

COMPSCI 320S2C, 2006: Algorithmics

Mark C. Wilson

August 9, 2006

Organizational matters

- Lecturer: Dr Mark Wilson.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email `mcw@cs.auckland.ac.nz`. Phone extn 86643.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email `mcw@cs.auckland.ac.nz`. Phone extn 86643.
- Office: City 303.588.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email `mcw@cs.auckland.ac.nz`. Phone extn 86643.
- Office: City 303.588.
- Lectures: will stick mostly to textbook, but there may be some extra material. Please ask questions.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email `mcw@cs.auckland.ac.nz`. Phone extn 86643.
- Office: City 303.588.
- Lectures: will stick mostly to textbook, but there may be some extra material. Please ask questions.
- Tutorial: please register immediately - they are compulsory.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email `mcw@cs.auckland.ac.nz`. Phone extn 86643.
- Office: City 303.588.
- Lectures: will stick mostly to textbook, but there may be some extra material. Please ask questions.
- Tutorial: please register immediately - they are compulsory.
- Other resources: course webpages, my handouts directory, lecturers, tutorials, class forum, library (books on reserve).

How to multiply two positive integers?

- The well-known algorithm taught at primary school uses the decimal representation, and requires a lookup table for 1-digit numbers.

How to multiply two positive integers?

- The well-known algorithm taught at primary school uses the decimal representation, and requires a lookup table for 1-digit numbers.
- Russian peasants apparently used a different algorithm, which requires only addition and multiplication/division by 2. It is well suited to computers.

How to multiply two positive integers?

- The well-known algorithm taught at primary school uses the decimal representation, and requires a lookup table for 1-digit numbers.
- Russian peasants apparently used a different algorithm, which requires only addition and multiplication/division by 2. It is well suited to computers.
- How to compare these two algorithms?

How to multiply two positive integers?

- The well-known algorithm taught at primary school uses the decimal representation, and requires a lookup table for 1-digit numbers.
- Russian peasants apparently used a different algorithm, which requires only addition and multiplication/division by 2. It is well suited to computers.
- How to compare these two algorithms?
- We define the **size** of an integer x to be the number of its binary digits, $b(x) := 1 + \lfloor \log_2 |x| \rfloor$.

How to multiply two positive integers?

- The well-known algorithm taught at primary school uses the decimal representation, and requires a lookup table for 1-digit numbers.
- Russian peasants apparently used a different algorithm, which requires only addition and multiplication/division by 2. It is well suited to computers.
- How to compare these two algorithms?
- We define the **size** of an integer x to be the number of its binary digits, $b(x) := 1 + \lfloor \log_2 |x| \rfloor$.
- How do the resource requirements grow as a function of the **problem size**?

How to multiply two positive integers?

- The well-known algorithm taught at primary school uses the decimal representation, and requires a lookup table for 1-digit numbers.
- Russian peasants apparently used a different algorithm, which requires only addition and multiplication/division by 2. It is well suited to computers.
- How to compare these two algorithms?
- We define the **size** of an integer x to be the number of its binary digits, $b(x) := 1 + \lfloor \log_2 |x| \rfloor$.
- How do the resource requirements grow as a function of the **problem size**?
- What is an **elementary operation** (basic unit of running time)? Addition of very large integers is probably not one.

“Russian peasant” multiplication

```
algorithm russmult(int  $x$ , int  $y$ )
{ reduce to case of positive input }
 $s \leftarrow 1$ ;
if ( $x < 0$ ) then  $s \leftarrow -s$ ;  $x \leftarrow -x$ ;
if ( $y < 0$ ) then  $s \leftarrow -s$ ;  $y \leftarrow -y$ ;
{ now multiply positive integers }
 $t \leftarrow 0$ ;
while  $x > 0$  do
    if ( $x \bmod 2 = 1$ ) then  $t \leftarrow t + y$ ;
     $x \leftarrow x \div 2$ ;
     $y \leftarrow 2 * y$ ;
return  $s * t$ ;
end
```

Correctness of russmult

First note that the algorithm always terminates, by basic properties of natural numbers.

It is easy to see that the algorithm is correct if and only if it is correct for nonnegative integer input, so we assume $x, y \geq 0$.

We use induction on x . If $x = 0$, the algorithm returns 0. Now suppose that $x \geq 1$. If x is even, $x = 2x'$, then t remains 0 after first iteration. The algorithm now proceeds as though it were being run on $x', 2y$. By induction, it returns $t = 2yx' = xy$ on these inputs, hence on the original inputs. On the other hand, if x is odd, $x = 2x' + 1$, then t has value y after one iteration. Let $T = t - y$ so $T = 0$ now. The algorithm now proceeds as though it were run on input $x', 2y$, with T in place of t . By induction it returns $T = 2x'y$ on this input. Hence the algorithm returns $t = 2x'y + y = xy$ on the original input.

Analysis of russmult

- If multiplying $x, y > 0$, problem size is measured by $\max\{b(x), b(y)\}$ or $b(x) + b(y)$ or just $(b(x), b(y))$. We can assume $b(x) \leq b(y)$.

Analysis of russmult

- If multiplying $x, y > 0$, problem size is measured by $\max\{b(x), b(y)\}$ or $b(x) + b(y)$ or just $(b(x), b(y))$. We can assume $b(x) \leq b(y)$.
- Number of additions is number of 1's in binary representation of x , $b(x)$ in worst case. Number of doublings/halvings is $b(x)$.

Analysis of russmult

- If multiplying $x, y > 0$, problem size is measured by $\max\{b(x), b(y)\}$ or $b(x) + b(y)$ or just $(b(x), b(y))$. We can assume $b(x) \leq b(y)$.
- Number of additions is number of 1's in binary representation of x , $b(x)$ in worst case. Number of doublings/halvings is $b(x)$.
- Time is dominated by additions and doubling/halving. If additions have unit cost then this gives a linear-time multiplication algorithm!

Analysis of russmult

- If multiplying $x, y > 0$, problem size is measured by $\max\{b(x), b(y)\}$ or $b(x) + b(y)$ or just $(b(x), b(y))$. We can assume $b(x) \leq b(y)$.
- Number of additions is number of 1's in binary representation of x , $b(x)$ in worst case. Number of doublings/halvings is $b(x)$.
- Time is dominated by additions and doubling/halving. If additions have unit cost then this gives a linear-time multiplication algorithm!
- More realistically, additions take time proportional to size of addends. In the worst case we have to add numbers of size $b(y), b(y) + 1, \dots, b(y) + b(x) - 1$; we end up with a total time of order $b(x)b(y)$, like the primary school algorithm.

Analysis of russmult

- If multiplying $x, y > 0$, problem size is measured by $\max\{b(x), b(y)\}$ or $b(x) + b(y)$ or just $(b(x), b(y))$. We can assume $b(x) \leq b(y)$.
- Number of additions is number of 1's in binary representation of x , $b(x)$ in worst case. Number of doublings/halvings is $b(x)$.
- Time is dominated by additions and doubling/halving. If additions have unit cost then this gives a linear-time multiplication algorithm!
- More realistically, additions take time proportional to size of addends. In the worst case we have to add numbers of size $b(y), b(y) + 1, \dots, b(y) + b(x) - 1$; we end up with a total time of order $b(x)b(y)$, like the primary school algorithm.
- What about average-case running time? It is still of the same order. See Assignment 1.

Review: asymptotic notation for $f : \mathbb{N} \rightarrow \mathbb{R}_+$

- $f \in O(g)$ means there is $C > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq Cg(n)$ for all $n \geq n_0$. “Eventually, f grows at most as fast as g ”.

Note that if always $g > 0$, then in the definition of O , we can remove n_0 at the expense of a bigger C (why?). Thus $f \in O(g)$ if and only if there is $C > 0$ with $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$, in other words f/g is bounded above.

Review: asymptotic notation for $f : \mathbb{N} \rightarrow \mathbb{R}_+$

- $f \in O(g)$ means there is $C > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq Cg(n)$ for all $n \geq n_0$. “Eventually, f grows at most as fast as g ”.
- $f \in \Omega(g)$ means $g(n) \in O(f(n))$:
 $f(n) \geq Cg(n)$ for all $n \geq n_0$. “Eventually, f grows at least as fast as g ”.

Note that if always $g > 0$, then in the definition of O , we can remove n_0 at the expense of a bigger C (why?). Thus $f \in O(g)$ if and only if there is $C > 0$ with $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$, in other words f/g is bounded above.

Review: asymptotic notation for $f : \mathbb{N} \rightarrow \mathbb{R}_+$

- $f \in O(g)$ means there is $C > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq Cg(n)$ for all $n \geq n_0$. “Eventually, f grows at most as fast as g ”.
- $f \in \Omega(g)$ means $g(n) \in O(f(n))$:
 $f(n) \geq Cg(n)$ for all $n \geq n_0$. “Eventually, f grows at least as fast as g ”.
- $f \in \Theta(g)$ means $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$:
 $f(n) \leq C_1g(n) \leq C_2f(n)$ for all $n \geq n_0$. “Eventually, f grows at the same rate as g ”.

Note that if always $g > 0$, then in the definition of O , we can remove n_0 at the expense of a bigger C (why?). Thus $f \in O(g)$ if and only if there is $C > 0$ with $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$, in other words f/g is bounded above.

Useful rules for asymptotics

- Limit rule: if eventually $g > 0$ and $L := \lim f(n)/g(n)$ exists,

When using the limit rule, L'Hôpital's rule from calculus is often useful.

Useful rules for asymptotics

- Limit rule: if eventually $g > 0$ and $L := \lim f(n)/g(n)$ exists,
 - if $L = 0$ then $f \in O(g)$ and $f \notin \Omega(g)$;

When using the limit rule, L'Hôpital's rule from calculus is often useful.

Useful rules for asymptotics

- Limit rule: if eventually $g > 0$ and $L := \lim f(n)/g(n)$ exists,
 - if $L = 0$ then $f \in O(g)$ and $f \notin \Omega(g)$;
 - if $0 < L < \infty$ then $f \in \Theta(g)$;

When using the limit rule, L'Hôpital's rule from calculus is often useful.

Useful rules for asymptotics

- Limit rule: if eventually $g > 0$ and $L := \lim f(n)/g(n)$ exists,
 - if $L = 0$ then $f \in O(g)$ and $f \notin \Omega(g)$;
 - if $0 < L < \infty$ then $f \in \Theta(g)$;
 - if $L = \infty$ then $f \in \Omega(g)$ and $f \notin O(g)$.

When using the limit rule, L'Hôpital's rule from calculus is often useful.

Useful rules for asymptotics

- Limit rule: if eventually $g > 0$ and $L := \lim f(n)/g(n)$ exists,
 - if $L = 0$ then $f \in O(g)$ and $f \notin \Omega(g)$;
 - if $0 < L < \infty$ then $f \in \Theta(g)$;
 - if $L = \infty$ then $f \in \Omega(g)$ and $f \notin O(g)$.
- Sum rule: if $f_1 \in O(g_1)$, $f_2 \in O(g_2)$, then $f_1 + f_2 \in O(\max\{g_1, g_2\})$.

When using the limit rule, L'Hôpital's rule from calculus is often useful.

