1. Suppose we are given a bit string $B[1..n]$. A triple of indices $1 \leq i < j < k \leq n$ is called a well-spaced triple if $B[i] = B[j] = B[k] = 1$ and $k - j = j - i$.

(a) Describe a brute-force algorithm to determine whether $B$ has a well-spaced triple in $O(n^2)$ time.

Solution:

```python
BruteForceTriple(B[1..n]):
    for i ← 1 to n
        for j ← i + 1 to n
            k ← 2j - i
            if k ≤ n and B[i] = 1 and B[j] = 1 and B[k] = 1
                return True
        return False
```

Rubric: 2 points. This is not the only brute-force solution.

(b) Describe an algorithm to determine whether $B$ has a well-spaced triple in $O(n \log n)$ time. [Hint: FFT!]

Solution: Let $BB$ denote the convolution of the input array $B$ with itself, that is, $BB = B * B$. We can compute $BB$ in $O(n \log n)$ time using any FFT algorithm.

Consider an index $j$ such that $B[j] = 1$. If $j$ is the middle index in an evenly-spaced triple $i, j, k$ in $B$, then each pair $(a, b) \in \{(i, k), (j, j), (k, i)\}$ contributes 1 to the summation $BB[2j] = \sum_{a+b=2j} B[a] \cdot B[b]$, and thus $BB[2j] \geq 3$. On the other hand, if $j$ is not the middle index of any evenly-spaced triple, then $BB[2j] = 1$.

Finally, if $B[j] = 0$, then $j$ cannot be part of an evenly-spaced triple.

```python
FFTTriple(B[0..n]):
    ⟨⟨pad for convolution⟩⟩
    for i ← n + 1 to 2n
        B[i] ← 0
    ⟨⟨compute convolution $B * B$⟩⟩
    $B^* = FFT(B)$
    for i ← 0 to 2n
        $BB^*[i] ← B^*[i] \cdot B^*[i]$  
    $BB ← InverseFFT(BB^*)$
    ⟨⟨look for evenly-spaced triples⟩⟩
    for j ← 0 to n
        if $B[j] = 1$ and $BB[2j] > 1$
            return True
    return False
```

This algorithm runs in $O(n \log n)$ time; the running time is dominated by the calls to FFT and InverseFFT.
Solution: If the algorithm from part (c) returns a positive integer, return True; otherwise, return False.

Rubric: 4 points. The second solution should get exactly the same score as part (c).

(c) Describe an algorithm to determine the number of well-spaced triples in B in $O(n \log n)$ time.

Solution: As in part (b), let $BB = B \ast B$. We can compute $BB$ in $O(n \log n)$ time using any FFT algorithm.

Consider an index $j$ such that $B[j] = 1$. If $j$ is the middle index in an evenly-spaced triple $i, j, k$ in $B$, then each pair $(a, b) \in \{(i, k), (j, j), (k, i)\}$ contributes 1 to the summation $BB[2j] = \sum_{a+b=2j} B[a] \cdot B[b]$. It follows that $j$ is the middle index of $c$ evenly-spaced triples if and only if $BB[2j] = 2c + 1$.

```
FFTTriple(B[0..n]):
   ⟨⟨pad for convolution⟩⟩
   for i ← n + 1 to 2n
      B[i] ← 0
   ⟨⟨compute convolution BB = B \ast B⟩⟩
   B* = FFT(B)
   for i ← 0 to 2n
      BB*[i] ← B*[i] \cdot B*[i]
   BB ← INVERSEFFT(BB*)
   ⟨⟨count evenly-spaced triples⟩⟩
   triples ← 0
   for j ← 0 to n
      if B[j] = 1
         triples ← triples + [BB[2j]/2]
   return triples
```

This algorithm runs in $O(n \log n)$ time; the running time is dominated by the calls to FFT and INVERSEFFT.

Rubric: 4 points.
2. This problem explores different algorithms for computing the factorial function \( n! \).

