CSCI 5451 Fall 2015

Week 3 Notes

Professor Ellen Gethner

August 30, 2015

Asymptotics

- Asymptotics is about the growth of functions;

Asymptotics

- Asymptotics is about the growth of functions;
- we'll use this as a tool to measure the run time (or possibly memory consumption) with respect to the input.

Asymptotics

- Asymptotics is about the growth of functions;
- we'll use this as a tool to measure the run time (or possibly memory consumption) with respect to the input.
- What follows is a very brief review of a subset of Chapter 3 in our textbook.

Review of Chapter 3

- Big "Oh" is a set of functions that provides an asymptotic upper bound.

Review of Chapter 3

- Big "Oh" is a set of functions that provides an asymptotic upper bound.

- In particular, for a given function $g(n)$,

  $O(g(n)) =$

  $\{f(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \; \forall n \geq n_0\}$.

Review of Chapter 3

- Big "Oh" is a set of functions that provides an asymptotic upper bound.

- In particular, for a given function $g(n)$,

  $O(g(n)) =$

  $\{f(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}$.

- We write $f(n) = O(g(n))$.

Review of Chapter 3

- Big "Oh" is a set of functions that provides an asymptotic upper bound.
- In particular, for a given function $g(n)$,

  $O(g(n)) =$

  $\{f(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}.$
- We write $f(n) = O(g(n))$.
- Note that $g(n)$ may not provide the best possible upper bound for $f(n)$.

Big "Oh" examples

- $n = O(n^2)$ because

Big "Oh" examples

- $n = O(n^2)$ because
- $n \leq n^2 \; \forall n \geq 1$.

Big "Oh" examples

- $n = O(n^2)$ because
- $n \leq n^2 \ \forall n \geq 1$.
- I used $c = 1$ and $n_0 = 1$.

Big "Oh" examples

- Similarly, $n^2, 17n^2, an^2 + bn + C = O(n^2)$ $(a, b \in \mathbb{R})$.

Big "Oh" examples

- Similarly, $n^2, 17n^2, an^2 + bn + C = O(n^2)$ $(a, b \in \mathbb{R})$.

- But $n^3 \neq O(n^2)$.

Big "Oh" examples

- Similarly, $n^2, 17n^2, an^2 + bn + C = O(n^2)$ $(a, b \in \mathbb{R})$.

- But $n^3 \neq O(n^2)$.

- To see why, suppose BWOC (by way of contradiction) that

Big "Oh" examples

- Similarly, $n^2, 17n^2, an^2 + bn + C = O(n^2)$ $(a, b \in \mathbb{R})$.
- But $n^3 \neq O(n^2)$.
- To see why, suppose BWOC (by way of contradiction) that
- $\forall n \geq n_0$ and some constant $c > 0$ we have $n^3 - cn^2 \leq 0$.

Big "Oh" examples

- Similarly, $n^2, 17n^2, an^2 + bn + C = O(n^2)$ $(a, b \in \mathbb{R})$.
- But $n^3 \neq O(n^2)$.
- To see why, suppose BWOC (by way of contradiction) that
- $\forall n \geq n_0$ and some constant $c > 0$ we have $n^3 - cn^2 \leq 0$.
- Then $n^3 - cn^2 \leq 0 \Rightarrow n^2(n - c) \leq 0$.

Big "Oh" examples

- Similarly, $n^2, 17n^2, an^2 + bn + C = O(n^2)$ $(a, b \in \mathbb{R})$.
- But $n^3 \neq O(n^2)$.
- To see why, suppose BWOC (by way of contradiction) that
- $\forall n \geq n_0$ and some constant $c > 0$ we have $n^3 - cn^2 \leq 0$.
- Then $n^3 - cn^2 \leq 0 \Rightarrow n^2(n - c) \leq 0$.
- But for $n > c$ we have $n^2(n - c) > 0$, which is a contradiction.

Big "Oh" examples

- Similarly, $n^2, 17n^2, an^2 + bn + C = O(n^2)$ $(a, b \in \mathbb{R})$.
- But $n^3 \neq O(n^2)$.
- To see why, suppose BWOC (by way of contradiction) that
- $\forall n \geq n_0$ and some constant $c > 0$ we have $n^3 - cn^2 \leq 0$.
- Then $n^3 - cn^2 \leq 0 \Rightarrow n^2(n - c) \leq 0$.
- But for $n > c$ we have $n^2(n - c) > 0$, which is a contradiction.
- Thus $n^3 \neq O(n^2)$.

Run time

- Generally speaking, the tools we use to find good asymptotic bounds are algebra, calculus, and common sense...

Run time

- Generally speaking, the tools we use to find good asymptotic bounds are algebra, calculus, and common sense...
- For example, recall that *fib*(n) had run time $f(n)$ with $2^{\frac{n}{2}} \leq f(n) \leq 2^{n-1}$.

Run time

- Generally speaking, the tools we use to find good asymptotic bounds are algebra, calculus, and common sense...

- For example, recall that *fib*(n) had run time $f(n)$ with $2^{\frac{n}{2}} \leq f(n) \leq 2^{n-1}$.

