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: _____

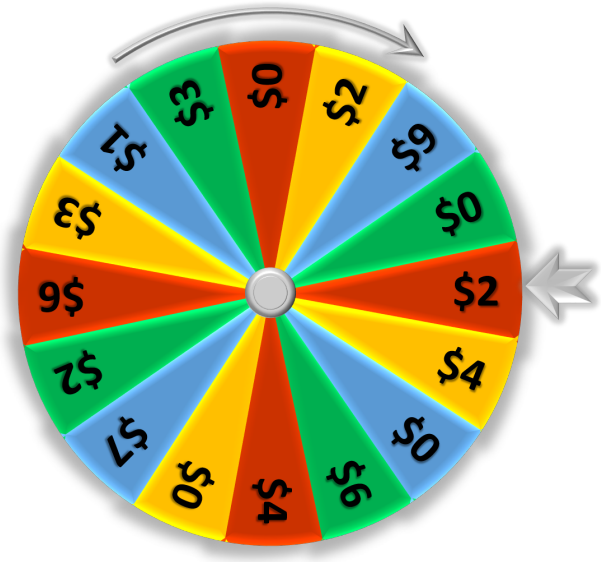

Figure 1: Specification of this wheel is $\langle 9, \delta[0:9]\rangle$, where $\delta[0..9] = \langle 4, 1, 3, 2, 2, 0, 2, 1, 0, 1\rangle$.

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 n do              {only single round scores of value ≤ n can contribute to a total score of n}
10.             if m ≤ N then 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.

Solution:

Running time for the given algorithm:

Line 1: $\theta(1) - \theta(R^2N)$

Line 2-3: $\theta(N)$

Line 4-6:

$$\sum_{r=2}^{R}\sum_{n=0}^{rN}\theta(1) = \sum_{r=2}^{R}\theta(rN) = \theta(R^2N)$$

Line 7-10:

$$\sum_{r=2}^{R}\sum_{n=0}^{rN}\sum_{m=0}^{n}\theta(1) = \sum_{r=2}^{R}\sum_{n=0}^{rN}\theta(n) = \sum_{r=2}^{R}\theta(\frac{rN(rN+1)}{2}) = \theta(R^3N^2)$$

Line 11: $\theta(1) - \theta(RN)$

Line 12-13: $\theta(RN)$

Line 14: $\theta(1) - \theta(R^2N)$

Grading Criteria:

- If you did line 7-10 correctly you get 4 points.
- If you did other parts correctly you get 1 point because you have to show the other parts as well.
- If you just wrote $\theta(n^3)$, you get 0 point, that means you did not understand the bound at all. However, if you did explain your reasoning, I tried to give you some partial points.
- If your final result is not in terms of N,R, we deducted 2 points.
- If your final result is $\theta(R^2N^2)$, we deducted 1.5 points

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?

[*Hint: You will need to construct a polynomial for* $\delta[0..N]$ *using its entries as coefficients.*]

Solution:

For each round we can express the problem in polynomial as-
$P(z) = c_0 + c_1z + c_2z^2 + \ldots c_Nz^N$ where $c_i = \delta[i]$
Multiplying the two polynomials of two rounds will give us the number of ways one can score n after two rounds of play. Degree bound of the multiplied polynomial would be $[0 \ldots 2N]$. So,
$P_2(z) = P(z) * P(z) = a_0 + a_1z + a_2z^2 + \ldots a_{2N}z^{2N}$
Here co-efficient a[0..2N] is the number of ways one can score n after two rounds.

– – – – – – – – – – – – – – – – – – – –5 points

Running Time: If we do regular polynomial multiplication running time is $\mathcal{O}\left(N^2\right)$. But if we use **FFT** for the polynomial multiplication, running time reduces to $\mathcal{O}\left(N \log N\right)$

– – – – – – – – – – – – – – – – – – – –3 points

Grading Criteria:

- For the first part, if you designed the polynomial and showed what are the coefficients are you will get full credit (i.e: 5 points). If you did write a polynomial but did not explain what are the elements of it you get zero (perhaps some partial credit depending on the explanation).
- For the second part, if you can correctly identify the complexity of you multiplication (either N^2 or $N \log N$), you get full credit (i.e: 3 points).

