CIS 3200 48490181

Homework 02

Collaborators: N/A

Q1.

(a) $T(n) = 4T(n/3) + n \log n$

Answer: $\Theta(n^{\log_3 4})$

We want to prove $T(n) = \Theta(n^{\log_3 4})$ by showing both $O(n^{\log_3 4})$ and $\Omega(n^{\log_3 4})$.

1. (Upper Bound): To prove the upper bound, we will show using strong induction that for some constant c, whose value we will determine later, $T(n) \leq cn^{\log_3 4}$ for all $n \geq 2$.

Base Case: Suppose that

Induction Hypothesis: Assume that the claim is true when $n = j$ for all j such that $2 \leq j \leq k$. in other words, $T(j) \leq cj^{\log_3 4}$

Induction Step: We want to show that $T(k+1) \leq c(k+1)^{\log_3 4}$. We have

$$\begin{aligned} T(k+1) &= 4T((k+1)/3) + (k+1)\log(k+1) \\ &\leq 4c(\frac{k+1}{3})^{\log_3 4} + (k+1)\log(k+1) \qquad \text{(by IH)} \end{aligned}$$

(b) $T(n) = 3T(n/3) + n/\log n$

Answer:

(c) $T(n) = 3T(n/3 - 2) + n/2$

Answer: $\Theta(n \log n)$ We want to prove $T(n) = \Theta(n \log n)$ by showing both $O(n \log n)$ and $\Omega(n \log n)$.

1) Upper Bound: To prove the upper bound, we will show using strong induction that for some constant $c > 0$, $T(n) \leq cn \log n$ for all $n \geq 18$.

Base Case: Choose $n_0 \geq 18$ and pick c large enough so that $T(n) \leq cn \log n$ holds for all $18 \leq n \leq n_0$. This is possible since it only requires checking finitely many n.

Induction Hypothesis: Assume that the claim holds for all j with $18 \leq j \leq k$, i.e. $T(j) \leq cj \log j$.

Induction Step: Consider $n = k + 1$:

$$\begin{aligned} T(k+1) &= 3T\left(\frac{k+1}{3} - 2\right) + \frac{k+1}{2} = 3T\left(\frac{k-5}{3}\right) + \frac{k+1}{2} \\ &\leq 3c\left(\frac{k-5}{3}\right)\log\left(\frac{k-5}{3}\right) + \frac{k+1}{2} \qquad \text{(by IH)} \\ &= c(k-5)\big(\log(k-5) - \log 3\big) + \frac{k+1}{2} \\ &\leq c(k+1)\log(k+1) \;-\; c(k-5)\log 3 \;+\; \frac{k+1}{2}. \end{aligned}$$

We now want to prove $T(k+1) \leq c(k+1)\log(k+1)$, which suffices to show

$$-c(k-5)\log 3 + \frac{k+1}{2} \leq 0.$$

This is equivalent to

$$c \;\geq\; \frac{k+1}{2\,(k-5)\log 3}.$$

Notice that the right-hand side decreases monotonically to $\frac{1}{2\log 3}$ as $k \to \infty$. Hence, if we choose

$$c \;\geq\; \frac{1}{\log 3},$$

then the inequality holds for all $k \geq n_0$ (where n_0 is chosen large enough to cover the finitely many base cases). Therefore

$$T(k+1) \leq c(k+1)\log(k+1),$$

completing the induction for the upper bound.

For $c \geq 1/\log 3$ and $k \geq n_0$, the negative term $-c(k-5)\log 3$ dominates the $+(k+1)/2$ term. Thus

$$T(k+1) \leq c(k+1)\log(k+1).$$

Hence $T(n) = O(n \log n)$.

2) Lower Bound: To prove the lower bound, we show by induction that for some $d > 0$, $T(n) \geq dn \log n$ for sufficiently large n.

Base Case: Choose n_1 large and d small enough so that $T(n) \geq dn \log n$ for $2 \leq n \leq n_1$.

Induction Hypothesis: Assume $T(j) \geq dj \log j$ for all $2 \leq j \leq k$.

