If you understand these problems you should not have difficulties in writing up your homework 1.

1 Formulae you should know

You should remember these properties from previous calculus classes:

- \( \log_a (b \cdot c) = \log_a(b) + \log_a(c) \),
- \( \log_a (b/c) = \log_a(b) - \log_a(c) \),
- \( \log_a (b^c) = c \cdot \log_a(b) \),
- \( e^{\log_a (b)} = b^{\log_a(c)} \),
- \( 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1} \).

2 Sample Problems

2.1 Summation

Problem 1:
Let \( S_n^k \) be the sum \( 1^k + 2^k + 3^k + \cdots + n^k \). Then
1. Calculate \( S_n^1 \) and \( S_n^2 \), and
2. Show that \( S_n^k = O(n^{k+1}) \).

Proof:
1. We recall the formulas \( (x + 1)^2 = x^2 + 2x + 1 \) and \( (x + 1)^3 = x^3 + 3x^2 + 3x + 1 \). Then

\[
S_n^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 \\
= 1 + (1^2 + 1) + (2^2 + 2^2 + 1) + \cdots + ((n - 1)^2 + 2(n - 1) + 1) \\
= (1^2 + 2^2 + \cdots + (n - 1)^2) + 2(1 + 2 + \cdots + (n - 1)) + n \\
= (S_n^2 - n^2) + 2(S_n^1 - n) + n \\
= S_n^2 + 2S_n^1 - n(n + 1).
\]

Simplifying by \( S_n^2 \), one gets \( S_n^1 = \frac{n(n+1)}{2} \).
Similarly,

\[
S_n^3 = 1^3 + 2^3 + 3^3 + \cdots + n^3 \\
= 1 + (1 + 1)^3 + (2 + 1)^3 + \cdots + ((n - 1) + 1)^3 \\
= 1 + (1^3 + 3 \cdot 1^2 + 3 \cdot 1 + 1) + (2^3 + 3 \cdot 2^2 + 3 \cdot 2 + 1) + \\
\cdots + ((n - 1)^3 + 3(n - 1)^2 + 3(n - 1) + 1) \\
= (1^3 + 2^3 + \cdots + (n - 1)^3) + 3(1^2 + 2^2 + \cdots + (n - 1)^2) + \\
+ 3(1 + 2 + \cdots + (n - 1)) + n \\
= (S_n^3 - n^3) + 3(S_n^2 - n^2) + 3S_{n-1}^1 + n \\
= S_n^2 + 3S_n^3 - n^3 - 3n^2 + n + 3 \cdot \frac{2n(n-1)}{2} \\
= S_n^2 + 3S_n^3 - \frac{n(2n^2 + 3n - 3)}{2} \\
= S_n^2 + 3S_n^3 - \frac{n(n+1)(2n+1)}{6}.
\]

Simplifying by \(S_n^3\), one gets \(S_n^3 = \frac{n(n+1)(2n+1)}{6}\).

2. We recall that \((n + 1)^{k+1} = n^{k+1} + (k + 1)n^k + P_{k-1}(n)\), where \(P_{k-1}\) is a polynomial of degree \(k - 1\). Then

\[
S_n^{k+1} = 1^{k+1} + 2^{k+1} + 3^{k+1} + \cdots + n^{k+1} \\
= 1 + (1 + 1)^{k+1} + (2 + 1)^{k+1} + \cdots + ((n - 1) + 1)^{k+1} \\
= 1 + (1^{k+1} + (k + 1) \cdot 1^k + P_{k-1}(1)) + (2^{k+1} + (k + 1) \cdot 2^k + P_{k-1}(2)) + \\
\cdots + ((n - 1)^{k+1} + (k + 1) \cdot (n - 1)^k + P_{k-1}(n-1)) \\
= (1^{k+1} + 2^{k+1} + \cdots + (n - 1)^{k+1}) + (k + 1)(1^k + 2^k + \cdots + (n - 1)^k) + \\
+ (1 + P_{k-1}(1) + P_{k-1}(2) + \cdots + P_{k-1}(n - 1)) \\
= (S_n^{k+1} - n^{k+1}) + (k + 1)(S_n^k - n^k) + O(n^k) \\
= S_n^{k+1} + (k + 1)S_n^k - n^{k+1} + O(n^k)
\]

Simplifying \(S_n^k\), one gets \(S_n^k = \frac{n^{k+1}}{k+1} - O(n^k)\), that is, \(S_n^k = O(n^{k+1})\).