Useful rules for asymptotics

- Limit rule: if eventually $g > 0$ and $L := \lim f(n)/g(n)$ exists,
 - if $L = 0$ then $f \in O(g)$ and $f \notin \Omega(g)$;
 - if $0 < L < \infty$ then $f \in \Theta(g)$;
 - if $L = \infty$ then $f \in \Omega(g)$ and $f \notin O(g)$.
- Sum rule: if $f_1 \in O(g_1)$, $f_2 \in O(g_2)$, then $f_1 + f_2 \in O(\max\{g_1, g_2\})$.
- Product rule: if $f_1 \in O(g_1)$, $f_2 \in O(g_2)$, then $f_1 f_2 \in O(g_1 g_2)$.

When using the limit rule, L'Hôpital's rule from calculus is often useful.

Useful rules for asymptotics

- Limit rule: if eventually $g > 0$ and $L := \lim f(n)/g(n)$ exists,
 - if $L = 0$ then $f \in O(g)$ and $f \notin \Omega(g)$;
 - if $0 < L < \infty$ then $f \in \Theta(g)$;
 - if $L = \infty$ then $f \in \Omega(g)$ and $f \notin O(g)$.
- Sum rule: if $f_1 \in O(g_1)$, $f_2 \in O(g_2)$, then $f_1 + f_2 \in O(\max\{g_1, g_2\})$.
- Product rule: if $f_1 \in O(g_1)$, $f_2 \in O(g_2)$, then $f_1 f_2 \in O(g_1 g_2)$.
- Transitivity: if $f \in O(g)$, $g \in O(h)$, then $f \in O(h)$.

When using the limit rule, L'Hôpital's rule from calculus is often useful.

Examples of asymptotics

- $n \lg n \in O(n^2)$, by limit rule;

Examples of asymptotics

- $n \lg n \in O(n^2)$, by limit rule;
- $\log_a n \in \Theta(\lg n)$ for each fixed $a > 1$, by change of base formula;

Examples of asymptotics

- $n \lg n \in O(n^2)$, by limit rule;
- $\log_a n \in \Theta(\lg n)$ for each fixed $a > 1$, by change of base formula;
- $2^n \in \Omega(n^{10})$, by limit rule;

Examples of asymptotics

- $n \lg n \in O(n^2)$, by limit rule;
- $\log_a n \in \Theta(\lg n)$ for each fixed $a > 1$, by change of base formula;
- $2^n \in \Omega(n^{10})$, by limit rule;
- $p(n) \in \Theta(n^{\deg p})$ if p is a polynomial, by limit rule.

Examples of asymptotics

- $n \lg n \in O(n^2)$, by limit rule;
- $\log_a n \in \Theta(\lg n)$ for each fixed $a > 1$, by change of base formula;
- $2^n \in \Omega(n^{10})$, by limit rule;
- $p(n) \in \Theta(n^{\deg p})$ if p is a polynomial, by limit rule.
- fix $k \in \mathbb{R}$ and let $S_k(n) = 1^k + 2^k + \dots + n^k$. Then

$$\begin{cases} S_k(n) \in \Theta(n^{k+1}) & \text{for } k \neq -1; \\ S_{-1}(n) \in \Theta(\log n). \end{cases}$$

One proof uses approximation by an integral.

Conditional asymptotics

- Sometimes we have a statement like: $f(n) \leq Cg(n)$ for all sufficiently large n such that n is a power of 2.

Conditional asymptotics

- Sometimes we have a statement like: $f(n) \leq Cg(n)$ for all sufficiently large n such that n is a power of 2.
- Without further assumptions we cannot say that $f \in O(g)$ unconditionally.

Conditional asymptotics

- Sometimes we have a statement like: $f(n) \leq Cg(n)$ for all sufficiently large n such that n is a power of 2.
- Without further assumptions we cannot say that $f \in O(g)$ unconditionally.
- However, the **smoothness rule** does allow us to conclude that $f \in O(g)$. The hypotheses are:

Conditional asymptotics

- Sometimes we have a statement like: $f(n) \leq Cg(n)$ for all sufficiently large n such that n is a power of 2.
- Without further assumptions we cannot say that $f \in O(g)$ unconditionally.
- However, the **smoothness rule** does allow us to conclude that $f \in O(g)$. The hypotheses are:
 - g is eventually increasing: $g(n+1) \geq g(n)$ for all sufficiently large n ;

Conditional asymptotics

- Sometimes we have a statement like: $f(n) \leq Cg(n)$ for all sufficiently large n such that n is a power of 2.
- Without further assumptions we cannot say that $f \in O(g)$ unconditionally.
- However, the **smoothness rule** does allow us to conclude that $f \in O(g)$. The hypotheses are:
 - g is eventually increasing: $g(n+1) \geq g(n)$ for all sufficiently large n ;
 - g has **subexponential growth**: there is $C > 0$ such that $g(2n) \leq Cg(n)$ for all sufficiently large n ;

Conditional asymptotics

- Sometimes we have a statement like: $f(n) \leq Cg(n)$ for all sufficiently large n such that n is a power of 2.
- Without further assumptions we cannot say that $f \in O(g)$ unconditionally.
- However, the **smoothness rule** does allow us to conclude that $f \in O(g)$. The hypotheses are:
 - g is eventually increasing: $g(n+1) \geq g(n)$ for all sufficiently large n ;
 - g has **subexponential growth**: there is $C > 0$ such that $g(2n) \leq Cg(n)$ for all sufficiently large n ;
 - f is eventually increasing.

Conditional asymptotics

- Sometimes we have a statement like: $f(n) \leq Cg(n)$ for all sufficiently large n such that n is a power of 2.
- Without further assumptions we cannot say that $f \in O(g)$ unconditionally.
- However, the **smoothness rule** does allow us to conclude that $f \in O(g)$. The hypotheses are:
 - g is eventually increasing: $g(n+1) \geq g(n)$ for all sufficiently large n ;
 - g has **subexponential growth**: there is $C > 0$ such that $g(2n) \leq Cg(n)$ for all sufficiently large n ;
 - f is eventually increasing.
- Similarly we can transfer a conditional $f \in \Omega(g)$ or $f \in \Theta(g)$ into the unconditional version.

Proofs by induction

- “Induction is just recursion for proofs”.

Proofs by induction

- “Induction is just recursion for proofs” .
- The technique of proof by induction is equivalent to the technique of the “minimal criminal” .

Proofs by induction

- “Induction is just recursion for proofs” .
- The technique of proof by induction is equivalent to the technique of the “minimal criminal” .
 - Suppose we have propositions $P(n)$ such that $P(0)$ is true and whenever $P(k)$ is true for all $k < n$ then $P(n)$ is necessarily true. We want to show that $P(n)$ is true for all n .

Proofs by induction

- “Induction is just recursion for proofs” .
- The technique of proof by induction is equivalent to the technique of the “minimal criminal” .
 - Suppose we have propositions $P(n)$ such that $P(0)$ is true and whenever $P(k)$ is true for all $k < n$ then $P(n)$ is necessarily true. We want to show that $P(n)$ is true for all n .
 - Suppose on the contrary that it is false for some n . Then there must be a least such n , call it n_0 (the minimal criminal), for which it is false.

Proofs by induction

- “Induction is just recursion for proofs” .
- The technique of proof by induction is equivalent to the technique of the “minimal criminal” .
 - Suppose we have propositions $P(n)$ such that $P(0)$ is true and whenever $P(k)$ is true for all $k < n$ then $P(n)$ is necessarily true. We want to show that $P(n)$ is true for all n .
 - Suppose on the contrary that it is false for some n . Then there must be a least such n , call it n_0 (the minimal criminal), for which it is false.
 - Now $n > 0$ by assumption and by minimality $P(k)$ is true for all $k < n$. Thus $P(n)$ is true, contradicting our assumption.

Proofs by induction

- “Induction is just recursion for proofs”.
- The technique of proof by induction is equivalent to the technique of the “minimal criminal”.
 - Suppose we have propositions $P(n)$ such that $P(0)$ is true and whenever $P(k)$ is true for all $k < n$ then $P(n)$ is necessarily true. We want to show that $P(n)$ is true for all n .
 - Suppose on the contrary that it is false for some n . Then there must be a least such n , call it n_0 (the minimal criminal), for which it is false.
 - Now $n > 0$ by assumption and by minimality $P(k)$ is true for all $k < n$. Thus $P(n)$ is true, contradicting our assumption.
- This can be used to show correctness of many algorithms.

Correctness proofs using induction

- Quicksort, mergesort are correct provided the partitioning/merge steps are correct (they work for size 1 input and reduce the problem to smaller instances, so there can't be a minimal criminal).

Correctness proofs using induction

- Quicksort, mergesort are correct provided the partitioning/merge steps are correct (they work for size 1 input and reduce the problem to smaller instances, so there can't be a minimal criminal).
- Any recursive algorithm can be treated in the same way.

Correctness proofs using induction

- Quicksort, mergesort are correct provided the partitioning/merge steps are correct (they work for size 1 input and reduce the problem to smaller instances, so there can't be a minimal criminal).
- Any recursive algorithm can be treated in the same way.
- Insertion sort reduces the number of inversions (pairs that are out of order), and works when there are no inversions, so we can use induction on the number of inversions.

Correctness proofs using induction

- Quicksort, mergesort are correct provided the partitioning/merge steps are correct (they work for size 1 input and reduce the problem to smaller instances, so there can't be a minimal criminal).
- Any recursive algorithm can be treated in the same way.
- Insertion sort reduces the number of inversions (pairs that are out of order), and works when there are no inversions, so we can use induction on the number of inversions.
- Any iterative algorithm that systematically reduces some nonnegative measure of badness can be treated in the same way.

The technique of constructive induction

- You are familiar with the use of induction to prove inequalities, such as “ $2^n < n!$ for $n \geq 4$ ”. However for asymptotics we more usually have something like “show that the Fibonacci numbers grow exponentially and determine their growth rate”.

The technique of constructive induction

- You are familiar with the use of induction to prove inequalities, such as “ $2^n < n!$ for $n \geq 4$ ”. However for asymptotics we more usually have something like “show that the Fibonacci numbers grow exponentially and determine their growth rate”.
- So we try to prove: “for some $C > 0$, some $n_0 \geq 0$ and some $\phi > 1$, we have $F(n) \geq C\phi^n$ for all $n \geq n_0$ ”.

The technique of constructive induction

- You are familiar with the use of induction to prove inequalities, such as “ $2^n < n!$ for $n \geq 4$ ”. However for asymptotics we more usually have something like “show that the Fibonacci numbers grow exponentially and determine their growth rate”.
- So we try to prove: “for some $C > 0$, some $n_0 \geq 0$ and some $\phi > 1$, we have $F(n) \geq C\phi^n$ for all $n \geq n_0$ ”.
- The induction step will have to use the recurrence defining $F(n)$. The inductive hypothesis will give $F(n) = F(n-1) + F(n-2) \geq C\phi^{n-1} + C\phi^{n-2}$. Notice that as long as $\phi^{n-1} + \phi^{n-2} \geq \phi^n$ the inductive step will follow.

The technique of constructive induction

- You are familiar with the use of induction to prove inequalities, such as “ $2^n < n!$ for $n \geq 4$ ”. However for asymptotics we more usually have something like “show that the Fibonacci numbers grow exponentially and determine their growth rate”.
- So we try to prove: “for some $C > 0$, some $n_0 \geq 0$ and some $\phi > 1$, we have $F(n) \geq C\phi^n$ for all $n \geq n_0$ ”.
- The induction step will have to use the recurrence defining $F(n)$. The inductive hypothesis will give $F(n) = F(n-1) + F(n-2) \geq C\phi^{n-1} + C\phi^{n-2}$. Notice that as long as $\phi^{n-1} + \phi^{n-2} \geq \phi^n$ the inductive step will follow.
- The largest ϕ for which this works is the root of $\phi^2 - \phi - 1 = 0$, namely $(1 + \sqrt{5})/2$.

