COMP 761: Lecture 31 – Dynamic Programming

David Rolnick

November 16, 2020

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

(Please don't post your ideas in the chat just yet, we'll discuss the problem soon in class.)

Course Announcements

Course Announcements

- Problem 1 – number of collisions e.g. in a slot with 4 keys is $3+2+1=6$.

Course Announcements

- Problem 1 – number of collisions e.g. in a slot with 4 keys is $3+2+1=6$.
- Problem 3 – the point is that in Ford-Fulkerson, we do not know how many augmenting paths we will use. The augmenting paths are added one at a time, in the sense that you add one, construct the residual network, then find another, and so one. There is no guarantee that this process ends after $|E|$ iterations – in particular, you might choose an augmenting path that backtracks on a previous one since the residual network has backward edges. But if you had hindsight, could you pick the augmenting paths to make it stop after $|E|$ iterations?

Course Announcements

- Problem 1 – number of collisions e.g. in a slot with 4 keys is $3+2+1=6$.
- Problem 3 – the point is that in Ford-Fulkerson, we do not know how many augmenting paths we will use. The augmenting paths are added one at a time, in the sense that you add one, construct the residual network, then find another, and so one. There is no guarantee that this process ends after $|E|$ iterations – in particular, you might choose an augmenting path that backtracks on a previous one since the residual network has backward edges. But if you had hindsight, could you pick the augmenting paths to make it stop after $|E|$ iterations?
- Office hours right after class!

Review: Universal hashing

Review: Universal hashing

- We have a problem if an adversary gets to pick the keys for a hash functions like $\lfloor mk \rfloor$.

Review: Universal hashing

- We have a problem if an adversary gets to pick the keys for a hash functions like $\lfloor mk \rfloor$.
- Can make them all collide very easily.

Review: Universal hashing

- We have a problem if an adversary gets to pick the keys for a hash functions like $\lfloor mk \rfloor$.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.

Review: Universal hashing

- We have a problem if an adversary gets to pick the keys for a hash functions like $\lfloor mk \rfloor$.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.
- Specifically, the goal is that for *any* pair of keys (even not random), we are likely to pick a hash function where they don't collide.

Review: Universal hashing

- We have a problem if an adversary gets to pick the keys for a hash functions like $\lfloor mk \rfloor$.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.
- Specifically, the goal is that for *any* pair of keys (even not random), we are likely to pick a hash function where they don't collide.
- It's basically moving the randomness assumption from the keys to the hash function, since we know we can control that.

Review: Universal hashing

- We have a problem if an adversary gets to pick the keys for a hash functions like $\lfloor mk \rfloor$.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.
- Specifically, the goal is that for *any* pair of keys (even not random), we are likely to pick a hash function where they don't collide.
- It's basically moving the randomness assumption from the keys to the hash function, since we know we can control that.
- Formally, let $\mathcal{H}$ be a set of possible hash functions, from which we pick an h at random.

Review: Universal hashing

- We have a problem if an adversary gets to pick the keys for a hash functions like $\lfloor mk \rfloor$.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.
- Specifically, the goal is that for *any* pair of keys (even not random), we are likely to pick a hash function where they don't collide.
- It's basically moving the randomness assumption from the keys to the hash function, since we know we can control that.
- Formally, let $\mathcal{H}$ be a set of possible hash functions, from which we pick an h at random.
- We say that $\mathcal{H}$ is *universal* if for any two different keys k_1, k_2, the probability of picking h with $h(k_1) = h(k_2)$ is at most $1/m$.

Universal hashing

Universal hashing

- Let's look at one way to design a universal set of hash functions.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.
- Since we assume more keys than slots, we have $p > m$.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.
- Since we assume more keys than slots, we have $p > m$.
- For every $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, 2, \ldots, p-1\}$, we define:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.
- Since we assume more keys than slots, we have $p > m$.
- For every $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, 2, \ldots, p-1\}$, we define:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p-1$.
- Since we assume more keys than slots, we have $p > m$.
- For every $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, 2, \ldots, p-1\}$, we define:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- $p-1$ choices for a and p for b, so $p(p-1)$ choices for h_{ab}.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p-1$.
- Since we assume more keys than slots, we have $p > m$.
- For every $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, 2, \ldots, p-1\}$, we define:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- $p-1$ choices for a and p for b, so $p(p-1)$ choices for h_{ab}.
- We will prove that this set of hash functions is universal.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

- Can we have $r_1 = r_2$?