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.

[*Hint: Since R is a power of 2, you can construct the polynomial for R rounds through repeated squaring of the polynomial for one round.*]

Solution:

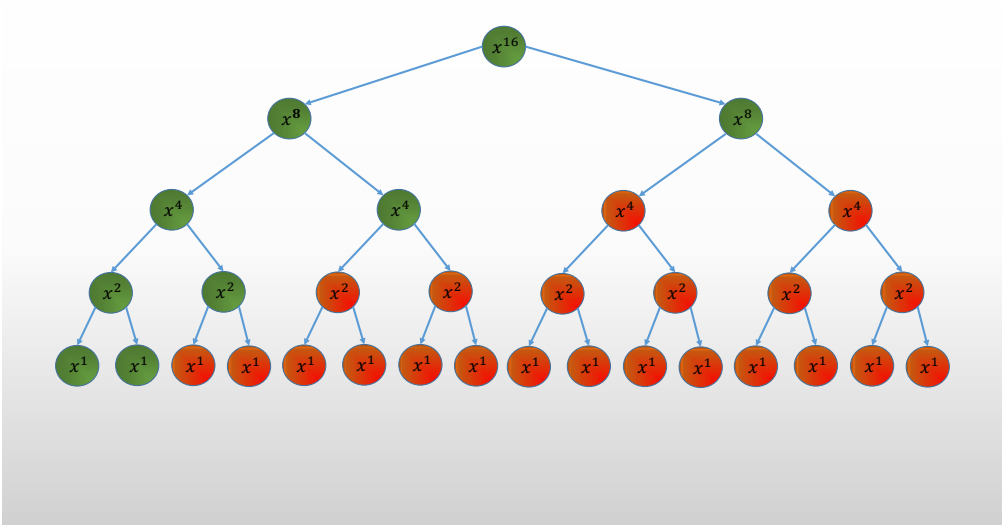

Figure 3: Tree expansion of repeated squaring. Note that, we do not need to explore the red colored branches because we already computed these values through exploring the green branches. Exploring red branches adds an extra *log* factor in the time complexity for this problem.

– – – – – – – – – – – – – – – – – – –4 points

[if anyone explored the red branches they will get only 2 points]

From 1(b), we know we can multiply two polynomials of degree bound [0..N] and get the polynomial of degree bound [0..2N] for two rounds using $\mathcal{O}(N \log N)$ running time. So,

Round	Degree bound for multiplication	Running Time
2	N	$NlongN$
4	2N	$2Nlog2N$
8	4N	$4Nlog4N$
...	...	...
R	$\frac{R}{2}N$	$\frac{R}{2}Nlog\frac{R}{2}N$

Since the degree bound is changing with each multiplication, we can add them to get the whole running time (note that, degree bound is changing geometrically so, we can apply master theorem case 3 as well):

$NlongN + 2Nlog2N + 4Nlog4N + ... + \frac{R}{2}Nlog\frac{R}{2}N < (1 + 2 + 4 + ...\frac{R}{2})NlogRN$

We know, $1 + 2 + 4 + ...\frac{R}{2} < R$ [Since, $R = 2^k$ rounds, where k is a given positive integer]
So we can write the time complexity as $\mathcal{O}(NR\log(NR))$

$$- - - - - - - - - - - - - - - - - - - -4 \text{ points}$$

[N.B: if anyone can correctly derive the running time of their algorithm (not necessarily NRlogNR) will get full score in this section]

Grading Criteria:

- For the first part, if you designed your repeated square algorithm expanding your tree by just green branches you will get full credit (4 points), if you did the whole tree expansion you get partial credit (2 point), Otherwise zero.
- If you got 4 points if first part and successfully derived the complexity $NR\log NR$ you get full credit (i.e: 4 points). But if you didn't get full credit in part 1, but yet you were able to prove the time complexity as $NR\log NR$, it means your derivation is wrong and you get Zero. However, if you were able to derive the complexity of you own algorithm you get full credits (i.e: 4 points).

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.