The technique of constructive induction

- You are familiar with the use of induction to prove inequalities, such as “ $2^n < n!$ for $n \geq 4$ ”. However for asymptotics we more usually have something like “show that the Fibonacci numbers grow exponentially and determine their growth rate”.
- So we try to prove: “for some $C > 0$, some $n_0 \geq 0$ and some $\phi > 1$, we have $F(n) \geq C\phi^n$ for all $n \geq n_0$ ”.
- The induction step will have to use the recurrence defining $F(n)$. The inductive hypothesis will give $F(n) = F(n-1) + F(n-2) \geq C\phi^{n-1} + C\phi^{n-2}$. Notice that as long as $\phi^{n-1} + \phi^{n-2} \geq \phi^n$ the inductive step will follow.
- The largest ϕ for which this works is the root of $\phi^2 - \phi - 1 = 0$, namely $(1 + \sqrt{5})/2$.
- We can now prove the result using any C that works for two consecutive base values. Can you find a good C and n_0 ?

Definitions

- A **digraph** is a finite set V of **nodes** together with a binary relation E on V . The element (v, w) of E is the **arc** from v to w .

Definitions

- A **digraph** is a finite set V of **nodes** together with a binary relation E on V . The element (v, w) of E is the **arc** from v to w .
- A **graph** has the same definition as a digraph, except that the arc relation is required to be symmetric. In this case we can represent the **edge** between u and v as an unordered pair $\{v, w\}$.

Definitions

- A **digraph** is a finite set V of **nodes** together with a binary relation E on V . The element (v, w) of E is the **arc** from v to w .
- A **graph** has the same definition as a digraph, except that the arc relation is required to be symmetric. In this case we can represent the **edge** between u and v as an unordered pair $\{v, w\}$.
- A (rooted, ordered) **tree** is a special type of graph (or digraph) that can be defined directly in a recursive way.

Definitions

- A **digraph** is a finite set V of **nodes** together with a binary relation E on V . The element (v, w) of E is the **arc** from v to w .
- A **graph** has the same definition as a digraph, except that the arc relation is required to be symmetric. In this case we can represent the **edge** between u and v as an unordered pair $\{v, w\}$.
- A (rooted, ordered) **tree** is a special type of graph (or digraph) that can be defined directly in a recursive way.
- (Di)graphs have enormously many applications in studying transportation/communication/social networks, as well as scheduling and other dependency relations. Trees arise in the analysis of recursive algorithms as well as explicit data structures.

Standard graph ADT operations

- Create empty (di)graph.

Standard graph ADT operations

- Create empty (di)graph.
- Return the **order** (number of nodes) and **size** (number of arcs).

Standard graph ADT operations

- Create empty (di)graph.
- Return the **order** (number of nodes) and **size** (number of arcs).
- Is there an arc between u and v ?

Standard graph ADT operations

- Create empty (di)graph.
- Return the **order** (number of nodes) and **size** (number of arcs).
- Is there an arc between u and v ?
- Add an arc between u and v .

Standard graph ADT operations

- Create empty (di)graph.
- Return the **order** (number of nodes) and **size** (number of arcs).
- Is there an arc between u and v ?
- Add an arc between u and v .
- Delete arc between u and v .

Standard graph ADT operations

- Create empty (di)graph.
- Return the **order** (number of nodes) and **size** (number of arcs).
- Is there an arc between u and v ?
- Add an arc between u and v .
- Delete arc between u and v .
- Add a node.

Standard graph ADT operations

- Create empty (di)graph.
- Return the **order** (number of nodes) and **size** (number of arcs).
- Is there an arc between u and v ?
- Add an arc between u and v .
- Delete arc between u and v .
- Add a node.
- Delete node u (and all arcs involving it).

Standard implementations

- Two main implementations for general digraphs: adjacency matrix and adjacency lists representation.

Standard implementations

- Two main implementations for general digraphs: adjacency matrix and adjacency lists representation.
- A graph can be represented as a symmetric digraph or using unordered pairs of nodes.

Standard implementations

- Two main implementations for general digraphs: adjacency matrix and adjacency lists representation.
- A graph can be represented as a symmetric digraph or using unordered pairs of nodes.
- More specialized classes of (di)graphs can have more efficient representations. Trees are an example.

Standard implementations

- Two main implementations for general digraphs: adjacency matrix and adjacency lists representation.
- A graph can be represented as a symmetric digraph or using unordered pairs of nodes.
- More specialized classes of (di)graphs can have more efficient representations. Trees are an example.
- We also need to deal with (arc)-**weighted** objects where a real number is associated to each arc. Simple modifications of the matrix and lists representations can be made.

Java graph implementation

- In CS220 we saw a particular implementation in Java.

Java graph implementation

- In CS220 we saw a particular implementation in Java.
- The files are available from the course website and a description is in the CS220 textbook.

Java graph implementation

- In CS220 we saw a particular implementation in Java.
- The files are available from the course website and a description is in the CS220 textbook.
- Please review them immediately and ask questions before the week 3 lab.

Review of important algorithms I

- Tree traversals: preorder, postorder, inorder (for binary tree).

Review of important algorithms I

- Tree traversals: preorder, postorder, inorder (for binary tree).
- Graph traversals: visit each node exactly once and explore each arc exactly once. At each step scan the neighbours of a frontier node and visit a new one.

Review of important algorithms I

- Tree traversals: preorder, postorder, inorder (for binary tree).
- Graph traversals: visit each node exactly once and explore each arc exactly once. At each step scan the neighbours of a frontier node and visit a new one.
 - Depth-first search: uses a stack to determine which node to use next as the frontier node. Can be implemented recursively.

Review of important algorithms I

- Tree traversals: preorder, postorder, inorder (for binary tree).
- Graph traversals: visit each node exactly once and explore each arc exactly once. At each step scan the neighbours of a frontier node and visit a new one.
 - Depth-first search: uses a stack to determine which node to use next as the frontier node. Can be implemented recursively.
 - Breadth-first search: uses a queue to determine which node to use next as the frontier node.

Review of important algorithms I

- Tree traversals: preorder, postorder, inorder (for binary tree).
- Graph traversals: visit each node exactly once and explore each arc exactly once. At each step scan the neighbours of a frontier node and visit a new one.
 - Depth-first search: uses a stack to determine which node to use next as the frontier node. Can be implemented recursively.
 - Breadth-first search: uses a queue to determine which node to use next as the frontier node.
 - Priority-first search: uses a priority queue to determine which node to use next as the frontier node. Includes depth-first and breadth-first as special cases.

Review of important algorithms I

- Tree traversals: preorder, postorder, inorder (for binary tree).
- Graph traversals: visit each node exactly once and explore each arc exactly once. At each step scan the neighbours of a frontier node and visit a new one.
 - Depth-first search: uses a stack to determine which node to use next as the frontier node. Can be implemented recursively.
 - Breadth-first search: uses a queue to determine which node to use next as the frontier node.
 - Priority-first search: uses a priority queue to determine which node to use next as the frontier node. Includes depth-first and breadth-first as special cases.
- Applications of graph traversal: (strongly) connected components, shortest cycle, 2-colouring, topological ordering. Can all be done in **linear time** ($\Theta(n + e)$ where n is order, e is size).

Review of important algorithms II

- Weighted (d)graphs have a number attached to each arc. Common question: find something (path, cycle, spanning tree) of minimum total weight.

Review of important algorithms II

- Weighted (d)graphs have a number attached to each arc. Common question: find something (path, cycle, spanning tree) of minimum total weight.
- Distance from source: Dijkstra (fails when negative weights allowed), Bellman-Ford.

Review of important algorithms II

- Weighted (d)graphs have a number attached to each arc. Common question: find something (path, cycle, spanning tree) of minimum total weight.
- Distance from source: Dijkstra (fails when negative weights allowed), Bellman-Ford.
- Pairwise distances: Floyd.

Review of important algorithms II

- Weighted (d)graphs have a number attached to each arc. Common question: find something (path, cycle, spanning tree) of minimum total weight.
- Distance from source: Dijkstra (fails when negative weights allowed), Bellman-Ford.
- Pairwise distances: Floyd.
- Spanning tree: Prim, Kruskal.

Review of important algorithms II

- Weighted (d)graphs have a number attached to each arc. Common question: find something (path, cycle, spanning tree) of minimum total weight.
- Distance from source: Dijkstra (fails when negative weights allowed), Bellman-Ford.
- Pairwise distances: Floyd.
- Spanning tree: Prim, Kruskal.
- The performance depends considerably on the data structures used. Prim and Dijkstra are based on priority-first traversal, and implementation of the priority queue is crucial.

Review of important algorithms II

- Weighted (d)graphs have a number attached to each arc. Common question: find something (path, cycle, spanning tree) of minimum total weight.
- Distance from source: Dijkstra (fails when negative weights allowed), Bellman-Ford.
- Pairwise distances: Floyd.
- Spanning tree: Prim, Kruskal.
- The performance depends considerably on the data structures used. Prim and Dijkstra are based on priority-first traversal, and implementation of the priority queue is crucial.
- Note for later: Prim, Kruskal, Dijkstra are **greedy** algorithms and Bellman-Ford and Floyd are based on **dynamic programming**.

Priority queues

- Recall: a **priority queue** is a container ADT where every element has a data field called a **key**. The basic operations are `insert`, `getmin`, `deletemin`.

Priority queues

- Recall: a **priority queue** is a container ADT where every element has a data field called a **key**. The basic operations are `insert`, `getmin`, `deletemin`.
- Stack and queue can be considered as special cases (but the usual implementations are more efficient).

Priority queues

- Recall: a **priority queue** is a container ADT where every element has a data field called a **key**. The basic operations are `insert`, `getmin`, `deletemin`.
- Stack and queue can be considered as special cases (but the usual implementations are more efficient).
- Many uses: sorting (heapsort), optimization algorithms (Dijkstra, Prim, branch-and-bound).

Priority queues

- Recall: a **priority queue** is a container ADT where every element has a data field called a **key**. The basic operations are `insert`, `getmin`, `deletemin`.
- Stack and queue can be considered as special cases (but the usual implementations are more efficient).
- Many uses: sorting (heapsort), optimization algorithms (Dijkstra, Prim, branch-and-bound).
- You have seen implementation by means of a binary (min-)heap (a type of tree nicely representable in an array). All basic operations take time in $O(\log n)$.

Priority queues

- Recall: a **priority queue** is a container ADT where every element has a data field called a **key**. The basic operations are `insert`, `getmin`, `deletemin`.
- Stack and queue can be considered as special cases (but the usual implementations are more efficient).
- Many uses: sorting (heapsort), optimization algorithms (Dijkstra, Prim, branch-and-bound).
- You have seen implementation by means of a binary (min-)heap (a type of tree nicely representable in an array). All basic operations take time in $O(\log n)$.
- Some applications (such as Dijkstra/Prim) require an additional `decreasekey` operation which dominates the computation, so it is important that this operation be efficient. In a binary heap, decreasing a key value is not very efficient (how would you do it?)

