 2, …., n]

The number of marbles in a staircase of size m is: m(m+1) / 2

We are trying to find the k such that f(k) <= n < f(k+1)

We perform a binary search on [1, 2, .., n] to find such k.

Solution

We perform a binary search to find such k.

We start the process of searching with k=n/2 and at every step:

- If g(k) >= n then we perform binary search in the 1…k array
- If g(k) < n then we perform binary search in the k, ….n array

Solution

We perform a binary search to find such k.

We start the process of searching with k=n/2 and at every step:

- If g(k) >= n then we perform binary search in the 1…k array
- If g(k) < n then we perform binary search in the k, ….n array

Time complexity:

$T(n) = T(n/2) + O(1) \rightarrow T(n) = O(\log n)$

Dynamic Programming

(T/F) Dynamic Programming approach only works on problems with non-overlapping subproblems.

False, Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into sub problems using recursion and storing the results of sub problems to avoid computing the same results again.

(T/F) The difference between dynamic programming and divide and conquer techniques is that in divide and conquer sub-problems are independent.

True, divide and conquer sub-problems are independent and DP uses solutions from the previous sub problems.

(T/F) In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems.

False, Consider the Dynamic Programming solution for Fibonacci Sequence, it just requires two variables to store the recently computed sub problem values. (Using Space Optimization)

Q3

USC students get a lot of free food at various events. Suppose you have a schedule of the next **n** days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the EVK Dining Hall for $7 . Alternatively, you can purchase one week's groceries for $21, which will provide dinner for each day that week. However, as you don't own a fridge, the groceries will go bad after seven days and any leftovers must be discarded. Due to your very busy schedule (midterms), these are your only three options for dinner each night.

Solution

If tonight is the last night, and there's free food, then OPT(n) = OP T (n − 1), because we can survive optimally until last night and then get free food tonight. Otherwise, we need to either go to the EVK Dining Hall or use the last of our groceries, as there's no sense leaving groceries here afterward. Accordingly, OPT(n) for nights without free food is the smaller of OPT(n−1) + $7 or OPT(n−7) + $21.

The base case is OPT(0) = 0, as it is for any OPT(n < 0) in the recursive formulation. When we fill this in iteratively, we will instead let the parameter to the second case be the larger of 0 or n−7 to avoid out-of-bounds exceptions, especially if our language doesn't permit negative array indices (many don't).

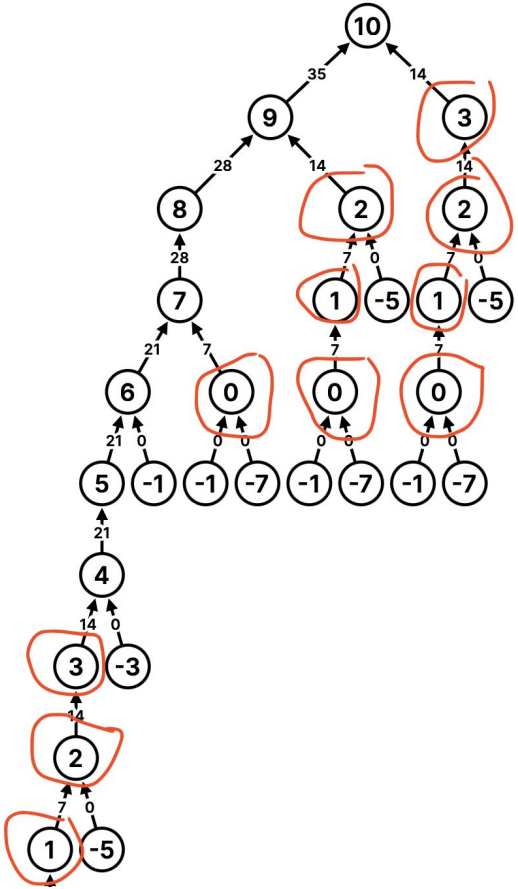
Draw recursion tree to observe overlapping subproblems

Schedule of free dinner as a boolean array:

[False, True, False, True, False, True, False, False, True, False, False]

Number of days = 11

(0-based indexing so calling on OPT(10)



For large values of n, the number of overlapping subproblems are significantly huge

Solution (continued) Bottom-Up Approach

Because each case requires only smaller parameter values as prerequisites to solving, we can fill it in increasing order:

OPT[0] = 0, OPT[< 0] = 0

for i = 1 → n do

    if free food on night i then

        OPT[i] = OPT[i − 1]

    else (visit dining hall / buy groceries)

        OPT[i] = min(OPT [i − 1] + $7 or OPT[max(0,i−7)]+$21)

The minimum amount to spend will be stored at OPT[n].