[*Hint: From every product polynomial* $P(z) = c_0 + c_1 z + c_2 z^2 + \ldots c_N z^N + c_{N+1} z^{N+1} \ldots$ *discard all powers of* z *above* z^N *before you use it for computing the next product.*]

Solution:
We will apply repeated squaring to get the solution of R rounds for this problem as well. It will take $\log R$ repeated multiplications like problem 1(c).

$- - - - - - - - - - - - - - - - - - -$ 5 points

[for correct tree expansion]

But in this case, we are keeping only first $N + 1$ entries of the multiplied polynomial in each step. So degree bound is not changing geometrically for the multiplications and it is always remains N. Using FFT we can do multiplication in $\mathcal{O}(N \log N)$ time. There are $\log R$ steps. So the running time complexity is $\mathcal{O}(N \log N * \log R)$

$- - - - - - - - - - - - - - - - - - -$ 4 points

Grading Criteria:

- For the first part, If you can successfully expand you algorithm from 1(c) by expanding just the green branches and showed it still requires $\log R$ multiplication to solve this problem you get full credit (i.e: 5 points). Since the problem is asking that your algorithm should have the complexity of $N \log N \log R$, any other algorithm with different complexity should earn you zero points. But depending on the explanation we might award you some partial credits.
- For the second part, you have to show that the degree bound is not changing after each multiplication like in 1(c). Why you cannot apply Master theorem case 3 in this algorithm, why we are multiplying $N \log N$ with $\log R$ directly, your answer should explain these facts to earn full credit.

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

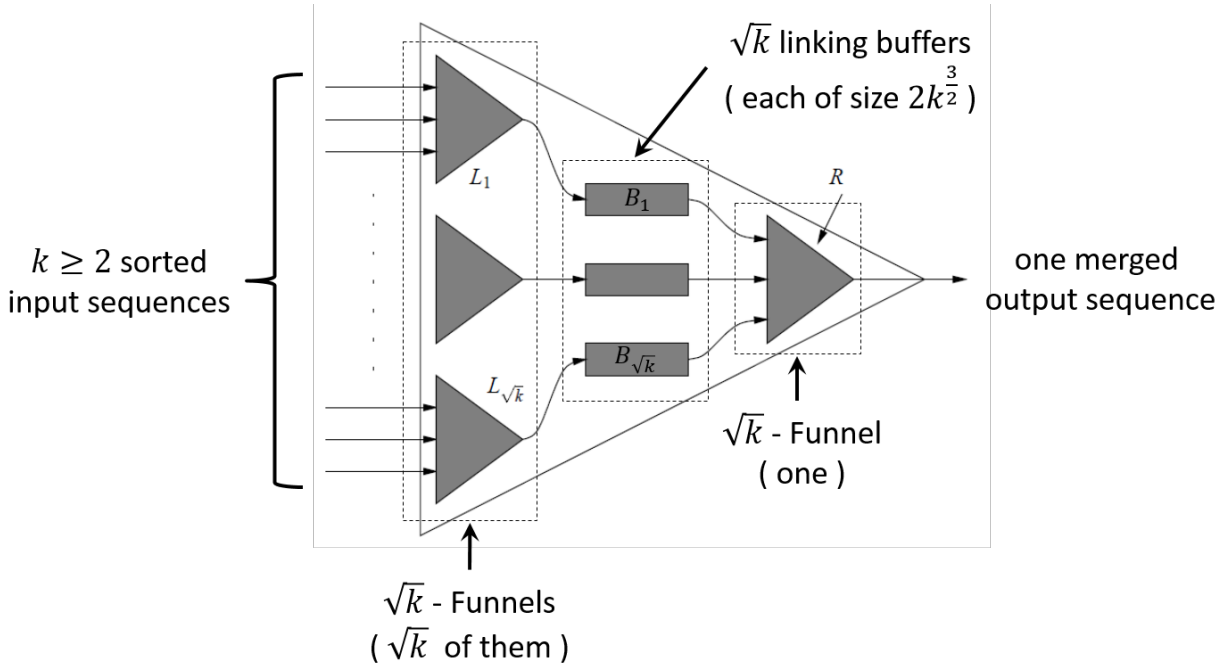

Figure 4: A k-Funnel.