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

- Can we have $r_1 = r_2$?
- If we did, we'd have $ak_1 \equiv ak_2 \mod p$. Is that possible?

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

- Can we have $r_1 = r_2$?
- If we did, we'd have $ak_1 \equiv ak_2 \mod p$. Is that possible?
- No, since p is prime, it's relatively prime to a, so can divide by a to get $k_1 \equiv k_2 \mod p$, a contradiction.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

- Can we have $r_1 = r_2$?
- If we did, we'd have $ak_1 \equiv ak_2 \mod p$. Is that possible?
- No, since p is prime, it's relatively prime to a, so can divide by a to get $k_1 \equiv k_2 \mod p$, a contradiction.
- Therefore, we have $r_1 \neq r_2$.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$\begin{aligned} r_1 &= (ak_1 + b) \mod p \\ r_2 &= (ak_2 + b) \mod p. \end{aligned}$$

- Therefore, we have $r_1 \neq r_2$.
- Even better, given p, r_1, r_2, k_1, k_2, we can solve for a and b:

$$\begin{aligned} b &\equiv r_1 - ak_1 \mod p \\ (k_1 - k_2)a &\equiv (r_1 - r_2) \mod p, \end{aligned}$$

and we can divide by $k_1 - k_2$ since it is relatively prime to p.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

- Therefore, we have $r_1 \neq r_2$.
- Even better, given p, r_1, r_2, k_1, k_2, we can solve for a and b:

$$b \equiv r_1 - ak_1 \mod p$$
$$(k_1 - k_2)a \equiv (r_1 - r_2) \mod p,$$

  and we can divide by $k_1 - k_2$ since it is relatively prime to p.
- Therefore, each (a, b) gives us a different (r_1, r_2).

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

- Therefore, we have $r_1 \neq r_2$.
- Even better, given p, r_1, r_2, k_1, k_2, we can solve for a and b:

$$b \equiv r_1 - ak_1 \mod p$$
$$(k_1 - k_2)a \equiv (r_1 - r_2) \mod p,$$

  and we can divide by $k_1 - k_2$ since it is relatively prime to p.
- Therefore, each (a, b) gives us a different (r_1, r_2).
- $p(p-1)$ choices for (a, b) and $p(p-1)$ choices for different (r_1, r_2).

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- Suppose we have keys $k_1 \neq k_2$. Define the remainders mod p:

$$\begin{aligned} r_1 &= (ak_1 + b) \mod p \\ r_2 &= (ak_2 + b) \mod p. \end{aligned}$$

- Therefore, we have $r_1 \neq r_2$.
- Even better, given p, r_1, r_2, k_1, k_2, we can solve for a and b:

$$\begin{aligned} b &\equiv r_1 - ak_1 \mod p \\ (k_1 - k_2)a &\equiv (r_1 - r_2) \mod p, \end{aligned}$$

  and we can divide by $k_1 - k_2$ since it is relatively prime to p.
- Therefore, each (a, b) gives us a different (r_1, r_2).
- $p(p-1)$ choices for (a, b) and $p(p-1)$ choices for different (r_1, r_2).
- So (r_1, r_2) must vary over each pair of different values mod p.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- For $k_1 \neq k_2$, we set:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

and found (r_1, r_2) is equally likely to be any pair of different values in $\{0, 1, 2, \ldots, p-1\}$.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- For $k_1 \neq k_2$, we set:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

  and found (r_1, r_2) is equally likely to be any pair of different values in $\{0, 1, 2, \ldots, p-1\}$.
- We have $h_{ab}(k_1) = h_{ab}(k_2)$ if and only if $r_1 \equiv r_2 \mod m$.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- For $k_1 \neq k_2$, we set:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

  and found (r_1, r_2) is equally likely to be any pair of different values in $\{0, 1, 2, \ldots, p-1\}$.
- We have $h_{ab}(k_1) = h_{ab}(k_2)$ if and only if $r_1 \equiv r_2 \mod m$.
- For any r_1, the number of values $r_2 \neq r_1$ in $\{0, 1, 2, \ldots, p-1\}$ where $r_1 \equiv r_2 \mod m$ is at most $(p-1)/m$.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- For $k_1 \neq k_2$, we set:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