The run-time of the algorithm is linear in n, i.e., polynomial with degree 1.

Heaps and Amortised Cost Analysis

Question:

If a binomial heap contains these three trees in the root list: B0, B1, and B3, after 2 DeletetMin operations it will have the following trees in the root list.

a) B0 and B3

b) B1 and B2

c) B0, B1 and B2

d) None of the above

Solution:

Observations:

- A binomial tree B_p has 2^p elements.
- A binomial heap of n elements (in total) is always unique (because binary representation of a decimal number is unique).
- An idea: If you know binary representations of total number of elements in the binomial heap, you can determine exactly which binomial trees are present in the heap, and vice versa.

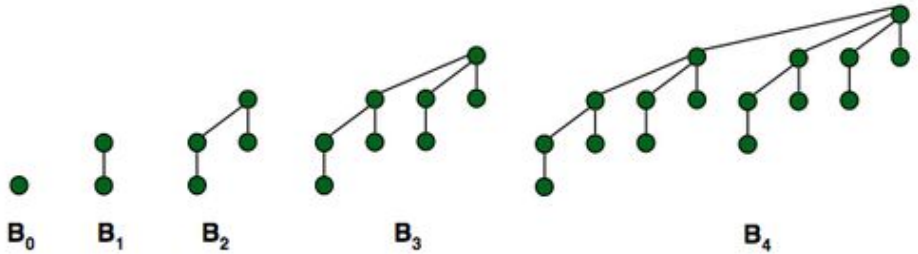


Image source: https://media.tumblr.com/tumblr_mcxot8jU371rzq63x.png

Solution:

Binary representation:

Rearrange B0, B1, and B3 => B3, B1, and B0 => 1011

Total number of elements before any deletion = 2^3 + 2^1 + 2^0 = 8+2+1 = 11

Number of elements after 2 deleteMins = 11-2 = 9

Representing 9 in binary, we get: 1001

Therefore, we would have B0 and B3 trees in the heap.

Question:

An ordered stack is a singly linked list data structure that stores a sequence of items in increasing order. The head of the list always contains the smallest item in the list. The ordered stack supports the following two operations. POP() deletes and returns the head (NULL if there are no elements in the list). PUSH(x) removes all items smaller than x from the beginning of the list, adds x and then adds back all previously removed items. PUSH and POP can only access items in the list starting with the head. What would be the amortized cost of PUSH operation, if we start with an empty list? Use the aggregate method.

Solution

The worst sequence of operations is pushing in order. Assume we push 1,2,3, ..., n, starting with an empty list. The first push costs 1. The second push costs 3 (pop, push(2), push(1)). The last push costs 2n-1 (n-1 pops followed by push(n), followed by n-1 pushes). The total cost is given by:

T(n) = 1 + 3 + 5 + ... +(2n-1) = O(n^2). The amortized cost is T(n)/n = O(n).

Question:

The United States Commission of Southern California Universities (USCSCU) is researching the impact of class rank on student performance. For this research, they want to find a list of students ordered by GPA containing every student in California. However, each school only has an ordered list of its own students by GPA and the commission needs an algorithm to combine all the lists. There are a few hundred colleges of interest, but each college can have thousands of students, and the USCSCU is terribly underfunded so the only computer they have on hand is an old vacuum tube computer that can do about a thousand operations per second. They are also on a deadline to produce a report so every second counts. Find the fastest algorithm for yielding the combined list and give its runtime in terms of the total number of students (m) and the number of colleges (n).

