COMP 3711

Midterm review

Course Intended Learning Outcomes

1. Describe fundamental concepts and techniques for determining the asymptotic behavior of real-valued functions defined in natural numbers.
2. Explain recurrence equations and solve common recurrences using a variety of techniques.
3. Analyze an algorithm described in plain language or some form of pseudocode in terms of its time (or space) efficiency as a function of the size of a problem instance.
4. Explain how various data structures, including trees, heaps and disjoint set structures, influence the time efficiency of algorithms.
5. Apply general algorithmic design and analysis techniques to solving problems, including greedy, divide-and-conquer and dynamic programming.
6. Identify randomization in algorithms and analyze basic randomized algorithms such as randomized quicksort and selection.

1. Describe fundamental concepts and techniques for determining the asymptotic behavior of real-valued functions defined in natural numbers.

- Definitions of O, Ω, Θ
- Given a function f(n), determine f(n) = Θ(?)
  - ignore lower order terms
  - ignore the constant coefficient of the most significant term (what's a constant?)
- Given two functions f(n), g(n), determine whether f(n) = O(g(n)), f(n)= Ω(g(n)), or f(n) = Θ(g(n))
  - Formal definition
  - Quick rules: $\log^c n < n^c < c^n$
  - How about functions involving two parameters, like n and k?
    - Two such functions may not always be comparable.
- Common summations (see background sheet)

2. Explain recurrence equations and solve common recurrences using a variety of techniques.

- Recurrences naturally arise in
  - Analysis of divide-and-conquer algorithms
  - Induction proofs
- Recursion tree method
- Master theorem

3. Analyze an algorithm described in plain language or some form of pseudocode in terms of its time (or space) efficiency as a function of the size of a problem instance.

- Default: Worst-case
  - What does it mean that the worst-case running time of an algorithm is $O(n \log n)$, $\Omega(n \log n)$, or $\Theta(n \log n)$?
  - Others: expected case vs average case
- Simple analysis
  - Single/double loops: insertion sort, counting sort
  - Iterative algorithms: radix sort
- Divide and conquer algorithms: Use recurrence

4. Explain how various data structures, including trees, heaps and ~~disjoint set structures~~, influence the time efficiency of algorithms.

- List, array, stack, queue
- Priority queue, heap
  - Heapify: O(log n)
  - Build-heap: O(n)
  - Extract-Min / Insertion / Decrease-Key / Increase-key: O(log n)

5. Apply general algorithmic design and analysis techniques to solving problems, including greedy, divide-and-conquer and ~~dynamic programming.~~

- Divide into two halves, recursively solve both, then combine
  - Mergesort
  - Counting inversions
  - Maximum contiguous subarray
- Divide into two halves, recursively solve one
  - Binary search
  - Local minimum (homework question)
- Divide into two halves, recursively solve more than two subproblems
  - Integer multiplication
  - Need to know how to analyze such an algorithm
  - Will not ask you to design such an algorithm on the exam

Greedy Algorithms

- General idea: make the choice that looks best now without thinking too much about the future
  - Sometimes it produces the optimal solution
- Examples
  - Interval scheduling
  - Interval partitioning
  - Fractional Knapsack
  - Huffman coding (correctness proof not required)
  - Many graph algorithms (later)
- The algorithms are usually very simple
  - But need to prove correctness!

Solution format

- The algorithm (language and/or pseudo code)
- Correctness proof (no fixed technique, the following is the most common one)
  - Let G be the greedy solution, O the optimal solution
  - Find one (usually the first or the last) difference between G and O.
  - Make O more similar to G by removing that difference, while making sure that the quality of O does not decrease (it can't increase either since O is already optimal).
  - Repeatedly applying the above transformations until G and O are the same
- Running time (usually easy)

6. Identify randomization in algorithms and analyze basic randomized algorithms such as randomized quicksort and selection.

- Basic probability theory
  - Probability distribution, expectation, independence,
  - Linearity of expectation
- Expected running-time for randomized algorithms
  - Expectation is only taken over the internal random choices, input can still be the worst
- The indicator random variable technique
  - The hiring problem, card guessing, waiting time, birthday paradox, coupon collector
  - Analysis of quicksort
  - Analysis of randomized selection
  - Analysis of bucket sort (average-case analysis)
- How to generate a random permutation

Other issues

- Ω(n log n) sorting lower bound
  - Understand what is a comparison-based algorithm
  - The decision-tree model
  - The lower bound proof
- NOT required:
  - The O(n)-time algorithm for maximum contiguous subarray
  - Matrix multiplication
  - Correctness proof of Huffman coding

About the Midterm Exam

- The midterm is on: 22-Oct (Thu) 19:00-21:00 in CTY G009A+B and 010
- A makeup session is scheduled at 17:00-19:00 in CYT G009B
  - Send me an email if you need to take the makeup session
- Coverage: Everything from the beginning to greedy algorithms.
- It will be a closed-book exam, but the math background sheet will be provided.
- You are recommended to use pencils (so that you can easily correct possible mistakes)
- Calculators or other electronic means are not allowed (or needed).