Priority-first traversal

```
algorithm PFSv(node  $s$ , string alg)
 $colour[s] \leftarrow GREY$ ;  $seen[s] \leftarrow time$ 
 $Q \leftarrow pqcreate()$ 
insert( $Q, s, key(s)$ )
while not isempty( $Q$ ) do
     $u \leftarrow getmin(Q)$ 
    for  $v$  adjacent to  $u$  do
        if  $colour[v] = WHITE$  then
             $time \leftarrow time + 1$ ;  $seen[v] \leftarrow time$ 
             $colour[v] \leftarrow GREY$ 
            insert( $Q, v, key(v)$ )
         $colour[u] \leftarrow BLACK$ ;  $done[u] \leftarrow time$ 
    deletemin( $Q$ )
 $time \leftarrow time + 1$ 
end
```

Dijkstra/Prim using priority queue

```
algorithm DijkPrim (weighted digraph  $(G, c)$ , node  $v$ , string alg)
 $Q \leftarrow \text{makepq}()$ 
for  $u \in V(G)$  do
    insert( $Q, u, \infty$ ); changekey( $Q, v, 0$ )
while not isempty( $Q$ ) do
     $u \leftarrow \text{getmin}(Q)$ ;  $k \leftarrow \text{getminkey}(Q)$ ; deletemin( $Q$ )
    for  $x \in Q$  do
         $l \leftarrow \text{getkey}(Q, x)$ 
        if (alg = DIJK)  $n \leftarrow \min\{l, k + c[u, x]\}$ 
        if (alg = PRIM)  $n \leftarrow \min\{l, c[u, x]\}$ 
        changekey( $Q, x, n$ )
```

If implemented properly, getkey/changekey are done e times;
getmin/getminkey/deletemin n times.

Priority queue implementations

- In Dijkstra/Prim, decreasekey can dominate the computation. A binary (min)-heap supports getmin/getminkey in $O(1)$ time and other operations in $O(\log n)$. Much research has been done into improving this.

Priority queue implementations

- In Dijkstra/Prim, decreasekey can dominate the computation. A binary (min)-heap supports getmin/getminkey in $O(1)$ time and other operations in $O(\log n)$. Much research has been done into improving this.
- It is possible to do decreasekey in $O(1)$ time without ruining other operations, provided we consider **amortized time complexity**. The first construction is called **Fibonacci heap**. A more recent one (T. Takaoka, University of Canterbury) is called **2-3 heap**. See me for references.

Priority queue implementations

- In Dijkstra/Prim, decreasekey can dominate the computation. A binary (min)-heap supports getmin/getminkey in $O(1)$ time and other operations in $O(\log n)$. Much research has been done into improving this.
- It is possible to do decreasekey in $O(1)$ time without ruining other operations, provided we consider **amortized time complexity**. The first construction is called **Fibonacci heap**. A more recent one (T. Takaoka, University of Canterbury) is called **2-3 heap**. See me for references.
- What is amortized time complexity? Roughly, the idea is to average over many consecutive operations of the same operation. Each time the operation changes the data structure and this affects the running time of the next time. The worst case may be bad but occur infrequently.

Amortized time complexity

- Suppose we have a function ϕ_i that measures the “messiness” of the data structure after the i th call on a given operation. Define the **amortized time** taken by the i th call as $\hat{t}_i = t_i + (\phi_i - \phi_{i-1})$, where t_i is time for i th call.

Amortized time complexity

- Suppose we have a function ϕ_i that measures the “messiness” of the data structure after the i th call on a given operation. Define the **amortized time** taken by the i th call as $\hat{t}_i = t_i + (\phi_i - \phi_{i-1})$, where t_i is time for i th call.
- Note that $\sum_{i=1}^n \hat{t}_i = \sum_{i=1}^n t_i + \phi_n - \phi_0$ by telescoping. Provided ϕ_i is never less than ϕ_0 (we don’t “over-clean”), total time is bounded above by total amortized time. If we choose ϕ cleverly, total amortized time can be easy to compute.

Amortized time complexity

- Suppose we have a function ϕ_i that measures the “messiness” of the data structure after the i th call on a given operation. Define the **amortized time** taken by the i th call as $\hat{t}_i = t_i + (\phi_i - \phi_{i-1})$, where t_i is time for i th call.
- Note that $\sum_{i=1}^n \hat{t}_i = \sum_{i=1}^n t_i + \phi_n - \phi_0$ by telescoping. Provided ϕ_i is never less than ϕ_0 (we don’t “over-clean”), total time is bounded above by total amortized time. If we choose ϕ cleverly, total amortized time can be easy to compute.
- Example: binary counter (BB p 115, detailed in class). Count from 0 to n . Let ϕ_i be the number of bits equal to 1 after i calls. Total amortized time is at most $2n$. However, worst case for single operation is $\lg n$.

Amortized time complexity

- Suppose we have a function ϕ_i that measures the “messiness” of the data structure after the i th call on a given operation. Define the **amortized time** taken by the i th call as $\hat{t}_i = t_i + (\phi_i - \phi_{i-1})$, where t_i is time for i th call.
- Note that $\sum_{i=1}^n \hat{t}_i = \sum_{i=1}^n t_i + \phi_n - \phi_0$ by telescoping. Provided ϕ_i is never less than ϕ_0 (we don’t “over-clean”), total time is bounded above by total amortized time. If we choose ϕ cleverly, total amortized time can be easy to compute.
- Example: binary counter (BB p 115, detailed in class). Count from 0 to n . Let ϕ_i be the number of bits equal to 1 after i calls. Total amortized time is at most $2n$. However, worst case for single operation is $\lg n$.
- Example: stack implemented as array with doubling. Detailed in class.

An ADT for set partitions

- Recall that a partition of a set S is a collection of disjoint subsets of S whose union is S . There is a 1-1 correspondence between partitions of S and equivalence relations on S .

An ADT for set partitions

- Recall that a partition of a set S is a collection of disjoint subsets of S whose union is S . There is a 1-1 correspondence between partitions of S and equivalence relations on S .
- There are 3 standard operations on the ADT that represents a partition of S .

We assume that the first operations performed after creating an empty object will create n singleton subsets using `MakeSet`.

An ADT for set partitions

- Recall that a partition of a set S is a collection of disjoint subsets of S whose union is S . There is a 1-1 correspondence between partitions of S and equivalence relations on S .
- There are 3 standard operations on the ADT that represents a partition of S .
 - `MakeUnionFind(S)` creates a partition in which each element x of S is in its own singleton set $\{x\}$.

We assume that the first operations performed after creating an empty object will create n singleton subsets using `MakeSet`.

An ADT for set partitions

- Recall that a partition of a set S is a collection of disjoint subsets of S whose union is S . There is a 1-1 correspondence between partitions of S and equivalence relations on S .
- There are 3 standard operations on the ADT that represents a partition of S .
 - `MakeUnionFind(S)` creates a partition in which each element x of S is in its own singleton set $\{x\}$.
 - `Union(x, y)` merges the subset containing x with the one containing y ;

We assume that the first operations performed after creating an empty object will create n singleton subsets using `MakeSet`.

An ADT for set partitions

- Recall that a partition of a set S is a collection of disjoint subsets of S whose union is S . There is a 1-1 correspondence between partitions of S and equivalence relations on S .
- There are 3 standard operations on the ADT that represents a partition of S .
 - `MakeUnionFind(S)` creates a partition in which each element x of S is in its own singleton set $\{x\}$.
 - `Union(x, y)` merges the subset containing x with the one containing y ;
 - `Find(x)` returns the unique subset containing x (we will label this set by a fixed representative element).

We assume that the first operations performed after creating an empty object will create n singleton subsets using `MakeSet`.

An ADT for set partitions

- Recall that a partition of a set S is a collection of disjoint subsets of S whose union is S . There is a 1-1 correspondence between partitions of S and equivalence relations on S .
- There are 3 standard operations on the ADT that represents a partition of S .
 - `MakeUnionFind(S)` creates a partition in which each element x of S is in its own singleton set $\{x\}$.
 - `Union(x, y)` merges the subset containing x with the one containing y ;
 - `Find(x)` returns the unique subset containing x (we will label this set by a fixed representative element).

We assume that the first operations performed after creating an empty object will create n singleton subsets using `MakeSet`.

- This ADT has applications whenever we have dynamically changing equivalence relation on a set. We will see examples soon.

Array implementation

- Maintain an array A indexed by elements of S , where $A[x]$ gives the representative of the set containing x . Thus **Find** takes constant time.

Array implementation

- Maintain an array A indexed by elements of S , where $A[x]$ gives the representative of the set containing x . Thus **Find** takes constant time.
- The union operation takes time in $\Theta(n)$ in worst case because we must update the values of $A[x]$ for all x in the two sets.

Array implementation

- Maintain an array A indexed by elements of S , where $A[x]$ gives the representative of the set containing x . Thus Find takes constant time.
- The union operation takes time in $\Theta(n)$ in worst case because we must update the values of $A[x]$ for all x in the two sets.
- There are some obvious speed improvements to be made (at the cost of more space):

Array implementation

- Maintain an array A indexed by elements of S , where $A[x]$ gives the representative of the set containing x . Thus Find takes constant time.
- The union operation takes time in $\Theta(n)$ in worst case because we must update the values of $A[x]$ for all x in the two sets.
- There are some obvious speed improvements to be made (at the cost of more space):
 - (Weighted union heuristic) Give the union the name of the larger of the two sets being merged, so less updating is required.

Array implementation

- Maintain an array A indexed by elements of S , where $A[x]$ gives the representative of the set containing x . Thus Find takes constant time.
- The union operation takes time in $\Theta(n)$ in worst case because we must update the values of $A[x]$ for all x in the two sets.
- There are some obvious speed improvements to be made (at the cost of more space):
 - (Weighted union heuristic) Give the union the name of the larger of the two sets being merged, so less updating is required.
 - Of course this requires us to keep track of the size, and we can use another array for that.

Array implementation

- Maintain an array A indexed by elements of S , where $A[x]$ gives the representative of the set containing x . Thus Find takes constant time.
- The union operation takes time in $\Theta(n)$ in worst case because we must update the values of $A[x]$ for all x in the two sets.
- There are some obvious speed improvements to be made (at the cost of more space):
 - (Weighted union heuristic) Give the union the name of the larger of the two sets being merged, so less updating is required.
 - Of course this requires us to keep track of the size, and we can use another array for that.
 - Maintain the list of elements in each set (how??) so we don't have to scan the whole array to find out what needs updating.

Array implementation

- Maintain an array A indexed by elements of S , where $A[x]$ gives the representative of the set containing x . Thus Find takes constant time.
- The union operation takes time in $\Theta(n)$ in worst case because we must update the values of $A[x]$ for all x in the two sets.
- There are some obvious speed improvements to be made (at the cost of more space):
 - (Weighted union heuristic) Give the union the name of the larger of the two sets being merged, so less updating is required.
 - Of course this requires us to keep track of the size, and we can use another array for that.
 - Maintain the list of elements in each set (how??) so we don't have to scan the whole array to find out what needs updating.
- Could also use a linked list.

Forest implementation

- Maintain each subset in a tree, with one set element per node, and choose the root as the representative.

Forest implementation

- Maintain each subset in a tree, with one set element per node, and choose the root as the representative.
- Use the predecessor representation so that we only store the parent of each node. Declare the parent of a root to be itself.

Forest implementation

- Maintain each subset in a tree, with one set element per node, and choose the root as the representative.
- Use the predecessor representation so that we only store the parent of each node. Declare the parent of a root to be itself.
- The `Find` operation takes time proportional to the depth of the node.

