Introduction to Algorithms | December 17, 2015
Massachusetts Institute of Technology | 6.006 Fall 2015
Instructors: Piotr Indyk and Ron Rivest | Final Exam

Final Exam

- Do not open this exam booklet until directed to do so. Read all the instructions on this page.
- When the exam begins, write your name on every page of this exam booklet.
- You have **180 minutes to earn 150 points.** Do not spend too much time on any one problem. Read them all first, and attack them in the order that allows you to make the most progress.
- **You may have three** $8.5'' \times 11''$ **double-sided cheat sheets**. No calculators, cell phones, programmable devices, or other communications devices allowed.
- Write your solutions in the space provided. If you need more space, use **the back of the sheet** containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. Simply cite them.
- Throughout the exam, **unless otherwise specified in the problem**:
  - You can utilize the Simple Uniform Hashing Assumption (SUHA).
  - You may assume that all input graphs are simple graphs given as adjacency lists.
  - It is OK if your algorithms work with "high probability".
- When writing an algorithm, a **clear description in English** will suffice. Pseudo-code is not required unless you find that it helps with the clarity of your presentation.
- Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points	Grade	Grader
0	2	2		
1	19	38		
2	10	30		
3	1	15		
4	1	20		
5	1	20		
6	1	25		
Total		150		

Name: ______________________________

Circle your recitation:

R01	R02	R03	R04	R05	R06	R07	R08	R09
Ilya	Anak	Alex	Szymon	Alex	Joe	Alex	Shalev Ben	Matthew
R	Y	Jaffe	Sidor	Jaffe	Paggi	Chen	David	Chang
10AM	**10AM**	**11AM**	**11AM**	**12PM**	**12PM**	**1PM**	**2PM**	**3PM**

Problem 0. What is Your Name? [2 points] (2 parts)

(a) Flip back to the cover page. Write your name and circle your recitation section.

(b) Write your name on top of each page.

Problem 1. True or False [38 points] (19 parts)

For each of the following questions, circle either T (True) or F (False). There is **no need to justify the answers;** you may include a remark regarding your interpretation of the question, or an assumption you made while answering it, but these should not be necessary, and it is better to ask a TA during the exam for clarification if necessary. The graders may ignore such remarks and assumptions. **Each correct answer is worth 2 points.**

(a) T F $2^{1/n} = \Theta(1)$ assuming $n \geq 1$.

Solution: True. $1 < 2^{1/n} \leq 2$ for any $n \geq 1$.

(b) T F Consider the recurrence $T(n) = 16T(n/2) + 8n^4$, with base case $T(n) = 1$ for $n \leq 1$. The asymptotic solution to this recurrence is $T(n) = \Theta(n^4)$.

Solution: False. By case 2 of the master theorem, $T(n) = \Theta(n^4 \log n)$.

(c) T F Suppose you have a sorted list of n numbers, to which you add 3 extra numbers in arbitrary places. One can sort this list in $O(n)$ time.

Solution: True.

(d) T F There is a comparison-based sorting algorithm that sorts n items using $O(n(\log\log n)^2)$ comparisons.

Solution: False. (Should be easy)

(e) T F A search can be performed on an AVL tree with n^{10} elements in $O(\log n)$ time.

Solution: True.

(f) T F For this problem, please use the simple uniform hashing assumption (SUHA). Suppose we have a hash table in which we resolve collisions using chaining. However, instead of using linked lists, we will use balanced binary search trees instead. Then, the expected runtime of checking whether the hash table contains a certain element is $\Omega(1 + \alpha \log \alpha)$.

Solution: False. It can't be worse than $\Theta(1 + \alpha)$, because we need $O(1)$ to perform the hash and at expected worst case $\Theta(\alpha)$ to search through the BBST. The actual runtime is around $\Theta(1 + \log \alpha)$. Each tree takes $\Theta(\log \alpha)$ to search through.

(g) T F There exists a comparison-based data structure maintaining a set of numbers that supports insertions of elements in $O(1)$ time and returns the minimum element in the structure at any time in $O(1)$ time.

Solution: True. Just maintain the minimum element since we do not require any deletion (or extract-min).

(h) T F Given a string s, finding the longest string w that appears at least k times as a substring of s can be done in $O(|s|^2)$ time with high probability. Note that the k occurrences of the substring do not have to be disjoint (i.e. they can overlap).

Solution: True.

(i) T F It is possible to compute the square root of a number to D digits of accuracy using $O(\log D)$ iterations of Newton's method.