Rotated sorted array

- Suppose you are given a sorted array A of n distinct numbers that has been rotated k steps, for some unknown integer k between 1 and $n - 1$. That is, A[1..k] is sorted in increasing order, and A[k + 1..n] is also sorted in increasing order, and A[n] < A[1]. The following array A is an example of n = 16 elements with k = 10.

  A = [9, 13, 16, 18, 19, 23, 28, 31, 37, 42, 0, 1, 2, 5, 7, 8].

(a) Design an O(log n)-time algorithm to find the value of k.

(b) Design an O(log n)-time algorithm that for any given x, finds x in the array, or reports that it does not exist. [You may do part (b) assuming you have solved part (a), even if you can't.]

O(log *n*) algorithm to find the value of *k*

Algorithm 1 Find-k(A, p, q)
m ← ⌊p+q/2⌋
if A[m + 1] < A[m] then return m
if A[m] ≥ A[1] then
    return Find-k(A, m, n)
    else return Find-k(A, 1, m − 1)
end if

The depth of recursion is O(log *n*) and it takes constant time for each recursion, so total cost is O(log *n*).

O(log *n*) algorithm that finds x

Find-x(A, x)

k ← Find-k(A, 1, n)

If x ≥ A[1] then return BinarySearch(A, 1, k, x)

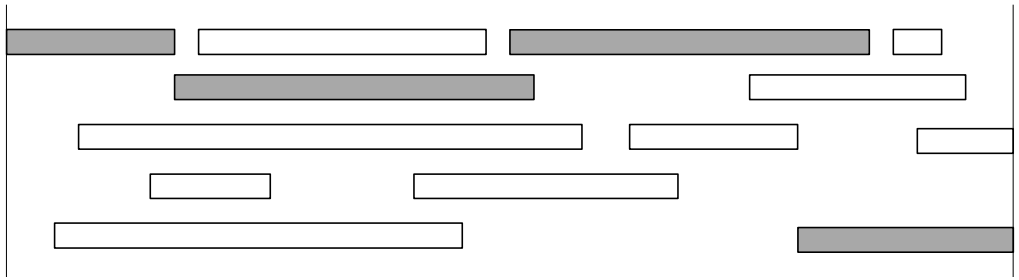
Else return BinarySearch(A, k + 1, n, x)

end if

Interval covering

A company wants to build cell phone base stations to cover a highway. However, not every location is allowed to build, and the government has designated a certain number of locations. Depending on how far the location is from the highway, its coverage on the highway also varies. Thus, this problem can be generally modeled as follows. The highway can be thought of as the real line from 0 to 1, and each location covers a certain interval $[x, y] \subseteq [0, 1]$ if a base station is built there. You are given a total of *n* such intervals and are guaranteed that all of them together cover [0, 1] (otherwise there is no solution to the problem). Design an algorithm that finds the minimum number of intervals that together cover [0, 1].

The figure shows an example where an optimal solution consists of the 4 shaded intervals (the optimal solution may not be unique). The intervals are given as two arrays $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ where the i-th interval is $[x_i, y_i]$. If you use a greedy algorithm, you must prove that it is correct.



Greedy algorithm

Choose the interval that covers 0 and has largest y_i . Then iteratively choose the interval that overlaps the already covered part with the largest y_i , until we reach 1. Naively picking such an interval takes $O(n)$ time, leading to $O(n^2)$ time in total. Below is a more efficient implementation of this greedy idea.

Interval-Covering
S = ∅, covered = 0;
sort the intervals by x_i such that $x_1 \le x_2 \le \cdots \le x_n$;
i ← 1;
while covered ≤ 1 **do**
    starting from the i-th interval, scan to find the last j such that $x_j \le$ covered;
    from the i-th interval to the j-th interval, find k such that y_k is maximized;
    S ← S ∪ {k}, covered ← y_k ;
    i ← j + 1;

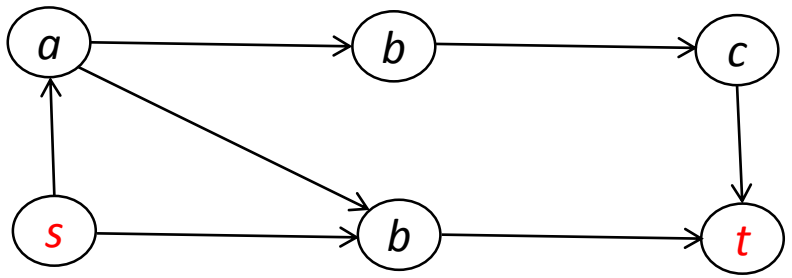
In the above algorithm, sorting takes $O(n\log n)$ time and the greedy loop takes $O(n)$ time because every interval is only examined once. So total running time is $O(n\log n)$.