**Problem 2:**

Calculate the sums \(\sum_{k=0}^{n} k2^k\) and \(\sum_{k=0}^{n} k^2 2^k\).

**Proof:** Recall that \(\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}\). By differentiation, one obtains:

\[
\sum_{k=0}^{n} kx^{k-1} = \frac{(n + 1)x^n(x - 1) - x^{n+1} + 1}{(x - 1)^2} = \frac{nx^{n+1} - (n + 1)x^n + 1}{(x - 1)^2}
\]

Multiplying by \(x\), one gets:

\[
\sum_{k=0}^{n} kx^k = \frac{nx^{n+2} - (n + 1)x^{n+1} + x}{(x - 1)^2},
\]

and replacing \(x\) by \(2\), \(\sum_{k=0}^{n} k2^k = (n - 1)2^{n+1} + 2\).

Differentiating \((*)\) again, one obtains

\[
\sum_{k=0}^{n} k^2 x^{k-1} = \frac{(n(n + 2)x^{n+1} - (n + 1)^2x^n + 1)(x - 1)^2 - 2(x - 1)(nx^{n+2} - (n + 1)x^{n+1} + x)}{(x - 1)^4}
\]

Replacing \(x\) by 2, we obtain:

\[
\sum_{k=0}^{n} k^2 2^k = n(n + 2)2^{n+1} - (n + 1)^2 2^n + 1 - 2(n2^{n+2} - (n + 1)2^{n+1} + 2) = (n^2 - 2n + 3)2^n - 3,
\]
that is, \( \sum_{k=0}^{n} k^2 2^k = (n^2 - 2n + 3)2^{n+1} - 6. \)

2.2 Function Order

**Problem 3:**
Is \( 2^{n+1} = O(2^n) \)? Is \( 2^{2n} = O(2^n) \)?

**Proof:** Yes. Take \( c = 2 \) and \( n_0 = 1 \); then, since \( 2^{n+1} = 2 \cdot 2^n \) for all \( n \), we have that for each \( n \geq 1 \), \( 2^{n+1} \leq c2^n \). No. There are no \( c, n_0 \) such that for each \( n \geq n_0, 2^{2n} \leq c2^n \). To show this, assume there exist such \( c, n_0 \). Then \( 2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \), which implies \( 2^n \leq c \). But no constant is bigger than \( 2^n \). Contradiction. In fact \( 2^{2n} = \omega(2^n) \), since \( \lim_{n \to \infty} \frac{2^n}{2^n} = \lim_{n \to \infty} \frac{1}{2^n} = 0. \)

**Problem 4:**
Prove that \( \log(n!) = O(n \log n) \) and \( n \log n = O(\log(n!)) \).

**Proof:** First we prove that \( \log(n!) = O(n \log n) \). Since \( n! = 1 \cdot 2 \cdot 3 \cdots n - 1 \cdot n \leq n \cdot n \cdots n = n^n \), we have that \( \log(n!) \leq \log(n^n) = n \log n \). Then \( \log(n!) = O(n \log n) \).

Now we prove that \( n \log n = O(\log(n!)) \). Since \( n! = 1 \cdot 2 \cdot 3 \cdots n - 1 \cdot n \geq \prod_{i=1}^{n/2} n/2 = (n/2)^{n/2} \), where the \( \geq \) step follows after noticing that the last \( n/2 \) product terms of \( n! \) are all greater than \( n/2 \). Then we have that \( \log(n!) \geq \log((n/2)^{n/2}) = (n \log(n/2))/2. \) Then \( n \log n = O(\log(n!)) \).

**Problem 5:**
Group the following functions by complexity category: \( n \ln n, (\log n)^2, 5n^2 + 7n, n^{5/2}, n!, 4^n, n^n, n^n + \ln n, 5^k n, \log(n!), (\log n)!, \sqrt{n}, e^n, 8n + 12, 10^n + n^{20} \).