(a) Recall that the standard lattice algorithm that you learned in elementary school multiplies any \( n \)-bit integer and any \( m \)-bit integer in \( O(mn) \) time. Describe and analyze a variant of Karatsuba’s algorithm that multiplies any \( n \)-bit integer and any \( m \)-bit integer, for any \( n \geq m \), in \( O(n \cdot m \cdot \log 3 - 1) = O(n \cdot m^{0.58496}) \) time.

**Solution:** Let \( A[0..n-1] \) and \( B[0..m-1] \) denote the input bit arrays, respectively representing the integers \( a = \sum_{i=0}^{n-1} A[i] \cdot 2^i \) and \( b = \sum_{i=0}^{m-1} B[i] \cdot 2^i \).

Intuitively, we split the longer array \( A \) into about \( n/m \) chunks of length about \( m \), multiply each chunk by \( b \) using Karatsuba’s algorithm, and then combine the partial products by brute force.

For simplicity, I’ll assume that \( n \) is a multiple of \( m \); otherwise, we can pad the array \( A \) with at most \( m - 1 \) zeros.

```plaintext
UnbalancedKaratsuba(A[0..n-1], B[0..m-1]):
product ← 0
for j ← 0 to n/m - 1
    shift ← m \cdot j
    inch = \sum_{i=0}^{m-1} A[shift + i] \cdot 2^i (\text{\textit{Input CHunk}})
    ouch ← Karatsuba(inch, b) (\text{\textit{Output CHunk}})
    product ← product + ouch \cdot 2^{shift}
return product
```

Each iteration of the main loop calls \texttt{Karatsuba} with two \( m \)-bit integers, and performs \( O(m) \) other work, so the running time of each iteration is \( O(m \log 3) \). The for-loop repeats \( n/m \) times, so the overall algorithm runs in \( O(n \cdot m \log 3 \cdot n/m) = O(n \cdot m^{0.58496}) \) time as required.

**Rubric:** 2 points = 1 for algorithm + 1 for analysis
(b) Analyze the running time of \textsc{Factorial}(n) using different algorithms for the multiplication in line (*):

i. Lattice multiplication

**Solution:** In the \(k\)th iteration of the main loop, we are multiplying \(k\) and \((k-1)!\), which have \(\Theta(\log k)\) and \(\Theta(k \log k)\) bits, respectively. If we use the lattice algorithm, this multiplication takes \(O(k \log^2 k) = O(k(\log k)^2)\) time. So the running time of \textsc{Factorial}(n) is

\[
\sum_{k=1}^{n} O(k \log^2 k) \leq \left(\sum_{k=1}^{n} O(k)\right) \cdot O(\log^2 n) = O(n^2 \log^2 n).
\]

**Rubric:** 2 points = 1 for running time + 1 for justification

ii. Your variant of Karatsuba’s algorithm from part (a)

**Solution:** Now the \(k\)th multiplication takes \(O((k \log k) \cdot (\log k)^{\lg 3-1}) = O(k \log^{\lg 3} k)\) time. So the running time of \textsc{Factorial}(n) is

\[
\sum_{k=1}^{n} O(k \log^{\lg 3} k) \leq \left(\sum_{k=1}^{n} O(k)\right) \cdot O(\log^{\lg 3} n) = O(n^2 \log^{\lg 3} n).
\]

Alas, faster multiplication only speeds up \textsc{Factorial} by a sub-logarithmic factor.

**Rubric:** 2 points = 1 for running time + 1 for justification
(c) Analyze the running time of \textsc{FasterFactorial}(n) using different algorithms for the multiplication in the last line of \textsc{Falling}:

i. Lattice multiplication

**Solution:** Let \(T(n, m)\) denote the running time of \textsc{Falling}(n, m). Both factors \(n^{m/2}\) and \((n - m/2)^{m/2}\) are \(\Theta(m \log n)\) bits long.

