4.4 The recursion-tree method

- Let us see how a recursion tree would provide a good guess for the recurrence
  \[ T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \]
- Start by finding an upper bound
- Floors and ceilings usually do not matter when solving recurrences
- Create a recursion tree for the recurrence \( T(n) = 3T(n/4) + cn^2 \) having written out the implied constant coefficient \( c > 0 \)
Subproblem sizes decrease by a factor of 4 each time we go down one level =>
- we eventually must reach a boundary condition

Subproblem size for a node at depth $i$ is $n/4^i$
- It hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$

Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$)
Each level has $3 \times$ the nodes of level above – the number of nodes at depth $i$ is $3^i$.

Subproblem sizes reduce by a factor of 4 for each level we go down, each node at depth $i$, $i = 0, 1, 2, \ldots, \log_4 n - 1$, has cost of $c(n/4^i)^2$.

Multiplying, we see that the total cost over all nodes at depth $i$ is $3^i c(n/4^i)^2 = (3/16)^i c n^2$.

The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} T(1)$, which is $\Theta(n^{\log_4 3})$, since $T(1)$ is a constant.

Add up the costs over all levels to determine the cost for the entire tree:

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left( \frac{3}{16} \right)^i c n^2 + \Theta(n^{\log_4 3})$$

$$= \frac{(3/16)^{\log_4 n - 1} - 1}{(3/16) - 1} c n^2 + \Theta(n^{\log_4 3})$$

By the value of a geometric (or exponential) series

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$
We can use an infinite decreasing geometric series as an upper bound

\[
T(n) = \sum_{i=0}^{\log_4 n-1} \left( \frac{3}{16} \right)^i cn^2 + \Theta(n^{\log_4 3})
\]

\[
< \sum_{i=0}^{\infty} \left( \frac{3}{16} \right)^i cn^2 + \Theta(n^{\log_4 3})
\]

\[
= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3})
\]

\[
= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2)
\]

We have derived a guess of \( T(n) = O(n^2) \) for our recurrence \( T(n) = 3T([n/4]) + \Theta(n^2) \)

- Root of the tree contributes \( cn^2 \) to the total cost – a constant fraction of the total cost
  - The cost of the root dominates the total cost
- If \( O(n^2) \) is indeed an upper bound for the recurrence, then it must be a tight bound
- First recursive call contributes cost of \( \Theta(n^2) \), \( \Omega(n^2) \) must be a lower bound for the recurrence
• Use the substitution method to verify that the guess was correct, i.e., $T(n) = O(n^2)$ is an upper bound for the recurrence

We want to show that $T(n) \leq dn^2$ for some constant $d > 0$

$$T(n) = 3T([n/4]) + cn^2$$
$$\leq 3d[n/4]^2 + cn^2$$
$$\leq 3d(n/4)^2 + cn^2$$
$$= \frac{3}{16}dn^2 + cn^2$$
$$\leq dn^2$$

as long as $d \geq (16/13)c$

• A more intricate example is the recurrence

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

• $c$ represents the constant in the $O(n)$ term

• Adding the values across the levels of the recursion tree yields value $cn$ for every level

• The longest simple path from the root to a leaf is $n \rightarrow (2/3)n \rightarrow (2/3)^2n \rightarrow \cdots \rightarrow 1$

• $(2/3)^k n = 1$ when $k = \log_{3/2} n$

• The height of the tree is $\log_{3/2} n$
We expect the solution to the recurrence to be \( \leq \) number of levels \( \times \) cost of each level: 
\[
O(cn \log_{3/2} n) = O(n \lg n)
\]

- Not every level contributes a cost of \( cn \)
- If this recursion tree were a complete binary tree of height \( \log_{3/2} n \), there would be \( 2^{\log_{3/2} n} = n^{\log_{3/2} 2} \) leaves
- The cost of each leaf is a constant
- The total cost of all leaves would then be \( \Theta(n^{\log_{3/2} 2}) \) which, since \( \log_{3/2} 2 > 1 \), is \( \omega(n \lg n) \)
• This recursion tree is not a complete binary tree, however, and has fewer than $n^{\log_{3/2} 2}$ leaves.
• As we go down from the root, more and more internal nodes are absent.
• Consequently, levels toward the bottom contribute less than $cn$ to the total cost.
• We can use the substitution method to verify that $O(n \lg n)$ is an upper bound for the solution to the recurrence.

