2.4: Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$g_1(n) = 2^{\sqrt{\log_2 n}}$
$g_2(n) = 2^n$
$g_4(n) = n^{4/3}$
$g_3(n) = n(\log_2 n)^3$
$g_5(n) = n^{\log_2 n}$
$g_6(n) = 2^{2^n}$
$g_7(n) = 2^{n^2}$

Answer:

By 2.9 we have $g_4(n) = O(g_2(n))$.

$\log_2 g_4(n) = \log_2(n^{(4/3)}) = (4/3)\log_2 n$
$\log_2 g_1(n) = \log_2(2^{\sqrt{\log_2 n}}) = \sqrt{\log_2 n} * \log_2 2 = (\log_2 n)^{(1/2)}$
Let $z = \log_2 n$.
So
$\log_2 g_4(n) = (4/3)z$
$\log_2 g_1(n) = z^{(1/2)}$
$z^{(1/2)} \leqslant (4/3)z$ for $z \geqslant 0$, or $n \geqslant 2^0 = 1$.
For $n \geqslant 1$ we have $\log_2 g_1(n) \leqslant \log_2 g_4(n)$ and hence $g_1(n) \leqslant g_4(n)$ and so $g_1(n) = O(g_4(n))$.

$g_2(n)$, $g_6(n)$, and $g_7(n)$ are exponential.
For $n \geqslant 0$ we have $2^n \leqslant 2^{n^2}$ and hence $g_2(n) \leqslant g_7(n)$, so $g_2(n) = O(g_7(n))$.
By 2.9 we have $g_7(n) = O(g_6(n))$ because $g_7(n)$ has a polynomial exponent and $g_6(n)$ has an exponential exponent.

We take the logarithm of the functions in order to compare $g_1(n)$, $g_4(n)$, $g_3(n)$, and $g_5(n)$.
$\log_2 g_1(n) = (\log_2 n)^{(1/2)}$ by above
$\log_2 g_4(n) = (4/3)\log_2 n$ by above
$\log_2 g_3(n) = \log_2(n(\log_2 n)^3) = \log_2 n + 3 * \log_2 \log_2 n$
$\log_2 g_5(n) = \log_2(n^{\log_2 n}) = (\log_2 n)^2$
Let $z = \log_2 n$
$\log_2 g_1(n) = z^{(1/2}$ $\log_2 g_4(n) = (4/3)z = z + (1/3)z$
$\log_2 g_3(n) = z + 3 * \log_2 z$
$\log_2 g_5(n) = z^2$

We have $z^{(1/2)} \leqslant z + 3 * \log_2 z$ for $z \geqslant 1$, or $n \geqslant 2$. For $n \geqslant 2$ we have $\log_2 g_1(n) \leqslant \log_2 g_3(n)$ and hence $g_1(n) \leqslant g_3(n)$ so $g_1(n) = O(g_3(n))$.

We have $z + 3 * \log_2 z \leqslant z + (1/3)z$ for $z \geqslant 1$, or $n \geqslant 2$. For $n \geqslant 2$ we have $\log_2 g_3(n) \leqslant \log_2 g_4(n)$ and hence $g_3(n) \leqslant g_4(n)$ so $g_3(n) = O(g_4(n))$.

We have $z + (1/3)z \leqslant z^2$ for $z \geqslant 0$, or $n \geqslant 1$. For $n \geqslant 1$ we have $\log_2 g_4(n) \leqslant \log_2 g_5(n)$ and hence $g_4(n) \leqslant g_5(n)$ so $g_4(n) = O(g_5(n))$.

Next, we take the logarithm of $g_2(n)$.
$\log_2 g_2(n) = \log_2(2^n) = n * \log_2 2 = n$
We have $(\log_2 n)^2 \leqslant n$ for $n \geqslant 1$ and hence $\log_2 g_5(n) \leqslant \log_2 g_2(n)$ so $g_5(n) \leqslant g_2(n)$ so $g_5(n) = O(g_2(n))$.