  and found (r_1, r_2) is equally likely to be any pair of different values in $\{0, 1, 2, \ldots, p-1\}$.
- We have $h_{ab}(k_1) = h_{ab}(k_2)$ if and only if $r_1 \equiv r_2 \mod m$.
- For any r_1, the number of values $r_2 \neq r_1$ in $\{0, 1, 2, \ldots, p-1\}$ where $r_1 \equiv r_2 \mod m$ is at most $(p-1)/m$.
- For example, if $p = 19$, $m = 3$, $r_1 = 12$, we have the $(19-1)/3 = 6$ possibilities:

$$0, 3, 6, 9, 15, 18.$$

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- For $k_1 \neq k_2$, we set:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

  and found (r_1, r_2) is equally likely to be any pair of different values in $\{0, 1, 2, \ldots, p-1\}$.
- We have $h_{ab}(k_1) = h_{ab}(k_2)$ if and only if $r_1 \equiv r_2 \mod m$.
- For any r_1, the number of values $r_2 \neq r_1$ in $\{0, 1, 2, \ldots, p-1\}$ where $r_1 \equiv r_2 \mod m$ is at most $(p-1)/m$.
- For example, if $p = 19$, $m = 3$, $r_1 = 12$, we have the $(19-1)/3 = 6$ possibilities:

$$0, 3, 6, 9, 15, 18.$$

- Since there are $p-1$ choices for r_2, the probability of having $r_1 \equiv r_2 \mod m$ is at most $1/m$.

Universal hashing

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- For $k_1 \neq k_2$, we set:

$$r_1 = (ak_1 + b) \mod p$$
$$r_2 = (ak_2 + b) \mod p.$$

  and found (r_1, r_2) is equally likely to be any pair of different values in $\{0, 1, 2, \ldots, p-1\}$.
- We have $h_{ab}(k_1) = h_{ab}(k_2)$ if and only if $r_1 \equiv r_2 \mod m$.
- For any r_1, the number of values $r_2 \neq r_1$ in $\{0, 1, 2, \ldots, p-1\}$ where $r_1 \equiv r_2 \mod m$ is at most $(p-1)/m$.
- For example, if $p = 19$, $m = 3$, $r_1 = 12$, we have the $(19-1)/3 = 6$ possibilities:

$$0, 3, 6, 9, 15, 18.$$

- Since there are $p-1$ choices for r_2, the probability of having $r_1 \equiv r_2 \mod m$ is at most $1/m$.
- This means that the set $\mathcal{H}$ of hash functions h_{ab} is universal.

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

- What's the brute force way to do this?

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

- What's the brute force way to do this?
- We can go through every possible way to write n as a sum of $k \leq n$.

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

- What's the brute force way to do this?
- We can go through every possible way to write n as a sum of $k \leq n$.
- Example: if $n = 10$, then can write $10 = 1 + 1 + 1 + 1 + 1 + 5$ or $10 = 1 + 2 + 7$ or $10 = 10$, etc.

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

- What's the brute force way to do this?
- We can go through every possible way to write n as a sum of $k \leq n$.
- Example: if $n = 10$, then can write $10 = 1 + 1 + 1 + 1 + 1 + 5$ or $10 = 1 + 2 + 7$ or $10 = 10$, etc.
- Then we have to see which is smallest out of $5p_1 + p_5, p_1 + p_2 + p_7, p_{10}, \ldots$.
- How many ways do we have to consider? (Assuming that order matters, e.g. $1 + 2 + 7$ is counted separately to $7 + 2 + 1$.)

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

- What's the brute force way to do this?
- We can go through every possible way to write n as a sum of $k \leq n$.
- Example: if $n = 10$, then can write $10 = 1 + 1 + 1 + 1 + 1 + 5$ or $10 = 1 + 2 + 7$ or $10 = 10$, etc.
- Then we have to see which is smallest out of $5p_1 + p_5, p_1 + p_2 + p_7, p_{10}, \ldots$.
- How many ways do we have to consider? (Assuming that order matters, e.g. $1 + 2 + 7$ is counted separately to $7 + 2 + 1$.)
- We can imagine all the balls in a row $O, O, O, O, O, O, O, O, O, O$.

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

- What's the brute force way to do this?
- We can go through every possible way to write n as a sum of $k \leq n$.
- Example: if $n = 10$, then can write $10 = 1 + 1 + 1 + 1 + 1 + 5$ or $10 = 1 + 2 + 7$ or $10 = 10$, etc.
- Then we have to see which is smallest out of $5p_1 + p_5, p_1 + p_2 + p_7, p_{10}, \ldots$.
- How many ways do we have to consider? (Assuming that order matters, e.g. $1 + 2 + 7$ is counted separately to $7 + 2 + 1$.)
- We can imagine all the balls in a row $O, O, O, O, O, O, O, O, O, O$.
- Each of the $n - 1$ commas is a possible break point, e.g. $O, O, O \mid O, O \mid O, O, O, O, O$.