Correctness proof

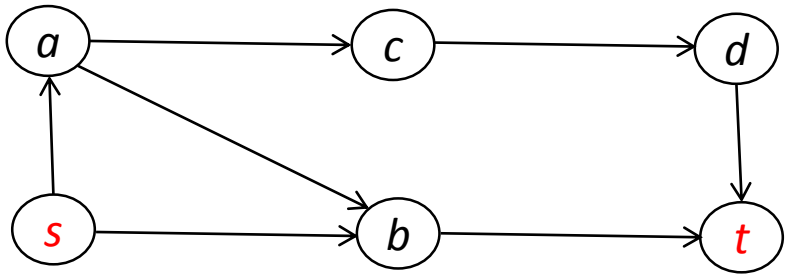
- Let G be the solution returned by this greedy algorithm, and O be an optimal solution. Consider the first interval where O is different from G. Suppose the interval chosen by G is $[x_i, y_i]$ and the one chosen by O is $[x_j, y_j]$. We must have both $x_i \leq$ covered and $x_j \leq$ covered, and by the greedy choice, $y_i \geq y_j$.
- Now we modify O by changing $[x_j, y_j]$ with $[x_i, y_i]$. This must still cover the interval [0, 1].
- Repeatedly applying this transformation will convert O into G. Thus G is also an optimal solution.

Random routing

- We can model a computer network as a directed graph, where each node is a computer and a directed edge (u, v) means that u can send messages to v. Ideally, messages should be routed along the shortest path, but this information may not be available since no node has a global view of the entire network. Thus, decisions have to be made by each node locally. Here we just consider one of the simplest routing protocols: RANDOM , where each node simply randomly chooses an outgoing link, and forwards the message along that link, hoping the message can find its way eventually.
- Consider the graph G1 below. Suppose we have a message at the source node *s* intending to reach destination *t*. Using RANDOM, s will forward the message to *a* or *b*, each with probability 1/2. If *b* gets it, it will forward the message to *t* in the next step, as *t* is its only outgoing neighbor. If *a* gets the message, it will choose *b* or *c* in the next step, each with probability 1/2, and so on so forth.



Random routing



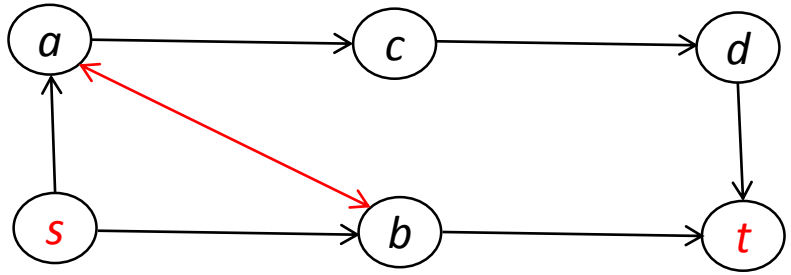
- On graph G1 , what's the probability that RANDOM indeed chooses the shortest path from *s* to *t* to route the message?

  The shortest path from *s* to *t* is (*s*, *b*, *t*). Pr(*s* → *b* → *t*) = 1/2.

- On graph G1 , what's the expected length (in terms of the number of edges) of the path chosen by RANDOM?

  There are 3 possible paths: Pr(*s* → *a* → *c* → *d* → *t*) = 1/4 , Pr(*s* → *a* → *b* → *t*) = 1/4, Pr(*s* → *b* →*t*) = 1/2 . So E[length of the path chosen randomly] = 4·1/4 + 3·1/4 + 2·1/2

Random routing



- On graph G2 , what's the probability that RANDOM indeed chooses the short- est path from s to t to route the message?

The shortest path from s to t is (s, b, t). Pr($s \rightarrow b \rightarrow t$) = 1/4.

- On graph G2 , what's the expected length of the path chosen by RANDOM?

Let X(e) be the number of times edge e appears on the randomly chosen path. By linearity of expectation, the expected total length is $E[\sum_e X(e)]$. For (s,a) and (s,b), each of them appears once with probability 1/2 and 0 times with probability 1/2, so E[X(s,a)] = E[X(s,b)] = 1/2. Because the random path exits from a or b with equal probability, we also have E[X(a,c)] = E[X(c,d)] = E[X(d,t)] = E[X (b,t)] = 1/2. It only remains to compute E[X(a,b)]. This is the same as the waiting time problem with success probability p = 1/2, except that we do not count the success coin. So E[(a,b)] = $1/p - 1 = 1$. Summing up all these expectations gives that the expected total length of the randomly chosen path is 4.