Therefore, the final answer is:
$g_1(n) \leqslant g_3(n) \leqslant g_4(n) \leqslant g_5(n) \leqslant g_2(n) \leqslant g_7(n) \leqslant g_6(n)$

2.6:

Answer:

(a)

We will prove this for $f(n) = n^3$. For the given algorithm, the outer loop runs exactly n times and the inner loop runs at most n times. Therefore, the lines within the inner loop are executed at most n^2 times. The line "Add up array entries $A[i]$ through $A[j]$" takes at most n operations. The line "Store the result in $B[i, j]$" takes constant time. Therefore, the given algorithm runs in $O(n^2 * n) = O(n^3)$ time.

(b)

Let $T(n)$ be the total number of operations for the given algorithm. Goal: show $T(n) \geqslant \epsilon * f(n)$ where $f(n) = n^3$.
Consider the iterations of the algorithm when $i \leqslant \frac{1}{3}n$ and $j \geqslant \frac{2}{3}n$. For these iterations, adding up the array entries $A[i]$ through $A[j]$ requires at least $\frac{1}{3}n$ operations, since there are $j - i + 1 = \frac{2}{3}n - \frac{1}{3}n + 1 = \frac{1}{3}n + 1$ terms to add.

For $i \leqslant \frac{1}{3}n$ and $j \geqslant \frac{2}{3}n$, there are $(\frac{1}{3}n)(\frac{1}{3}n) = \frac{1}{9}n^2$ possible pairs (i, j). For each of these pairs (i, j), at least $\frac{1}{3}n$ operations are required to add up the array entries $A[i]$ through $A[j]$, as shown above.

Therefore, the algorithm must perform at least $(\frac{1}{3}n)(\frac{1}{9}n^2) = \frac{1}{27}n^3$ total operations and $T(n) \geqslant \frac{1}{27}n^3$ as required.

$\Rightarrow T(n) = \Omega(n^3)$
$\Rightarrow T(n) = \Theta(n^3)$ by part (a).

(c)

Here's an improved algorithm to solve this problem:

```
sum = 0
For i = 1, 2, ..., n
    sum = A[i]
    For j = i + 1, i + 2, ..., n
        sum = sum + A[j]
        Store the value of the variable sum in B[i, j]
    Endfor
Endfor
```

By using a variable *sum* in the inner for loop, we can reduce the process "Add up array entries $A[i]$ through $A[j]$" from linear running time to constant running time. For each iteration of the outer for loop, we initialize *sum* to $A[i]$. Then, for each iteration of the inner for loop, store $A[i] + A[i + 1] + ... + A[j]$ in *sum*. For the next iteration of the inner for loop, *sum* contains $A[i] + A[i + 1] + ... + A[j - 1]$. Therefore adding $A[j]$ to *sum* produces the desired sum, which is a constant time operation.

The outer loop still runs n times and the inner loop still runs at most n times. Because the lines within the inner loop now all run in constant time, the running time of the new algorithm is $O(n * n) = O(n^2)$ and $\lim_{n\to\infty} \frac{n^2}{n^3} = 0$, as desired.

2.7:

Answer:

As noted in the question, suppose that the song runs for n words total. Let k be the number of unique lines required to convey the song, where each verse consists of the previous verse, with one extra line added on.
We encode the song using the following script:

```
line_1 = "text of line 1"
line_2 = "text of line 2"
...
line_k = "text of line k"
For i = 1, 2, ..., k
    For j = i, i - 1, ..., 1
        Sing line j
    Endfor
Endfor
```

The output of the script will be the following:

```
Output:

(Verse 1) line_1
(Verse 2) line_2 line_1
(Verse 3) line_3 line_2 line_1
...
(Verse k) line_k ... line_2 line_1
```

If each line has at least one word, then we have:

$$\begin{aligned} n &\geqslant 1 + 2 + ... + k \\ n &\geqslant \frac{k(k+1)}{2} \\ n &\geqslant \frac{k^2}{2} \\ 2n &\geqslant k^2 \\ k &\leqslant \sqrt{2n} \end{aligned}$$