**Proof:** Much of the ranking is based on the following facts:

1. Exponential functions grow faster than polynomial functions \( \lim_{n \to \infty} \frac{n^b}{a^n} = 0 \), polynomial functions grow faster than logarithmic functions \( \lim_{n \to \infty} \frac{(\log n)^b}{n^c} = 0. \)

2. The base of a logarithm doesn’t matter asymptotically, but the base of an exponential and the degree of a polynomial do matter.

3. If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \) then \( g(n) \in \Theta(f(n)) \) if \( c > 0 \). If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \) then \( g(n) \in o(f(n)) \). If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \) then \( f(n) \in o(g(n)) \).

4. L’Hopitals rule: If \( f(n) \) and \( g(n) \) are both differentiable with derivatives \( f'(n) \) and \( g'(n) \), respectively, and if \( \lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty \)

then

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)} \]

5. If \( \lim_{n \to \infty} \log \left[ \frac{f(n)}{g(n)} \right] = \infty \) then \( \lim_{n \to \infty} \left[ \frac{f(n)}{g(n)} \right] = \infty \).

6. Some useful identities:
(a) \((\frac{n}{2})^{n/2} \leq n! \leq n^n\)
(b) Stirlings: \(n! \sim (n/e)^n \sqrt{2\pi n}\)
(c) \(\log_a a = \frac{\log e}{\log n}\)
(d) \(a^{\log_a n} = n^{\log_a a}\)

Then the ordering is the following:

- \(n^n\) and \((n^n + \ln n)\)
- \(n!\)
- \(10^n + n^{20}\)
- \(4^n\)
- \(e^n\)
- \((\ln n)!\)
- \(n^{5/2}\)
- \(5^{\ln n}\)
- \(5n^2 + 7n\)
- \(n \log n\) and \(\log(n!n)\)
- \(8n + 12\)
- \(n^{1/2}\)
- \(\log^2 n\)

By fact 3, \(\lim_{n \to \infty} \frac{n^n + \ln n}{n^n} = \lim_{n \to \infty} \left(1 + \frac{\ln n}{n^n}\right) = 1\)

By facts 3 and 6b, \(\lim_{n \to \infty} \frac{n^n}{n!} = \lim_{n \to \infty} \frac{n^n}{(n/e)^n \sqrt{2\pi n}} = \lim_{n \to \infty} \frac{e^n}{\sqrt{2\pi n}} = \infty\)

By facts 3 and 6a and by noticing that \(10^n + n^{20} \leq 2(10^n)\) as \(n\) goes to infinity, \(\lim_{n \to \infty} \frac{10^n + n^{20}}{n!} < \lim_{n \to \infty} \frac{2(10^n)}{(n/2)^{(n/2)}} = 0\)

By fact 3, \(\lim_{n \to \infty} \frac{10^n + n^{20}}{4^n} = \lim_{n \to \infty} \left(\frac{10^n}{4^n} + \frac{n^{20}}{4^n}\right) = \infty\)

By fact 3, \(\lim_{n \to \infty} \frac{4^n}{e^n} = \lim_{n \to \infty} \frac{4^n}{e^n} = 0\)

By facts 3, 5, and 6a, \(\lim_{n \to \infty} \frac{e^n}{(\ln n)!} \leq \lim_{n \to \infty} \frac{e^n}{(\ln n)!^2} = \lim_{n \to \infty} n \ln e - \ln n \ln \ln n = \infty\)

By facts 6a, 6d, and 3, \(\lim_{n \to \infty} \frac{(\ln n)!}{n^{5/2}} > \lim_{n \to \infty} \frac{(\ln n/2)!^2}{n^{5/2}} = \lim_{n \to \infty} \frac{n^{(\ln \ln n)/2}}{n^{1/2}n^{5/2}} = \infty\)

By facts 3 and 5, \(\lim_{n \to \infty} \frac{n^{5/2}}{5^{\ln n}} = \lim_{n \to \infty} (5/2) \ln n - \ln 5 \ln n \approx \lim_{n \to \infty} \ln (.18) \ln n = \infty\)