- Moreover, since $2^{\frac{n}{2}} = \sqrt{2}^n$ and $2^{n-1} = \frac{1}{2}2^n$,

Run time

- Generally speaking, the tools we use to find good asymptotic bounds are algebra, calculus, and common sense...
- For example, recall that $fib(n)$ had run time $f(n)$ with $2^{\frac{n}{2}} \leq f(n) \leq 2^{n-1}$.
- Moreover, since $2^{\frac{n}{2}} = \sqrt{2}^n$ and $2^{n-1} = \frac{1}{2}2^n$,
- we conclude that $f(n) = O(C^n)$ for some positive constant C satisfying

Run time

- Generally speaking, the tools we use to find good asymptotic bounds are algebra, calculus, and common sense...
- For example, recall that *fib*(n) had run time $f(n)$ with $2^{\frac{n}{2}} \leq f(n) \leq 2^{n-1}$.
- Moreover, since $2^{\frac{n}{2}} = \sqrt{2}^n$ and $2^{n-1} = \frac{1}{2}2^n$,
- we conclude that $f(n) = O(C^n)$ for some positive constant C satisfying
- $\sqrt{2} \leq C \leq 2$.

What about an asymptotic lower bound?

- The counterpart to Big Oh is Big Omega, which provides an asymptotic lower bound.

What about an asymptotic lower bound?

- The counterpart to Big Oh is Big Omega, which provides an asymptotic lower bound.
- In particular,

What about an asymptotic lower bound?

- The counterpart to Big Oh is Big Omega, which provides an asymptotic lower bound.
- In particular,
- $\Omega(g(n)) =$

  $\{f(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \; \forall n \geq n_0\}.$

Big Ω examples

- $n = \Omega(lg\ n)$, where $lg\ n = log_2(n)$.

Big Ω examples

- $n = \Omega(lg\ n)$, where $lg\ n = log_2(n)$.
- Any nonconstant polynomial is in $\Omega(n)$. Why?

Big Ω examples

- $n = \Omega(lg\ n)$, where $lg\ n = log_2(n)$.
- Any nonconstant polynomial is in $\Omega(n)$. Why?
- $2^n = \Omega(n)$.

Big Ω examples

- $n = \Omega(lg\ n)$, where $lg\ n = log_2(n)$.
- Any nonconstant polynomial is in $\Omega(n)$. Why?
- $2^n = \Omega(n)$.
- $lg(lg(n) \neq \Omega(n)$.

Big Ω examples

- $n = \Omega(lg\ n)$, where $lg\ n = log_2(n)$.
- Any nonconstant polynomial is in $\Omega(n)$. Why?
- $2^n = \Omega(n)$.
- $lg(lg(n) \neq \Omega(n)$.
- Why?

Why is $lg(lg(n)) \neq \Omega(n)$?

- Suppose BWOC that $\exists n_0, c > 0$ such that

Why is $lg(lg(n)) \neq \Omega(n)$?

- Suppose BWOC that $\exists n_0, c > 0$ such that
- $lg(lg(n)) \geq c lg(n) \forall n > n_0$.

Why is $lg(lg(n)) \neq \Omega(n)$?

- Suppose BWOC that $\exists n_0, c > 0$ such that
- $lg(lg(n)) \geq clg(n) \forall n > n_0$.
- Since the function 2^x is strictly increasing (the derivative is always positive), it follows that

Why is $lg(lg(n)) \neq \Omega(n)$?

- Suppose BWOC that $\exists n_0, c > 0$ such that
- $lg(lg(n)) \geq clg(n) \forall n > n_0$.
- Since the function 2^x is strictly increasing (the derivative is always positive), it follows that
- $2^{lg(lg(n))} \geq 2^{clg(n)}$

Why is $lg(lg(n)) \neq \Omega(n)$?

- Suppose BWOC that $\exists n_0, c > 0$ such that
- $lg(lg(n)) \geq clg(n) \forall n > n_0$.
- Since the function 2^x is strictly increasing (the derivative is always positive), it follows that
- $2^{lg(lg(n))} \geq 2^{clg(n)}$
- $\Rightarrow 2^{lg(lg(n))} \geq 2^{lg(n^c)}$

Why is $lg(lg(n)) \neq \Omega(n)$?

- Suppose BWOC that $\exists n_0, c > 0$ such that
- $lg(lg(n)) \geq c lg(n) \forall n > n_0$.
- Since the function 2^x is strictly increasing (the derivative is always positive), it follows that
- $2^{lg(lg(n))} \geq 2^{c lg(n)}$
- $\Rightarrow 2^{lg(lg(n))} \geq 2^{lg(n^c)}$
- $\Rightarrow lg(n) \geq n^c$, a contradiction.

Best kind of asymptotic bound: Big Θ

- The best possible asymptotic bound is Big Θ.

Best kind of asymptotic bound: Big Θ

- The best possible asymptotic bound is Big Θ.
- In particular, Θ will give both an upper and lower bound at the same time,

Best kind of asymptotic bound: Big Θ

- The best possible asymptotic bound is Big Θ.
- In particular, Θ will give both an upper and lower bound at the same time,
- which means you've found a function that mimics the exact run time of your algorithm.