Next, we prove a bound on the length of the script in words, given by the function $f(n)$. Let the length of the two For loops be bounded by a constant f. Let the length of each line of text be bounded by a constant l, where l is c (the upper bound on the line length) plus the length of the code for the variable assignment. So $f(n) = f + l * k$. By substitution we have $f(n) \leqslant f + l * \sqrt{2n}$.
$\Rightarrow f(n) = O(\sqrt{n})$

2.8:

Answer:

(a)

In order for $f(n)$ to grow slower than linearly, $f(n)$ could be $O(\log n)$ or $O(\sqrt{n})$. With a budget of $k = 2$ jars we can't do $O(\log n)$, but we can do $O(\sqrt{n})$.
The strategy can be described as follows. Assume n is a perfect square. Drop the first jar from rungs $i * \sqrt{n}$ for $i = 1, 2, ..., \sqrt{n}$ until it breaks. Suppose it breaks at rung $j * \sqrt{n}$. Then we know the highest safest rung is between $(j-1) * \sqrt{n} + 1$ and $j * \sqrt{n} - 1$, inclusive. So we drop the second jar at each rung starting at rung $(j-1) * \sqrt{n} + 1$. Following this strategy, we incur at most $\sqrt{n} + \sqrt{n} = 2 * \sqrt{n}$ drops and $f(n) \leqslant 2 * \sqrt{n}$.
$\Rightarrow f(n) = O(\sqrt{n})$ and $\lim_{n\to\infty} \frac{\sqrt{n}}{n} = 0$, as desired.
If n is not a perfect square we can use the same strategy, but using multiples of $\lfloor\sqrt{n}\rfloor$ and still get an algorithm that runs in $O(\sqrt{n})$ time.

(b)

The recursive strategy for a budget of $k > 2$ jars can be described as follows. We can extend the strategy from part (a).
Instead of using multiples of $\lfloor\sqrt{n}\rfloor$, we generalize to using multiples of $\left\lfloor n^{\frac{(k-1)}{k}} \right\rfloor$. Notice that for the first jar we will have at most $\frac{2n}{n^{\frac{k-1}{k}}} = 2n^{(1/k)}$ drops. Similar to part (a), if the jar breaks at rung $j * n^{\frac{k-1}{k}}$ we know that the highest safest rung is between $(j-1) * n^{\frac{k-1}{k}} + 1$ and $j * n^{\frac{k-1}{k}} - 1$, inclusive.
We want to prove by induction that $f_k(n) \leqslant 2kn^{(1/k)}$.
Basis Step: k = 2. We proved in part (a) that $f_2(n) \leqslant 2 * n^{(1/2)} \leqslant 4 * n^{(1/2)}$. $\Rightarrow$ basis step true.
Inductive Step:
Inductive Hypothesis: Let $f_{k-1}(n) \leqslant 2(k-1)n^{\frac{1}{k-1}}$.
We know we will have at most $2n^{\frac{1}{k}}$ drops for the first jar by above. For $k-1$ jars it takes $f_{k-1}(n^{\frac{k-1}{k}})$ because there are $n^{\frac{k-1}{k}}$ rungs between $(j-1) * n^{\frac{k-1}{k}} + 1$ and $j * n^{\frac{k-1}{k}} - 1$. By the inductive hypothesis:
$f_{k-1}(n^{\frac{k-1}{k}}) \leqslant 2(k-1)(n^{\frac{k-1}{k}})^{\frac{1}{k-1}} = 2(k-1)n^{(1/k)}$
Therefore, the $f_k(n) \leqslant 2(k-1)n^{(1/k)} + 2n^{\frac{1}{k}} = 2kn^{(1/k)}$, as desired.

The result follows by the principle of mathematical induction and $\lim_{n\to\infty} \frac{f_k(n)}{f_{k-1}(n)} = 0$ for each k as required.