By facts 3 and 6c, \(\lim_{n \to \infty} \frac{5n^2 + 7n}{5^{\ln n}} \approx \lim_{n \to \infty} \left(\frac{5}{n^{0.3}} + \frac{7}{n^{1.3}}\right) = 0\)

By facts 3, \(\lim_{n \to \infty} \frac{5n^2 + 7n}{\ln n} = \lim_{n \to \infty} \frac{5n + 7}{\ln n} = \infty\)

By facts 3, 6a, and 6c, \(\lim_{n \to \infty} \frac{\ln(n!)}{n \ln n} \leq \lim_{n \to \infty} \frac{n \ln n}{(1/\ln e)n \ln n} = \ln e \lim_{n \to \infty} \frac{\ln(n!)}{n \ln n} \geq \lim_{n \to \infty} \frac{n/2 \ln n/2}{(1/\ln e)n \ln n} = \frac{\ln e}{2}\)
By facts 3 and 4, \( \lim_{n \to \infty} \frac{n \ln n}{8n + 12} = \lim_{n \to \infty} \frac{n/n + \ln n}{8} = \infty \)

By fact 3, \( \lim_{n \to \infty} \frac{8n + 12}{n^{1/2}} = \lim_{n \to \infty} (8\sqrt{n} + 12/\sqrt{n}) = \infty \)

By fact 1, \( \lim_{n \to \infty} \frac{(\lg n)^2}{n^{1/2}} = 0 \)

### 2.3 Recurrence

**Problem 6:**
Argue that the solution to the recurrence \( T(n) = T(n/3) + T(2n/3) + n \) is \( \Omega(n \log n) \) by appealing to a recursion tree.

**Proof:** The shortest path from the root to a leaf in the recursion tree is \( n \to (1/3)n \to (1/3)^2n \to \cdots \to 1 \). Since \( (1/3)^k n = 1 \) when \( k = \log_{3/2} n \), the height of the part of the tree in which every node has two children is \( \log_{3/2} n \). Since the values at each of theses levels of the tree add up to \( n \), the solution to the recurrence is at least \( n \log_{3/2} n = \Omega(n \log n) \).

**Problem 7:**
Solve the recurrence \( T(n) = T(\sqrt{n}) + 1 \).

**Proof:** First, make the substitution \( n = 2^m \) and let \( S(m) \) denote \( T(2^m) \). Replacing \( n \) with \( 2^m \), we obtain the recurrence \( T(2^m) = T(2^{m/2}) + 1 \), that is, \( S(m) = S(m/2) + 1 \). Solving this equation using trees, we get the solution \( S(m) = O(\log m) \). Replacing \( S(m) \) with \( T(2^m) \), one obtains \( T(2^m) = O(\log m) \), which means that \( T(n) = O(\log \log n) \).

**Problem 8:** (a more difficult one)
Professor Diogenes has supposedly identical VLSI chips that in principle are capable of testing each other. The professor’s test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:

<table>
<thead>
<tr>
<th>Chip A says</th>
<th>Chip B says</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>B is good</td>
<td>A is good</td>
<td>both are good, or both are bad</td>
</tr>
<tr>
<td>B is good</td>
<td>A is bad</td>
<td>at least one is bad</td>
</tr>
<tr>
<td>B is bad</td>
<td>A is good</td>
<td>at least one is bad</td>
</tr>
<tr>
<td>B is bad</td>
<td>A is bad</td>
<td>at least one is bad</td>
</tr>
</tbody>
</table>

a) Show that if more than \( n/2 \) chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

b) Consider the problem of finding a single good chip from among \( n \) chips, assuming that more than \( n/2 \) of the chips are good. Show that \( [n/2] \) pairwise tests are sufficient to reduce the problem to one of nearly half the size.

c) Show that the good chips can be identified with \( \Theta(n) \) (proportional to \( n \)) pairwise tests, assuming that more than \( n/2 \) of the chips are good. Give and solve the recurrence that describes the number of tests.