Best kind of asymptotic bound: Big Θ

- The best possible asymptotic bound is Big Θ.
- In particular, Θ will give both an upper and lower bound at the same time,
- which means you've found a function that mimics the exact run time of your algorithm.
- That is, $\Theta(g(n)) =$

  $\{f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2(g(n) \ \forall n \geq n_0\}$.

Big Θ Example

- $\forall a, b, C \in \mathbb{R}$ with $a \neq 0$ we have that $an^2 + bn + C = \Theta(n^2)$.

Big Θ Example

- $\forall a, b, C \in \mathbb{R}$ with $a \neq 0$ we have that $an^2 + bn + C = \Theta(n^2)$.
- Unravel the definition of Θ and come up with values for n_0, c_1, and c_2.

Artist's Interpretation of O, Ω, and Θ

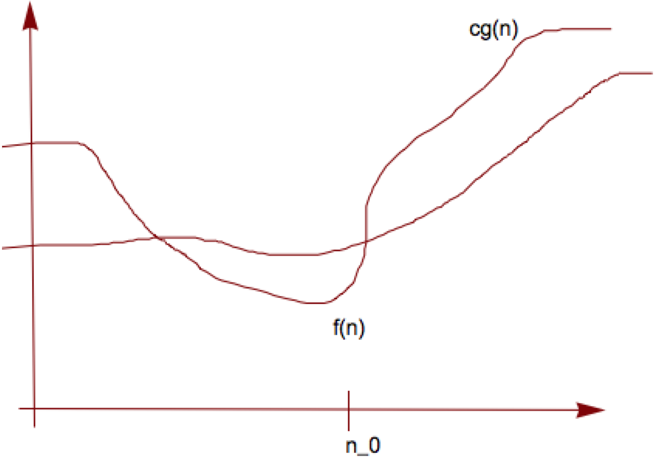


Figure: $f(n) = O(g(n))$

Artist's Interpretation of O, Ω, and Θ

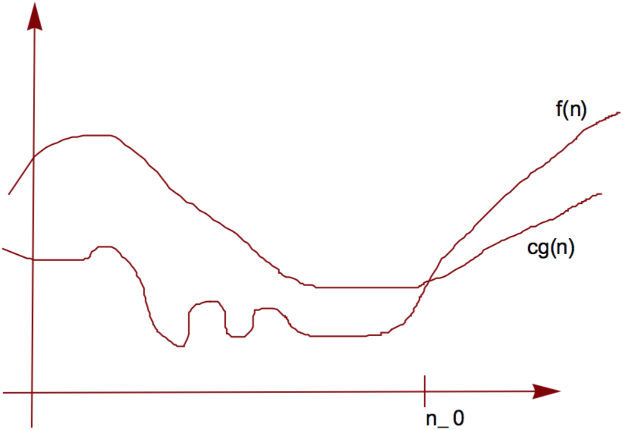


Figure: $f(n) = \Omega(g(n))$

Artist's Interpretation of O, Ω, and Θ

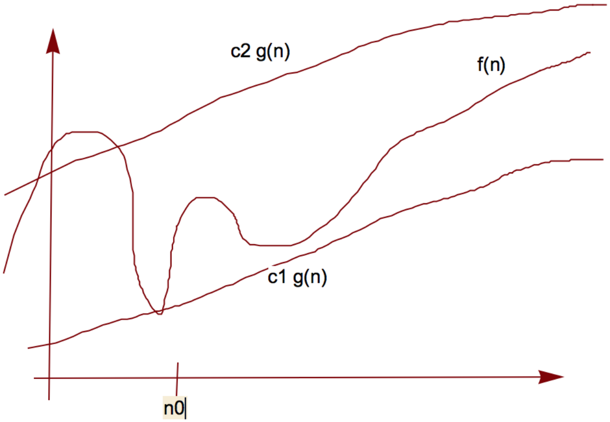


Figure: $f(n) = \Theta(g(n))$

New Topic: Master Theorem for Recursion

- We are continuing our review of the first few chapters of the textbook.

New Topic: Master Theorem for Recursion

- We are continuing our review of the first few chapters of the textbook.
- Suppose you have a recursive algorithm whose run time is given by $T(n)$ such that

New Topic: Master Theorem for Recursion

- We are continuing our review of the first few chapters of the textbook.

- Suppose you have a recursive algorithm whose run time is given by $T(n)$ such that

- for some $a, b \in \mathbb{Z}$ we have $T(n) = aT(\frac{n}{b}) + f(n)$.

New Topic: Master Theorem for Recursion

- We are continuing our review of the first few chapters of the textbook.

- Suppose you have a recursive algorithm whose run time is given by $T(n)$ such that

- for some $a, b \in \mathbb{Z}$ we have $T(n) = aT(\frac{n}{b}) + f(n)$.

- For a familiar example of this scenario, read **2-3** in our text on Mergesort.

New Topic: Master Theorem for Recursion

- We are continuing our review of the first few chapters of the textbook.
- Suppose you have a recursive algorithm whose run time is given by $T(n)$ such that
- for some $a, b \in \mathbb{Z}$ we have $T(n) = aT(\frac{n}{b}) + f(n)$.
- For a familiar example of this scenario, read **2-3** in our text on Mergesort.
- **Goal.** We want a closed-form asymptotic formula for $T(n)$.