### 4.5 The master method

Theorem 4.1 (*Master theorem*)

Let $a \geq 1$ and $b > 1$, and $f(n)$ a function, $T(n)$ is defined on nonneg. integers by recurrence

$$T(n) = aT(n/b) + f(n),$$

where we $n/b$ means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. $T(n)$ has the following asymptotic bounds:
1. If \( f(n) = O(n^{\log_b a-\epsilon}) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b a}) \)

2. If \( f(n) = \Theta(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a \log n}) \)

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

In each of the three cases, we compare the function \( f(n) \) with the function \( n^{\log_b a} \):

- The larger of the two determines the solution to the recurrence:
  - In case 1, the function \( n^{\log_b a} \) is the larger, then the solution is \( T(n) = \Theta(n^{\log_b a}) \)
  - In case 3, the function \( f(n) \) is the larger, then the solution is \( T(n) = \Theta(f(n)) \)
  - In case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is \( T(n) = \Theta(n^{\log_b a \log n}) \)
• Be aware of some technicalities:
• In case 1, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be polynomially smaller.
• $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of $n^\epsilon$ for some $\epsilon > 0$.
• In case 3, not only must $f(n)$ be larger than $n^{\log_b a}$, it also must be polynomially larger and satisfy the “regularity” condition that $af(n/b) \leq cf(n)$.

Using the master method

$T(n) = 9T(n/3) + n$

• We have $a = 9$, $b = 3$, $f(n) = n$, and thus $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.
• Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.
\[ T(n) = T\left(\frac{2n}{3}\right) + 1 \]

- Here \( a = 1, \ b = 3/2, \ f(n) = 1, \) and \( n^{\log_b a} = n^{\log_3 1} = n^0 = 1 \)
- Case 2 applies, since \( f(n) = \Theta\left(n^{\log_b a}\right) = \Theta(1), \) and thus the solution to the recurrence is \( T(n) = \Theta(\lg n) \)

\[ T(n) = 3T\left(\frac{n}{4}\right) + n\lg n \]

- We have \( a = 3, \ b = 4, \ f(n) = n\lg n, \) and \( n^{\log_b a} = n^{\log_4 3} = O(n^{0.793}) \)
- Since \( f(n) = \Omega(n^{\log_4 3 + \epsilon}), \) where \( \epsilon \approx 0.2, \) case 3 applies provided that the regularity condition holds for \( f(n) \)
- For sufficiently large \( n, \) we have that \( af\left(n/b\right) = 3\left(n/4\right)\lg\left(n/4\right) \leq (3/4)n\lg n = cf(n) \) for \( c = 3/4 \)
- By case 3, the solution is \( T(n) = \Theta(n\lg n) \)
The master method does not apply to
\[ T(n) = 2T(n/2) + n \log n \]
even if it seems to have the proper form:
\[ a = 2, \ b = 2, \ f(n) = n \log n, \ \text{and} \ n^{\log_b a} = n \]
One might think that case 3 applies, since
\[ f(n) \] is asymptotically larger than \( n^{\log_b a} \)
It is, however, not polynomially larger: ratio
\[ f(n)/n^{\log_b a} = \log n \] is asymptotically less than \( n^\varepsilon \) for any positive constant \( \varepsilon \)
Consequently, the recurrence falls into the gap between case 2 and case 3

The recurrence of the running time of the simple divide-and-conquer algorithm for matrix multiplication
\[ T(n) = 8T(n/2) + \Theta(n^2) \]
Now \( a = 8, \ b = 2, \) and \( f(n) = \Theta(n^2) \) and so
\[ n^{\log_b a} = n^{\log_2 8} = n^3 \]
Since \( n^3 \) is polynomially larger than \( f(n) \)
– i.e., \( f(n) = O(n^{3-\varepsilon}) \) for \( \varepsilon = 1 \)
case 1 applies, and \( T(n) = \Theta(n^3) \)
The recurrence of the running time of Strassen’s algorithm

\[ T(n) = 7T(n/2) + \Theta(n^2) \]

Here we have \( a = 7, \ b = 2, \) and \( f(n) = \Theta(n^2) \)
and so \( n^{\log_b a} = n^{\log_2 7} = n^{\lg 7} \)

Recalling that \( 2.80 < \lg 7 < 2.81 \), we see that \( f(n) = \Theta(n^{\lg 7}) \) for \( \epsilon = 0.8 \)

Again, case 1 applies, and we have the solution \( T(n) = \Theta(n^{\lg 7}) \)

---

5 Probalistic Analysis and Randomized Algorithms

We need to hire a new office assistant with the help of an employment agency that sends us a candidate each day.