**Proof:** Question a): Let \( g \) be the number of good chips and \( n - g \) be the number of bad chips. Also, assume \( n - g \geq g \).

From this assumption we have that we can always find a set \( G \) of good chips and a set \( B \) of bad chips of equal size \( g \). Now, assume that the set \( B \) of bad chips conspire to fool the professor in the following way: for any test made by the professor, they declare themselves as ‘good’ and the chips in \( G \) as ‘bad’. Notice that the chips in \( G \) report correct answers and then exhibit a symmetric behaviour: they declare themselves as ‘good’ and the chips in \( B \) as ‘bad’. This implies that whichever is the strategy based on the kind of test considered and used by the professor, the behaviour of
the chips in \( G \) is indistinguishable from the behaviour of the chips in \( B \). This does not allow the professor to determine which chips are good.

**Question b):** Assume \( n \) is even. Then we can match chip in pairs \((c_{2i-1}, c_{2i})\), for \( i = 1, \ldots, n/2 \) and test all these pairs. Then we throw away all pairs of chips that do not say ‘good, good’ and finally we take a chip from each pair. In this way, the final set contains at most \( n/2 \) chips and we are still keeping more good chips than bad ones, since all the pairs that we have discarded contain at least a bad chip.

Now, assume \( n \) is odd. Then, we test \( \lfloor n/2 \rfloor \) pairs constructed as before; this time, however, we don’t test one chip, say, \( c_n \). Our final set will contain one chip for every pair tested that reported ‘good, good’. Moreover, according to whether the number of such chips has the form \( 4k + 3 \) or \( 4k + 1 \), we discard chip \( c_n \) or we put it into our final set. Now, why this construction works?

First of all, as in the case in which \( n \) is even, by discarding all pairs of chips that report anything but ‘good,good’, we discard pairs of chips in which at least one is bad, and thus we still keep more good chips than bad ones.

Now, assume that we have discarded the pairs of chips that report anything but ‘good,good’, and we are left with \( 4k+3 \) chips, for some integer \( k \). Then, in this case there are at least \( 2k + 2 \) good chips and thus there are at least \( k + 1 \) pairs of ‘good,good’ chips, and at most \( k \) pairs of ‘bad,bad’ chips. Thus chip \( c_n \) can also be discarded, and only a chip in each of these \( 2k + 2 \) pairs is not discarded.

Finally, assume that we have discarded the pairs of chips that report anything but ‘good,good’, and we are left with \( 4k + 1 \) chips, for some integer \( k \). Then, in this case there are at least \( 2k + 1 \) good chips.

Now, assume \( c_n \) is bad; then there are at least \( 2k + 2 \) good chips, that is, \( k + 1 \) pairs of ‘good,good’ chips; and at most \( k – 1 \) pairs of bad chips. Thus, taking a single chip from every pair, plus \( c_n \) gives us at least \( k + 1 \) good chips and at most \( (k – 1) + 1 = k \) bad chips; that is, a majority of good chips and our final set is at most of size \( \lceil k/2 \rceil \).

On the other hand, assume \( c_n \) is good. Then we have at least \( k \) pairs of ‘good,good’ chips and taking a chip from each pair plus \( c_n \) gives us a majority of good chips and a final set of size at most \( \lceil k/2 \rceil \).

**Question c):** If at least \( n/2 \) of the chips are good, using the answer to question b), we can compute a single good chip. Now, in order to find all good chips, we can just test the good chip with all the others: for each pair of this type, knowing that at least one of the two chips in the pair is good is enough to decide whether the other chip is good or not. This last stage takes \( n – 1 \) more tests. Then, in order to prove that the goof chips can be found with \( \Theta(n) \) pairwise tests, we just need to show that the number of tests to find a single good chip is \( \Theta(n) \). To show this, we can write the number of tests \( T(n) \) that we do in the answer to question b) as

\[ T(n) \leq T([n/2]) + [n/2], \]

and \( T(1) = 0 \). The solution to this recurrence is \( T(n) = O(n) \) and can be computed for instance by arguing and proving by induction that \( T(n) \leq 2^{\log n} \).