Induction Step: For $n = k + 1$,

$$\begin{aligned} T(k+1) &= 3T\left(\frac{k-5}{3}\right) + \frac{k+1}{2} \\ &\geq 3d\left(\frac{k-5}{3}\right)\log\left(\frac{k-5}{3}\right) + \frac{k+1}{2} \\ &= d(k-5)\big(\log(k-5) - \log 3\big) + \frac{k+1}{2}. \end{aligned}$$

By the mean value theorem for $f(x) = x \log x$,

$$(k-5)\log(k-5) \;\geq\; (k+1)\log(k+1) - 6(\log(k+1)+1).$$

Therefore

$$\begin{aligned} T(k+1) &\geq d\big[(k+1)\log(k+1) - 6(\log(k+1)+1) - (k-5)\log 3\big] + \frac{k+1}{2} \\ &= d(k+1)\log(k+1) \;+\; \Big(\tfrac{k+1}{2} - d(k-5)\log 3\Big) \;-\; 6d(\log(k+1)+1). \end{aligned}$$

Choosing $d \leq 1/(4\log 3)$ ensures that the linear term $\frac{k+1}{2} - d(k-5)\log 3$ is positive and dominates $-6d(\log(k+1)+1)$ for large k. Thus

$$T(k+1) \geq d(k+1)\log(k+1).$$

Therefore, combining both bounds, we conclude

$$T(n) = \Theta(n \log n).$$

(d) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

(e) $T(n) = T(n-1) + 1/n$
Answer: $\Theta(\log n)$

(f) $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$
Answer: $\Theta(n \log \log n)$

Q2. Sorting Almost Sorted Arrays

Algorithm:

English Description:

We first initialize a min-heap H of size $k+1$ by inserting the first $k+1$ elements of the input array A into H. This guarantees that the true first element of the sorted array is contained in H, since in a k-sorted array the smallest element cannot be located beyond the first $k+1$ positions. We also initialize an output array B of size n, which will hold the sorted elements. Next, we repeatedly perform the `extract-min` operation on H. At each step, we remove the smallest element from H and place it into the next available position of B. After each `extract-min`, if there are unseen elements left in A, we insert the next element of A into H using the `insert` operation. This makes sure that H always stores the next $k+1$ candidates for the output. In other words, as one element leaves H via `extract-min`, another new element from A enters H via `insert`, thereby sliding the window forward. (And by doing so, the heap never stores more than $k+1$ elements)

We continue this process for exactly n iterations. During the first phase, when unseen elements of A remain, each `extract-min` is paired with inserting the next unseen element of A into H. Once all elements of A have been inserted, the loop continues with only `extract-min` operations, gradually emptying H. After n iterations, all elements have been written into B, which is the fully sorted version of A.

Pseudocode:
Input: $A[1..n]$ (distinct numbers), k.
Output: $B[1..n]$ (sorted array).

(a) Initialize empty array $B[1..n]$ to 0.

(b) $H \leftarrow$ `MAKE-HEAP`().

(c) Insert $A[1], \ldots, A[\min(n, k+1)]$ into H.

(d) $p \leftarrow k+2$ (*next index of A not yet inserted*).

(e) **for** $i = 1$ to n:

i. $x \leftarrow$ `EXTRACT-MIN`(H).

ii. $B[i] \leftarrow x$.

iii. **if** $p \leq n$ **then**

A. `INSERT`($H, A[p]$).

B. $p \leftarrow p + 1$.

(f) **end for**

(g) **return** B.

Proof of Correctness: We prove that the algorithm outputs the sorted array $B = S$, where S is the sorted version of A, by carefully maintaining a loop invariant. That is, we use a loop invariant argument supported by a key lemma to prove the correctness of algorithm.

Loop Invariant. At the start of iteration i of the main for loop ($1 \leq i \leq n$):

(I1) The output array B already contains the first $i-1$ elements of S. In other words, $B[1..i-1] = S[1..i-1]$, so the prefix of B is correctly sorted and consists of the $(i-1)$ smallest elements of A.

(I2) The heap H contains exactly those elements of A that have been inserted but not yet output. More specifically, H contains all elements of A that could legally appear next in the sorted order without violating the k-sortedness property, and only those elements.

Supporting Lemma. The correct next element $S[i]$ must always be among the next $k+1$ unseen elements of A.

