In-Class Midterm

(7:05 PM – 8:20 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 four (4) 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*. But *no books* and *no computers*.

Good Luck!

Question	Pages	Score	Maximum
1. The Wheel of Fortune	2–6		30
2. Funnels and Funnelsort	8–11		30
3. Grocery Shopping	13–14		15
Total			75

Name: ______________________________

Figure 1: Specification of this wheel is $\langle 9, \delta[0:9] \rangle$, where $\delta[0..9] = \langle 4, 1, 3, 2, 2, 0, 2, 1, 0, 1 \rangle$.

Question 1. [30 Points] The Wheel of Fortune. A *wheel of fortune* is divided into many small wedges where each wedge is labeled with a dollar amount between \$0 and $\$N$ (inclusive) for some given $N \geq 0$. While the same dollar amount may appear in multiple wedges, some dollar amount between \$0 and $\$N$ may not appear at all. Figure 1 shows an example with $N = 9$.

The game of *wheel of fortune* is played in multiple rounds. In each round, each player spins the wheel once and when the wheel stops spinning he/she collects the dollar amount written on the wedge closest to the arrowhead placed right outside the wheel as a prize. For example, in Figure 1 the red \$2 wedge is the closest to the arrowhead. Let's assume for simplicity that there will be no ties (i.e., two wedges are the closest to the arrowhead). After each player plays for $R > 0$ rounds, the player with the highest total prize amount wins the game.

A *wheel specification* is a tuple $\langle N, \delta[0..N] \rangle$, where the first entry (i.e., N) in the tuple says that the maximum possible score is N, and the second entry (i.e., $\delta[0..N]$) means that for each $n \in [0, N]$, $\delta[n]$ will give the number of ways one can score n in a single round of play (i.e., number of times n appears on the wheel). Figure 1 shows an example.

Given a wheel specification $\langle N, \delta[0..N] \rangle$ and the number of rounds R of play, where $R = 2^k$ for some integer $k \geq 0$, this task asks you to return an array $\beta[0..RN]$, where for each $n \in [0, RN]$, $\beta[n]$ gives the number of ways one can score n after R rounds of play. We will call this the *wheel of fortune problem.*

```
FIND-NUMBER-OF-WAYS-TO-SCORE( N, R, δ[0..N] )
    {Input N > 0 is the maximum possible score from one round of play. Hence, possible
    scores are: 0, 1, 2, ..., N. The array δ[0..N] gives the number of ways one can get the
    scores in a single round, e.g., δ[n] gives the number of ways one can get score n ∈ [0, N].
    Finally, R = 2^k is the number of rounds to play, where k ≥ 0 is an integer. This function
    returns the number of ways one can score n after R rounds of play, where 0 ≤ n ≤ RN.}

1.  allocate array W[1..R][0..RN]        {for 1 ≤ r ≤ R and 0 ≤ n ≤ rN, W[r][n] will
                                          store the number of ways one can score n after r rounds of play}
2.  for n ← 0 to N do
3.      W[1][n] ← δ[n]                   {initialize W[1][0..N] with the first round data from δ[0..N]}

4.  for r ← 2 to R do                    {initialize for rounds 2 to R}
5.      for n ← 0 to rN do               {maximum possible score in round r is rN}
6.          W[r][n] ← 0                  {initialize W[r][n] to 0}

7.  for r ← 2 to R do                    {compute round r numbers from numbers computed for round r − 1}
8.      for n ← 0 to rN do               {compute W[r][n]}
9.          for m ← 0 to min{n, N} do    {only single round scores of value ≤ n can contribute to a total score of n}
10.             W[r][n] ← W[r][n] + W[r − 1][n − m] × δ[m]   {one way of scoring n in r rounds is to score m
                                          in round r and n − m in rounds 1 to r − 1, and the
                                          number of ways that can be done is W[r − 1][n − m] × δ[m]}
11. allocate array β[0..RN]              {β[0..RN] is used to store and return results computed for round R}
12. for n ← 0 to RN do                   {maximum possible score in round R is RN}
13.     β[n] ← W[R][n]                   {copy results from W[R][0..RN] to β[0..RN]}

14. free array W[1..R][0..RN]            {free temporary array}

15. return β[0..RN]                      {return results computed for round R}
```

Figure 2: For each $n \in [0, RN]$, this function returns the number of ways one can score n after R rounds of play of the *Wheel of Fortune*, where N is the maximum possible score in a single round.

