(a) When the input size is doubled, the algorithms get slower by

(i) a factor of 4.

(ii) a factor of 8.

(iii) a factor of 4.

(iv) a factor of 2, plus an additive $2n$.

(v) the square of the previous running time.

(b) When the input size is increased by an additive one, the algorithms get slower by

(i) an additive $2n + 1$.

(ii) an additive $3n^2 + 3n + 1$.

(iii) an additive $200n + 100$.

(iv) an additive $\log(n+1) + n[\log(n+1) - \log n]$.

(v) a factor of 2.

[1] ex561.359.766

(i) $6,000,000$.

(ii) 33015.

(iii) $600,000$.

(iv) About 9×10^{11}.

(v) 45.

(vi) 5.

[1] ex695.516.801

We know from the text that polynomials (i.e. a sum of terms where n is raised to fixed powers, even if they are not integers) grow slower than exponentials. Thus, we will consider f_1, f_2, f_3, f_6 as a group, and then put f_4 and f_5 after them.

For polynomials f_i and f_j, we know that f_i and f_j can be ordered by comparing the highest exponent on any term in f_i to the highest exponent on any term in f_j. Thus, we can put f_2 before f_3 before f_1. Now, where to insert f_6? It grows faster than n^2, and from the text we know that logarithms grow slower than polynomials, so f_6 grows slower than n^c for any $c > 2$. Thus we can insert f_6 in this order between f_3 and f_1.

Finally come f_4 and f_5. We know that exponentials can be ordered by their bases, so we put f_4 before f_5.

[1] ex831.202.488

We order the functions as follows.

- g_1 comes before g_5. This is like the solved exercise in which we saw $2^{\sqrt{\log n}}$. If we take logarithms, we are comparing $\sqrt{\log n}$ to $\log n + \log(\log n) \geq \log n$; changing variables via $z = \log n$, this is $\sqrt{z} = z^{1/2}$ versus $z + \log z \geq z$.

- g_5 comes before g_3, since $(\log n)^3$ grows faster than $\log n$. (They're both polynomials in $\log n$, but $(\log n)^3$ has the larger degree.)

- g_3 comes before g_4: Dividing both by n, we are comparing $(\log n)^3$ with $n^{1/3}$, or (taking cube roots), $\log n$ with $n^{1/9}$. Now we use the fact that logarithms grow slower than exponentials.

- g_4 comes before g_2, since polynomials grow slower than exponentials.

- g_2 comes before g_7: Taking logarithms, we are comparing n to n^2, and n^2 is the polynomial of larger degree.

- g_7 comes before g_6: Taking logarithms, we are comparing n^2 to 2^n, and polynomials grow slower than exponentials.

[1] ex413.866.86

(i) This is false in general, since it could be that $g(n) = 1$ for all n, $f(n) = 2$ for all n, and then $\log_2 g(n) = 0$, whence we cannot write $\log_2 f(n) \leq c \log_2 g(n)$.

On th other hand, if we simply require $g(n) \geq 2$ for all n beyond some n_1, then the statement holds. Since $f(n) \leq cg(n)$ for all $n \geq n_0$, we have $\log_2 f(n) \leq \log_2 g(n) + \log_2 c \leq (\log_2 c)(\log_2 g(n))$ once $n \geq \max(n_0, n_1)$.

(ii) This is false: take $f(n) = 2n$ and $g(n) = n$. Then $2^{f(n)} = 4^n$, while $2^{g(n)} = 2^n$.

(iii) This is true. Since $f(n) \leq cg(n)$ for all $n \geq n_0$, we have $(f(n))^2 \leq c^2(g(n))^2$ for all $n \geq n_0$.

[1] ex66.350.972

(a) We prove this for $f(n) = n^3$. The outer loop of the given algorithm runs for exactly n iterations, and the inner loop of the algorithm runs for at most n iterations every time it is executed. Therefore, the line of code that adds up array entries $A[i]$ through $A[j]$ (for various i's and j's) is executed at most n^2 times. Adding up array entries $A[i]$ through $A[j]$ takes $O(j - i + 1)$ operations, which is always at most $O(n)$. Storing the result in $B[i, j]$ requires only constant time. Therefore, the running time of the entire algorithm is at most $n^2 \cdot O(n)$, and so the algorithm runs in time $O(n^3)$.