We interview that person and then decide either to hire her or not.

We pay the agency a fee for each interview.

To actually hire an applicant is more costly
- We must fire the current office assistant and
- pay a substantial hiring fee to the agency.
5.1 The hiring problem

- We are committed to having, at all times, the best possible person for the job
- After interviewing an applicant, if she is better qualified than the current assistant, we will
  - fire the current office assistant and hire the new applicant
- We are willing to pay the price of this strategy, but wish to estimate what that price will be

\[
\text{HIRE-ASSISTANT}(n)
\]

1. \( \text{best} \leftarrow 0 \) // candidate 0 is a least-qualified dummy candidate
2. \( \text{for } i \leftarrow 1 \text{ to } n \)
3. interview candidate \( i \)
4. if candidate \( i \) is better than \( \text{best} \)
5. \( \text{best} \leftarrow i \)
6. hire candidate \( i \)
Interviewing has a low cost, \( c_i \), whereas hiring is expensive, costing \( c_h \).

Letting \( m \) be the number of people hired, the total cost of the algorithm is \( O(c_i n + c_h m) \).

No matter how many people we hire, we always interview \( n \) candidates and thus always incur the cost \( c_i n \).

Let us concentrate on analyzing \( c_h m \), the hiring cost.

This quantity varies with each run of the algorithm.

---

**Worst-case analysis**

In the worst case, we hire every candidate that we interview

– candidates come in increasing order of quality

– we hire \( n \) times, for a total hiring cost of \( c_h n \).

We have no idea about the order in which the candidates arrive, nor do we have any control over this order.

It is natural to ask what we expect to happen in a typical or average case.
Probabilistic analysis

• We can assume that the applicants come in a random order
• Assume that we can compare any two candidates and decide the better qualified
  ▶ there is a total order on the candidates
• Rank each candidate with a unique number 1,...,n
• Use \textit{rank}(i) to denote the rank of applicant \textit{i}

A higher rank corresponds to a better qualified applicant

• Ordered list \( \langle \text{rank}(1), \text{rank}(2),...,\text{rank}(n) \rangle \) is a permutation of the list \( \langle 1,2,...,n \rangle \)
• “The applicants come in a random order” ≡
  – “this list of ranks is equally likely to be any one of the \( n! \) permutations of the numbers 1 through \( n \)”
• Alternatively, the ranks form a uniform random permutation; i.e., each of the possible \( n! \) permutations appears with equal probability
Randomized algorithms

- We call an algorithm randomized if its behavior is determined also by values produced by a random-number generator \texttt{RANDOM}
- A call to \texttt{RANDOM}(a, b) returns an integer between \(a\) and \(b\), inclusive, with each such integer being equally likely
- E.g., \texttt{RANDOM}(0, 1) produces 0 with probability \(1/2\), and 1 with probability \(1/2\)

- A call to \texttt{RANDOM}(3, 7) returns either 3, 4, 5, 6, 7, each with probability \(1/5\)
- Each integer returned by \texttt{RANDOM} is independent of the integers returned on previous calls
- In analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the \texttt{RANDOM}
- We refer to the running time of a randomized algorithm as an expected running time
5.2 Indicator random variables

- Indicator random variables provide a convenient method for converting between probabilities and expectations.
- Suppose we are given a sample space $S$ and an event $A$.
- Then the indicator random variable $I\{A\}$ associated with $A$ is defined as
  
  $I\{A\} = \begin{cases} 
  1 & \text{if } A \text{ occurs} \\
  0 & \text{if } A \text{ does not occur} 
  \end{cases}$

Let us determine the expected number of heads that we obtain when flipping a fair coin.

- Our sample space is $S = \{H, T\}$, with $\Pr\{H\} = \Pr\{T\} = \frac{1}{2}$.
- We define an indicator random variable $X_H$, associated with the coin coming up heads ($H$).
- This variable counts the number of heads obtained in this flip.
  