Solution

Given:

n: Total number of colleges (we have n different arrays)

m: Total number of students in all of n arrays combined.

Thoughts?

- n-way merge - slow - bad! O(m*n)
- using heaps - faster - good! O(m*logn)

Solution

- Use a min heap H of size n.
- Use an Array named CombinedSort[1,.....m] to store the combined result (our final answer!)
- Use another array named CP[1,....,n] store the current pointer locations of n different arrays.

Solution

Insert the first elements of each sorted array into the heap. The objects entered into the heap will consist of the pair (GPA, college ID).

Set pointers into all n arrays to the second element of the array CP(j) = 2 for j=1 to n.

Loop over all students (i= 1 to m) —> loop runs m times

S = ExtractMin(H) —>O(logn)

CombinedSort (i) = S.GPA —> O(1)

j = S. collegeID —> O(1)

Insert element at CP(j) from college j into the heap —> O(logn)

Increment CP(j) —> O(1)

endloop

Total runtime complexity - O(mlog n)

Greedy Algorithm

T/F questions

1. A greedy algorithm always makes the choice that looks best at the moment.

T/F questions

1. A greedy algorithm always makes the choice that looks best at the moment.

   Answer: True

Multiple Choices Question

Which of the following is a Greedy algorithm?

1. Kruskal’s algorithm
2. Bellman-Ford algorithm
3. Prim’s algorithm
4. Dijkstra’s algorithm

Multiple Choices Question

Which of the following is a Greedy algorithm?

1. Kruskal’s algorithm
2. Bellman-Ford algorithm
3. Prim’s algorithm
4. Dijkstra’s algorithm

1,3,4

Bellman-Ford algorithm is based on dynamic programming

Greedy: 2 Common Methods of Proof of Optimality

- **Stay-ahead:** an inductive method to show that for each step you take, you are always ahead of any other solution according to some metric

- **Exchange:** show that for your greedy solution, any swaps between steps/elements cannot achieve a better result

Practice Exam Question

You have a finite set of points $P = (p_1, p_2, p_3, ..., p_n)$ along a real line. Your task is to find the smallest set of intervals, each of length 2, which would contain all the given points.

Example (may or may not be an optimal solution/smallest set): Let $X = 1.5, 2.0, 2.1, 5.7, 8.8, 9.1, 10.2$. Then the three intervals [1.5, 3.5], [4, 6], and [8.7, 10.7] are length-2 intervals such that every element is contained in one of the intervals.

Device a greedy algorithm for the above problem and argue that your algorithm correctly finds the smallest set of intervals.

Practice Exam Question

Algorithm: Sort P and call it S. Initialize T as an empty set. While S is not empty, select the smallest p from S, add $[p, p+2]$ into T, and remove all elements within $[p, p+2]$ from S. At last, return T as the answer.

Practice Exam Question

Proof:

1) Statement: Intervals in our solution are never to the left of the corresponding intervals in the optimal solution. We can prove this by induction. Let the nth interval in our solution be t_n, the nth interval in the optimal solution be t'_n, and $p^{(n)}$ be the starting point of t_n.

Base Case: When $n = 1$, we pick $p^{(1)}$, the smallest point in S. To cover $p^{(1)}$, the rightmost starting point of the first interval is $p^{(1)}$. Thus, our first interval t_1 could not be to the left of t'_1 in the optimal solution. The base case holds.

Induction Hypothesis: Assume the statement holds for the first $n = k$ intervals.

Show $n = k + 1$ holds: Our t_{k+1} starts at $p^{(k+1)}$ and $p^{(k+1)} \notin t_k$. by induction hypothesis, we know $p^{(k+1)} \notin t'_k$ as well. To cover $p^{(k+1)}$, the rightmost starting point is $p^{(k+1)}$, so t_{k+1} can not be to the left of t'_{k+1}. Hence, the statement holds for $n = k + 1$.