(b) Consider the times during the execution of the algorithm when $i \leq n/4$ and $j \geq 3n/4$. In these cases, $j - i + 1 \geq 3n/4 - n/4 + 1 > n/2$. Therefore, adding up the array entries $A[i]$ through $A[j]$ would require at least $n/2$ operations, since there are more than $n/2$ terms to add up. How many times during the execution of the given algorithm do we encounter such cases? There are $(n/4)^2$ pairs (i, j) with $i \leq n/4$ and $j \geq 3n/4$. The given algorithm enumerates over all of them, and as shown above, it must perform at least $n/2$ operations for each such pair. Therefore, the algorithm must perform at least $n/2 \cdot (n/4)^2 = n^3/32$ operations. This is $\Omega(n^3)$, as desired.

(c) Consider the following algorithm.

```
For i = 1, 2, ... n
  Set B[i, i+1] to A[i] + A[i+1]
For k = 2, 3, ..., n - 1
  For i - 1, 2, ..., n - k
    Set j = i + k
    Set B[i, j] to be B[i, j-1] + A[j]
```

This algorithm works since the values $B[i, j-1]$ were already computed in the previous iteration of the outer for loop, when k was $j - 1 - i$, since $j - 1 - i < j - i$. It first computes $B[i, i+1]$ for all i by summing $A[i]$ with $A[i+1]$. This requires $O(n)$ operations. For each k, it then computes all $B[i, j]$ for $j - i = k$ by setting $B[i, j] = B[i, j-1] + A[j]$. For each k, this algorithm performs $O(n)$ operations since there are at most n $B[i, j]$'s such that $j - i = k$. There are less than n values of k to iterate over, so this algorithm has running time $O(n^2)$.

---

[1] ex474.221.961

Suppose that to obtain n words, we need L lines (most of which will get repeated many times, as described above). We write the script as follows

```
line 1 = <text of line 1 here>
line 2 = <text of line 2 here>
...
line L = <text of line L here>
For i = 1, 2, ..., L
  For j = 1, 2, ..., i
    Sing lines j through 1
  Endfor
Endfor
```

Now, the nested `For` loops have length bounded by a constant c_1, so the real space in the script is consumed by the text of the lines. Each of these lines in the script has length at most c_2 (where c_2 is the maximum line length c plus the space to write the variable assignment). So in total, the space required by the script is $S = c_1 + c_2 L$.

Recall that n denotes the number of words this produces when sung. n is at least $1 + 2 + \cdots + L = \frac{1}{2}L(L-1)$; hence, $\frac{1}{2}(L-1)^2 \leq n$, and so $L \leq 1 + \sqrt{2n}$. Plugging this into our bound on the length of the script, we have $f(n) = S \leq c_1 + c_2\sqrt{2n} = O(\sqrt{n})$.

[1] ex434.486.949

(a) Suppose for simplicity that n is a perfect square. We drop the first jar from heights that are multiples of $\sqrt{n}$ (i.e. from $\sqrt{n}, 2\sqrt{n}, 3\sqrt{n}, \ldots$) until it breaks.

If we drop it from the top rung and it survives, then we're also done. Otherwise, suppose it breaks from height $j\sqrt{n}$. Then we know the highest safe rung is between $(j-1)\sqrt{n}$ and $j\sqrt{n}$, so we drop the second jar from rung $1+(j-1)\sqrt{n}$ on upward, going up by one each time.

In this way, we drop each of the two jars at most $\sqrt{n}$ times, for a total of at most $2\sqrt{n}$. If n is not a perfect square, then we drop the first jar from heights that are multiples of $\lfloor\sqrt{n}\rfloor$, and then apply the above rule for the second jar. In this way, we drop the first jar at most $2\sqrt{n}$ times (quite an overestimate if n is reasonably large) and the second jar at most $\sqrt{n}$ times, still obtaining a bound of $O(\sqrt{n})$.

(b) We claim by induction that $f_k(n) \leq 2kn^{1/k}$. We begin by dropping the first jar from heights that are multiples of $\lfloor n^{(k-1)/k}\rfloor$. In this way, we drop the first jar at most $2n/n^{(k-1)/k} = 2n^{1/k}$ times, and thus narrow the set of possible rungs down to an interval of length at most $n^{(k-1)/k}$.

We then apply the strategy for $k-1$ jars recursively. By induction it uses at most $2(k-1)(n^{(k-1)/k})^{1/(k-1)} = 2(k-1)n^{1/k}$ drops. Adding in the $\leq 2n^{1/k}$ drops made using the first jar, we get a bound of $2kn^{1/k}$, completing the induction step.

[1] ex291.532.145