New Topic: Master Theorem for Recursion

- We are continuing our review of the first few chapters of the textbook.
- Suppose you have a recursive algorithm whose run time is given by $T(n)$ such that
- for some $a, b \in \mathbb{Z}$ we have $T(n) = aT(\frac{n}{b}) + f(n)$.
- For a familiar example of this scenario, read **2-3** in our text on Mergesort.
- **Goal.** We want a closed-form asymptotic formula for $T(n)$.
- The final answer will depend on the nature of $f(n)$.

Theorem 4.1

- **Case 1.** If for some $\epsilon > 0$ we have $f(n) = O(n^{\log_b(a)-\epsilon})$ then $T(n) = \Theta(n^{\log_b(a)})$.

Theorem 4.1

- **Case 1.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)-\epsilon})$ then $T(n) = \Theta(n^{log_b(a)})$.
- **Case 2.** If $f(n) = \Theta(n^{log_b(a)})$ then $T(n) = \Theta(n^{log_b(a)} lg(n))$.

Theorem 4.1

- **Case 1.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)-\epsilon})$ then $T(n) = \Theta(n^{log_b(a)})$.
- **Case 2.** If $f(n) = \Theta(n^{log_b(a)})$ then $T(n) = \Theta(n^{log_b(a)} lg(n))$.
- **Case 3.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)+\epsilon})$ and

Theorem 4.1

- **Case 1.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)-\epsilon})$ then $T(n) = \Theta(n^{log_b(a)})$.

- **Case 2.** If $f(n) = \Theta(n^{log_b(a)})$ then $T(n) = \Theta(n^{log_b(a)} lg(n))$.

- **Case 3.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)+\epsilon})$ and

- if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$, n large, and $\epsilon > 0$ then

Theorem 4.1

- **Case 1.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)-\epsilon})$ then $T(n) = \Theta(n^{log_b(a)})$.

- **Case 2.** If $f(n) = \Theta(n^{log_b(a)})$ then $T(n) = \Theta(n^{log_b(a)} lg(n))$.

- **Case 3.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)+\epsilon})$ and

- if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$, n large, and $\epsilon > 0$ then

- $T(n) = \Theta(f(n))$.

Theorem 4.1

- **Case 1.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)-\epsilon})$ then $T(n) = \Theta(n^{log_b(a)})$.
- **Case 2.** If $f(n) = \Theta(n^{log_b(a)})$ then $T(n) = \Theta(n^{log_b(a)} lg(n))$.
- **Case 3.** If for some $\epsilon > 0$ we have $f(n) = O(n^{log_b(a)+\epsilon})$ and
- if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$, n large, and $\epsilon > 0$ then
- $T(n) = \Theta(f(n))$.
- We will prove Case 1 for the special case that n is an exact power of b.

Proof of Case 1 with $n = b^k$ for some $k \in \mathbb{Z}^+$.

- **Proof.** Let $n = b^k$ for some $k \in \mathbb{Z}^+$

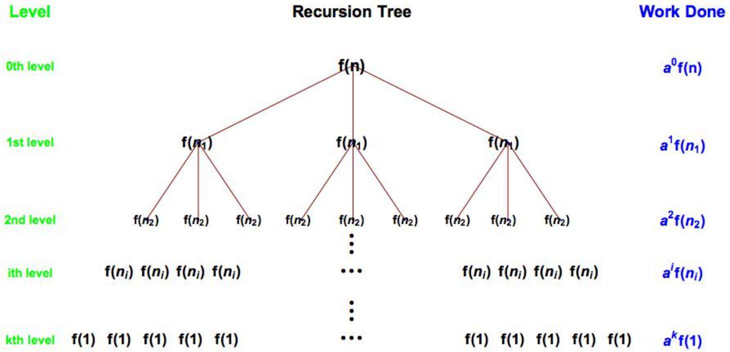
Proof of Case 1 with $n = b^k$ for some $k \in \mathbb{Z}^+$.

- **Proof.** Let $n = b^k$ for some $k \in \mathbb{Z}^+$
- Let's use a recursion tree in the analysis of $T(n)$.

Proof of Case 1 with $n = b^k$ for some $k \in \mathbb{Z}^+$.

- **Proof.** Let $n = b^k$ for some $k \in \mathbb{Z}^+$
- Let's use a recursion tree in the analysis of $T(n)$.
- For notational convenience, define $n_i = \frac{n}{b^i}$.

Proof of Case 1 with $n = b^k$ for some $k \in \mathbb{Z}^+$



Proof of Case 1, continued

- How many levels are there when $n = b^k$?

Proof of Case 1, continued

- How many levels are there when $n = b^k$?

- We keep going until we reach $f(1)$, which occurs at $\frac{n}{b^k} = \frac{b^k}{b^k}$, the kth level.

Proof of Case 1, continued

- How many levels are there when $n = b^k$?
- We keep going until we reach $f(1)$, which occurs at $\frac{n}{b^k} = \frac{b^k}{b^k}$, the kth level.
- Note that $log_b(n) = log_b(b^k) = k$ and thus the depth of the tree is exactly $k = log(n)$.