If we use the lattice algorithm, the final multiplication in \textsc{Falling}(n, m) takes \(O(m^2 \log^2 n)\) time, which gives us the recurrence

\[T(n, m) = T(n, m/2) + T(n - m/2, m/2) + O(m^2 \log^2 n).\]

Multiplying larger numbers takes longer, so we can reasonably assume that \(T(n - m/2, m/2) \leq T(n, m/2)\) and simplify our recurrence:

\[T(n, m) \leq 2T(n, m/2) + O(m^2 \log^2 n).\]

Now define a new function \(t(m) = T(n, m)/\log^2 n\), which satisfies the even simpler recurrence

\[t(m) \leq 2t(m/2) + O(m^2),\]

whose solution via recursion trees is \(t(m) = O(m^2)\). It follows that \(T(n, m) \leq t(m) \log^2 n = O(m^2 \log n)\).

We conclude that \textsc{FasterFactorial}(n) runs in \(O(n^2 \log^2 n)\) time, just like \textsc{Factorial}(n).

**Rubric:** 2 points = 1 for running time + 1 for justification

ii. Your variant of Karatsuba’s algorithm from part (a)

**Solution:** Again, let \(T(n, m)\) denote the running time of \textsc{Falling}(n, m). If we use Karatsuba’s algorithm, the final multiplication in \textsc{Falling}(n, m) takes \(O(m^{\lg^3} \log^{\lg^3} n)\) time, giving us the recurrence

\[T(n, m) = T(n, m/2) + T(n - m/2, m/2) + O(m^{\lg^3} \log^{\lg^3} n).\]

Again, we can simplify using the inequality \(T(n - m/2, m/2) \leq T(n, m/2)\):

\[T(n, m) \leq 2T(n, m/2) + O(m^{\lg^3} \log^{\lg^3} n).\]

The helper function \(t(m) = T(n, m)/\log^{\lg^3} n\) satisfies the even simpler recurrence

\[t(m) \leq 2t(m/2) + O(m^{\lg^3}),\]

whose solution via recursion trees is \(t(m) = O(m^{\lg^3})\). It follows that \(T(n, m) \leq t(m) \log^{\lg^3} n = O(m^{\lg^3} \log^{\lg^3} n)\).

We conclude that \textsc{FasterFactorial}(n) runs in \(O(n^{\lg^3} \log^{\lg^3} n)\) time, which is significantly faster than \textsc{Factorial}(n)!

**Rubric:** 2 points = 1 for running time + 1 for justification
3. Your new boss at the Dixon Ticonderoga Pencil Factory asks you to design an algorithm to solve the following problem. Suppose you are given \( N \) pencils, each with one of \( c \) different colors, and a non-negative integer \( k \). \textbf{How many different ways are there to choose a set of \( k \) pencils?} Two pencil sets are considered identical if they contain the same number of pencils of each color.

Describe an algorithm to solve this problem, and analyze its running time. Your input is an array \( \text{Pencils}[1..c] \) and an integer \( k \), where \( \text{Pencils}[i] \) stores the number of pencils with color \( i \). Your output is a single non-negative integer. For full credit, report the running time of your algorithm as a function of the parameters \( N \), \( c \), and \( k \). Assume that \( k \ll c \ll N \).

This problem is based on a Jane Street interview question.

\begin{quote}
\textbf{Solution:} Following the hint, we associate the polynomial \( \sum_{i=1}^{\text{Pencils}[j]} x^i \) with each color \( j \). We need to compute the coefficient of \( x^k \) in the product polynomial \( \prod_{j=1}^{c} P_j(x) \). This coefficient is equal to the number of different ways of choosing one term from each polynomial \( P_j \) (that is, choosing a number of pencils of color \( j \)) such that the degrees of the chosen terms (that is, the chosen number of pencils of each color) sum to \( k \). \((\text{So far this is worth 2 points.})\)

First, the output polynomial \( P \) has degree \( N \). Thus, if we multiply the polynomials in a simple for-loop, each intermediate polynomial has degree at most \( N \). So each multiplication can be performed in \( O(N^2) \) time using the lattice algorithm. There are \( c \) factor polynomials, so computing their product requires \( c - 1 \) pairwise multiplications. The overall running time of this algorithm is \( O(cN^2) \). \((\text{So far this is worth 4 points.})\)

If instead we multiply polynomials using fast Fourier transforms, each multiplication takes \( O(N \log N) \) time, so the overall running time drops to \( O(cN \log N) \). \((\text{So far this is worth 6 points.})\)