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 4 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

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

Solution:

$$T(k) = k(k^2-1)\widehat{T}(k)$$
$$T(\sqrt{k}) = \sqrt{k}(k-1)\widehat{T}(\sqrt{k}) - - - - - -(1)$$

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

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

note that:

$$\frac{\left(2k^{\frac{3}{2}} + 2\sqrt{k}\right)}{k(k^2-1)} T\left(\sqrt{k}\right) = \frac{\left(2\sqrt{k}(k+1)\right)}{k(k+1)(k-1)} T\left(\sqrt{k}\right)$$

$$from(1), \frac{\left(2\sqrt{k}(k+1)\right)}{k(k+1)(k-1)} T\left(\sqrt{k}\right) = \frac{2\sqrt{k}}{k(k-1)} \sqrt{k}(k-1)\widehat{T}(\sqrt{k}) = 2\widehat{T}(\sqrt{k})$$

$$\frac{k^2}{k(k^2-1)} = \frac{1}{k-\frac{1}{k}}$$

so we have:

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

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 $\tilde{T}(x) = \mathcal{O}(x)$.

[*Hint: For all* $x > 0$, $2^x - 2^{-x} > x$. *Use the Akra-Bazzi method to solve the recurrence. Also, recall that for* $t \neq -1$, $\int_1^x u^t du = \left[\frac{u^{t+1}}{t+1}\right]_1^x$.]

Solution:

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

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

form the hint: $2^x - 2^{-x} > x$, we have $\frac{1}{2^x - 2^{-x}} < \frac{1}{x}$, so

$$\widetilde{T}(x) = \begin{cases} \Theta(1) & \text{if } 1 \leq x \leq 2, \\ 2\widetilde{T}\left(\frac{x}{2}\right) + \mathcal{O}\left(\frac{1}{x}\right) & \text{otherwise.} \end{cases} \quad ----------3 points$$

Use the Akra-Bazzi method, we have:

$$a = 2, b = 0.5, ab^p = 1, p = 1 ----------2 points$$

$$x_0 = 2, x_0 \geq max\{1/b, 1/(1-b)\} ----------1 points$$

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

$$\tilde{T}(x) = \Theta\left(x^1\left(1 + \int_1^x \frac{u^{-1}}{u^{1+1}} du\right)\right)$$

$$\tilde{T}(x) = \Theta\left(x\left(1 + \int_1^x u^{-3} du\right)\right)$$

$$\tilde{T}(x) = \Theta\left(x + x\left[\frac{u^{-3+1}}{-3+1}\right]_1^x\right)$$

$$\tilde{T}(x) = \Theta\left(x + x/2 - \frac{1}{2u^2}\right)$$

$$\tilde{T}(x) = \mathcal{O}(x) ----------4 points$$

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

Solution:

$$\tilde{T}(x) = \mathcal{O}(x)$$

$$\widehat{T}(2^x) = \mathcal{O}(x)$$

$$\widehat{T}(k) = \mathcal{O}(\log k)$$

$$- - - - - - - - - - - - - - - - - - - -2 \text{ points}$$

$$T(k) = k(k^2-1)\widehat{T}(k)$$

$$T(k) = k(k^2-1)\mathcal{O}(\log k)$$

$$T(k) = k^3\mathcal{O}(\log k)$$

$$- - - - - - - - - - - - - - - - - - - -2 \text{ points}$$

```
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 5: Funnelsort.

2(*d*) [**12 Points**] The *Funnelsort* algorithm is shown in Figure 5. Let $S(n)$ be its running time on a sequence of length n. Write a recurrence relation describing $S(n)$ and solve it.

[*Hint: Follow steps similar to those in parts* 2(*a*)–2(*c*)*. The recurrene for* $S(n)$ *will need your solution for* $T(k)$ *from part* 2(*c*).]

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

Solution:

$$S(n) = \begin{cases} \Theta(1) & \text{if } n \leq 8, \\ n^{\frac{1}{3}} S\left(n^{\frac{2}{3}}\right) + T(n^{\frac{1}{3}}) & \text{otherwise.} \end{cases}$$

− − − − − − − − − 4 points (1 point for each term, 4 points for correct recurrence)

$$S(n) = n^{\frac{1}{3}} S\left(n^{\frac{2}{3}}\right) + n\mathcal{O}(\log n)$$

$$\frac{S(n)}{n} = \frac{S(n^{\frac{2}{3}})}{n^{\frac{2}{3}}} + \mathcal{O}(\log n)$$

$$\tilde{S}(n) = \tilde{S}(n^{\frac{2}{3}}) + \mathcal{O}(\log n)$$

$$\widehat{S}(2^x) = \widehat{S}(2^{\frac{2}{3}x}) + \mathcal{O}(x)$$

$$\tilde{S}(x) = \tilde{S}(\frac{2}{3}x) + \mathcal{O}(x)$$

$$\tilde{S}(x) = \begin{cases} \Theta(1) & \text{if } 1 \leq x \leq 3, \\ \tilde{S}(\frac{2}{3}x) + \mathcal{O}(x) & \text{otherwise.} \end{cases}$$

− −2 points

Use Akra-Bazzi method to solve the recurrence.

$$\text{Since } a = 1, b = \frac{2}{3}, \text{ we have } p = 0.$$

− −1 point

$$\text{Validate } x_0 \geq \max\{\frac{1}{b}, \frac{1}{1-b}\}.$$

− −1 point

$$\tilde{S}(x) = \Theta\left(x^0\left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right)$$

$$\tilde{S}(x) = \Theta\left(x^0\left(1 + \int_1^x \frac{\mathcal{O}(u)}{u^1} du\right)\right)$$

$$\tilde{S}(x) = \Theta(1) + \mathcal{O}\left(\int_1^x du\right)$$

$$\tilde{S}(x) = \mathcal{O}(x)$$

− −3 points

$$\tilde{S}(2^x) = \mathcal{O}(x)$$

$$\widehat{S}(n) = \mathcal{O}(\log n)$$

$$S(n) = n\mathcal{O}(\log n) = \mathcal{O}(n \log n)$$

− −1 point

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.

[*Hint: You need to use an algorithm we have seen in the class.*]

Solution:

We can turn this problem into a matrix multiplication problem.

$Matrix_1$: We can use the $U_{i,k}$ values to create a $i \times k$ matrix where rows represents households and columns represents grocery items. Value at i, k is $U_{i,k}$.

$Matrix_2$**:** We can use the $P_{k,j}$ values to create a $k \times j$ matrix where rows represents grocery items and columns represents grocery stores. Value at k, j is $P_{k,j}$.

$Result = Matrix_1 \times Matrix_2$

Result: A $i \times j$ matrix that consists of result $C_{i,j}$

We can use Strassen's algorithm to multiply this two matrices. As we know 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 and it's a recursive divide and conquer method.

Grading Criteria:

- If you were able to create the matrices correctly you get $2 + 2 = 4$ points, 2 points for each matrix.
- If you explained that you would use Strassen's algorithm, you get 3 points.

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 $\Theta\left(n^2\right)$.

[*Hint: Recall that we discussed in the class how one can come up with the algorithm you used in part* 3(*a*) *and the observations that led to its efficiency.*]

Solution:

If all the $U_{i,k} = \alpha$ then for 2×2 matrix blocks the multiplication would look like this,

$$\begin{bmatrix} \alpha & \alpha \\ \alpha & \alpha \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} \alpha(e+g) & \alpha(f+h) \\ \alpha(e+g) & \alpha(f+h) \end{bmatrix} \tag{0.1}$$

From Equation 0.1, we can see that we only need 2 multiplication (instead of 7 in Strassen) and 2 addition (instead of 18 in Strassen).

So the recurrence would look like this,

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1, \\ 2T\left(\frac{n}{2}\right) + \theta(n^2), & \text{otherwise,} \end{cases}$$

We could solve this recurrence using master theorem. Here, a=2, b=2 and $f(n) = \theta(n^2)$ Using master theorem we can easily show that running time = $\theta(n^2)$

Grading Criteria:

- If you did not use recursive divide and conquer, you get 0 points.
- Algorithm part is worth 5 points, if you came up with the algorithm, you got all 5 points. If you did not get the whole algorithm correct, based on your explanation you got partial points.
- Coming up with the recurrence is worth 2 points and solving the recurrence worth 1 points.
- All these points are independent, so if you got one right you got points for that part.

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).