  $X_H = I\{H\} = \begin{cases} 
  1 & \text{if } H \text{ occurs} \\
  0 & \text{if } T \text{ occurs} 
  \end{cases}$
Recall that $E[X] = \sum_x x \cdot P_r\{X = x\}$

The expected number of heads obtained in one flip of the coin is simply the expected value of our indicator variable $X_H$

$$E[X_H] = E[I\{H\}]$$
$$= 1 \cdot P_r\{H\} + 0 \cdot P_r\{T\}$$
$$= 1 \cdot \left(\frac{1}{2}\right) + 0 \cdot \left(\frac{1}{2}\right)$$
$$= \frac{1}{2}$$

Thus the expected number of heads obtained by one flip of a fair coin is $\frac{1}{2}$

The expected value of an indicator random variable associated with an event $A$ is equal to the probability that $A$ occurs

**Lemma 5.1**  \textit{Given a sample space $S$ and an event $A$ in $S$, let $X_A = I\{A\}$. Then $E[X_A] = P_r\{A\}$}

**Proof**  By the definition of an indicator and the definition of expected value, we have $E[X_A] = E[I\{A\}] = 1 \cdot P_r\{A\} + 0 \cdot P_r\{\bar{A}\} = P_r\{A\}$, where $\bar{A}$ denotes $S - A$, the complement of $A$
• Compute the number of heads in $n$ coin flips
• Let $X_i$ be the indicator associated with the event in which the $i$th flip comes up heads:
  
  \[ X_i = I\{\text{the } i\text{th flip results in the event } H\} \]

• Let $X$ denote the total number of heads in the $n$ coin flips, so that
  
  \[ X = \sum_{i=1}^{n} X_i \]

• To compute the expected number of heads, take the expectation of both sides to obtain
  
  \[ E[X] = E\left[ \sum_{i=1}^{n} X_i \right] \]

• Linearity of expectation
  
  \[ E[X_1 + X_2] = E[X_1] + E[X_2] \]

  makes the use of indicator random variables a powerful analytical technique
  
  – it applies even when there is dependence among the variables

• We now can easily compute the expected number of heads:
  
  \[
  E[X] = E\left[ \sum_{i=1}^{n} X_i \right] = \sum_{i=1}^{n} E[X_i] \\
  \sum_{i=1}^{n} 1/2 = n/2
  \]
Analysis of the hiring problem

- Let $X$ be the random variable whose value is the number of times we hire a new assistant.

- Let $X_i$ be the indicator random variable associated with the event in which the $i$th candidate is hired.

  $$X_i = \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{if candidate } i \text{ is not hired} \end{cases}$$

- $X = X_1 + X_2 + \cdots + X_n$

- By Lemma 5.1

  $$E[X_i] = \Pr\{\text{candidate } i \text{ is hired}\}$$

  and we must compute the probability that lines 5–6 of `HIRE-ASSISTANT` are executed.

- Candidate $i$ is hired (line 6) exactly when she is better than each of candidates $1, \ldots, i - 1$.

- We assume that the candidates arrive in a random order: the first $i$ candidates have appeared in a random order.

- Any one of these first $i$ candidates is equally likely to be the best-qualified so far.
• Candidate $i$ has a probability of $1/i$ of being better qualified than candidates $1, \ldots, i - 1$ and thus a probability of $1/i$ of being hired.
• By Lemma 5.1, we conclude that $E[X_i] = 1/i$
• Now we can compute $E[X]$:

$$E[X] = E \left[ \sum_{i=1}^{n} X_i \right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} 1/i = \ln n + O(1)$$

by harmonic series

$$H_n = \sum_{k=1}^{n} 1/k = \ln n + O(1)$$

• We interview $n$ people, but actually hire only approx. $\ln n$ of them, on average
• The following lemma summarize this

**Lemma 5.2** Assuming that candidates come in random order, algorithm HIRE-ASSISTANT has an average-case total hiring cost of $O(c_h \ln n)$

**Proof** Follows immediately from our definition of the hiring cost and the equation, which shows that the expected number of hires is approximately $\ln n$
5.3 Randomized algorithms

- The algorithm above is deterministic
- The number of times we hire a new office assistant differs for different inputs
- Given the rank list $A_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, a new assistant is always hired 10 times
- For the list of ranks $A_2 = \{10, 9, 8, 7, 6, 5, 4, 3, 2, 1\}$, a new assistant is hired only once
- $A_3 = \{5, 2, 1, 8, 4, 7, 10, 9, 3, 6\}$, a new assistant is hired three times

Consider an algorithm that first permutes the input before determining the best candidate
- Now we randomize in the algorithm, not in the input distribution
- For input $A_3$ we cannot say how many times the best is updated – it differs with each run
- Not even an enemy can produce a bad input, random permutation yields input order irrelevant
- The algorithm performs badly only if the random-number generator produces an “unlucky” permutation
**Randomized-Hire-Assistant**(\(n\))