We can further improve this algorithm using a divide-and-conquer strategy instead of a simple for-loop, similarly to the FALLING function in problem 2. To compute the product of \( c \) polynomials, we first recursively multiply the first \( c/2 \) polynomials, then recursively multiply the last \( c/2 \) polynomials, and finally multiply the two products using FFTs. The recursion tree has depth \( O(\log c) \), and the total time spent on multiplications in each level of the tree is \( O(N \log N) \), so the overall running time becomes \( O(N \log N \log c) \).\((\text{So far this is worth 8 points.})\)

Alternatively, because \( k \ll N \), we can improve our algorithm algorithm by discarding all terms with degree greater than \( k \) as soon as they arise. With this optimization, each multiplication involves two polynomials of degree at \( k \), and thus can be performed in \( O(k \log k) \) time via FFT. The overall running time drops to \( O(ck \log k) \). \((\text{So far this is worth 10 points (full credit) even without the binary tree optimization.})\)

But we can still do better! Because \( k \ll c \), we can improve the algorithm even further by observing that, after discarding all terms larger than \( x^k \), there are at most \( k \) distinct truncated polynomials \( P_j \). For any integer \( 1 \leq j \leq k \), let \( Q_j(x) = \sum_{i=1}^{j} x^i \). For all \( j < k \), let \( c_j \) denote the number of colors for which we have exactly \( j \) pencils, and let \( c_k \) denote the number of colors for which we have at least \( k \) pencils. Then the
first $k$ coefficients of $P(x)$ are equal to the first $k$ coefficients of the polynomial

\[ \tilde{P}(x) = \prod_{i=1}^{k} (Q_j(x))^{c_j}. \]

So we can proceed as follows:

- In $O(c + k)$ time, compute the exponent $c_j$ for every index $1 \leq j \leq k$.
- For each index $j$, compute the first $k$ coefficients of $(Q_j(x))^{c_j}$ by repeated squaring with $O(\log c_j)$ multiplications. If each multiplication uses FFTs, we can computing the first $k$ coefficients of $(Q_j(x))^{c_j}$ in $O(k \log k \log c_j) = O(k \log k \log c)$ time.\(^b\)
- Finally, compute the first $k$ coefficients of $\tilde{P}(k)$ from these $k$ polynomials in $O(k \cdot k \log k)$ time.

The overall running time of this algorithm is $O(c + k^2 \log k \log c)$, which is $o(ck \log k)$ if $k$ is sufficiently small compared to $c$.\(^c\) \(\langle\text{This is worth 13 points.}\rangle\)

Finally, if $k$ is sufficiently small—specifically, if $k = o(\sqrt{c} / \log c)$—the running time of this algorithm simplifies to $O(c)$.\(^d\) This is clearly optimal, because any correct algorithm must read the entire input! \(\langle\text{This is worth 15 points.}\rangle\)

---

I’ve described a sequence of optimizations, each leading to a different time bound. It is not obvious from the stated time bounds, but in fact, each optimization makes the algorithm faster (or at least no slower).

- Multiply pairs of polynomials using FFTs instead of the lattice algorithm.
- Organize multiplications in a binary tree instead of a simple for-loop.
- Discard terms with degree greater than $k$ whenever they arise. (This makes the binary tree optimization useful only for factors with degree less than $k$, but that’s still better than nothing.)
- Compute powers of (truncated) polynomials using repeated squaring instead of multiplying one by one.

\(^a\)In practice, this can be optimized further by using a Huffman code to optimally organize the multiplications; however, this optimization does not reduce the worst-case running time, when Pencils\(^i\) has roughly the same value for all $i$.

\(^b\)More careful analysis implies an even smaller upper bound $O(k \log k \log(c/k))$ for this part of the repeated-squaring algorithm; the running time is maximized when all exponents $c_j$ are equal.

\(^c\)If we used lattice multiplication everywhere instead of FFTs, the running time would be $O(c + k^3 \log c)$.

\(^d\)If $k = o(c^{1/3} / \log^{1/3} c)$, the running time is $O(c)$ even if we use lattice multiplication everywhere instead of FFTs.
Solution (dynamic programming): To save a bit of space, I’ll call the input array $P[1..c]$ instead of $Pencils[1..c]$.