### 2.4 Sorting

**Problem 9:**

Insertion sort can be expressed as a recursive procedure as follows. In order to sort \( A[1..n] \), we recursively sort \( A[1..n-1] \) and then insert \( A[n] \) into the sorted array \( A[1..n-1] \). Write a recurrence for the running time of this recursive version of insertion sort.

**Proof:** Let \( T(n) \) be the time needed to sort an array of \( n \) elements using this recursive version of insertion sort. In the recursive step of this version, the same algorithm is run on an array of \( n-1 \) elements, and there is an additional step of at most \( n-1 \) comparison in order to insert the last element input in the sorted array of \( n-1 \) elements. Then the recurrence for the running time is

\[ T(n) = T(n-1) + O(n). \]
**Problem 10: (2-5, Skiena)**

Given two sets $S_1$ and $S_2$ (each of size $n$), and a number $x$, describe an $O(n \lg n)$ algorithm for finding whether there exists a pair of elements, one from $S_1$ and one from $S_2$, that add up to $x$.

**Proof:** (Solution 1) A $\Theta(n^2)$ algorithm would entail examining each element $y_1$ in $S_1$ and determining if there is an element $y_2$ in $S_2$ such that $y_1 + y_2 = x$.

FINDSUM($S_1$, $S_2$, $x$)
1. for $y_i \in S_1$
2. for $y_j \in S_2$
3. if $y_i + y_j = x$ then return $(y_i, y_j)$

**Proof:** (Solution 2) A more efficient solution takes advantage of sorting as a subroutine. First, we sort the numbers in $S_2$. Next, we loop through each element $y_i$ in $S_1$ and then do a binary search on the sorted $S_2$ for $x - y_i$.

FINDSUM($S_1$, $S_2$, $x$)
1. MERGESORT($S_2$)
2. for $y_i \in S_1$
3. $y_j = \text{BINARYSEARCH}(S_2, x - y_i)$
4. if ($y_j \neq \text{notfound}$) then return $(y_i, y_j)$

The MERGESORT takes time $\Theta(n \lg n)$. For each element in $S_1$ the BINARYSEARCH on $S_2$ will take $O(\lg n)$. Since this binary search is done $n$ times the for loop requires $O(n \lg n)$ worst-case time. The worst-case time for the entire algorithm is also $O(n \lg n)$.

**Problem 11: (Convex Hull; page 35, Skiena)**

Given $n$ points in two dimensions, find the convex polygon of smallest area that contains them all.

**Proof:** Let $(x_1, y_1), \ldots, (x_n, y_n)$ be $n$ points in two dimensions. Very often in geometrical problems, it is a good idea to sort the points by one or both coordinates. In our problem, we sort the points $(x_i, y_i)$ such that either $x_i < x_{i+1}$ or $x_i = x_{i+1}$ and $y_i < y_{i+1}$. This sorting assures us that the points are visited from the left to the right and from the bottom to the top when the counter increases from 1 to $n$.

The strategy to solve this problem is to iteratively determine the convex hull given by the points 1..$i$, where $i$ increases from 2 to $n$. Notice that the $i$th point is always a vertex in the convex hull given by the points 1..$i$. We have to understand how the convex hull is modified when a new point is added. First, let us consider two arrays $\text{high}[1..n]$ and $\text{low}[1..n]$ where $\text{high}[i]$ and $\text{low}[i]$ are going to be the high and the low neighbors of $i$ in the convex hull given by 1..$i$, respectively. The key observation is that if there is a point $k \in 1..i - 1$ such that $k$ is below the line determined by the points $i$ and $\text{high}[k]$ then $k$ cannot be a vertex in the convex hull of 1..$i$; similarly, $k$ cannot be a vertex in the convex hull of 1..$i$ if $k$ is above the line determined by $i$ and $\text{low}[k]$. Summarizing all these up, we get the following algorithm:

**Step 1:**
Sort the pairs $(x_i, y_i)$ using a standard comparison algorithm (for example MERGE_SORT) where the order relation is given by: $(x_i, y_i) \leq (x_j, y_j)$ iff $x_i < y_i$ or $x_i = x_j$ and $y_i \leq y_j$.