Proof of Lemma. Suppose, for contradiction, that $S[i]$ lies outside the next $k+1$ unseen elements of A. Then its distance between its original index in A and its final index in S would be at least $k+1$. This contradicts the definition of k-sortedness, which guarantees that no element is displaced by more than k positions. Therefore, $S[i]$ is always within the next $k+1$ unseen elements and thus is inserted into H before or at iteration i. □

Initialization (Base Case): Before the first iteration, B is empty, so (I1) holds. The algorithm inserts the first $\min(n, k+1)$ elements of A into H. By the lemma, $S[1]$ must be among these elements. Hence, (I2) holds: H contains exactly the possible candidates for the first output element.

Maintenance: Assume as the induction hypothesis (IH) that (I1)–(I2) hold at the start of iteration i for some $1 \leq i \leq n$. We prove that (I1)–(I2) also hold at the start of iteration $i+1$.

First, by the lemma, the correct next element $S[i]$ is among the candidates in H. By (I1) and the IH, all smaller elements $S[1..i-1]$ have already been placed in B, so no element in H is smaller than $S[i]$. Therefore, when the algorithm calls `EXTRACT-MIN`(H), the element returned is exactly $S[i]$. Placing $S[i]$ into $B[i]$ means that $B[1..i] = S[1..i]$, so (I1) holds at the end of iteration i, and hence at the start of iteration $i+1$ (by IH).

Next, to reestablish (I2), we consider two cases:

Case 1: There are unseen elements of A remaining ($p \leq n$). By IH, at the start of iteration i, H contained exactly the candidates for $B[i]$. After removing $S[i]$ by `EXTRACT-MIN`, the algorithm inserts $A[p]$, the next unseen element, into H. Thus, at the start of iteration $i+1$, H again contains exactly the elements that could serve as $B[i+1]$, preserving (I2).

Case 2: All elements of A have been inserted already ($p > n$). By IH, at the start of iteration i, H contained exactly the not-yet-output elements. After removing $S[i]$ by `EXTRACT-MIN`, no new insertion occurs, so H now contains precisely the set $\{S[i+1], \ldots, S[n]\}$. This is exactly the set of candidates for $B[i+1]$, so (I2) also holds at the start of iteration $i+1$.

Thus in both cases, (I1) and (I2) hold at the start of iteration $i+1$, completing the inductive step.

Termination: The loop executes exactly n iterations. During the iterations where $p \leq n$, each extraction is paired with an insertion, so the heap size stays bounded by $k+1$. Once $p > n$ (all inputs have been inserted), iterations proceed with *extractions only*. By (I2), at the start of iteration i in this phase the heap contains exactly the remaining elements $\{S[i], \ldots, S[n]\}$; extracting the minimum writes $S[i]$ to $B[i]$, and the heap strictly decreases in size by one. Consequently, after iteration n the heap is empty and (by repeated application of (I1)) we have $B[1..n] = S[1..n]$. Thus the algorithm outputs the fully sorted array and the heap has been completely drained. □

Conclusion: We showed that the loop invariant is true initially, maintained at every iteration, and implies that at termination $B = S$. Therefore, the algorithm is correct. □

Running Time Analysis:

For the initialization step, building the initial heap of size $k+1$ can be done in $O(k)$ time using the standard heapify procedure. Initializing B an empty array of size n takes $O(n)$ time.

For the iteration step, each of the n iterations performs one `extract-min` and possibly one `insert` on a heap of size at most $k+1$. Each of these operations costs $O(\log k)$ time. Thus, the total iteration step takes $O(n \log k)$ time.

Thus the total running time is

$$O(n) + n \cdot O(\log k) = O(n \log k).$$

Q3. The Hadamard Matrix

1. Orthogonality of distinct rows

We prove that distinct rows are orthogonal:

Theorem 1. *For any $m \geq 0$ and any $i \neq j$ with $1 \leq i, j \leq 2^m$,*

$$\sum_{\ell=1}^{2^m} H_{i\ell}^{(m)} H_{j\ell}^{(m)} \;=\; 0.$$

Equivalently, $(H^{(m)} \cdot (H^{(m)})^\top) = 2^m I_{2^m}$.

We proceed by strong induction on m.

Base case ($m = 0$). $H^{(0)}$ is 1×1; there are no distinct indices $i \neq j$ to check, so the claim is vacuously true. Moreover, $H^{(0)}(H^{(0)})^\top = [1] = 2^0 I_1$.