For any integers $0 \leq i \leq c$ and $0 \leq s \leq k$, let $NumSets(i, n)$ denote the number of pencil sets of size $s$ that use only the first $i$ colors. This function satisfies the following recurrence:

$$NumSets(i, s) = \begin{cases} 
1 & \text{if } s = 0 \\
0 & \text{if } i = 0 \text{ and } s > 0 \\
\sum_{n=0}^{s} NumSets(i - 1, s - ni) & \text{if } i > 0 \text{ and } s \leq P[i] \\
\sum_{n=0}^{s} NumSets(i - 1, s - ni) & \text{otherwise} 
\end{cases}$$

We need to compute $NumSets(c, k)$. The variable $ni$ represents the number of pencils with color $i$. As a sanity check, notice that $NumSets(i, 0) = 1$ for all $i$; there is exactly one way to make an empty pencil set!

We can memoize this function into a two-dimensional array $NumSets[0..c, 0..k]$, which we can fill in standard row-major order in $O(k^2 c)$ time. (For each $i$ and $s$, the innermost loop iterates at most $k + 1$ times.) (So far this is worth 8 points.)

The main bottleneck in this algorithm is evaluating the sums inside the inner loop; we can speed up this evaluation using a technique called prefix sums. Let $NSAtMost(i, s)$ denote the number of pencil sets of size at most $s$ that can be made using only the first $i$ colors:

$$NSAtMost(i, s) := \sum_{j=0}^{s} NumSets(i, j).$$

This helper function and our earlier function $NumSets$ satisfy the following mutual recurrence:

$$NSAtMost(i, s) = \begin{cases} 
1 & \text{if } s = 0 \\
NSAtMost(i, s - 1) + NumSets(i, s) & \text{otherwise} 
\end{cases}$$

$$NumSets(i, s) = \begin{cases} 
1 & \text{if } s = 0 \\
0 & \text{if } i = 0 \text{ and } s > 0 \\
NSAtMost(i - 1, s) & \text{if } i > 0 \text{ and } s \leq P[i] \\
NSAtMost(i - 1, s) - NSAtMost(i - 1, s - P[i]) & \text{otherwise} 
\end{cases}$$

We can memoize these function into two $c \times k$ arrays. We can fill both arrays simultaneously in row-major order, increasing $i$ in the outer loop, increasing $s$ in the inner loop, and computing $NumSets[i, s]$ and then $NSAtMost[i, s]$ inside the inner loop. The resulting algorithm runs in $O(ck)$ time. (This is worth 11 points.)

But we can do better! Each set of pencils that we can build consists of 1 pencil each from $x_1$ different colors, 2 pencils each from $x_2$ different colors, 3 pencils each from $x_3$ different colors, and so on. So $k = x_1 + 2x_2 + 3x_3 + \cdots$.
from $x_3$ different colors, and so on, for some non-negative integers $x_1, x_2, \ldots, x_k$ such that

$$\sum_{i=1}^{k} i \cdot x_i = k.$$ 

The main idea behind our faster dynamic programming algorithm is to consider the possible choices of $x_1, x_2, \ldots, x_k$ separately from the actual color assignments. For example, if we are given two red pencils, four green pencils, and one blue pencil, we can build four different types of four-pencil sets:

- $(x_1, x_2, x_3, x_4) = (0, 0, 0, 1) \rightarrow GGGG$
- $(x_1, x_2, x_3, x_4) = (1, 0, 1, 0) \rightarrow GGGR$ and $GGGB$
- $(x_1, x_2, x_3, x_4) = (0, 2, 0, 0) \rightarrow GGRR$
- $(x_1, x_2, x_3, x_4) = (2, 1, 0, 0) \rightarrow GGRB$ and $RRGB$

Imagine choosing the counts $x_i$ (and counting the choices for the corresponding colors) in decreasing index order. In some intermediate stage of this decision process, we have already chosen $x_{k}, x_{k-1}, \ldots, x_{\ell}+1$ and we need to count decompositions of the form

$$\sum_{i=1}^{\ell} i \cdot x_i = s,$$

assuming we have already used $u = \sum_{j=\ell+1}^{k} x_j$ colors of pencils. The remaining subproblem can be specified by three integers, each between 0 and $k$:

- The size $s$ of the subset we have left to build
- The largest number $\ell$ of pencils of one color we can include in the subset
- The number $u$ of colors that we have already used.