Proof of Case 1, continued

- How many levels are there when $n = b^k$?
- We keep going until we reach $f(1)$, which occurs at $\frac{n}{b^k} = \frac{b^k}{b^k}$, the kth level.
- Note that $log_b(n) = log_b(b^k) = k$ and thus the depth of the tree is exactly $k = log(n)$.
- At the bottom of the tree there are a^k constant time (ie $\Theta(1)$) operations to be performed.

Proof of Case 1, continued

- How many levels are there when $n = b^k$?
- We keep going until we reach $f(1)$, which occurs at $\frac{n}{b^k} = \frac{b^k}{b^k}$, the kth level.
- Note that $log_b(n) = log_b(b^k) = k$ and thus the depth of the tree is exactly $k = log(n)$.
- At the bottom of the tree there are a^k constant time (ie $\Theta(1)$) operations to be performed.
- How much work is done at the remaining $k - 1$ levels?

Work done in remaining $k-1$ levels

- Recall that in Case 1, we have $f(n) = O(n^{\log_b(a)-\epsilon})$

Work done in remaining $k-1$ levels

- Recall that in Case 1, we have $f(n) = O(n^{log_b(a)-\epsilon})$

- In that case, $n^{log_b(a)-\epsilon} = b^{k(log_b(a)-\epsilon)}$

Work done in remaining $k-1$ levels

- Recall that in Case 1, we have $f(n) = O(n^{log_b(a)-\epsilon})$
- In that case, $n^{log_b(a)-\epsilon} = b^{k(log_b(a)-\epsilon)}$
- $= b^{log_b(a^k)-k\epsilon}$

Work done in remaining $k-1$ levels

- Recall that in Case 1, we have $f(n) = O(n^{log_b(a)-\epsilon})$
- In that case, $n^{log_b(a)-\epsilon} = b^{k(log_b(a)-\epsilon)}$
- $= b^{log_b(a^k)-k\epsilon)}$
- $= a^k b^{-k\epsilon}$ (*).

Work done in remaining $k-1$ levels

- Recall that in Case 1, we have $f(n) = O(n^{log_b(a)-\epsilon})$
- In that case, $n^{log_b(a)-\epsilon} = b^{k(log_b(a)-\epsilon)}$
- $= b^{log_b(a^k)-k\epsilon)}$
- $= a^k b^{-k\epsilon}$ (*).
- It then follows that when we replace n with $n_j = \frac{b^k}{b^j} = b^{k-j}$ in (*), we have

Work done in remaining $k-1$ levels

- Recall that in Case 1, we have $f(n) = O(n^{log_b(a)-\epsilon})$
- In that case, $n^{log_b(a)-\epsilon} = b^{k(log_b(a)-\epsilon)}$
- $= b^{log_b(a^k)-k\epsilon}$
- $= a^k b^{-k\epsilon}$ (*).
- It then follows that when we replace n with $n_j = \frac{b^k}{b^j} = b^{k-j}$ in (*), we have
- $f(n_j) = O(a^{k-j} b^{-(k-j)\epsilon})$

Work done in remaining $k-1$ levels

- Recall that in Case 1, we have $f(n) = O(n^{log_b(a)-\epsilon})$
- In that case, $n^{log_b(a)-\epsilon} = b^{k(log_b(a)-\epsilon)}$
- $= b^{log_b(a^k)-k\epsilon)}$
- $= a^k b^{-k\epsilon}$ (*).
- It then follows that when we replace n with $n_j = \frac{b^k}{b^j} = b^{k-j}$ in (*), we have
- $f(n_j) = O(a^{k-j} b^{-(k-j)\epsilon})$
- $= O(a^{k-j} b^{j\epsilon} b^{-k\epsilon})$.

Total cost at jth level

- At the jth level we compute $f(n_j)$ exactly a^j times.

Total cost at jth level

- At the jth level we compute $f(n_j)$ exactly a^j times.
- Thus the total cost at the jth level is $O(a^j a^{k-j} b^{j\epsilon} b^{-k\epsilon})$

Total cost at jth level

- At the jth level we compute $f(n_j)$ exactly a^j times.
- Thus the total cost at the jth level is $O(a^j a^{k-j} b^{j\epsilon} b^{-k\epsilon})$
- $= O(\frac{a^k b^{j\epsilon}}{b^{k\epsilon}})$.

Total Cost Over the Whole Tree

- Since the cost of work done at the jth level is $O(\frac{a^k b^{j\epsilon}}{b^{k\epsilon}})$,

Total Cost Over the Whole Tree

- Since the cost of work done at the jth level is $O(\frac{a^k b^{j\epsilon}}{b^{k\epsilon}})$,
- the total cost over the whole recursion tree is

Total Cost Over the Whole Tree

- Since the cost of work done at the jth level is $O(\frac{a^k b^{j\epsilon}}{b^{k\epsilon}})$,
- the total cost over the whole recursion tree is
- $\Theta(a^k) + O(\frac{a^k}{b^{k\epsilon}} \sum_{j=0}^{k-1} (b^\epsilon)^j)$, *where*

Total Cost Over the Whole Tree

- Since the cost of work done at the jth level is $O(\frac{a^k b^{j\epsilon}}{b^{k\epsilon}})$,
- the total cost over the whole recursion tree is
- $\Theta(a^k) + O(\frac{a^k}{b^{k\epsilon}} \sum_{j=0}^{k-1} (b^\epsilon)^j)$, *where*
- $\Theta(a^k)$ is the work at the bottom (i.e. kth) level of the tree.

