CSE548, AMS542: Analysis of Algorithms, Spring 2015 **Date:** March 12

In-Class Midterm

(4:05 PM – 5:20 PM : 75 Minutes)

- This exam will account for either 15% or 30% of your overall grade depending on your relative performance in the midterm and the final. The higher of the two scores (midterm and final) will be worth 30% of your grade, and the lower one 15%.
- There are three (3) questions, worth 75 points in total. Please answer all of them in the spaces provided.
- There are 18 pages including two (2) blank pages and two (2) pages of appendices. Please use the blank pages if you need additional space for your answers.
- The exam is *open slides* and *open notes*.

Good Luck!

Question	Pages	Score	Maximum
1. JFK AirTrains	2–3		20
2. Climbing Stairs with a Ham Sandwich	5–9		30
3. Incremental Mergesort	10–15		25
Total			75

Name: _____

Question 1. [20 Points] JFK AirTrains. The AirTrain system of the John F. Kennedy International Airport connects the airport terminals and parking areas with railway and subway lines. When an AirTrain stops at a station it aligns its doors with the doors of the platform before opening them. This question is about this alignment when some of the doors are nonfunctional (i.e., do not open). We will consider an AirTrain $\mathcal{A}$ with $n > 0$ equispaced doors[1] which is trying to stop at a platform $\mathcal{P}$ with exactly the same number of doors with the same inter-door spacing. If you know exactly which doors of $\mathcal{A}$ and $\mathcal{P}$ do not open, how can you align $\mathcal{A}$ and $\mathcal{P}$ so that the maximum number of working doors of $\mathcal{A}$ align with working doors of $\mathcal{P}$?

We number the doors of $\mathcal{A}$ from front to rear by consecutive integers starting from 0, and do the same for $\mathcal{P}$. For $0 \leq i < n$, let

$$a_i = \begin{cases} 1 & \text{if door } i \text{ of } \mathcal{A} \text{ is functional,} \\ 0 & \text{otherwise;} \end{cases} \quad \text{and} \quad p_i = \begin{cases} 1 & \text{if door } i \text{ of } \mathcal{P} \text{ is functional,} \\ 0 & \text{otherwise.} \end{cases}$$

Each alignment of $\mathcal{A}$ and $\mathcal{P}$ is identified by a unique integer $k \in (-n, n)$. If $k \geq 0$, then door 0 of $\mathcal{A}$ is aligned with door k of $\mathcal{P}$, otherwise door 0 of $\mathcal{P}$ is aligned with door $-k$ of $\mathcal{A}$.

1(*a*) [**5 Points**] For any given alignment $k \in (-n, n)$, show that the number of working doors of $\mathcal{A}$ aligned with working doors of $\mathcal{P}$ can be computed in $\mathcal{O}(n)$ time.

[1] though JFK AirTrains are very small, and a typical one includes only a couple of cars

1(b) [**15 Points**] Show that the alignment that aligns the maximum number of working doors of $\mathcal{A}$ with working doors of $\mathcal{P}$ can be found $\Theta(n \log n)$ time.

[*Hint: Cast the problem as that of computing the product of two polynomials of degree at most n, where one polynomial is constructed from a_i's and the other from p_i's. Recall how one can multiply two polynomials efficiently.*]

Use this page if you need additional space for your answers.

Question 2. [30 Points] Climbing Stairs with a Ham Sandwich. This question is on recurrences.

2(*a*) **[10 Points]** The following recurrence gives the time needed (i.e., the number $Q(m)$ of tree nodes visited) to answer a line query in a *ham-sandwich tree*[2] for m points in a plane.

$$Q(m) = \begin{cases} \Theta(1) & \text{if } m \leq 4, \\ Q\left(\frac{m}{2}\right) + Q\left(\frac{m}{4}\right) + \Theta(1) & \text{otherwise.} \end{cases}$$

Use the Akra-Bazzi method to solve the recurrence.

[*Hint: When solving* $\sum_{i=1}^{k} a_i b_i^p = 1$ *for* p*, replace* 2^p *with* x *and solve for* x *first.*]

[2] This is a serious data structure in computational geometry, invented by H. Edelsbrunner and E. Welzl in 1983!