Forest implementation

- Maintain each subset in a tree, with one set element per node, and choose the root as the representative.
- Use the predecessor representation so that we only store the parent of each node. Declare the parent of a root to be itself.
- The `Find` operation takes time proportional to the depth of the node.
- The `Union` operation takes constant time: make the root of one tree point to the root of the other.

Forest implementation

- Maintain each subset in a tree, with one set element per node, and choose the root as the representative.
- Use the predecessor representation so that we only store the parent of each node. Declare the parent of a root to be itself.
- The `Find` operation takes time proportional to the depth of the node.
- The `Union` operation takes constant time: make the root of one tree point to the root of the other.
- We can show that the amortized time for `Find` is $O(\log n)$.
Can we do better?

Forest implementation

- Maintain each subset in a tree, with one set element per node, and choose the root as the representative.
- Use the predecessor representation so that we only store the parent of each node. Declare the parent of a root to be itself.
- The `Find` operation takes time proportional to the depth of the node.
- The `Union` operation takes constant time: make the root of one tree point to the root of the other.
- We can show that the amortized time for `Find` is $O(\log n)$. Can we do better?
- (Path compression heuristic) after a `Find` operation, go back along the path and reset all the pointers on that path to point directly to the root.

Analysis of Find

- If we use weighted union, then every time the set containing x has to change its name, its size must have at least doubled.

Analysis of Find

- If we use weighted union, then every time the set containing x has to change its name, its size must have at least doubled.
- Since it starts at size 1 and does not exceed $n = |S|$, it can change name at most $\lg n$ times. Thus Find runs in $\Theta(\lg n)$ time in the worst case.

Analysis of Find

- If we use weighted union, then every time the set containing x has to change its name, its size must have at least doubled.
- Since it starts at size 1 and does not exceed $n = |S|$, it can change name at most $\lg n$ times. Thus Find runs in $\Theta(\lg n)$ time in the worst case.
- If we also use path compression, Find takes more work, but never by more than a constant factor. The cleaning up done by the compression more than compensates.

Analysis of Find

- If we use weighted union, then every time the set containing x has to change its name, its size must have at least doubled.
- Since it starts at size 1 and does not exceed $n = |S|$, it can change name at most $\lg n$ times. Thus Find runs in $\Theta(\log n)$ time in the worst case.
- If we also use path compression, Find takes more work, but never by more than a constant factor. The cleaning up done by the compression more than compensates.
- It can be shown that the amortized time for Find is $O(\alpha(n))$ where α is the **inverse Ackermann function**. This function grows so slowly that it can't take a value more than 4 on any practical problem. See book by Cormen *et al.* for details.

Design paradigm 1: greedy algorithms

- Build up a solution step-by-step, making the locally best choice, with no regrets. Usually solution is a vector. We select an element from a set of **candidates**, using a **selection criterion**, and add to our **partial solution**. A candidate not chosen may also be **rejected**. Once a candidate has been rejected, it is never considered again. On termination, we have a **solution**.

Design paradigm 1: greedy algorithms

- Build up a solution step-by-step, making the locally best choice, with no regrets. Usually solution is a vector. We select an element from a set of **candidates**, using a **selection criterion**, and add to our **partial solution**. A candidate not chosen may also be **rejected**. Once a candidate has been rejected, it is never considered again. On termination, we have a **solution**.
- Usually we want an **optimal** solution. Greedy algorithms are usually fast and sometimes give optimal solutions. Proving that they do is usually hard.

Design paradigm 1: greedy algorithms

- Build up a solution step-by-step, making the locally best choice, with no regrets. Usually solution is a vector. We select an element from a set of **candidates**, using a **selection criterion**, and add to our **partial solution**. A candidate not chosen may also be **rejected**. Once a candidate has been rejected, it is never considered again. On termination, we have a **solution**.
- Usually we want an **optimal** solution. Greedy algorithms are usually fast and sometimes give optimal solutions. Proving that they do is usually hard.
- A partial solution that can be extended to a solution is called **feasible** (often easy to check), and one that extends to an optimal solution is **promising** (usually hard).

Design paradigm 1: greedy algorithms

- Build up a solution step-by-step, making the locally best choice, with no regrets. Usually solution is a vector. We select an element from a set of **candidates**, using a **selection criterion**, and add to our **partial solution**. A candidate not chosen may also be **rejected**. Once a candidate has been rejected, it is never considered again. On termination, we have a **solution**.
- Usually we want an **optimal** solution. Greedy algorithms are usually fast and sometimes give optimal solutions. Proving that they do is usually hard.
- A partial solution that can be extended to a solution is called **feasible** (often easy to check), and one that extends to an optimal solution is **promising** (usually hard).
- Examples you (may) have seen: Dijkstra, Prim/Kruskal. Other examples: scheduling, change-making,

A scheduling problem

- We have n jobs to schedule on a single resource. The i th job has a requested start time $s(i)$ and finish time $f(i)$.

A scheduling problem

- We have n jobs to schedule on a single resource. The i th job has a requested start time $s(i)$ and finish time $f(i)$.
- There may be conflicts that make it impossible to fulfil all requests. We want to maximize the number of jobs scheduled.

A scheduling problem

- We have n jobs to schedule on a single resource. The i th job has a requested start time $s(i)$ and finish time $f(i)$.
- There may be conflicts that make it impossible to fulfil all requests. We want to maximize the number of jobs scheduled.
- Some greedy ideas:

A scheduling problem

- We have n jobs to schedule on a single resource. The i th job has a requested start time $s(i)$ and finish time $f(i)$.
- There may be conflicts that make it impossible to fulfil all requests. We want to maximize the number of jobs scheduled.
- Some greedy ideas:
 - Accept in increasing order of s (“earliest start time”);

A scheduling problem

- We have n jobs to schedule on a single resource. The i th job has a requested start time $s(i)$ and finish time $f(i)$.
- There may be conflicts that make it impossible to fulfil all requests. We want to maximize the number of jobs scheduled.
- Some greedy ideas:
 - Accept in increasing order of s (“earliest start time”);
 - Accept in increasing order of $f - s$ (“shortest job time”);

A scheduling problem

- We have n jobs to schedule on a single resource. The i th job has a requested start time $s(i)$ and finish time $f(i)$.
- There may be conflicts that make it impossible to fulfil all requests. We want to maximize the number of jobs scheduled.
- Some greedy ideas:
 - Accept in increasing order of s (“earliest start time”);
 - Accept in increasing order of $f - s$ (“shortest job time”);
 - Accept in increasing order of number of conflicts (“fewest conflicts”);

A scheduling problem

- We have n jobs to schedule on a single resource. The i th job has a requested start time $s(i)$ and finish time $f(i)$.
- There may be conflicts that make it impossible to fulfil all requests. We want to maximize the number of jobs scheduled.
- Some greedy ideas:
 - Accept in increasing order of s (“earliest start time”);
 - Accept in increasing order of $f - s$ (“shortest job time”);
 - Accept in increasing order of number of conflicts (“fewest conflicts”);
 - Accept in increasing order of f (“earliest finish time”).

A scheduling problem

- We have n jobs to schedule on a single resource. The i th job has a requested start time $s(i)$ and finish time $f(i)$.
- There may be conflicts that make it impossible to fulfil all requests. We want to maximize the number of jobs scheduled.
- Some greedy ideas:
 - Accept in increasing order of s (“earliest start time”);
 - Accept in increasing order of $f - s$ (“shortest job time”);
 - Accept in increasing order of number of conflicts (“fewest conflicts”);
 - Accept in increasing order of f (“earliest finish time”).
- Only the last one always gives an optimal solution. They all give feasible solutions. In formal greedy framework:
candidates: jobs; partial solution: list of compatible jobs;
selection rule: varies; reject if job causes a conflict with what is already chosen; solution: list of jobs such that no more can be added.

Proof of optimality of EFT rule

- Basic idea: the greedy algorithm stays ahead throughout.

Proof of optimality of EFT rule

- Basic idea: the greedy algorithm stays ahead throughout.
- Let $\mathcal{O} = [j_1, \dots, j_m]$ be an optimal list of requests and let $A = [i_1, \dots, i_k]$ be the list created by the algorithm. We want to show that $k = m$.

Proof of optimality of EFT rule

- Basic idea: the greedy algorithm stays ahead throughout.
- Let $\mathcal{O} = [j_1, \dots, j_m]$ be an optimal list of requests and let $A = [i_1, \dots, i_k]$ be the list created by the algorithm. We want to show that $k = m$.
- Assume the requests are ordered in time in the obvious way.

Proof of optimality of EFT rule

- Basic idea: the greedy algorithm stays ahead throughout.
- Let $\mathcal{O} = [j_1, \dots, j_m]$ be an optimal list of requests and let $A = [i_1, \dots, i_k]$ be the list created by the algorithm. We want to show that $k = m$.
- Assume the requests are ordered in time in the obvious way.
- Prove by induction that $f(i_r) \leq f(j_r)$. Clear for $r = 1$ by greedy property. Induction step is easy.

Proof of optimality of EFT rule

- Basic idea: the greedy algorithm stays ahead throughout.
- Let $\mathcal{O} = [j_1, \dots, j_m]$ be an optimal list of requests and let $A = [i_1, \dots, i_k]$ be the list created by the algorithm. We want to show that $k = m$.
- Assume the requests are ordered in time in the obvious way.
- Prove by induction that $f(i_r) \leq f(j_r)$. Clear for $r = 1$ by greedy property. Induction step is easy.
- If A is not optimal, then $m > k$. But then $f(i_k) \leq f(j_k)$. There is another feasible job to add after time $f(j_k)$. But the greedy algorithm would not have stopped while there was a compatible job left. Contradiction: so A is optimal.

Graph algorithms in greedy framework

- (minimum spanning tree) Prim's algorithm: choose root and grow tree by adding shortest edge possible. Candidates: edges; partial solution: set of edges forming subtree; selection: cheapest edge connected to tree; rejection: if cycle formed.

Graph algorithms in greedy framework

- (minimum spanning tree) Prim's algorithm: choose root and grow tree by adding shortest edge possible. Candidates: edges; partial solution: set of edges forming subtree; selection: cheapest edge connected to tree; rejection: if cycle formed.
- (minimum spanning tree) Kruskal's algorithm: grow forest by adding cheapest edge possible. Candidates: edges; partial solution: edges forming forest; selection: cheapest edge; rejection: if cycle formed.

Graph algorithms in greedy framework

- (minimum spanning tree) Prim's algorithm: choose root and grow tree by adding shortest edge possible. Candidates: edges; partial solution: set of edges forming subtree; selection: cheapest edge connected to tree; rejection: if cycle formed.
- (minimum spanning tree) Kruskal's algorithm: grow forest by adding cheapest edge possible. Candidates: edges; partial solution: edges forming forest; selection: cheapest edge; rejection: if cycle formed.
- (single source shortest path) Dijkstra's algorithm: choose root and grow tree by adding currently closest node to root; update distances. Candidates: nodes (indices of distance array); partial solution: set of indices; selection: smallest array element; rejection: never.

Another scheduling problem

- We have a single resource and n jobs to schedule. Job i has deadline $d(i)$ and takes time $t(i)$ to perform. We must specify for each i the start time $s(i)$. The finish time is $f(i) := s(i) + t(i)$.

Another scheduling problem

- We have a single resource and n jobs to schedule. Job i has deadline $d(i)$ and takes time $t(i)$ to perform. We must specify for each i the start time $s(i)$. The finish time is $f(i) := s(i) + t(i)$.
- Define the lateness $l(i)$ to be $\max\{f(i) - d(i), 0\}$. We aim to minimize the maximum lateness $\max_i l(i)$.