Practice Exam Question

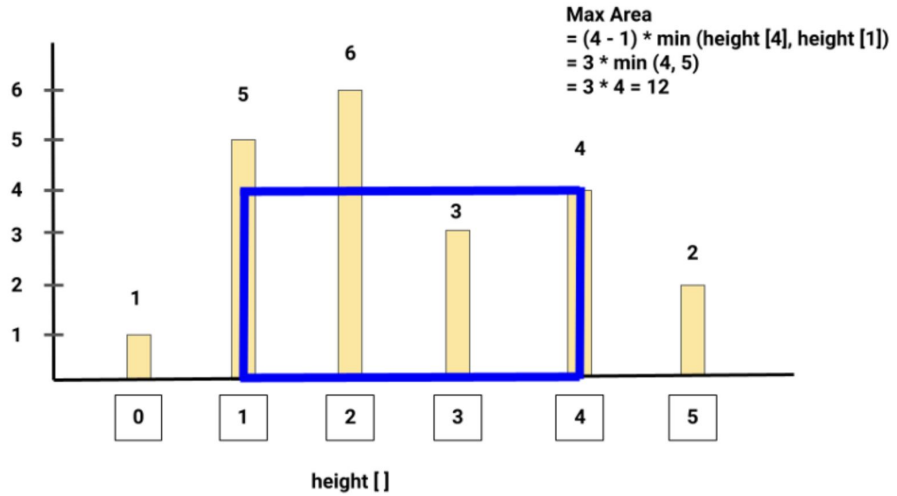
2) Statement: Our solution has m intervals and the optimal solution O has fewer intervals. We will prove that this statement is not possible by contradiction. We look at the first point x which is not covered by t_{m-1}. Using the statement in (1), x will also not be covered in the t'_{m-1}. O which has at most $m-1$ intervals doesn't cover x. This contradicts to O is the optimal solution. Thus, the optimal solution can not contain fewer intervals than our solution.

With (1) and (2), we show our algorithm returns the optimal solution.

Extra Question

You are given an integer array with distinct heights of length n. There are n vertical lines drawn such that the two endpoints of the i^{th} line are (i, 0) and (i, height[i]).
Find two lines that together with the x-axis form a container, such that the container contains the most water.

- Calculate *the maximum amount of water a container can store*.



Extra Question

Intuition

- Be greedy
- Start with the two endpoints. As larger the distance between two lines, more water it can hold.
- At each moment, we either move left or right depending on which line is smaller in height.

Extra Question

Algorithm:

Two pointer approach.

```
left = 0
right = n
water =0

While left<right
    water = max(water,min(height[left],height[right]) * (right-left))
    if(height[left]<height[right])
        left++
    else
        right - -
```

Maximize the value of water

Time complexity ?

O(n)

Extra Question

We can prove this using an **exchange argument.** Let us consider that there exists an optimal solution O that does NOT choose the smaller line at every point. Let us consider left and right to be two lines. Let W be the water capacity of these two lines.

Therefore W(left,right) = (right-left) * min(height[left] , height[right])

There are two cases

Case 1: height[left] < height[right]
So W(left,right) = (right-left) * height[left]

In this case, there are two possibilities, optimal solution can go either left+1 or right-1. If the optimal solution O chooses left+1, then it is same as our solution. **However, optimal solution O may choose right-1.**

If height[left] > height[right-1]:

W(left,right-1) = (right-1-left)* min(height[left],height[right-1]) = (right-1-left) * height[right-1]

<= (right-1-left) * height[left]

Therefore

W(left,right-1) <=(right - left) * height[left]

i.e. W(left,right-1) <= W(left,right)

Same is true for W(left,right-2), W(left,right-3)....

Extra Question

If height[left] < height[right-1]:

W(left,right-1) = (right-1-left)* min(height[left],height[right-1]) = (right-1-left)*height[left]

<= (right-1-left) * height[left]

Therefore

W(left,right-1) <= (right - left) * height[left]

i.e. W(left,right-1) <= W(left,right)

Same is true for W(left,right-2), W(left,right-3)....

In this case, in either height[left] > height[right-1] or height[left] < height[right-1], W(left,right-1) <= W(left,right), so there is no need of calculating W(left,right-1). The optimal solution explores worse values than ours so our solution is optimal. Case 2 (height[left] > height[right]) can be proven similarly.