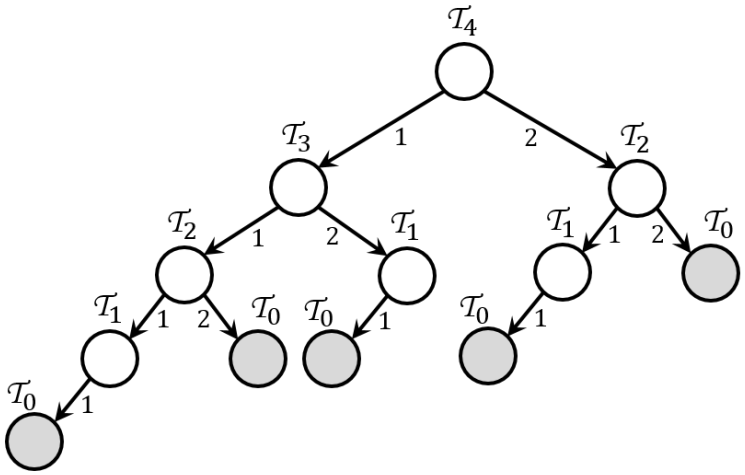

Figure 1: Climb tree $\mathcal{T}_4$ for $n = 4$ stairs (ref: questions 2(b)-2(c)).

BUILD-CLIMB-TREE(n)

Input: Inputs is the number $n \geq 0$ of stairs to climb by climbing either 1 stair or 2 stairs per step.

Output: Returns a pointer to a *climb tree* $\mathcal{T}_n$ for n stairs.

Algorithm:

1. ***if*** $n \leq 0$ ***then return*** NIL {*nothing to climb*}
2. ***else***
3. $\mathcal{T} \leftarrow$ new tree node {*this is the root of a* $\mathcal{T}_n$}
4. $\mathcal{T}.left \leftarrow$ BUILD-CLIMB-TREE($n-1$) {*climb 1 stair, and recursively build a* $\mathcal{T}_{n-1}$ *in* $\mathcal{T}.left$}
5. $\mathcal{T}.right \leftarrow$ BUILD-CLIMB-TREE($n-2$) {*climb 2 stairs, and recursively build a* $\mathcal{T}_{n-2}$ *in* $\mathcal{T}.right$}
6. ***return*** $\mathcal{T}$

Figure 2: Build a climb tree $\mathcal{T}_n$ for n stairs (ref: questions 2(b)-2(c)).

2(*b*) [**5 Points**] Suppose you want to generate all possible ways you can climb $n \geq 0$ stairs by climbing either 1 stair or 2 stairs per step. One can store all possible solutions to the problem in a binary *climb tree* $\mathcal{T}_n$. The left pointer of the root of $\mathcal{T}_n$ represents the case in which the first step climbs only 1 stair, and points to a $\mathcal{T}_{n-1}$ for the remaining $n-1$ stairs. The right pointer represents the case in which the first step climbs 2 stairs, and points to a $\mathcal{T}_{n-2}$ for the remaining $n-2$ stairs. The number of leafs in $\mathcal{T}_n$ is equal to the number of ways one can climb the n stairs under the constraints, and every root to leaf path represents a different way of climbing the stairs. Figure 1 shows an example.

The algorithm BUILD-CLIMB-TREE shown in Figure 2 accepts the number n of stairs as input, and returns a pointer to a $\mathcal{T}_n$.

Let $T(n)$ be the running time of BUILD-CLIMB-TREE for n stairs. Write down a recurrence relation describing $T(n)$.

2(c) [**8 Points**] Solve the recurrence for $T(n)$ in part 2(b) using your solution for $Q(m)$ in part 2(a).

[*Hint: A substitution of the form* $m = 2^n$ *in* $Q(m)$ *may be helpful.*]

2(d) [**7 Points**] Use your solution from part 2(a) to solve the following recurrence.

$$S(n) = \begin{cases} \Theta(1) & \text{if } n \leq 16, \\ S(\sqrt{n}) + S(\sqrt[4]{n}) + \Theta(1) & \text{otherwise.} \end{cases}$$

Question 3. [25 Points] Incremental Mergesort. In this task we will consider a data structure $\mathcal{D}$ that can efficiently maintain a set of numbers in almost sorted order under insertion. More specifically, for every $n > 0$, after the insertion of n numbers into $\mathcal{D}$, it can output the earliest $m = 2^{\lfloor \log_2 n \rfloor}$ inserted numbers in sorted order. Observe that m is the largest power of 2 not larger than n, and so $\frac{n}{2} \leq m \leq n$.