Total Cost Over the Whole Tree

- Since the cost of work done at the jth level is $O(\frac{a^k b^{j\epsilon}}{b^{k\epsilon}})$,
- the total cost over the whole recursion tree is
- $\Theta(a^k) + O(\frac{a^k}{b^{k\epsilon}} \sum_{j=0}^{k-1} (b^\epsilon)^j)$, *where*
- $\Theta(a^k)$ is the work at the bottom (i.e. kth) level of the tree.
- But $\sum_{j=0}^{k-1} (b^\epsilon)^j = \frac{b^{\epsilon k} - 1}{b^\epsilon - 1}$ because the summation is a finite geometric series.

Total Cost, continued

- Thus the total cost of the computation is

Total Cost, continued

- Thus the total cost of the computation is

- $\Theta(a^k) + O(\frac{a^k}{b^{\epsilon k}} \frac{b^{\epsilon k} - 1}{b^{\epsilon} - 1})$

Total Cost, continued

- Thus the total cost of the computation is
- $\Theta(a^k) + O(\frac{a^k}{b^{\epsilon k}} \frac{b^{\epsilon k}-1}{b^{\epsilon}-1})$
- $= \Theta(a^k) + O(\frac{a^k}{b^{\epsilon}-1} \frac{b^{\epsilon k}-1}{b^{\epsilon k}})$.

Total Cost, continued

- Thus the total cost of the computation is
- $\Theta(a^k) + O(\frac{a^k}{b^{\epsilon k}} \frac{b^{\epsilon k} - 1}{b^{\epsilon} - 1})$
- $= \Theta(a^k) + O(\frac{a^k}{b^{\epsilon} - 1} \frac{b^{\epsilon k} - 1}{b^{\epsilon k}})$.
- Recall that $\epsilon > 0$. In that case $\frac{b^{\epsilon k} - 1}{b^{\epsilon k}} < 1 \; \forall k \in \mathbb{Z}^+$.

Total Cost, continued

- Thus the total cost of the computation is
- $\Theta(a^k) + O(\frac{a^k}{b^{\epsilon k}} \frac{b^{\epsilon k}-1}{b^{\epsilon}-1})$
- $= \Theta(a^k) + O(\frac{a^k}{b^{\epsilon}-1} \frac{b^{\epsilon k}-1}{b^{\epsilon k}})$.
- Recall that $\epsilon > 0$. In that case $\frac{b^{\epsilon k}-1}{b^{\epsilon k}} < 1 \; \forall k \in \mathbb{Z}^{+}$.
- Morever, $b^{\epsilon} - 1$ is a constant; hence $O(\frac{a^k}{b^{\epsilon}-1} \frac{b^{\epsilon k}-1}{b^{\epsilon k}}) = O(a^k)$.

Total Cost, continued

- Thus the total cost of the computation is
- $\Theta(a^k) + O(\frac{a^k}{b^{\epsilon k}} \frac{b^{\epsilon k}-1}{b^{\epsilon}-1})$
- $= \Theta(a^k) + O(\frac{a^k}{b^{\epsilon}-1} \frac{b^{\epsilon k}-1}{b^{\epsilon k}})$.
- Recall that $\epsilon > 0$. In that case $\frac{b^{\epsilon k}-1}{b^{\epsilon k}} < 1 \; \forall k \in \mathbb{Z}^+$.
- Morever, $b^{\epsilon} - 1$ is a constant; hence $O(\frac{a^k}{b^{\epsilon}-1} \frac{b^{\epsilon k}-1}{b^{\epsilon k}}) = O(a^k)$.
- In that case, $T(n) = \Theta(a^k) + O(a^k) = \Theta(a^k) = \Theta(n^{\log_b a})$.

Total Cost, continued

- Thus the total cost of the computation is
- $\Theta(a^k) + O(\frac{a^k}{b^{\epsilon k}} \frac{b^{\epsilon k}-1}{b^{\epsilon}-1})$
- $= \Theta(a^k) + O(\frac{a^k}{b^{\epsilon}-1} \frac{b^{\epsilon k}-1}{b^{\epsilon k}})$.
- Recall that $\epsilon > 0$. In that case $\frac{b^{\epsilon k}-1}{b^{\epsilon k}} < 1 \; \forall k \in \mathbb{Z}^+$.
- Morever, $b^{\epsilon} - 1$ is a constant; hence $O(\frac{a^k}{b^{\epsilon}-1} \frac{b^{\epsilon k}-1}{b^{\epsilon k}}) = O(a^k)$.
- In that case, $T(n) = \Theta(a^k) + O(a^k) = \Theta(a^k) = \Theta(n^{log_b a})$.
- **QED**

New Topic/s: Dynamic Programming and Greedy Algorithms

- **Dynamic Programming** usually solves an optimization problem;

New Topic/s: Dynamic Programming and Greedy Algorithms

- **Dynamic Programming** usually solves an optimization problem;
- These are problems where you want the best possible solution such as, for example,

New Topic/s: Dynamic Programming and Greedy Algorithms

- **Dynamic Programming** usually solves an optimization problem;
- These are problems where you want the best possible solution such as, for example,
- the least costly or most profitable (time? money? etc).