Solution: True.

(j) T F In the undirected, unweighted graph G, the shortest path between u and v has length 3, and the shortest path between u and w has length 5. We can conclude that there is no edge between v and w.

Solution: True.

(k) T F Given a weighted undirected graph $G = (V, E)$. For any starting node s, in order to find a path of least total weight from s to any node in G, it suffices to apply Bellman-Ford algorithm with only $|V|/2$ iterations.

Solution: False, it is actually $|V| - 1$ iterations.

(l) T F The triangle inequality, when applied to the shortest path distances over a graph $G = (V, E)$ with the weight function $w : E \to \mathbb{R}$, states that for any three nodes a, b, and c, we have that $w(a, c) \leq w(a, b) + w(b, c)$.

Solution: False. The triangle inequality refers to the actual shortest path distances, not the edge weights.

(m) T F Given a weighted directed graph G with no *positive-weight* cycles, the *longest* (heaviest-weight) paths from a source node s to all reachable nodes can be computed with Dijkstra's algorithm after negating all the edge weights.

Solution: False. There can originally be edges with positive weights on G, so the new weights can be negative. Dijkstra's algorithm does not support negative weights.

(n) T F If a directed graph G contains negative-weight edges but no negative-weight cycles, then the Bellman-Ford algorithm can be used to compute the shortest path between any two nodes.

Solution: True. Bellman-Ford can handle a graph with negative weight edges.

(o) T F Given a weighted undirected graph $G = (V, E)$ with no negative weights, such that the degree (the number of incident edges) of each node is exactly 6, the shortest paths lengths from a source node s to all reachable nodes can be computed in $O(V \log V)$ time.

Solution: True. Since $|E| = \Theta(V)$ and there are no negative weights, Dijkstra's algorithm takes time $O(E \log V) = O(V \log V)$.

(p) T F One can find the shortest path distance between two nodes in an unweighted graph of maximum degree d in time $O(d^\ell)$, where ℓ is the distance between the two nodes.

Solution: True. Just use BFS and terminate when you find ℓ. The maximum number of nodes that will be searched is $O(d^\ell)$.

(q) T F Given a weighted directed graph G with positive edge weights, bidirectional Dijkstra's algorithm always correctly computes a shortest (least-weight) path from a source node s to a destination node t (assuming such a path exists).

Solution: True. We proved that in class.

(r) T F In order for dynamic programming to be applicable, the subproblem dependence graph for a recursive approach must form a rooted tree.

Solution: False. The only requirement is that the subproblem dependence graph be acyclic.

(s) T F When using Longest Common Subsequence algorithm to recover an optimal solution (as opposed to only computing its length), the computational complexity becomes worse than $O(n^2)$, where n is the length of each input string.

Solution: False.

Problem 2. Algorithmic Techniques [30 points] (10 parts)

For each of the following problems, indicate which of the provided algorithms or algorithmic techniques you would choose in order to solve it **as quickly as possible**. Select **only one** answer for each problem by circling the number in front of your answer. There is no need to justify the answers. **Each correct answer is worth 3 points.**

(a) Tweedledum and Tweedledee have two pet monkeys which enjoy typing random stuff on keyboards. The monkeys are identical twins, and the sequences of letters they type are almost identical. However, the monkeys occasionally hit different keys, or miss some keys altogether. They ask you for an efficient algorithm to identify those differences.

1. Binary Search Trees

2. Dynamic Programming

3. Numerics

Solution: Dynamic Programming.

(b) You have a weighted directed complete graph such that

$$w(u, v) = -w(v, u)$$

for all vertices u and v. (The weight of the edge from u to v is the negative of the weight of the edge from v to u.) Which algorithm would you use to find a shortest path from specified vertex s to specified vertex t (if one exists)?

1. DFS

2. Dijkstra's Algorithm

3. Bellman-Ford Algorithm

Solution: Bellman-Ford.

(c) A search engine called *Poodle* asked you to implement a system that, given a word, lists all web pages stored in Poodle's computers' memory containing that word.

1. Heap Sort

2. Binary Search Trees

3. Hashing

Solution: Hashing

(d) Given a directed acyclic graph and a source node s, find the number of paths from s to each other node in the graph. Here two paths are defined to be different if there is at least one edge they do not share.