Inductive hypothesis. Assume the claim holds for $m - 1 \geq 0$, i.e.,

$$\sum_{\ell=1}^{2^{m-1}} H_{a\ell}^{(m-1)} H_{b\ell}^{(m-1)} = \begin{cases} 2^{m-1}, & a = b, \\ 0, & a \neq b. \end{cases}$$

Inductive step. Fix $m \geq 1$. Index the rows and columns of $H^{(m)}$ from 1 to 2^m. Split the column index ℓ into the left half $\ell \in L := \{1, \ldots, 2^{m-1}\}$ and the right half $\ell \in R := \{2^{m-1} + 1, \ldots, 2^m\}$. Similarly, split the rows into the top half $T := \{1, \ldots, 2^{m-1}\}$ and bottom half $B := \{2^{m-1}+1, \ldots, 2^m\}$.

By the block form,

$$H^{(m)} = \begin{array}{c} \\ T \\ B \end{array} \begin{array}{c} \begin{array}{cccc} L & R & H^{(m-1)} & H^{(m-1)} \end{array} \\ \begin{bmatrix} & & H^{(m-1)} & H^{(m-1)} \\ & & H^{(m-1)} & -H^{(m-1)} \end{bmatrix} \end{array}.$$

Thus, writing any row index as either $i \in T$ or $i \in B$, we have the following explicit descriptions:

- If $i \in T$ (i.e. $1 \leq i \leq 2^{m-1}$), then the i-th row equals

$$\left(H_{i,\cdot}^{(m-1)} \;\middle|\; H_{i,\cdot}^{(m-1)} \right),$$

  i.e. the concatenation of the same length-2^{m-1} row twice.

- If $i \in B$ (i.e. $i = i' + 2^{m-1}$ with $1 \le i' \le 2^{m-1}$), then the i-th row equals

$$\left(H^{(m-1)}_{i',\cdot} \;\middle|\; -H^{(m-1)}_{i',\cdot} \right),$$

  i.e. the concatenation of a row and its negation.

Fix distinct $i \neq j$. We compute the dot product $\sum_{\ell=1}^{2^m} H^{(m)}_{i\ell} H^{(m)}_{j\ell}$ by cases.

Case A: $i, j \in T$**.** Then

$$\sum_{\ell=1}^{2^m} H^{(m)}_{i\ell} H^{(m)}_{j\ell} = \sum_{\ell \in L} H^{(m-1)}_{i\ell} H^{(m-1)}_{j\ell} + \sum_{\ell \in R} H^{(m-1)}_{i,\ell-2^{m-1}} H^{(m-1)}_{j,\ell-2^{m-1}} = S + S = 2S,$$

where $S = \sum_{\ell=1}^{2^{m-1}} H^{(m-1)}_{i\ell} H^{(m-1)}_{j\ell}$. By the inductive hypothesis, since $i \neq j$, $S = 0$, hence the sum is 0.

Case B: $i, j \in B$**.** Write $i = i' + 2^{m-1}$ and $j = j' + 2^{m-1}$ with $i' \neq j'$. Then

$$\sum_{\ell=1}^{2^m} H^{(m)}_{i\ell} H^{(m)}_{j\ell} = \sum_{\ell \in L} H^{(m-1)}_{i'\ell} H^{(m-1)}_{j'\ell} + \sum_{\ell \in R} (-H^{(m-1)}_{i',\ell-2^{m-1}}) \, (-H^{(m-1)}_{j',\ell-2^{m-1}}) = S + S = 2S,$$

with the same S as above but for (i', j'). Since $i' \neq j'$, the inductive hypothesis gives $S = 0$, so the sum is 0.

Case C: $i \in T$**,** $j \in B$ **(the case** $i \in B$**,** $j \in T$ **is symmetric).** Write $j = j' + 2^{m-1}$. Then

$$\sum_{\ell=1}^{2^m} H^{(m)}_{i\ell} H^{(m)}_{j\ell} = \sum_{\ell \in L} H^{(m-1)}_{i\ell} H^{(m-1)}_{j'\ell} + \sum_{\ell \in R} H^{(m-1)}_{i,\ell-2^{m-1}} \cdot \left(-H^{(m-1)}_{j',\ell-2^{m-1}}\right) = S - S = 0,$$

where $S = \sum_{\ell=1}^{2^{m-1}} H^{(m-1)}_{i\ell} H^{(m-1)}_{j'\ell}$ (no condition on i and j' is needed here, since S cancels against itself).