1(a) [**5 Points**] A straightforward algorithm for solving the wheel of fortune problem is shown in Figure 2. Derive a tight bound (i.e., Θ bound) on its worst-case running time.

1(b) [**8 Points**] Explain how you will reduce the wheel of fortune problem with $R = 2$ rounds into a problem of multiplying two polynomials. What will be the running time of your algorithm?

1(c) [**8 Points**] Extend your algorithm from part 1(b) to handle $R = 2^k$ rounds, where k is a given positive integer. Derive the running time of your algorithm.

For full score describe a correct algorithm that runs in $\mathcal{O}(NR\log(NR))$ time.

1(d) [**9 Points**] Suppose we want to solve the wheel of fortune problem for $R = 2^k$ rounds for some given positive integer k, but want to compute only the first $N + 1$ entries of β, i.e., compute $\beta[0..N]$ instead of $\beta[0..RN]$. Show that you can do that in $\mathcal{O}(N \log N \log R)$ time.

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

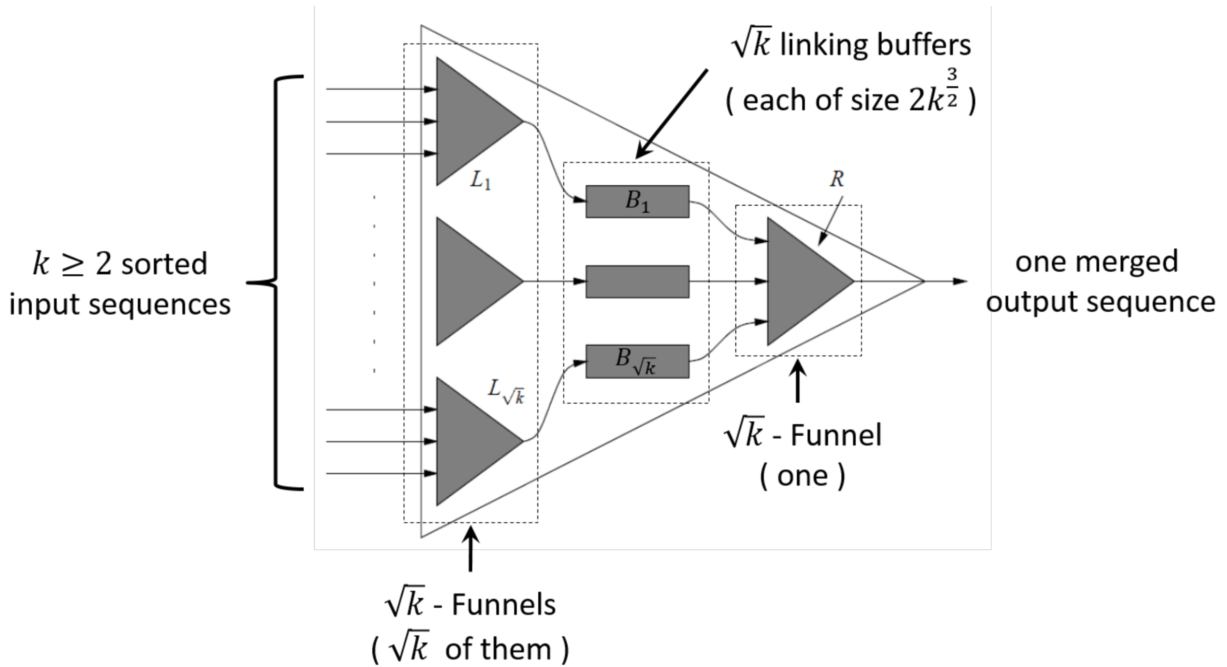

Figure 3: A k-Funnel.

Question 2. [**30 Points**] **Funnels and Funnelsort.** A k-Funnel (also known as a k-merger) is a data structure that takes $k \geq 2$ sorted sequences as inputs and supports an *Extract* operation which outputs the sorted sequence of the smallest k^3 elements from the input sequences whenever it is invoked. As Figure 3 shows a k-Funnel is built recursively from $\sqrt{k}+1$ smaller $\sqrt{k}$-Funnels and $\sqrt{k}$ linking buffers of size $2k^{\frac{3}{2}}$ each.