1. Dynamic Programming

2. Sorting

3. Graph Transformation, then Bellman-Ford algorithm

Solution: Dynamic Programming. Subproblems: $N(v, k) =$ number of paths from s to v with length at most k. $N(v, k) = \sum_{u \in v.\pi} N(u, k-1)$.

(e) You want to multiply two large integers together. These integers are given in binary.

1. Divide-and-Conquer

2. Newton's Method

3. Dynamic Programming

Solution: Divide-and-Conquer – use Karatsuba algorithm or some better variations. Better techniques not covered in this class should not collide with any choices above.

(f) You want to sort n English words, each of which is represented as a string of length at most 10. Pick the choice that will result in the *smallest asymptotic runtime*.

1. Heap

2. Merge Sort

3. Radix Sort

Solution: Radix Sort – $O(n)$ but others take $\Omega(n \log n)$.

(g) You need to find a signature of a virus in a large file of length n, where the signature is a string of m bytes. You would use:

1. Rolling hash

2. Divide-and-Conquer

3. Radix sort

Solution: Rolling hash.

(h) You are given the function $f(x) = x^2 - x^4 + 2$. Plotting it, you see that there is local maximum somewhere between $x = 1/2$ and $x = 1.0$. What algorithm would you use find the x-coordinate of this local maximum to 100,000 digits of precision?

1. Newton's Method

2. Binary Search

3. Repeated squaring

Solution: **Newton's Method** (on the derivative of f).

(i) Archeological excavations at MIT have uncovered a rare working copy of a ZX Spectrum computer. To satisfy your HASS requirements, you are writing a "Labirynth" computer game where the user has to find a path in a labyrinth drawn on a computer screen. You need to make sure that, for each labyrinth generated by the computer, there is a path from the starting point to the exit that avoids obstacles and whose length is at most d pixels. You need to design a fast algorithm for checking whether such a path exists. (You can model the computer screen as a 256×192 array of 0s and 1s, where 1 denotes a pixel covered by an obstacle).

1. DAG-SP (algorithm for finding shortest paths in DAGs)

2. Dijkstra's Algorithm

3. BFS

Solution: BFS

(j) To relax after the exam, you sit down to watch n cat videos. As you watch, you want to keep a list of the best d cat videos you've seen so far; you will update this list throughout your cat-video-watching binge. Each time you watch a new cat video that's better than the worst of the d you have so far, you will replace the worst of the d with the new video. You want to minimize the time it takes to make these updates. How should you store this list?

1. Hash Table

2. Heap

3. Linked List

Solution: Heap. We need a data structure that supports insert/delete and find-min, which is exactly what a Heap is.

Problem 3. Apples and Oranges [15 points]

Farmer Ben lined up n apples and n oranges, mixing them up within the same sequence. He wants to group the same type of fruits together, but due to the lack of additional space, he can only rearrange his fruits in-place via reversal of substrings (contiguous blocks).

For example, consider the string `AOAAOAAOOO`, representing an initial ordering of five apples (`A`s) and five oranges (`O`s). He may group these fruits by reversing the substrings from locations 3 thru 5, then from locations 2 thru 7:

$$\texttt{AO}\underline{\texttt{AAO}}\texttt{AAOOO} \rightarrow \texttt{A}\underline{\texttt{OOAAAA}}\texttt{OOO} \rightarrow \texttt{AAAAAOOOOO}.$$

Define the *arranging time* as the total length of the reversed substrings. (In the above example, the arranging time is $3 + 6 = 9$.)

Describe an algorithm that produces an efficient sequence of substrings to reverse that groups Ben's fruits together. Analyze the asymptotic arranging time of your algorithm's output. For full credit, *both* your algorithm's running time, and the arranging time for your algorithm's output, must be $O(n \log n)$.

Hint: Use divide-and-conquer to recursively arrange each half, then combine them. Note that you *don't need the smallest arranging time*, just an arranging time that is $O(n \log n)$.

Solution:

Let F denote the sequence of fruits, and REVERSE(i, j) denote the action of reversing $F[i \ldots j]$. Without loss of generality, we will aim to place apples to the left of the sequence, and the oranges to the right of the sequence. Consider the following procedure FRUIT-SORT$(F, low, high)$ which arranges the fruits from $F[low \ldots high]$ into the form `AA`...`AAOO`...`OO`.

```
FRUIT-SORT($F, low, high$)
1  if $low < high$
2     then $mid \leftarrow \lfloor (low + high)/2 \rfloor$
3          FRUIT-SORT($F, low, mid$)
4          FRUIT-SORT($F, mid + 1, high$)
5          $x \leftarrow$ number of oranges in $F[low \ldots mid]$
6          $y \leftarrow$ number of apples in $F[mid + 1 \ldots high]$
7          REVERSE($mid - x + 1, mid + y$)
```

Clearly when $low < high$, there is only one fruit and thus is already correctly grouped. Otherwise, this algorithm divides the problem into 2 roughly equal halves, and calls FRUIT-SORT on each respective half.