In all cases, the dot product is 0. This proves the off-diagonal orthogonality. For $i = j$, the same computations give

$$\sum_{\ell=1}^{2^m} \left(H^{(m)}_{i\ell}\right)^2 = \begin{cases} 2 \cdot \sum_{\ell=1}^{2^{m-1}} \left(H^{(m-1)}_{i\ell}\right)^2, & i \in T, \\ 2 \cdot \sum_{\ell=1}^{2^{m-1}} \left(H^{(m-1)}_{i'\ell}\right)^2, & i \in B, \end{cases} = 2 \cdot 2^{m-1} = 2^m,$$

so each row has squared norm 2^m. Hence $H^{(m)}(H^{(m)})^\top = 2^m I$, completing the induction and the proof of Theorem 1. □

2. An $O(m2^m)$ algorithm for $H^{(m)}x$

2.1 English description (butterfly structure)

Let $x \in \mathbb{Z}^{2^m}$ (or $\mathbb{R}^{2^m}$) be a column vector. We describe an in-place algorithm (no extra asymptotic memory) that transforms x into $y = H^{(m)}x$ using only additions and subtractions in m *stages*. At each stage $s = 0, 1, \ldots, m-1$, we process the vector in contiguous *blocks* of length 2^{s+1}. Inside each block, we pair entries at distance 2^s and replace each pair (u, v) by the two numbers $(u+v,\ u-v)$. Concretely:

- Stage s partitions indices into blocks $B = \{t, t+1, \ldots, t+2^{s+1}-1\}$ for $t \in \{1,\ 1+2^{s+1},\ 1+2\cdot 2^{s+1}, \ldots\}$.
- Within each block B, for every *offset* $r = 0, 1, \ldots, 2^s - 1$, take the pair of positions $t+r$ and $t+r+2^s$ with current values (u, v) and overwrite them by

$$(u', v') = (u+v,\ u-v).$$

Intuitively, stage s applies in parallel the 2×2 transform $\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ to all pairs separated by stride 2^s. After m stages, the cumulative effect is exactly multiplication by $H^{(m)}$. Because each value participates in one addition and one subtraction per stage, the total work is $m \cdot 2^m$ arithmetic operations, and the algorithm runs in $O(m2^m)$ time and $O(1)$ extra space (beyond the output).

2.2 Pseudocode

Let us index arrays from 1 for consistency with the problem statement.

Input: integer $m \geq 0$, vector $x[1..2^m]$.

Output: $y = H^{(m)}x$ (stored back in x in-place or copied to y).

Algorithm (in-place):

for $s = 0, 1, \ldots, m-1$ **do**
  stride $\leftarrow 2^s$, blockLen $\leftarrow 2^{s+1}$
  for $t = 1,\ 1+\text{blockLen},\ 1+2\cdot\text{blockLen}, \ldots,\ 2^m - \text{blockLen} + 1$ **do**
    for $r = 0, 1, \ldots, \text{stride} - 1$ **do**
      $u \leftarrow x[t+r]$, $v \leftarrow x[t+r+\text{stride}]$
      $x[t+r] \leftarrow u+v$
      $x[t+r+\text{stride}] \leftarrow u-v$
    end for
  end for
end for

Return x.

2.3 Correctness proof (by induction on m)

We prove that after completing the m stages, the resulting vector equals $H^{(m)}x$.

Base case ($m = 0$). The vector has length 1 and the loop has no iterations; the output is $x = H^{(0)}x$, as required.

Inductive step. Assume the algorithm produces $H^{(m-1)}z$ from any input $z \in \mathbb{R}^{2^{m-1}}$ in $m-1$ stages. Consider an input $x \in \mathbb{R}^{2^m}$ and split it as a vertical concatenation $x = \begin{bmatrix} x^{(L)} \\ x^{(R)} \end{bmatrix}$ with each half in $\mathbb{R}^{2^{m-1}}$.