Problem

You would like to buy exactly n identical balls. A bag of k costs p_k, for $k = 1, 2, \ldots, n$. (There are no conditions on the p_k, except that they are positive.) What is an efficient algorithm for determining the minimum price you have to pay?

- What's the brute force way to do this?
- We can go through every possible way to write n as a sum of $k \leq n$.
- Example: if $n = 10$, then can write $10 = 1 + 1 + 1 + 1 + 1 + 5$ or $10 = 1 + 2 + 7$ or $10 = 10$, etc.
- Then we have to see which is smallest out of $5p_1 + p_5, p_1 + p_2 + p_7, p_{10}, \ldots$.
- How many ways do we have to consider? (Assuming that order matters, e.g. $1 + 2 + 7$ is counted separately to $7 + 2 + 1$.)
- We can imagine all the balls in a row $O, O, O, O, O, O, O, O, O, O$.
- Each of the $n - 1$ commas is a possible break point, e.g. $O, O, O \mid O, O \mid O, O, O, O, O$.
- So 2^{n-1} possibilities, which is a lot to check.

Recursive approach

Recursive approach

- Any way to make this faster?

Recursive approach

- Let's try doing this recursively, like we did with sorting.

Recursive approach

- Let's try doing this recursively, like we did with sorting.
- Suppose the first ball is in a group of k.

Recursive approach

- Let's try doing this recursively, like we did with sorting.
- Suppose the first ball is in a group of k.
- Then, we have $n - k$ balls left.

Recursive approach

- Let's try doing this recursively, like we did with sorting.
- Suppose the first ball is in a group of k.
- Then, we have $n - k$ balls left.
- So if $f(n)$ is what we want to determine, we have:

$$f(n) = \min_k (f(n-k) + p_k).$$

Recursive approach

- Let's try doing this recursively, like we did with sorting.
- Suppose the first ball is in a group of k.
- Then, we have $n - k$ balls left.
- So if $f(n)$ is what we want to determine, we have:

$$f(n) = \min_k (f(n-k) + p_k).$$

- Suppose that $f(n)$ takes $t(n)$ steps to calculate – then, we have:

$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$

Recursive approach

- Let's try doing this recursively, like we did with sorting.
- Suppose the first ball is in a group of k.
- Then, we have $n - k$ balls left.
- So if $f(n)$ is what we want to determine, we have:

$$f(n) = \min_k (f(n-k) + p_k).$$

- Suppose that $f(n)$ takes $t(n)$ steps to calculate – then, we have:

$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$

- What is the solution to this recurrence relation?

Recursive approach

- Let's try doing this recursively, like we did with sorting.
- Suppose the first ball is in a group of k.
- Then, we have $n - k$ balls left.
- So if $f(n)$ is what we want to determine, we have:

$$f(n) = \min_k (f(n-k) + p_k).$$

- Suppose that $f(n)$ takes $t(n)$ steps to calculate – then, we have:

$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$

- Solving this gives $t(n) = \Theta(2^n)$, as witnessed by:

$$2^n = 1 + \sum_{k=1}^{n} 2^{n-k}.$$

Recursive approach

- Let's try doing this recursively, like we did with sorting.
- Suppose the first ball is in a group of k.
- Then, we have $n - k$ balls left.
- So if $f(n)$ is what we want to determine, we have:

$$f(n) = \min_k (f(n-k) + p_k).$$

- Suppose that $f(n)$ takes $t(n)$ steps to calculate – then, we have:

$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$

- Solving this gives $t(n) = \Theta(2^n)$, as witnessed by:

$$2^n = 1 + \sum_{k=1}^{n} 2^{n-k}.$$

- What went wrong? This was supposed to be faster!

More efficient approach

More efficient approach

- We had:

$$f(n) = \min_k (f(n-k) + p_k).$$

More efficient approach

- We had:

$$f(n) = \min_k (f(n-k) + p_k).$$

- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:

$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$

More efficient approach

- We had:

$$f(n) = \min_k (f(n-k) + p_k).$$

- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:

$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$

- How can we do better?

More efficient approach

- We had:

$$f(n) = \min_k (f(n-k) + p_k).$$

- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:

$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$

- When we calculate $f(n)$, we're calculating $f(n-1)$, $f(n-2)$, $f(n-3)$, etc.