New Topic/s: Dynamic Programming and Greedy Algorithms

- **Dynamic Programming** usually solves an optimization problem;
- These are problems where you want the best possible solution such as, for example,
- the least costly or most profitable (time? money? etc).
- How can we recognize such problems?

Recognizing a dynamic programming problem: two properties

1. **Optimal Substructure.** The best solution of the Big Problem is composed of the best solutions of smaller problems of the same kind.

Recognizing a dynamic programming problem: two properties

1. **Optimal Substructure.** The best solution of the Big Problem is composed of the best solutions of smaller problems of the same kind.

2. **Overlapping Subproblems.** Solving the Big Problem recusively involves solving the same little problem over and over.

Big Example 1: The Knapsack Problem

- **The Problem.** You are given an integer K (size of your knapsack) and n items of varying sizes such that the ith item has size k_i.

Big Example 1: The Knapsack Problem

- **The Problem.** You are given an integer K (size of your knapsack) and n items of varying sizes such that the ith item has size k_i.

- Find a subset of these items whose sizes sum to K, or else determine if no such subset exists.

Big Example 1: The Knapsack Problem

- **The Problem.** You are given an integer K (size of your knapsack) and n items of varying sizes such that the ith item has size k_i.

- Find a subset of these items whose sizes sum to K, or else determine if no such subset exists.

- **Set-up and Notation.**
  - Let $P(n, K)$ denote the original problem (n items and knapsack of size K)
  - Assume that the n items long with their sizes $k_1, k_2, \ldots k_n$ are given as input.
  - Let $P(i, k)$ denote the problem with the first i items and a knapsack of size k.

Knapsack problem, continued

- For now, concentrate on the *decision problem*, namely

Knapsack problem, continued

- For now, concentrate on the *decision problem*, namely
- whether or not a solution exists, and

Knapsack problem, continued

- For now, concentrate on the *decision problem*, namely
- whether or not a solution exists, and
- later on, if a solution exists, work on the particulars.

Algorithmic Solution to the Knapsack Problem

- **Algorithm Sadsack**
  - List out all subsets of the n items,
  - compute the sum of each subset's element sizes;
  - if at least one such sum is exactly K then done.
  - Otherwise, there is no solution.

Algorithmic Solution to the Knapsack Problem

- **Algorithm Sadsack**
  - List out all subsets of the n items,
  - compute the sum of each subset's element sizes;
  - if at least one such sum is exactly K then done.
  - Otherwise, there is no solution.
- What is the run time?

Algorithmic Solution to the Knapsack Problem

- **Algorithm Sadsack**
  - List out all subsets of the n items,
  - compute the sum of each subset's element sizes;
  - if at least one such sum is exactly K then done.
  - Otherwise, there is no solution.
- What is the run time?
- Exponential in n. Why?

Find a better solution!

- To design an algorithmic solution, we'll use the strong form of mathematical induction.

Find a better solution!

- To design an algorithmic solution, we'll use the strong form of mathematical induction.

- **Induction Hypothesis.** We know how to solve $P(n-1, k)$ for all $0 \leq k \leq K$.

Find a better solution!

- To design an algorithmic solution, we'll use the strong form of mathematical induction.

- **Induction Hypothesis.** We know how to solve $P(n-1,k)$ for all $0 \leq k \leq K$.

- **Base Case.** When $n = 1$, the problem $P(1, K)$ has a solution iff the first (and only) item satisfies $k_1 = K$.

Find a better solution!

- To design an algorithmic solution, we'll use the strong form of mathematical induction.
- **Induction Hypothesis.** We know how to solve $P(n-1,k)$ for all $0 \leq k \leq K$.
- **Base Case.** When $n = 1$, the problem $P(1,K)$ has a solution iff the first (and only) item satisfies $k_1 = K$.
- **Design.** If $P(n-1,K)$ has a solution, then the same solution solves $P(n,K)$,

Find a better solution!

- To design an algorithmic solution, we'll use the strong form of mathematical induction.
- **Induction Hypothesis.** We know how to solve $P(n-1, k)$ for all $0 \leq k \leq K$.
- **Base Case.** When $n = 1$, the problem $P(1, K)$ has a solution iff the first (and only) item satisfies $k_1 = K$.
- **Design.** If $P(n-1, K)$ has a solution, then the same solution solves $P(n, K)$,
- and we simply choose not to pack the nth item, and we are done.