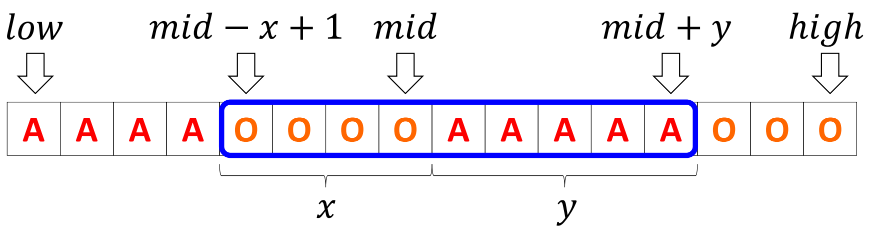


Figure 1: $F[low \dots high]$ after the conquer step

By induction, we assume that the algorithm recursively groups the fruits in each half. This yields the sequence of fruits as shown in Figure 1. One call to REVERSE (on the blue subsequence in the figure) is sufficient to group the apples to the left, and the oranges to the right, as desired. (Note that the number of apples and oranges in each half can be computed in linear time, but this does not affect the arranging time.)

As we perform only one addition REVERSE per a call to FRUIT-SORT, the arranging time incurred is bounded by the subsequence length. Thus, we obtain the recurrence $A(n) = 2A(n/2) + O(n)$ for the arranging time, with base condition $A(1) = O(1)$. This is the same recurrence as merge sort, so it has solution $A(n) = O(n \log n)$.

Problem 4. Wingardium Leviosa [20 points]

You are a magician traveling in the magical world *Graphland*, which is a graph, $G(V, E)$. Some edges, $(u, v) \in E$, are "magical" and will confer upon you unimaginable abilities if your path traverses exactly k magical edges. (If a particular magical edge is traversed more than once, it gets counted as many times as it is traversed; you could, for example, have a looping path (i.e. cycle) that traverses a single magical edge k times.) Each edge, even the magical ones, has a non-negative weight $w(u, v)$. You would like to find the shortest (least total weight) path from your current vertex s to an ending vertex t that traverses exactly k "magical" edges.

Describe an algorithm that runs in $O(kE + kV \log kV)$ time that finds the shortest path between s and t that traverses exactly k "magical" edges. (Your algorithm should also report "no such path exists" when such is the case.)

Hint: Consider transforming the original graph.

Solution: Make $k + 1$ copies of the graph, where ordinary edges stay within a copy, but magical edges move you from the ith copy to the $i + 1$st copy. To find a path from the s in the first copy to the t in the last copy, use Dijkstra on this transformed version of the input graph.

Problem 5. Card Flipping [20 points]

You have in front of you an array of n face-up cards, where each card shows an integer. (These numbers shown may be represented as an array $A[1..n]$.) You first flip over the rightmost (n^{th}) card. You then flip $n-1$ more cards, each time flipping either the leftmost unflipped card or the rightmost unflipped card (your choice). If you flip a card showing x after just having flipped a card showing y, you receive a reward of $|x-y|$.

Describe an algorithm that computes the best order in which to flip the cards, in order to maximize your total reward. Give and justify a running-time analysis for your solution.

Solution: Use Dynamic Programming. Each subproblem considers a subarray $A[i..j]$ of unflipped cards as well as the value x of the most recently flipped card.

```
def f(A, i, j, x):
  if i==j:
    return abs(x-A[i])
  else:
    return max( abs(x-A[i]) + f(A,i+1,j,A[i]),
                abs(x-A[j]) + f(A,i,j-1,A[j]) )
```

Memoize this recursive approach. This only computes the max reward; as with most dynamic programming code you can add extra record-keeping to be able to actually reconstruct the optimal solution.

The subproblem dependence graph is $\Theta(n^2)$, and the running time is also.

Solution: Define subproblems $S(i,j,0) =$ optimal score for $A[i...j]$ having previously removed $i-1$ and $S(i,j,1) =$ optimal score for $A[i...j]$ having previously removed $j+1$. We solve these subproblems for all $0 < i < n$ and $i \le j < n$ with the exception of $S(1, n-1, 0)$.