A sorting algorithm known as *Funnelsort* uses k-Funnels to sort data cache-efficiently.

Let $T(k)$ be the time needed by a k-Funnel to output the smallest k^3 elements from the input sequences in sorted order. One can show that

$$T(k) = \begin{cases} \Theta(1) & \text{if } k \leq 4, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right) T\left(\sqrt{k}\right) + \Theta\left(k^2\right) & \text{otherwise.} \end{cases}$$

This task asks you to solve this recurrence and use that solution to find the running time of Funnelsort.

2(a) [**4 Points**] Show that if $T(k) = k(k^2-1)\widehat{T}(k)$ then

$$\widehat{T}(k) = \begin{cases} \Theta(1) & \text{if } k \le 4, \\ 2\widehat{T}\left(\sqrt{k}\right) + \Theta\left(\frac{1}{k-\frac{1}{k}}\right) & \text{otherwise.} \end{cases}$$

2(b) [**10 Points**] Show that if $k = 2^x$ for some $x \geq 1$ and $\widehat{T}(k) = \tilde{T}(x)$ then

$$\widetilde{T}(x) = \begin{cases} \Theta(1) & \text{if } 1 \leq x \leq 2, \\ 2\widetilde{T}\left(\frac{x}{2}\right) + \mathcal{O}\left(\frac{1}{x}\right) & \text{otherwise.} \end{cases}$$

Solve this recurrence to show that $\tilde{T}(x) = \Theta(x)$.

2(c) [**4 Points**] Use your solution for $\tilde{T}(x)$ from part 2(b) to find a solution for $T(k)$.

```
FUNNELSORT( A, n )     {Takes a sequence A of n numbers as input. Outputs the entries of A in sorted order.}

1. if n ≤ 8 then
2.     sort A using any sorting algorithm in Θ(1) time
3.     return the sorted sequence
4. else
5.     split A into n^{1/3} contiguous subsequences A_1, A_2, ..., A_{n^{1/3}} of length n^{2/3} each
                {for simplicity, we are not using precise expressions for #subsequences and subsequence lengths}
6.     for i ← 1 to n^{1/3} do
7.         A_i ← FUNNELSORT( A_i, n^{2/3} )
8.     merge the n^{1/3} sorted subsequences (from lines 6–7) using an n^{1/3}-Funnel
9.     return the merged sequence
```

Figure 4: Funnelsort.

2(d) [**12 Points**] The *Funnelsort* algorithm is shown in Figure 4. Let $S(n)$ be its running time on a sequence of length n. Write a recurrence relation describing $S(n)$ and solve it.

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

Question 3. [**15 Points**] **Grocery Shopping.** There are n households in a town and there are n grocery stores within reasonable distance from the households. Each store carries the same set of n grocery items. For convenience, we will identify the households using unique integers from 1 to n. Let's do the same with the grocery stores and the grocery items.

Each month household $i \in [1, n]$ needs to buy $u_{i,k}$ units of grocery item $k \in [1, n]$. The price of one unit of item $k \in [1, n]$ at grocery store $j \in [1, n]$ is $p_{k,j}$. Let $c_{i,j}$ be the total cost if household i buys all its grocery items from grocery store j.

Given all $u_{i,k}$ values for $1 \leq i, k \leq n$ and all $p_{k,j}$ for all $1 \leq k, j \leq n$, your task is to find all $c_{i,j}$ values for $1 \leq i, j \leq n$.

3(a) [**7 Points**] Explain how you can use a recursive divide-and-conquer algorithm to find all $c_{i,j}$ values in $\mathcal{O}\left(n^{\log_2 7}\right)$ time.

3(b) [**8 Points**] Suppose $u_{i,k} = \alpha$ for all $i, k \in [1, n]$, where α is a constant. Explain how you will modify the recursive divide-and-conquer algorithm you used in part 3(a) to find all $c_{i,j}$ values in $\Theta\left(n^2\right)$ time under this constraint. Write down the recurrence relation describing the running time of the modified algorithm and show that it solves to $\Theta\left(n^2\right)$.

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

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 $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$. 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).