**Step 2:**

\[
\text{high}[1..n], \text{low}[1..n]; \text{high}[1] = 0, \text{low}[1] = 0
\]

for $i = 2$ to $n$

\[
k = i - 1
\]

while $\text{BELOW}(k, i, \text{high}[k]) = \text{true}$

\[
k = \text{high}[k]
\]

$k = i - 1$

while $\text{ABOVE}(k, i, \text{low}[k]) = \text{true}$

\[
k = \text{low}[k]
\]

\[
\text{low}[i] = k
\]
where \( \text{BELOW}(k, i, j) \) is true iff \( k \) is below the line determined by the points \( i \) and \( j \), and \( \text{ABOVE}(k, i, j) \) is true iff \( k \) is above the line determined by the points \( i \) and \( j \).

So we started with \( k = i - 1 \) and moved along the high neighbors until the first point \( k \) which is a vertex in the convex hull determined by \( 1, i \) was found; this \( k \) is the high neighbor of \( i \). Similarly (but dually) for the low neighbor of \( i \).

Once we have the high and low neighbors of \( n \), we recurrently can obtain and print the convex hull by first following the high path of \( n \) and then following the low path of \( n \).

Now, let us analyze the complexity of this algorithm. Step 1 obviously takes \( O(n \log n) \). The analyze of Step 2 is more complicated because we have to analyze it globally, counting how many while iterations will be in the whole execution. Notice that once a point \( k \) is processed in the first while loop (i.e., \( \text{BELOW}(k, i, \text{high}[k]) \) is true), it will not be processed again for a larger \( i \). This is because \( \text{high}[k] \) will become the high neighbor of \( i \), so \( k \) will be hidden for all larger \( i \). Consequently, \( k \) will never be visited again within the first while loop. Similarly for the second while loop. Therefore, the while loops are executed \( O(n) \) times.

**Problem 12: (Onion; 2-17, Skiena)**

The onion of a set of \( n \) points is the series of convex polygons that result from finding the convex hull, striping it from the point set, and repeating until no more points are left. Give an \( O(n^2) \) algorithm for determining the onion of a point set.

**Proof:** Make sure that you understand the convex hull problem before you go further.

Following exactly the algorithm described in the text of the problem, it seems that \( O(n) \) iterations of convex hull are needed. Since the complexity of convex hull is \( O(n \log n) \), the complexity of this problem is going to be \( O(n^2 \log n) \). Therefore we have to find something else.

The key point we have to observe is that the points do not have to be sorted again when a new convex hull is generated, because they are already sorted from previous convex hulls. So the points can be sorted only once, at the beginning, and the natural algorithm follows:

**Step 1:**
Sort the points as in the convex hull problem,

**Step 2:**
while there still exist points in the set
(a) generate the current convex hull
(b) output the convex hull
(c) strip the convex hull from the point set

The algorithm terminates because each iteration strips some points from the point set. In order for (a) to work properly, we need to make sure that the remained points are still sorted under the ordering described in the convex hull problem after the previous convex hull is removed. Perhaps the easiest way to assure that is to have an array of \( n \) flags (boolean values) associated with the points in the set, and in (c) only to change the values of those flags corresponding to the convex hull. Then all \( n \) points are scanned in (a) each time when a convex hull is generated, but only those with the proper flag are taken into consideration.

As in the convex hull problem, Step 1 has the complexity \( O(n \log n) \). Because the generation of a new convex hull can be done in linear time once the points are sorted (see the convex hull problem), (a) takes \( O(n) \); obviously, (b) and (c) need each \( O(n) \). So the body of the while loop has a linear complexity. Since each iteration there are some points striped from the point set, the loop has less than \( n \) iterations. Therefore, Step 2 has the complexity \( O(n^2) \) which is the complexity of the whole algorithm.

**Problem 13: (k-way Merging)**

Give an \( O(n \log_k k) \) algorithm to merge \( k \) sorted lists into one sorted list, where \( n \) is the total number of elements in all the input lists. Hint: Use a heap for \( k \)-way merging. Analyze the time complexity of your algorithm.