By the recursive definition,

$$H^{(m)}x = \begin{bmatrix} H^{(m-1)} & H^{(m-1)} \\ H^{(m-1)} & -H^{(m-1)} \end{bmatrix} \begin{bmatrix} x^{(L)} \\ x^{(R)} \end{bmatrix} = \begin{bmatrix} H^{(m-1)}(x^{(L)} + x^{(R)}) \\ H^{(m-1)}(x^{(L)} - x^{(R)}) \end{bmatrix}.$$

Now examine the algorithm's first stage ($s = 0$). It pairs entries at distance 1, i.e. it replaces each adjacent pair (u, v) by $(u + v, u - v)$. Applied across the vector, this transforms

$$\begin{bmatrix} x^{(L)} \\ x^{(R)} \end{bmatrix} \longmapsto \begin{bmatrix} x^{(L)} + x^{(R)} \\ x^{(L)} - x^{(R)} \end{bmatrix}.$$

Crucially, after stage $s = 0$ the top half and bottom half are exactly these two vectors. The remaining $m-1$ stages act *independently* on each half, because the block length doubles every stage: at stage $s \geq 1$ the stride is $2^s \geq 2$, so no pair crosses the boundary between the top and bottom halves. By the inductive hypothesis, the top half becomes $H^{(m-1)}(x^{(L)}+x^{(R)})$ and the bottom half becomes $H^{(m-1)}(x^{(L)}-x^{(R)})$. Concatenating these halves yields exactly $H^{(m)}x$. This completes the induction. □

2.4 Running time and space

At stage s, there are 2^m entries and each participates in exactly one pair operation $(u, v) \mapsto (u+v, u-v)$. Thus each stage performs $\dfrac{2^m}{2}$ pairs, i.e. 2^m additions/subtractions, counted as $O(2^m)$ arithmetic operations. There are m stages, so the total work is $O(m2^m)$. The algorithm is in-place, using $O(1)$ auxiliary space beyond the input/output storage.

2.5 Remarks (normalization and signs)

As written, the transform computes the *unnormalized* Hadamard product $y = H^{(m)}x$. If an orthonormal transform is desired (so that the transform matrix is orthogonal), one scales by $2^{-m/2}$: $\widetilde{y} = 2^{-m/2}H^{(m)}x$. Orthogonality from Theorem 1 immediately implies energy preservation $\|\widetilde{y}\|_2 = \|x\|_2$.

Summary

We proved by induction on m that distinct rows of $H^{(m)}$ are orthogonal (indeed, $H^{(m)}(H^{(m)})^\top = 2^m I$). We then gave a fast, in-place *butterfly* algorithm (the Fast Walsh–Hadamard Transform) that computes $y = H^{(m)}x$ in $O(m2^m)$ time using only additions and subtractions, and proved its correctness and runtime bound in detail.

Q4. Filling a Grid of Cells

Algorithm:

Let $n = 2^m$. The algorithm proceeds recursively by dividing the board into four quadrants and creating one artificial hole in each of the three quadrants that do not contain the original hole so that all four subproblems have exactly one hole each.

(a) **Base case ($m = 0$).** If the board is 1×1, it already consists solely of the prescribed hole; no tiles are placed and the algorithm returns.

(b) **Recursive step ($m \geq 1$).** Draw the horizontal and vertical midlines that split the board into four congruent $2^{m-1} \times 2^{m-1}$ quadrants:

$$\begin{aligned} Q_{11} &:= \{1, \ldots, 2^{m-1}\} \times \{1, \ldots, 2^{m-1}\} \quad \text{(top-left)}, \\ Q_{12} &:= \{1, \ldots, 2^{m-1}\} \times \{2^{m-1}+1, \ldots, 2^m\} \quad \text{(top-right)}, \\ Q_{21} &:= \{2^{m-1}+1, \ldots, 2^m\} \times \{1, \ldots, 2^{m-1}\} \quad \text{(bottom-left)}, \\ Q_{22} &:= \{2^{m-1}+1, \ldots, 2^m\} \times \{2^{m-1}+1, \ldots, 2^m\} \quad \text{(bottom-right)}. \end{aligned}$$

Let the four *central cells* adjacent to the intersection of the midlines be

$$c_{11} = (2^{m-1}, 2^{m-1}), \;\; c_{12} = (2^{m-1}, 2^{m-1}+1), \;\; c_{21} = (2^{m-1}+1, 2^{m-1}), \;\; c_{22} = (2^{m-1}+1, 2^{m-1}+1).$$

Exactly one quadrant, denote it $Q_{a^*b^*}$, contains the given hole (i, j).