Another scheduling problem

- We have a single resource and n jobs to schedule. Job i has deadline $d(i)$ and takes time $t(i)$ to perform. We must specify for each i the start time $s(i)$. The finish time is $f(i) := s(i) + t(i)$.
- Define the lateness $l(i)$ to be $\max\{f(i) - d(i), 0\}$. We aim to minimize the maximum lateness $\max_i l(i)$.
- Some greedy ideas:

Another scheduling problem

- We have a single resource and n jobs to schedule. Job i has deadline $d(i)$ and takes time $t(i)$ to perform. We must specify for each i the start time $s(i)$. The finish time is $f(i) := s(i) + t(i)$.
- Define the lateness $l(i)$ to be $\max\{f(i) - d(i), 0\}$. We aim to minimize the maximum lateness $\max_i l(i)$.
- Some greedy ideas:
 - (shortest job first) choose jobs in increasing order of $t(i)$;

Another scheduling problem

- We have a single resource and n jobs to schedule. Job i has deadline $d(i)$ and takes time $t(i)$ to perform. We must specify for each i the start time $s(i)$. The finish time is $f(i) := s(i) + t(i)$.
- Define the lateness $l(i)$ to be $\max\{f(i) - d(i), 0\}$. We aim to minimize the maximum lateness $\max_i l(i)$.
- Some greedy ideas:
 - (shortest job first) choose jobs in increasing order of $t(i)$;
 - (greatest slack time) choose jobs in decreasing order of $d(i) - t(i)$;


Another scheduling problem

- We have a single resource and n jobs to schedule. Job i has deadline $d(i)$ and takes time $t(i)$ to perform. We must specify for each i the start time $s(i)$. The finish time is $f(i) := s(i) + t(i)$.
- Define the lateness $l(i)$ to be $\max\{f(i) - d(i), 0\}$. We aim to minimize the maximum lateness $\max_i l(i)$.
- Some greedy ideas:
 - (shortest job first) choose jobs in increasing order of $t(i)$;
 - (greatest slack time) choose jobs in decreasing order of $d(i) - t(i)$;
 - (earliest deadline first) choose jobs in increasing order of $d(i)$.

Another scheduling problem

- We have a single resource and n jobs to schedule. Job i has deadline $d(i)$ and takes time $t(i)$ to perform. We must specify for each i the start time $s(i)$. The finish time is $f(i) := s(i) + t(i)$.
- Define the lateness $l(i)$ to be $\max\{f(i) - d(i), 0\}$. We aim to minimize the maximum lateness $\max_i l(i)$.
- Some greedy ideas:
 - (shortest job first) choose jobs in increasing order of $t(i)$;
 - (greatest slack time) choose jobs in decreasing order of $d(i) - t(i)$;
 - (earliest deadline first) choose jobs in increasing order of $d(i)$.
- The last one, denoted EDF, works (proof coming up) and the others do not (as seen by easy examples).

Proof of optimality of EDF

- Basic idea: **exchange argument**. Gradually transform an optimal solution into the greedy solution without sacrificing optimality.
- Let O be an optimal schedule. We may assume that O has no idle time.
- Renumber the jobs so that $d(1) \leq d(2) \leq \dots \leq d(n)$ and let A be the schedule $1, 2, \dots, n$ found by EDF.
- We show that if i precedes j in O and $d(i) > d(j)$, then swapping i and j gives an optimal schedule. The only thing to check is the new lateness of i :
$$\tilde{l}(i) = f(j) - d(i) < f(j) - d(j)$$
 so the maximum lateness has not increased.
- Iterating this we obtain a schedule with no idle time and no inversions. Up to a permutation of jobs with identical deadline, it is the same as A . In particular it has the same objective value so that A is also optimal. 

Optimal caching

- We have a set U of n data items in main memory. We have a memory **cache** that holds $k < n$ data items. A sequence D of **requests** (elements of U) is given. We want to minimize the number of **cache misses** (a requested element is not in the cache).

Optimal caching

- We have a set U of n data items in main memory. We have a memory **cache** that holds $k < n$ data items. A sequence D of **requests** (elements of U) is given. We want to minimize the number of **cache misses** (a requested element is not in the cache).
- When an item is added to a full cache, another item must be evicted. We need to specify an **eviction schedule**.

Optimal caching

- We have a set U of n data items in main memory. We have a memory **cache** that holds $k < n$ data items. A sequence D of **requests** (elements of U) is given. We want to minimize the number of **cache misses** (a requested element is not in the cache).
- When an item is added to a full cache, another item must be evicted. We need to specify an **eviction schedule**.
- Under real conditions we must solve the **online version** of this problem because we don't know D in advance. But studying the offline version is important because it gives an upper bound on performance of online algorithms.

Optimal caching

- We have a set U of n data items in main memory. We have a memory **cache** that holds $k < n$ data items. A sequence D of **requests** (elements of U) is given. We want to minimize the number of **cache misses** (a requested element is not in the cache).
- When an item is added to a full cache, another item must be evicted. We need to specify an **eviction schedule**.
- Under real conditions we must solve the **online version** of this problem because we don't know D in advance. But studying the offline version is important because it gives an upper bound on performance of online algorithms.
- FIF (**Farthest-in-future**) rule: evict the data item whose next request is latest among all elements currently in the cache.

Proof of optimality of FIF

- Exchange argument: let S be an optimal eviction schedule and A the one chosen by FIF.

Proof of optimality of FIF

- Exchange argument: let S be an optimal eviction schedule and A the one chosen by FIF.
- Consider the first point where S deviates from A : element d is brought in, S evicts e and A evicts $f \neq e$. We transform to an optimal S' which agrees with A at this step also.

Proof of optimality of FIF

- Exchange argument: let S be an optimal eviction schedule and A the one chosen by FIF.
- Consider the first point where S deviates from A : element d is brought in, S evicts e and A evicts $f \neq e$. We transform to an optimal S' which agrees with A at this step also.
- Specifically, S' evicts e at this step. In future steps, it copies S until one of the following things happens:

Proof of optimality of FIF

- Exchange argument: let S be an optimal eviction schedule and A the one chosen by FIF.
- Consider the first point where S deviates from A : element d is brought in, S evicts e and A evicts $f \neq e$. We transform to an optimal S' which agrees with A at this step also.
- Specifically, S' evicts e at this step. In future steps, it copies S until one of the following things happens:
 - There is a request for $g \neq e, f$ not in cache of S , and S evicts e . Then g is not in cache of S' so we make S' evict f .

Proof of optimality of FIF

- Exchange argument: let S be an optimal eviction schedule and A the one chosen by FIF.
- Consider the first point where S deviates from A : element d is brought in, S evicts e and A evicts $f \neq e$. We transform to an optimal S' which agrees with A at this step also.
- Specifically, S' evicts e at this step. In future steps, it copies S until one of the following things happens:
 - There is a request for $g \neq e, f$ not in cache of S , and S evicts e . Then g is not in cache of S' so we make S' evict f .
 - There is a request for f and S evicts e' . If $e' = e$, S' does not need to evict and simply accesses f from cache. If $e' \neq e$, then S' evicts e' and brings in e .

Proof of optimality of FIF

- Exchange argument: let S be an optimal eviction schedule and A the one chosen by FIF.
- Consider the first point where S deviates from A : element d is brought in, S evicts e and A evicts $f \neq e$. We transform to an optimal S' which agrees with A at this step also.
- Specifically, S' evicts e at this step. In future steps, it copies S until one of the following things happens:
 - There is a request for $g \neq e, f$ not in cache of S , and S evicts e . Then g is not in cache of S' so we make S' evict f .
 - There is a request for f and S evicts e' . If $e' = e$, S' does not need to evict and simply accesses f from cache. If $e' \neq e$, then S' evicts e' and brings in e .
 - In each case S' now now has the same cache as S and copies S from now on. Under FIF, one of the cases above must occur before e is requested. Thus S' makes no more misses than S .

Knapsack problem

- We have a container with capacity W and must choose items from a list of n items. The i th item has a weight w_i and a value v_i . We aim to maximize the total value in our container.

Knapsack problem

- We have a container with capacity W and must choose items from a list of n items. The i th item has a weight w_i and a value v_i . We aim to maximize the total value in our container.
- Linear programming formulation: maximize $\sum_j a_j v_j$ subject to $\sum_j a_j w_j \leq W$ and $0 \leq a_j \leq 1$.

Knapsack problem

- We have a container with capacity W and must choose items from a list of n items. The i th item has a weight w_i and a value v_i . We aim to maximize the total value in our container.
- Linear programming formulation: maximize $\sum_j a_j v_j$ subject to $\sum_j a_j w_j \leq W$ and $0 \leq a_j \leq 1$.
- Greedy idea: take items in decreasing order of value density v_i/w_i .

Knapsack problem

- We have a container with capacity W and must choose items from a list of n items. The i th item has a weight w_i and a value v_i . We aim to maximize the total value in our container.
- Linear programming formulation: maximize $\sum_j a_j v_j$ subject to $\sum_j a_j w_j \leq W$ and $0 \leq a_j \leq 1$.
- Greedy idea: take items in decreasing order of value density v_i/w_i .
- This is optimal if we can take arbitrary amounts of each type (a_j are real). But it is not if we cannot subdivide objects (a_j must be integer). Example: $W = 3$,
 $v_1 = 1, v_2 = 2, w_1 = 1, w_2 = 3$.

Knapsack problem

- We have a container with capacity W and must choose items from a list of n items. The i th item has a weight w_i and a value v_i . We aim to maximize the total value in our container.
- Linear programming formulation: maximize $\sum_j a_j v_j$ subject to $\sum_j a_j w_j \leq W$ and $0 \leq a_j \leq 1$.
- Greedy idea: take items in decreasing order of value density v_i/w_i .
- This is optimal if we can take arbitrary amounts of each type (a_j are real). But it is not if we cannot subdivide objects (a_j must be integer). Example: $W = 3$,
 $v_1 = 1, v_2 = 2, w_1 = 1, w_2 = 3$.
- Implementation: sort items in order of decreasing v_i/w_i , or build a priority and extract repeatedly.

Proof of optimality of greedy algorithm

- Proof of optimality: let j be the first place where an optimal solution $O = [b_1, \dots, b_n]$ deviates from the greedy solution $A = [a_1, \dots, a_n]$. Then $b_j < a_j$, for all $i < j$ we have $a_i = 1 = b_i$, and $a_i = 0$ for all $i > j$.
- Since O is optimal there must be some $k > j$ with $b_k > 0$. But then we can change b_j to $b_j + b_k w_k / w_j$ and maintain the weight constraint while not decreasing the objective. This gives an optimal solution agreeing with A till step $j + 1$. Now use induction on j .
- This is an exchange argument, and also shows that the greedy algorithm stays ahead. For each k , it gives the optimum to the restricted problem where only objects $1..k$ can be chosen.

Yet another scheduling problem

- We have n jobs to execute each taking one time unit, and exactly one job can be done at each time step. The i th job earns us profit g_i if and only if executed before the deadline, time step d_i . We want to maximize total profit.

Yet another scheduling problem

- We have n jobs to execute each taking one time unit, and exactly one job can be done at each time step. The i th job earns us profit g_i if and only if executed before the deadline, time step d_i . We want to maximize total profit.
- Greedy approach: choose jobs in order of decreasing profit, provided we maintain feasibility (there is some way to permute current vector and schedule all jobs in it). How to check feasibility efficiently?