1. randomly permute the list of candidates
2. \(\text{best} \leftarrow 0\) // candidate 0 is a least-qualified
   // dummy candidate
3. for \(i \leftarrow 1\) to \(n\)
4. interview candidate \(i\)
5. if candidate \(i\) is better than candidate \(\text{best}\)
6. \(\text{best} \leftarrow i\)
7. hire candidate \(i\)

---

**Randomly permuting arrays**

- We assume that we are given an array \(A\) which, w.l.o.g., contains the elements \(1, 2, \ldots, n\)
- Produce a random permutation of the array
- Assign element \(A[i]\) a random priority \(P[i]\), and sort the elements according to priorities
- E.g., if our initial array is \(A = \langle 1, 2, 3, 4 \rangle\) and we choose random priorities \(P = \langle 36, 3, 62, 19 \rangle\), we would produce array \(B = \langle 2, 4, 1, 3 \rangle\)
- We call this procedure **PERMUTE-BY-SORTING**
**Permuted-by-Sorting**

1. \( n \leftarrow A.\text{length} \)
2. let \( P[1..n] \) be a new array
3. for \( i \leftarrow 1 \) to \( n \)
4. \( P[i] \leftarrow \text{RANDOM}(1,n^3) \)
5. sort \( A \), using \( P \) as sort keys
   - We use a range of \( 1 \) to \( n^3 \) for random numbers to make it likely that all the priorities in \( P \) are unique

It remains to prove that the procedure produces a **uniform random permutation**,
   - It is equally likely to produce every permutation of the numbers \( 1 \) through \( n \)

**Lemma 5.4** **Permuted-by-Sorting** produces a uniform random permutation of the input, assuming that all priorities are distinct

**Proof** See the book.
• It is better to permute the given array in place
• RANDOMIZE-IN-PLACE does so in \( O(n) \) time
• In its \( i \)th iteration, it chooses the element \( A[i] \) randomly from among elements \( A[i] \) through \( A[n] \)
• After the \( i \)th iteration, \( A[i] \) is never altered

\[
\text{RANDOMIZE-IN-PLACE}(A) \\
1 \quad n \leftarrow A.\ length \\
2 \quad \text{for } i \leftarrow 1 \text{ to } n \\
3 \quad \text{swap } A[i] \text{ with } A[\text{RANDOM}(i,n)]
\]

• A \( k \)-permutation on a set of \( n \) elements is a sequence containing \( k \) of the \( n \) elements, with no repetitions
• There are \( n!/(n-k)! \) \( k \)-permutations

**loop invariant:**
• Just prior to the \( i \)th iteration of the for loop of lines 2–3, for each possible \( (i-1) \)-permutation of the \( n \) elements, the subarray \( A[1..i-1] \) contains this \( (i-1) \)-permutation with probability \( (n-i+1)!/n! \)
**Initialization:** loop invariant trivially holds

**Maintenance:**
- Consider a particular $i$-permutation, and denote the elements in it by $\langle x_1, x_2, \ldots, x_i \rangle$
- This permutation consists of an $(i-1)$-permutation $\langle x_1, x_2, \ldots, x_{i-1} \rangle$ followed by the value $x_i$ that the algorithm places in $A[i]$
- Let $E_1$ denote the event in which the first $(i-1)$ iterations have created the particular $(i-1)$-permutation $\langle x_1, x_2, \ldots, x_{i-1} \rangle$ in $A[1..i-1]$

By the loop invariant, $\Pr\{E_1\} = (n-i+1)!/n!$
- Let $E_2$ be the event that $i$th iteration puts $x_i$ in position $A[i]$
- The $i$-permutation $\langle x_1, x_2, \ldots, x_i \rangle$ appears in $A[1..i]$ precisely when both $E_1$ and $E_2$ occur
  $$\Pr\{E_2 \cap E_1\} = \Pr\{E_2|E_1\}\Pr\{E_1\}$$
- $\Pr\{E_2|E_1\} = 1/(n-i+1)$ because in line 3 the algorithm chooses $x_i$ randomly from the $n-i+1$ values in positions $A[i..n]$
- $\Pr\{E_2 \cap E_1\} = \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} = \frac{(n-i)!}{n!}$
Termination:

- At termination, \( i = n + 1 \), and we have that the subarray \( A[1..n] \) is a given \( n \)-permutation with probability

\[
\frac{(n - (n + 1) + 1)!}{n!} = \frac{0!}{n!} = 1/n!
\]

- Thus, RANDOMIZE-IN-PLACE produces a uniform random permutation