- **Place one central L-tromino.** Place an L-shaped tromino covering the three central cells

$$\{c_{ab} : (a, b) \neq (a^*, b^*)\}.$$

  This leaves $c_{a^*b^*}$ uncovered, while all other quadrants now treat their covered central cell as their "hole."

- **Recurse on quadrants.** For each quadrant Q_{ab}:

$$\text{hole}(Q_{ab}) = \begin{cases} (i, j), & \text{if } (a, b) = (a^*, b^*), \\ c_{ab}, & \text{otherwise.} \end{cases}$$

  Apply the same procedure recursively to Q_{ab} with its designated hole.

The output is the union of all recursively placed trominoes plus the single central tromino at this level.

Pseudocode:

Procedure TILE(m, Q, h) // Tile a $2^m \times 2^m$ board Q with hole at h.

(a) **if** $m = 0$ **then return** // base case

(b) Partition Q into four quadrants $Q_{11}, Q_{12}, Q_{21}, Q_{22}$.

(c) Let $c_{11}, c_{12}, c_{21}, c_{22}$ be the four central cells at the intersection of the partition.

(d) Determine which quadrant $Q_{a^*b^*}$ contains the hole h.

(e) Place one L-tromino covering $\{c_{ab} : (a, b) \neq (a^*, b^*)\}$.

(f) For each $(a, b) \in \{1, 2\}^2$:

i. **if** $(a, b) = (a^*, b^*)$ **then** $h_{ab} \leftarrow h$.

ii. **else** $h_{ab} \leftarrow c_{ab}$.

iii. TILE$(m - 1, Q_{ab}, h_{ab})$.

(g) **return**

Proof of Correctness:

We prove by induction on $m \geq 0$ that the algorithm constructs a valid tiling of the $2^m \times 2^m$ board that covers every cell except the designated hole.

Base case ($m = 0$). The board has one cell, which is the hole. The algorithm places no tiles; thus every non-hole cell (there are none) is covered, satisfying the specification vacuously.

Inductive step. Assume the claim holds for boards of size $2^{m-1} \times 2^{m-1}$, i.e., for any chosen hole in such a board, the algorithm returns a valid tiling that covers all and only the non-hole cells.

Consider a $2^m \times 2^m$ board with hole (i, j). The algorithm identifies the quadrant $Q_{a^*b^*}$ containing the hole and places one L-triomino on the three central cells not in $Q_{a^*b^*}$. This placement is valid because:

- The three chosen central cells are pairwise adjacent in an L-configuration (two share a corner with the third), hence they form exactly one L-triomino shape.
- None of these three cells equals the designated hole (i, j) since the latter lies within $Q_{a^*b^*}$.
- This central triomino does not overlap with any future tile because all subsequent tiles are restricted to their respective quadrants (see below).

After this placement, each quadrant Q_{ab} has exactly one missing cell:

- $Q_{a^*b^*}$ has the original hole (i, j).
- Any Q_{ab} with $(a, b) \neq (a^*, b^*)$ has its central-adjacent cell c_{ab} designated as the hole.

By the inductive hypothesis applied independently to each quadrant (all are of size $2^{m-1} \times 2^{m-1}$), the recursive calls produce valid tilings within their respective quadrant boundaries that cover precisely the non-hole cells of those quadrants.

Finally, the union of the four quadrant tilings together with the one central L-triomino covers:

- all cells in the three quadrants $(a, b) \neq (a^*, b^*)$ except their chosen holes c_{ab} (which are already accounted for by the central L-triomino), and
- all cells in the quadrant $Q_{a^*b^*}$ except (i, j).

No overlaps occur because (1) the central L-triomino lies entirely within the 2×2 block of the four central cells, and (2) the quadrant tilings are confined to their quadrants. Therefore every cell of the $2^m \times 2^m$ board is covered exactly once except the designated hole (i, j), as required.

By induction, the algorithm is correct for all $m \geq 0$.

Running-Time Analysis

Let $T(m)$ denote the time to tile a $2^m \times 2^m$ board with a designated hole using the algorithm above. The work at level m consists of:

- identifying the quadrant containing (i, j) and placing *one* central L-triomino: $O(1)$ time,
- making four recursive calls on boards of size $2^{m-1} \times 2^{m-1}$.