Yet another scheduling problem

- We have n jobs to execute each taking one time unit, and exactly one job can be done at each time step. The i th job earns us profit g_i if and only if executed before the deadline, time step d_i . We want to maximize total profit.
- Greedy approach: choose jobs in order of decreasing profit, provided we maintain feasibility (there is some way to permute current vector and schedule all jobs in it). How to check feasibility efficiently?
- Fact: a list I is feasible if and only if the permutation that has jobs in increasing deadline order is feasible. Proof: fairly obvious. So only need to check one order for a subset. There is a better method (only check the schedule built by considering jobs in order, putting each job as late as possible). See later.

Yet another scheduling problem

- We have n jobs to execute each taking one time unit, and exactly one job can be done at each time step. The i th job earns us profit g_i if and only if executed before the deadline, time step d_i . We want to maximize total profit.
- Greedy approach: choose jobs in order of decreasing profit, provided we maintain feasibility (there is some way to permute current vector and schedule all jobs in it). How to check feasibility efficiently?
- Fact: a list I is feasible if and only if the permutation that has jobs in increasing deadline order is feasible. Proof: fairly obvious. So only need to check one order for a subset. There is a better method (only check the schedule built by considering jobs in order, putting each job as late as possible). See later.
- We always obtain an optimal solution this way. Proof follows.

Proof of optimality

- Exchange argument. Let I be job list chosen by greedy, and J an optimal one. Claim: we can feasibly reorder so each job $a \in I \cap J$ occurs at same time in each schedule.

Proof of optimality

- Exchange argument. Let I be job list chosen by greedy, and J an optimal one. Claim: we can feasibly reorder so each job $a \in I \cap J$ occurs at same time in each schedule.
- Assuming this, suppose that

Proof of optimality

- Exchange argument. Let I be job list chosen by greedy, and J an optimal one. Claim: we can feasibly reorder so each job $a \in I \cap J$ occurs at same time in each schedule.
- Assuming this, suppose that
 - $a \in I$ is opposite a gap in J . Then can feasibly insert a in J , yielding more profit than J , contradiction.

Proof of optimality

- Exchange argument. Let I be job list chosen by greedy, and J an optimal one. Claim: we can feasibly reorder so each job $a \in I \cap J$ occurs at same time in each schedule.
- Assuming this, suppose that
 - $a \in I$ is opposite a gap in J . Then can feasibly insert a in J , yielding more profit than J , contradiction.
 - $b \in J$ is opposite a gap in I . Then can feasibly insert b in I , so greedy would have chosen b already, contradiction.

Proof of optimality

- Exchange argument. Let I be job list chosen by greedy, and J an optimal one. Claim: we can feasibly reorder so each job $a \in I \cap J$ occurs at same time in each schedule.
- Assuming this, suppose that
 - $a \in I$ is opposite a gap in J . Then can feasibly insert a in J , yielding more profit than J , contradiction.
 - $b \in J$ is opposite a gap in I . Then can feasibly insert b in I , so greedy would have chosen b already, contradiction.
 - a in $I \setminus J$ is opposite b in $J \setminus I$. If $g_a \neq g_b$, can replace one by the other and get contradiction, so $g_a = g_b$, and hence I is also optimal.

Proof of optimality

- Exchange argument. Let I be job list chosen by greedy, and J an optimal one. Claim: we can feasibly reorder so each job $a \in I \cap J$ occurs at same time in each schedule.
- Assuming this, suppose that
 - $a \in I$ is opposite a gap in J . Then can feasibly insert a in J , yielding more profit than J , contradiction.
 - $b \in J$ is opposite a gap in I . Then can feasibly insert b in I , so greedy would have chosen b already, contradiction.
 - a in $I \setminus J$ is opposite b in $J \setminus I$. If $g_a \neq g_b$, can replace one by the other and get contradiction, so $g_a = g_b$, and hence I is also optimal.
- Proof of claim: if $a \in I \cap J$, order them feasibly with a occurring at times t_I, t_J . If $t_I = t_J$, done. If $t_I < t_J$, move a in I to time t_J , swapping with anything that may be there. Still feasible. Similarly if $t_I > t_J$. Once a is moved, it is never moved again. Hence eventually all such a match up.

Analysis of the algorithm

- Using first feasibility criterion, and array implementation, it takes total time in $\Theta(n^2)$ in worst case to do all feasibility checks. At each stage need to find place to insert latest job, and check that moving others doesn't violate deadline.

Analysis of the algorithm

- Using first feasibility criterion, and array implementation, it takes total time in $\Theta(n^2)$ in worst case to do all feasibility checks. At each stage need to find place to insert latest job, and check that moving others doesn't violate deadline.
- Using second criterion, we can do better using the data structure (disjoint sets ADT). At each stage find latest time available (corresponds to finding which set deadline belongs to) and merge two sets. Can be done in ALMOST linear time.

Dijkstra's algorithm

```
algorithm Dijkstra(weighted digraph  $(G, c)$ , node  $v$ )  
for  $u \in V(G)$  do  
     $dist[u] \leftarrow c[v, u]$   
 $S \leftarrow \{v\}$   
while  $S \neq V(G)$  do  
    find  $u \in V(G) \setminus S$  so that  $dist[u]$  is minimum  
     $S \leftarrow S \cup \{u\}$   
    for  $x \in V(G) \setminus S$  do  
         $dist[x] \leftarrow \min\{dist[x], dist[u] + c[u, x]\}$ 
```

At top of **while** loop, this property holds:

P: if $w \in S$, $dist[w]$ is the minimum weight of a path to w .

Dijkstra's algorithm stays ahead

- Proof by induction on m that P holds after m iterations of the `while` loop. Call the corresponding statement $P(m)$.

Dijkstra's algorithm stays ahead

- Proof by induction on m that P holds after m iterations of the `while` loop. Call the corresponding statement $P(m)$.
- When $m = 0$, $S_0 = \{v\}$ and clearly $P(0)$ holds.

Dijkstra's algorithm stays ahead

- Proof by induction on m that P holds after m iterations of the `while` loop. Call the corresponding statement $P(m)$.
- When $m = 0$, $S_0 = \{v\}$ and clearly $P(0)$ holds.
- Suppose $P(m)$ holds and that at iteration $m + 1$, the newest special node is u (so $S_{m+1} = S_m \cup \{u\}$), and the last arc used is (v, u) .

Dijkstra's algorithm stays ahead

- Proof by induction on m that P holds after m iterations of the `while` loop. Call the corresponding statement $P(m)$.
- When $m = 0$, $S_0 = \{v\}$ and clearly $P(0)$ holds.
- Suppose $P(m)$ holds and that at iteration $m + 1$, the newest special node is u (so $S_{m+1} = S_m \cup \{u\}$), and the last arc used is (v, u) .
- Let $w \in S_{m+1}$. Note $dist_{m+1}[w] = dist_m[w]$. If $w \neq u$ then since $P(m)$ holds, so does $P(m + 1)$.

Dijkstra's algorithm stays ahead

- Proof by induction on m that P holds after m iterations of the `while` loop. Call the corresponding statement $P(m)$.
- When $m = 0$, $S_0 = \{v\}$ and clearly $P(0)$ holds.
- Suppose $P(m)$ holds and that at iteration $m + 1$, the newest special node is u (so $S_{m+1} = S_m \cup \{u\}$), and the last arc used is (v, u) .
- Let $w \in S_{m+1}$. Note $dist_{m+1}[w] = dist_m[w]$. If $w \neq u$ then since $P(m)$ holds, so does $P(m + 1)$.
- If $w = u$, then by choice of u , any S_{m+1} -path to u of weight less than $dist[u]$ must go straight to u on exiting S_m . But the algorithm rejected this choice when it chose the arc (v, u) . Thus $P(m + 1)$ holds.

Dijkstra's algorithm stays ahead

- Proof by induction on m that P holds after m iterations of the `while` loop. Call the corresponding statement $P(m)$.
- When $m = 0$, $S_0 = \{v\}$ and clearly $P(0)$ holds.
- Suppose $P(m)$ holds and that at iteration $m + 1$, the newest special node is u (so $S_{m+1} = S_m \cup \{u\}$), and the last arc used is (v, u) .
- Let $w \in S_{m+1}$. Note $dist_{m+1}[w] = dist_m[w]$. If $w \neq u$ then since $P(m)$ holds, so does $P(m + 1)$.
- If $w = u$, then by choice of u , any S_{m+1} -path to u of weight less than $dist[u]$ must go straight to u on exiting S_m . But the algorithm rejected this choice when it chose the arc (v, u) . Thus $P(m + 1)$ holds.
- By induction $P(m)$ is true for all m .

Minimum spanning tree problem

- Given a connected weighted graph, find a **spanning tree** (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.

Minimum spanning tree problem

- Given a connected weighted graph, find a **spanning tree** (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient **greedy** algorithms presented here: Prim's and Kruskal's. This problem has more greedy algorithms that work.

Minimum spanning tree problem

- Given a connected weighted graph, find a **spanning tree** (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient **greedy** algorithms presented here: Prim's and Kruskal's. This problem has more greedy algorithms that work.
- Each selects edges in order of increasing weight, subject to not obviously creating a cycle.

Minimum spanning tree problem

- Given a connected weighted graph, find a **spanning tree** (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient **greedy** algorithms presented here: Prim's and Kruskal's. This problem has more greedy algorithms that work.
- Each selects edges in order of increasing weight, subject to not obviously creating a cycle.
- Prim maintains a tree at each stage that grows to span; Kruskal maintains a forest whose trees coalesce into one spanning tree.

Minimum spanning tree problem

- Given a connected weighted graph, find a **spanning tree** (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient **greedy** algorithms presented here: Prim's and Kruskal's. This problem has more greedy algorithms that work.
- Each selects edges in order of increasing weight, subject to not obviously creating a cycle.
- Prim maintains a tree at each stage that grows to span; Kruskal maintains a forest whose trees coalesce into one spanning tree.
- Prim implementation very similar to Dijkstra, get $O(e + n \log n)$; Kruskal uses disjoint sets ADT and can be implemented to run in time $O(e \log n)$.

Prim's algorithm

```
algorithm Prim(weighted digraph  $(G, c)$ , node  $v$ )  
for  $u \in V(G)$  do  
     $d[u] \leftarrow \infty$   
 $d[v] \leftarrow 0$   
 $S \leftarrow \emptyset$   
while  $S \neq V(G)$  do  
    find  $u \in V(G) \setminus S$  so that  $d[u]$  is minimum  
     $S \leftarrow S \cup \{u\}$   
    for  $x \in V(G) \setminus S$  do  
         $d[x] \leftarrow \min\{d[x], c[u, x]\}$ 
```

Very similar to Dijkstra - uses a priority queue to hold elements of d . Most time taken by EXTRACT-MIN and DECREASE-KEY operations.

Kruskal's algorithm

```
algorithm Kruskal(weighted digraph  $(G, c)$ )  
 $T \leftarrow \emptyset$   
sort  $E(G)$  by increasing order of cost  
for  $e = \{u, v\} \in E(G)$  do  
    if  $u$  and  $v$  are not in the same tree then  
         $T \leftarrow T \cup \{e\}$   
        merge the trees of  $u$  and  $v$ 
```

Keep track of the trees using disjoint sets ADT, with standard operations FIND and UNION. They can be implemented efficiently so that the main time taken is the sorting step.