More efficient approach

- We had:

$$f(n) = \min_k (f(n-k) + p_k).$$

- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:

$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$

- When we calculate $f(n)$, we're calculating $f(n-1)$, $f(n-2)$, $f(n-3)$, etc.
- When we calculate $f(n-1)$, we calculate $f(n-2)$, $f(n-3)$, etc. *again*.

More efficient approach

- We had:
$$f(n) = \min_k (f(n-k) + p_k).$$
- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:
$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$
- When we calculate $f(n)$, we're calculating $f(n-1)$, $f(n-2)$, $f(n-3)$, etc.
- When we calculate $f(n-1)$, we calculate $f(n-2)$, $f(n-3)$, etc. *again*.
- That's wasted computation, why not calculate $f(n-2)$ once and reuse it!

More efficient approach

- We had:
$$f(n) = \min_k (f(n-k) + p_k).$$
- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:
$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$
- When we calculate $f(n)$, we're calculating $f(n-1)$, $f(n-2)$, $f(n-3)$, etc.
- When we calculate $f(n-1)$, we calculate $f(n-2)$, $f(n-3)$, etc. *again*.
- That's wasted computation, why not calculate $f(n-2)$ once and reuse it!
- New approach: Compute $f(1)$, then use that to compute $f(2)$, use the stored $f(1)$ and $f(2)$ to compute $f(3)$, etc.

More efficient approach

- We had:
$$f(n) = \min_k (f(n-k) + p_k).$$
- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:
$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$
- When we calculate $f(n)$, we're calculating $f(n-1)$, $f(n-2)$, $f(n-3)$, etc.
- When we calculate $f(n-1)$, we calculate $f(n-2)$, $f(n-3)$, etc. *again*.
- That's wasted computation, why not calculate $f(n-2)$ once and reuse it!
- New approach: Compute $f(1)$, then use that to compute $f(2)$, use the stored $f(1)$ and $f(2)$ to compute $f(3)$, etc.
- Defining $t(n)$ as the time required to compute and store *all of* $f(1), f(2), \ldots, f(n)$, we have:
$$t(n) = t(n-1) + cn,$$
where the cn is just from doing a min over n numbers.

More efficient approach

- We had:
$$f(n) = \min_k (f(n-k) + p_k).$$
- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:
$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$
- New approach: Compute $f(1)$, then use that to compute $f(2)$, use the stored $f(1)$ and $f(2)$ to compute $f(3)$, etc.
- Defining $t(n)$ as the time required to compute and store *all of* $f(1), f(2), \ldots, f(n)$, we have:
$$t(n) = t(n-1) + cn,$$
where the cn is just from doing a min over n numbers.
- What is the solution to $t(n)$ for this algorithm?

More efficient approach

- We had:
$$f(n) = \min_k (f(n-k) + p_k).$$
- Our algorithm was essentially doing all the work of considering all the possible configurations, just framed as a recursion:
$$t(n) = \sum_{k=1}^{n} (t(n-k) + c).$$
- New approach: Compute $f(1)$, then use that to compute $f(2)$, use the stored $f(1)$ and $f(2)$ to compute $f(3)$, etc.
- Defining $t(n)$ as the time required to compute and store *all of* $f(1), f(2), \ldots, f(n)$, we have:
$$t(n) = t(n-1) + cn,$$
where the cn is just from doing a min over n numbers.
- What is the solution to $t(n)$ for this algorithm?
- It's just
$$t(n) = cn + c(n-1) + \cdots + c = c(n + (n-1) + \cdots + 1) = \Theta(n^2).$$

Dynamic programming

Dynamic programming

- *Dynamic programming* is a paradigm of algorithm design in which solutions to smaller subproblems are calculated *and reused*.

Dynamic programming

- *Dynamic programming* is a paradigm of algorithm design in which solutions to smaller subproblems are calculated *and reused*.
- One way of doing this is to solve all the possible subproblems in order of increasing size, so each one has all the solutions it needs.

Dynamic programming

- *Dynamic programming* is a paradigm of algorithm design in which solutions to smaller subproblems are calculated *and reused*.
- One way of doing this is to solve all the possible subproblems in order of increasing size, so each one has all the solutions it needs.
- Alternatively, they can be computed as they are needed, but then stored in case they must be reused. This is called *memoization* (not a typo – we are making *memos* of useful solutions).