**Proof:** An \( O(n \log k) \) algorithm for the problem (different from the one treated in the discussion section) is to pair the lists and merge each pair, and recursively merge the \( k/2 \) resulting lists. The time is given by the recurrence
Problem 14: (Weighted Median)

For \( n \) distinct elements \( x_1, x_2, \ldots, x_n \) with positive weights \( w_1, w_2, \ldots, w_n \) such that \( \sum_{i=1}^{n} w_i = 1 \), the weighted median is the element \( x_k \) satisfying

\[
\sum_{x_i < x_k} w_i \leq \frac{1}{2} \quad \text{and} \quad \sum_{x_i > x_k} w_i \leq \frac{1}{2}
\]

1. Argue that the median of \( x_1, x_2, \ldots, x_n \) is the weighted median of the \( x_i \) with weights \( w_i = 1/n \) for \( i = 1, 2, \ldots, n \).

2. Show how to compute the weighted median of \( n \) elements in \( O(n \log n) \) worst-case time using sorting.

3. Show how to compute the weighted median in \( \Theta(n) \) worst-case time using a linear-time median algorithm.

Proof:

1.

2.

3. We are given an unsorted list \( x_1, \ldots, x_n \) and positive weights \( w_1, \ldots, w_n \) with \( \sum w_i = 1 \), and wish to find an \( x_i \) so that \( \sum_{j|x_j < x_i} w_j \leq \frac{1}{2} \) and \( \sum_{k|x_k > x_i} w_k \leq \frac{1}{2} \).

For a weighted unsorted list \( (x_1, w_1), \ldots, (x_n, w_n) \), let \( W = \sum_{1 \leq i \leq n} w_i \). We will give an algorithm which solves a more general problem: given a weighted array, and a number \( 0 \leq \alpha \leq W \) find an \( x_i \) so that \( \sum_{j|x_j < x_i} w_j \leq \alpha \) and \( \sum_{k|x_k > x_i} w_k \leq W - \alpha \).

We know there is an algorithm \( \text{Select} \) which can find the \( j \)’th largest element of a list of length \( n \) in \( O(n) \) time. To find the weighted median (or in general, the \( \alpha \)-sample), we first find the non-weighted median \( x_i \) using \( \text{Select} \). Then we divide the list into the subset \( S \) of elements smaller than \( x_i \) and the subset \( B \) of elements bigger than or equal to \( x_i \). Since \( x_i \) is the unweighted median, both \( S \) and \( B \) have at most \( n/2 \) elements. (If our list contains only a single element, the recursion bottoms out, and we return that element, which is a correct solution since both sums in the two inequalities are over empty sets, hence are \( 0 \), and \( 0 \leq \alpha \leq W \).)

We then calculate \( W_S = \sum_{x_j \in S} w_j \). If \( W_S > \alpha \), we recursively call our algorithm on \( S, \alpha \). We output the result of this recursive call, \( x_k \), since \( x_k \) satisfies: \( \alpha \geq \sum_{j|x_j < x_k} w_j = \sum_{j|x_j, j \in S} w_j \), since only elements of \( S \) can be smaller than an element of \( S \), and \( \sum_{j|x_j > x_k} w_j = \sum_{j|x_j \in S} w_j + \sum_{j|x_j \in S, x_j > x_k} w_j \leq (W - W_S) + (W_S - \alpha) = W - \alpha \) since all elements not in \( S \) are larger than \( x_k \). Otherwise, recursively call the algorithm on \( B, \alpha - W_S \). The proof of correctness in this case is similar to the last. Thus, the above algorithm always finds a valid solution.

Since the time used by the sub-routine \( \text{Select} \) is \( O(n) \) as is that to divide the list into \( B \) and \( S \) and to compute \( W_S \), there is a constant \( c \) so that the time taken by the algorithm on an \( n \) element input, \( T(n) \), is at most \( cn + \text{the time taken by a recursive call on an input of } 1/2 \text{ the size. I.e., } T(n) \leq cn + T(n/2) \). Thus \( T(n) \leq cn + cn/2 + cn/4 + \ldots \leq 2cn \), so the algorithm takes \( O(n) \) time.