Use recurrences $S(i,j,0) = \max(S(i+1,j,0)+|A[i-1]-A[i]|\,, S(i,j-1,1)+|A[i-1]-A[j]|)$ and $S(i,j,1) = \max(S(i+1,j,0) + |A[j+1]-A[i]|\,, S(i,j-1,1) + |A[j+1]-A[j]|)$.

Can solve from length one subproblems iteratively up to length n subproblems.

Final solution is $S(1, n-1, 1)$.

Analysis: There are $\binom{n}{2}/2 \in O(n^2)$ subproblems, each of which can be solved in constant time.

Problem 6. Sawmills [25 points]

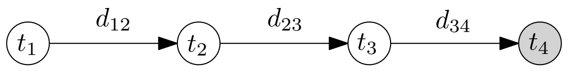
A river flows left to right, passing by villages 1, 2, . . . , n in order.

The villagers in village i cut t_i trees per year, and transport them (if necessary) to the next village that has a sawmill (by floating them down the river). Currently, there is one big sawmill that handles all the logs, located in the last (n^{th}) village.

Let d_{ij} denote the (positive) distance from the ith village to the jth village. We will be looking at the *total distance all the logs are travelling*. The total distance is currently

$$\sum_{i=1}^{n} t_i \cdot d_{in}$$

Here is an example for $n = 4$, where the n^{th} village with the sawmill is marked (we only mark the distances between adjacent villages, but you are given *all the pairwise distances* as an input).

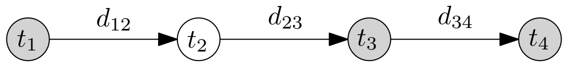


The total distance with one sawmill is:

$$t_1 \cdot d_{14} + t_2 \cdot d_{24} + t_3 \cdot d_{34}.$$

Your goal is to build at most k *additional sawmills* to minimize the total distance all the logs need to float.

For example, if $k = 2$ and $n = 4$, and we build new sawmills in the villages 1 and 3, then the new total distance is equal to $t_2 \cdot d_{23}$. (This may not be the best solution.)



Describe an algorithm that:

- takes as input: n, k, t_1, . . . , t_n, and a subroutine that computes d_{ij} efficiently for all i and j (i.e. you are given all pairwise distances between villages),
- and produces as output: the best locations for the k (or less) new sawmills, that minimize the *total distance logs will travel*.

The running time of your algorithm should be $O(n^2k)$. Slower solutions will receive partial credit.

Hint: Use dynamic programming; you should perform the following steps.

(a) Define the subproblems for the dynamic programming.

(b) Write down the recursive formula for the subproblems.

(c) Show how to compute the cost of the optimal solution in $O(n^2k)$ time.

(d) Sketch in a couple of sentences how to recover the *optimal solution itself* (as opposed to the optimal total distance).

Solution: Use dynamic programming. Let $\mathrm{OPT}[p][q]$ denote the best distance for the first p villages if:

- we assume we have a sawmill in the p-th village;
- we are allowed to build at most q *additional* sawmills.

Clearly, the cost of the answer is $\mathrm{OPT}[n][k]$.

If $q = 0$, then we may only use the sawmill in the p-th village, and, one has:

$$\mathrm{OPT}[p][0] = \sum_{i=1}^{p-1} t_i \cdot d_{ip} \ .$$

If $p = 1$, then $\mathrm{OPT}[1][q] = 0$ for every q.

Suppose that $p > 1$ and $q > 0$. Then, we try all the possible ways to open the last out of q sawmills as follows:

$$\mathrm{OPT}[p][q] = \min_{1 \le l \le p-1} \left(\mathrm{OPT}[l][q-1] + \sum_{i=l+1}^{p-1} t_i \cdot d_{ip} \right) .$$

If implemented straightforwardly, the running time is $O(n^2k)$, since we have $O(nk)$ states, in each state we try $O(n)$ values of l.

Then, we precompute all the values $F(u, v)$ by noticing that $F(u, v) = F(u + 1, v) + t_u \cdot d_{u,v-1}$. Using these precomputed values, we can compute the DP table in time $O(n^2k)$ ($O(nk)$ states, $O(n)$ work in each).

One can recover the solution itself (not just the cost) using standard ideas (by remembering how we achieved optimum for every state).