Designing a solution using induction

- Otherwise, if $P(n-1, K)$ has no solution then in order for a solution for $P(n, K)$ to exist,

Designing a solution using induction

- Otherwise, if $P(n-1, K)$ has no solution then in order for a solution for $P(n, K)$ to exist,

- we must pack the nth item

Designing a solution using induction

- Otherwise, if $P(n-1, K)$ has no solution then in order for a solution for $P(n, K)$ to exist,
- we must pack the nth item
- and we must be able to pack a subset of items 1 through $n-1$ into a knapsack of size $K - k_n$.

Designing a solution using induction

- Otherwise, if $P(n-1, K)$ has no solution then in order for a solution for $P(n, K)$ to exist,

- we must pack the nth item

- and we must be able to pack a subset of items 1 through $n-1$ into a knapsack of size $K - k_n$.

- That is, we must now solve $P(n-1, K)$ and $P(n-1, K-k_n)$.

Designing a solution using induction

- Otherwise, if $P(n-1, K)$ has no solution then in order for a solution for $P(n, K)$ to exist,
- we must pack the nth item
- and we must be able to pack a subset of items 1 through $n-1$ into a knapsack of size $K - k_n$.
- That is, we must now solve $P(n-1, K)$ and $P(n-1, K-k_n)$.
- **Observation.** We have reduced a problem of size n to two problems of size $n-1$ and will lead to an exponential algorithm.

Designing a solution using induction

- Otherwise, if $P(n-1, K)$ has no solution then in order for a solution for $P(n, K)$ to exist,
- we must pack the nth item
- and we must be able to pack a subset of items 1 through $n-1$ into a knapsack of size $K - k_n$.
- That is, we must now solve $P(n-1, K)$ and $P(n-1, K-k_n)$.
- **Observation.** We have reduced a problem of size n to two problems of size $n-1$ and will lead to an exponential algorithm.
- Again ☹

The Fix: Dynamic Programming

- We remember previous computations so as not have to compute the same things repeatedly.

The Fix: Dynamic Programming

- We remember previous computations so as not have to compute the same things repeatedly.

- **The Specifics.** In the problem $P(n, K)$ there are only $(n+1)(K+1)$ subproblems to consider.

The Fix: Dynamic Programming

- We remember previous computations so as not have to compute the same things repeatedly.

- **The Specifics.** In the problem $P(n, K)$ there are only $(n+1)(K+1)$ subproblems to consider.

- Store all of the results in an $(n+1) \times (K+1)$ array;

The Fix: Dynamic Programming

- We remember previous computations so as not have to compute the same things repeatedly.

- **The Specifics.** In the problem $P(n, K)$ there are only $(n+1)(K+1)$ subproblems to consider.

- Store all of the results in an $(n+1) \times (K+1)$ array;

- the (i, k)th entry contains information about the solution of $P(i, k)$.

The Fix: Dynamic Programming

- We remember previous computations so as not have to compute the same things repeatedly.

- **The Specifics.** In the problem $P(n, K)$ there are only $(n+1)(K+1)$ subproblems to consider.

- Store all of the results in an $(n+1) \times (K+1)$ array;

- the (i, k)th entry contains information about the solution of $P(i, k)$.

- If we want to know the actual subset of items to be packed, we can add a field to the (i, k)th entry that indicates whether or not item i is to be packed.

Algorithm Knapsack(S, K)

Input. S, a $1 \times n$ array storing item sizes $k_1, \ldots, k_n$ and K, the size of the knapsack. So $S[i] = k_i$.

Output. P, a 2-dimensional array such that

$$\begin{cases} P[i,k].\mathit{exist} = \text{true} & \text{if } \exists \text{a solution to } P(i,k) \\ P[i,k].\mathit{belong} = \text{true} & \text{if the ith item is to be packed} \end{cases}$$

Algorithm Knapsack, continued

- **begin**
- $P[0,0].exist = \text{true}$
- **for** $k = 1$ **to** K **do**
- $P[0,k].belong = \text{false}$
- **for** $i = 1$ **to** n **do**
- for $k = 0$ to K do
- $P[i,k].exist = \text{false}$ *(default value)*
- **if** $P[i-1,k].exist = \text{true}$ **then**
- $P[i,k].exist = \text{true}$
- **else if** $k - S[i] \geq 0$ **then**
- **if** $P[i-1, k - S[i].exist = \text{true}$ **then**
- $P[i,k].exist = \text{true}$
- $P[i,k].belong = \text{true}$
- **end**

Complexity of Algorithm Knapsack

- There are $(n+1)(K+1)$ to compute in the output array,

Complexity of Algorithm Knapsack

- There are $(n+1)(K+1)$ to compute in the output array,
- each of which is computed in constant time from two other entries in the array

Complexity of Algorithm Knapsack

- There are $(n+1)(K+1)$ to compute in the output array,
- each of which is computed in constant time from two other entries in the array
- $\Rightarrow$ the total run time for the decision problem is $O(nK)$, which is better than exponential in general.

Complexity of Algorithm Knapsack

- There are $(n+1)(K+1)$ to compute in the output array,
- each of which is computed in constant time from two other entries in the array
- $\Rightarrow$ the total run time for the decision problem is $O(nK)$, which is better than exponential in general.
- Tracing back through the array $P[n, K]$ to find the valid "belong" elements takes time $O(n)$.

Next

We'll do another dynamic programming example next week, and then move on to greedy algorithms.