The Initialize-Global function given below is called first to initialize the data structure. Then the numbers are inserted into it by calling the Insert function.

An example of how the data structure works is given in Figure 3.

```
Initialize-Global( )                                   {initialize global variables and data structures}
1. allocate an array A[0 : N − 1], where N is the maximum number of insertions supported by the data structure
2. n ← 0                                               {array A is currently empty}
3. S ← ∅                                               {stack S will store the lengths of sorted segments of A from left to right}
```

```
Insert( x )                                            {insert the number x into the data structure}
1.  A[n] ← x                                           {store x in the next available location}
2.  n ← n + 1                                          {now we have one more number in the data structure}
3.  l ← 1,  q ← n − l                                  {the new number forms the rightmost (sorted) segment A[q : n − 1] of A,
                                                        and has length (n − 1) − q + 1 = n − q = l = 1}
4.  while S ≠ ∅ and S.top( ) = l do                    {if the sorted segment to the left of the current segment
                                                        A[q : n − 1] of length l is also of length l}
5.      S.pop( )                                       {remove that segment from the stack}
6.      p ← q − l                                      {A[p : q − 1] is a sorted segment of length l immediately
                                                        to the left of current segment A[q : n − 1] of length l}
7.      Merge( A, p, q, n )                            {merge sorted segments A[p : q − 1] and A[q : n − 1] of length l each,
                                                        and store the merged sorted segment of length 2l in A[p : n − 1]}
8.      q ← p,  l ← 2l                                 {now A[q : n − 1] is the rightmost sorted segment of length 2l}
9.  endwhile
10. S.push( l )                                        {push the length of the rightmost sorted segment into the stack}
```

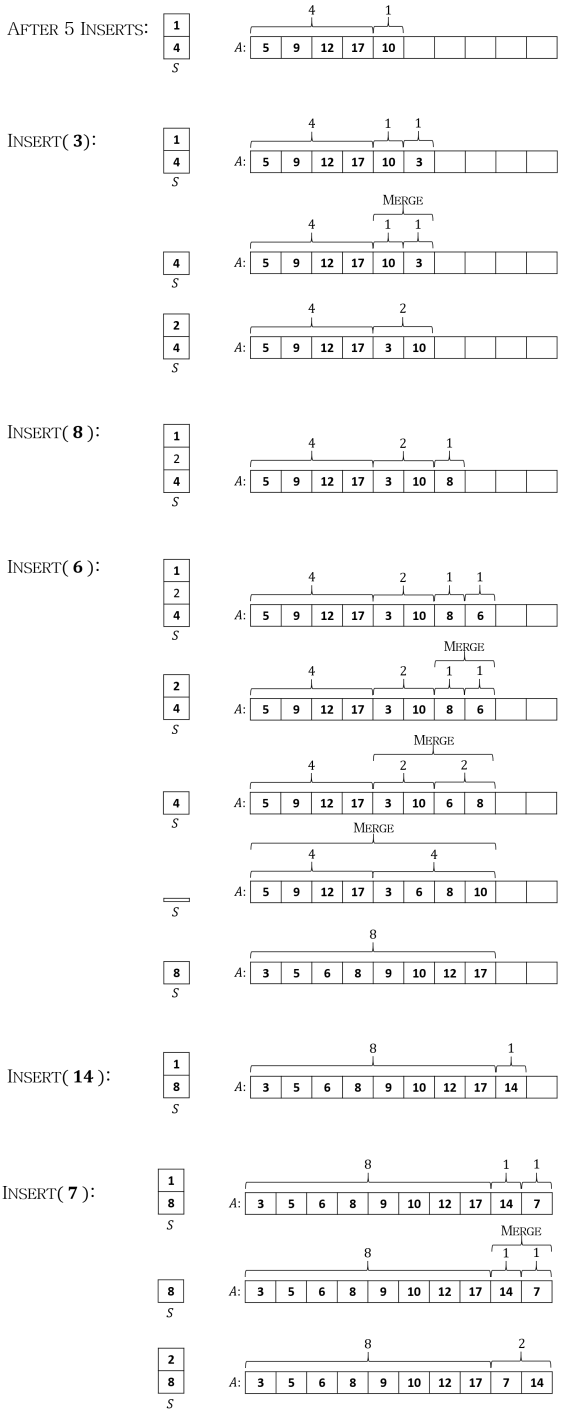

Figure 3: An example showing how the data structure works (ref: question 3).