Thus,

$$T(m) \;=\; 4\,T(m-1) \;+\; O(1), \qquad T(0) = O(1).$$

Now $T(m)$ denote the running time on a $2^m \times 2^m$ board. As argued,

$$T(m) = 4\,T(m-1) + f(m), \qquad T(0) = d,$$

where $f(m)$ is the non-recursive work at level m (identifying the hole's quadrant and placing the single central tromino). In our algorithm as described earlier, $f(m) = \Theta(1)$.

Part A. Exact solution when $f(m) \equiv c$ (a fixed constant).

Claim. If $T(m) = 4T(m-1) + c$ for all $m \geq 1$ and $T(0) = d$, then

$$T(m) = \Big(d + \frac{c}{3}\Big)4^m - \frac{c}{3}\,.$$

Proof (by induction on m). Base $m = 0$: $T(0) = d$ and the formula gives $\big(d + \frac{c}{3}\big)4^0 - \frac{c}{3} = d$, which matches.

IH: Assume the formula holds for $m - 1$.

Step to m: By the recurrence and the IH,

$$T(m) = 4\Big[\Big(d + \frac{c}{3}\Big)4^{m-1} - \frac{c}{3}\Big] + c = \Big(d + \frac{c}{3}\Big)4^m - \frac{4c}{3} + c = \Big(d + \frac{c}{3}\Big)4^m - \frac{c}{3}.$$

Thus the formula holds for m **by IH**. □

Immediate corollary: $T(m) = \Theta(4^m) = \Theta\big((2^m)^2\big)$, i.e., $\Theta(n^2)$ with side length $n = 2^m$.

Part B. Θ-bounds with only constant envelopes for $f(m)$.

Assume there exist constants $c_{\min}, c_{\max} \geq 0$ and constants $d_{\min}, d_{\max}$ such that

$$d_{\min} \leq T(0) \leq d_{\max}, \qquad c_{\min} \leq f(m) \leq c_{\max} \quad \text{for all } m \geq 1.$$

Then $T(m)$ satisfies the pair of recurrences

$$\underbrace{4T(m-1)+c_{\min}}_{\text{lower}} \;\le\; T(m) \;\le\; \underbrace{4T(m-1)+c_{\max}}_{\text{upper}}.$$

We prove matching upper and lower bounds by induction.

Upper bound. We claim there exist constants $A, B > 0$ with $B \ge \frac{c_{\max}}{3}$ and $A \ge d_{\max} + B$ such that

$$T(m) \;\le\; A \cdot 4^m - B \qquad \text{for all } m \ge 0.$$

Proof (by induction). Base $m = 0$: $T(0) \le d_{\max} \le A - B$, so $T(0) \le A \cdot 4^0 - B$ holds.

IH: Assume $T(m-1) \le A \cdot 4^{m-1} - B$.

Step to m: Using $T(m) \le 4T(m-1) + c_{\max}$ and **by IH**,

$$T(m) \le 4\big(A \cdot 4^{m-1} - B\big) + c_{\max} = A \cdot 4^m - (4B - c_{\max}) \le A \cdot 4^m - B,$$

since $4B - c_{\max} \ge B$ is ensured by $B \ge c_{\max}/3$. □

Lower bound. We claim there exist constants a, b with $0 \le b \le \frac{c_{\min}}{3}$ and $a \le d_{\min} + b$ such that

$$T(m) \;\ge\; a \cdot 4^m - b \qquad \text{for all } m \ge 0.$$

Proof (by induction). Base $m = 0$: $T(0) \ge d_{\min} \ge a - b$, so $T(0) \ge a \cdot 4^0 - b$ holds.

IH: Assume $T(m-1) \ge a \cdot 4^{m-1} - b$.

Step to m: Using $T(m) \ge 4T(m-1) + c_{\min}$ and **by IH**,

$$T(m) \ge 4\big(a \cdot 4^{m-1} - b\big) + c_{\min} = a \cdot 4^m - (4b - c_{\min}) \ge a \cdot 4^m - b,$$

since $4b - c_{\min} \le b$ is ensured by $b \le c_{\min}/3$. □

Combining the two bounds,

$$a \cdot 4^m - b \;\le\; T(m) \;\le\; A \cdot 4^m - B,$$

for appropriate constants a, b, A, B depending only on $d_{\min}, d_{\max}, c_{\min}, c_{\max}$. Hence

$$T(m) = \Theta(4^m) = \Theta\big((2^m)^2\big).$$

Q5. Binary Search with a Twist