Define $\text{NSets}(s, \ell, u)$ to be the number of pencil sets of size $s$ that use at most $\ell$ pencils of each color, if $u$ colors (each with at least $\ell$ pencils) are unavailable. Our top-level problem asks us to compute $\text{NSets}(k, k, 0)$.

Before we describe a recurrence for this function, we need some simple preprocessing. If any color has more than $k$ pencils, we throw the extra pencils away; for all $i$, replace $P[i]$ with $\min\{P[i], k\}$. Then we compute an array $\text{Cols}[1..k]$, where $\text{Cols}[i]$ is the number of different colors for which we have at least $i$ pencils. We can compute this array in $O(c)$ time by sorting the input array $P[1..c]$ (using counting sort) and then scanning.

The function $\text{NSets}$ satisfies the following rather intimidating recurrence:

$$\text{NSets}(s, \ell, u) = \begin{cases} 
1 & \text{if } s = 0 \\
\sum_{x_\ell=0}^{s/\ell} \binom{\text{Cols}[\ell] - u}{x_\ell} \cdot \text{NSets}(s - \ell \cdot x_\ell, \ell - 1, u + x_\ell) & \text{otherwise}
\end{cases}$$

The recurrence looks at all possible values of $x_\ell$, and for each such value, counts the number of subsets with $x_\ell$ colors contributing exactly $\ell$ pencils. For each value
of \( x_\ell \), the binomial coefficient counts number of ways to choose \( x_\ell \) colors from the \( \text{Cols}[^\ell \ldots u] \) available colors that have at least \( \ell \) pencils, and the recursive call counts our choices for the remaining \( s - \ell \cdot x_\ell \) pencils.

We can memoize this recurrence into a three-dimensional array, indexed by \( s, \ell \) and \( u \). We can fill the array using three nested for-loops, increasing \( \ell \) in the outer loop, decreasing \( u \) in the middle loop, and increasing \( s \) in the inner loop. We can evaluate each entry \( \text{NSets}[s, \ell, u] \) in \( O(1 + s/\ell) \) time; in particular, we can evaluate each binomial coefficient in \( O(1) \) time using the recurrence

\[
\binom{n}{i} = \begin{cases} 
1 & \text{if } i = 0 \\
\binom{n-1}{i-1} \frac{n-i}{i} & \text{otherwise}
\end{cases}
\]

Thus, the overall time to fill the entire memoization array is at most

\[
\sum_{s=1}^{k} \sum_{\ell=1}^{k} \sum_{u=0}^{k} O(1 + s/\ell) = O(k^3) + O(k) \left( \sum_{s=1}^{k} s \right) \left( \sum_{\ell=1}^{k} \frac{1}{\ell} \right) = O(k^3 \log k).
\]

The overall running time of our algorithm is \( O(c + k^3 \log k) \) time, which is \( o(ck) \) if \( k \) is sufficiently small compared to \( c \).\(^a\) \( \langle \text{This is worth 13 points.} \rangle \)

Finally, if \( k \) is sufficiently small—specifically, if \( k = o(c^{1/3} / \log^{1/3} c) \)—the running time of this algorithm simplifies to \( O(c) \). This is clearly optimal, because any correct algorithm must read the entire input! \( \langle \text{This is worth 15 points.} \rangle \)

\(^a\)This is slightly faster than the previous solution based on truncated polynomials if we use Lattice multiplication everywhere instead of FFTs.

**Rubric:** 10 points for a correct \( O(ck \log k) \)-time algorithm, partial credit and extra credit as indicated in the solution. Partial credit for dynamic programming solutions follows the standard dynamic programming rubric given in Homework 2. These are almost certainly not the only correct algorithms, or the fastest possible algorithms for all combinations of \( c, k, \) and \( N \).