3(a) [**6 Points**] The stack S used by the data structure stores the lengths of sorted segments of A from left to right. Argue that (i) the segment lengths stored in S are always powers of 2, (ii) they strictly decrease from bottom to top of S, and (iii) after inserting n numbers the segment lengths in S sum up to n.

[*Hint: Show that if the properties holds before an insertion, they continue to hold after the insertion.*]

3(b) [**4 Points**] Prove that for every $n > 0$, after inserting n numbers into the data structure, the leftmost sorted segment of A will contain the earliest $2^{\lfloor \log_2 n \rfloor}$ inserted numbers in sorted order.

[*Hint: Use the properties proved in part* 3(a)*, and observe that* $2^{\lfloor \log_2 n \rfloor}$ *is the largest power of 2 not larger than* n*.*]

3(c) [**5 Points**] Prove that the worst-case cost (i.e., number of comparisons performed) of the n-th INSERT is $\Theta(n)$.

[*Hint: Recall that fewer than* $2k$ *comparisons are needed in order to merge two sorted segments of length* k *each. Now reason about the costs of all merge operations triggered by an* INSERT*. You may find the results proved in part* 3(a) *useful.*]

3(d) [**10 Points**] Show that if you perform a sequence of $n > 0$ INSERT operations, the amortized cost (i.e., number of comparisons performed) of each INSERT is only $\mathcal{O}(\log n)$.

[*Hint: All elements involved in a merge operation should share the cost of merging. So each element needs to bear a very small portion of the total cost of the entire merge operation. Now reason about the number of times an element can be part of a merge operation.*]

Use this page if you need additional space for your answers.

Appendix: Recurrences

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

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1, \\ aT\left(\frac{n}{b}\right) + f(n), & \text{otherwise,} \end{cases}$$

where, $\frac{n}{b}$ is interpreted to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then $T(n)$ has the following bounds:

Case 1: If $f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.

Case 2: If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for some constant $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.

Case 3: If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Akra-Bazzi Recurrences. Consider the following recurrence:

$$T(x) = \begin{cases} \Theta(1), & \text{if } 1 \leq x \leq x_0, \\ \sum_{i=1}^{k} a_i T(b_i x) + g(x), & \text{otherwise,} \end{cases}$$

where,

1. $k \geq 1$ is an integer constant,
2. $a_i > 0$ is a constant for $1 \leq i \leq k$,
3. $b_i \in (0, 1)$ is a constant for $1 \leq i \leq k$,
4. $x \geq 1$ is a real number,
5. x_0 is a constant and $\geq \max\left\{\frac{1}{b_i}, \frac{1}{1-b_i}\right\}$ for $1 \leq i \leq k$, and
6. $g(x)$ is a nonnegative function that satisfies a polynomial growth condition (e.g., $g(x) = x^\alpha \log^\beta x$ satisfies the polynomial growth condition for any constants $\alpha, \beta \in \Re$).

Let p be the unique real number for which $\sum_{i=1}^{k} a_i b_i^p = 1$. Then

$$T(x) = \Theta\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right).$$

Appendix: Computing Products

Integer Multiplication. Karatsuba's algorithm can multiply two n-bit integers in $\Theta\left(n^{\log_2 3}\right) = \mathcal{O}\left(n^{1.6}\right)$ time (improving over the standard $\Theta\left(n^2\right)$ time algorithm).

Matrix Multiplication. Strassen's algorithm can multiply two $n \times n$ matrices in $\Theta\left(n^{\log_2 7}\right) = \mathcal{O}\left(n^{2.81}\right)$ time (improving over the standard $\Theta\left(n^3\right)$ time algorithm).

Polynomial Multiplication. One can multiply two n-degree polynomials in $\Theta\left(n \log n\right)$ time using the FFT (Fast Fourier Transform) algorithm (improving over the standard $\Theta\left(n^2\right)$ time algorithm).