Proof of optimality of Prim and Kruskal

- Basic idea is an exchange argument.

Proof of optimality of Prim and Kruskal

- Basic idea is an exchange argument.
- Claim: let $B \subset V$ and let $T \subseteq E$ be promising, and no edge in T leaves B . Let e be minimum weight edge leaving B . Then $T \cup \{e\}$ is promising.

Proof of optimality of Prim and Kruskal

- Basic idea is an exchange argument.
- Claim: let $B \subset V$ and let $T \subseteq E$ be promising, and no edge in T leaves B . Let e be minimum weight edge leaving B . Then $T \cup \{e\}$ is promising.
- Assuming claim, proof follows by taking $B =$ nodes of component including endpoint of next edge e (Kruskal) or $B =$ nodes of current tree (Prim).

Proof of optimality of Prim and Kruskal

- Basic idea is an exchange argument.
- Claim: let $B \subset V$ and let $T \subseteq E$ be promising, and no edge in T leaves B . Let e be minimum weight edge leaving B . Then $T \cup \{e\}$ is promising.
- Assuming claim, proof follows by taking $B =$ nodes of component including endpoint of next edge e (Kruskal) or $B =$ nodes of current tree (Prim).
- Proof of claim: let U be MST containing T . If $e \in U$, done. Else there is another edge e' leaving B (to close the cycle). Then removing e' and adding e to U gives MST containing T .

Design paradigm 2: divide and conquer

- Basic idea: decompose a problem instance into smaller instances (“divide”); solve these instances (recursively); combine their solutions to give solution for original instance (“conquer”). We can also solve very small instances nonrecursively if that is more efficient.

Design paradigm 2: divide and conquer

- Basic idea: decompose a problem instance into smaller instances (“divide”); solve these instances (recursively); combine their solutions to give solution for original instance(“conquer”). We can also solve very small instances nonrecursively if that is more efficient.
- Often most work is in the “divide” step or most work is in the “conquer” step.

Design paradigm 2: divide and conquer

- Basic idea: decompose a problem instance into smaller instances (“divide”); solve these instances (recursively); combine their solutions to give solution for original instance (“conquer”). We can also solve very small instances nonrecursively if that is more efficient.
- Often most work is in the “divide” step or most work is in the “conquer” step.
- Examples you (should) have already seen: mergesort (“conquer” type), quicksort (“divide” type), quickselect, binary search (simplest case).

Design paradigm 2: divide and conquer

- Basic idea: decompose a problem instance into smaller instances (“divide”); solve these instances (recursively); combine their solutions to give solution for original instance (“conquer”). We can also solve very small instances nonrecursively if that is more efficient.
- Often most work is in the “divide” step or most work is in the “conquer” step.
- Examples you (should) have already seen: mergesort (“conquer” type), quicksort (“divide” type), quickselect, binary search (simplest case).
- New examples: multiplication, median-finding,

Design paradigm 2: divide and conquer

- Basic idea: decompose a problem instance into smaller instances (“divide”); solve these instances (recursively); combine their solutions to give solution for original instance (“conquer”). We can also solve very small instances nonrecursively if that is more efficient.
- Often most work is in the “divide” step or most work is in the “conquer” step.
- Examples you (should) have already seen: mergesort (“conquer” type), quicksort (“divide” type), quickselect, binary search (simplest case).
- New examples: multiplication, median-finding,
- Running time analysis leads to a certain type of *recurrence* relation.

Divide and conquer recurrences

- If the subinstances for a size n instance have sizes p_1, \dots, p_k , the overhead cost for dividing and combining is $f(n)$, and n_0 is the threshold below which we use another algorithm for small instances, then the cost $T(n)$ for this instance is (roughly) given by

$$T(n) = \sum_i T(p_i) + f(n) \quad \text{for } n \geq n_0.$$

Divide and conquer recurrences

- If the subinstances for a size n instance have sizes p_1, \dots, p_k , the overhead cost for dividing and combining is $f(n)$, and n_0 is the threshold below which we use another algorithm for small instances, then the cost $T(n)$ for this instance is (roughly) given by

$$T(n) = \sum_i T(p_i) + f(n) \quad \text{for } n \geq n_0.$$

- Examples (cost = number of comparisons):

Divide and conquer recurrences

- If the subinstances for a size n instance have sizes p_1, \dots, p_k , the overhead cost for dividing and combining is $f(n)$, and n_0 is the threshold below which we use another algorithm for small instances, then the cost $T(n)$ for this instance is (roughly) given by

$$T(n) = \sum_i T(p_i) + f(n) \quad \text{for } n \geq n_0.$$

- Examples (cost = number of comparisons):
 - binary search: $k = 1, p_1 = \lceil n/2 \rceil$ or $\lfloor n/2 \rfloor, f(n) \in \Theta(1)$

Divide and conquer recurrences

- If the subinstances for a size n instance have sizes p_1, \dots, p_k , the overhead cost for dividing and combining is $f(n)$, and n_0 is the threshold below which we use another algorithm for small instances, then the cost $T(n)$ for this instance is (roughly) given by

$$T(n) = \sum_i T(p_i) + f(n) \quad \text{for } n \geq n_0.$$

- Examples (cost = number of comparisons):
 - binary search: $k = 1, p_1 = \lceil n/2 \rceil$ or $\lfloor n/2 \rfloor, f(n) \in \Theta(1)$
 - mergesort: $k = 2, p_1 = \lceil n/2 \rceil, p_2 = \lfloor n/2 \rfloor, f(n) \in \Theta(n)$

Divide and conquer recurrences

- If the subinstances for a size n instance have sizes p_1, \dots, p_k , the overhead cost for dividing and combining is $f(n)$, and n_0 is the threshold below which we use another algorithm for small instances, then the cost $T(n)$ for this instance is (roughly) given by

$$T(n) = \sum_i T(p_i) + f(n) \quad \text{for } n \geq n_0.$$

- Examples (cost = number of comparisons):
 - binary search: $k = 1, p_1 = \lceil n/2 \rceil$ or $\lfloor n/2 \rfloor, f(n) \in \Theta(1)$
 - mergesort: $k = 2, p_1 = \lceil n/2 \rceil, p_2 = \lfloor n/2 \rfloor, f(n) \in \Theta(n)$
 - quicksort: $k = 2, p_1 = l, p_2 = n - l - 1$ (where $0 \leq l \leq n - 1$ can have any value), $f(n) \in \Theta(n)$.

Divide and conquer recurrences

- If the subinstances for a size n instance have sizes p_1, \dots, p_k , the overhead cost for dividing and combining is $f(n)$, and n_0 is the threshold below which we use another algorithm for small instances, then the cost $T(n)$ for this instance is (roughly) given by

$$T(n) = \sum_i T(p_i) + f(n) \quad \text{for } n \geq n_0.$$

- Examples (cost = number of comparisons):
 - binary search: $k = 1, p_1 = \lceil n/2 \rceil$ or $\lfloor n/2 \rfloor, f(n) \in \Theta(1)$
 - mergesort: $k = 2, p_1 = \lceil n/2 \rceil, p_2 = \lfloor n/2 \rfloor, f(n) \in \Theta(n)$
 - quicksort: $k = 2, p_1 = l, p_2 = n - l - 1$ (where $0 \leq l \leq n - 1$ can have any value), $f(n) \in \Theta(n)$.
- Need a general method for solving such recurrences, at least asymptotically. Restrict to the case where k is independent of n and the instance.

Solution of divide and conquer recurrences

Fix $a > 0, b > 1$ and consider

$$T(n) = aT(n/b) + f(n) \quad (\text{when } n > n_0), \quad T(n_0) = c.$$

First solve this when n/n_0 is a power of b : put $U(i) = T(n_0b^i)$ and $g(i) = f(n_0b^i)$. Get

$$U(i) = aU(i-1) + g(i) \quad (\text{when } i > 0), \quad U(0) = c.$$

Iterate to obtain

$$U(i) = a^i c + \sum_{k=0}^{i-1} a^k g(i-k).$$

Important special case: $f(n) = n^p$ for some fixed p . Write $B = b^p$.

Then by above (sum a geometric series) we obtain

$$U(i) = \begin{cases} ca^i + n_0^p B(a^i - B^i)/(a - B) & \text{if } a \neq B; \\ ca^i + in_0^p B^i & \text{if } a = B. \end{cases}$$

Let $e = \log_b a$. Then we have, for n/n_0 a power of b ,

$$T(n) = \begin{cases} (n_0)^{-e} \left(c + \frac{n_0^p b^p}{a - b^p} \right) n^e - \frac{b^p}{a - b^p} n^p & \text{if } e \neq p; \\ n^e \log_b n + (cn_0^{-e} - \log_b n_0) n^e & \text{if } e = p. \end{cases}$$

So (conditional on $n/n_0 = b^i$) we have

$$T(n) \in \begin{cases} \Theta(f(n)) & \text{if } e < p; \\ \Theta(n^e) & \text{if } e > p; \\ \Theta(f(n) \log n) & \text{if } e = p. \end{cases}$$

Is this result true unconditionally? For general $f(n)$? What about O, Ω ?

Solution of divide and conquer recurrences

- Exact solution when n/n_0 not an exact power is complicated.
Example: mergesort $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ has solution $T(n) = n \lg n + n\theta(n)$ where $\theta(n) = 0$ if n is a power of 2 and $0 < \theta(n) < 0.086$ otherwise.

Solution of divide and conquer recurrences

- Exact solution when n/n_0 not an exact power is complicated.
Example: mergesort $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ has solution $T(n) = n \lg n + n\theta(n)$ where $\theta(n) = 0$ if n is a power of 2 and $0 < \theta(n) < 0.086$ otherwise.
- We only consider asymptotic results from now on. The main idea is as for exact powers, but there are several technical details.

Solution of divide and conquer recurrences

- Exact solution when n/n_0 not an exact power is complicated.
Example: mergesort $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ has solution $T(n) = n \lg n + n\theta(n)$ where $\theta(n) = 0$ if n is a power of 2 and $0 < \theta(n) < 0.086$ otherwise.
- We only consider asymptotic results from now on. The main idea is as for exact powers, but there are several technical details.
- The smoothness rule plays a big part.

Solution of divide and conquer recurrences

- Exact solution when n/n_0 not an exact power is complicated. Example: mergesort $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ has solution $T(n) = n \lg n + n\theta(n)$ where $\theta(n) = 0$ if n is a power of 2 and $0 < \theta(n) < 0.086$ otherwise.
- We only consider asymptotic results from now on. The main idea is as for exact powers, but there are several technical details.
- The smoothness rule plays a big part.
- Often we have a recurrence *inequality*; we can analyse by relating to analogous recurrence equation.

Solution of divide and conquer recurrences

- Exact solution when n/n_0 not an exact power is complicated. Example: mergesort $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ has solution $T(n) = n \lg n + n\theta(n)$ where $\theta(n) = 0$ if n is a power of 2 and $0 < \theta(n) < 0.086$ otherwise.
- We only consider asymptotic results from now on. The main idea is as for exact powers, but there are several technical details.
- The smoothness rule plays a big part.
- Often we have a recurrence *inequality*; we can analyse by relating to analogous recurrence equation.
- Previous result with $f(n) = n^p$ does generalize to case where subproblem sizes are *almost* equal and $f(n) \in \Theta(n^p(\log n)^q)$. This is often called the “Master Theorem”. See me for proof.

Details for example of mergesort

- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n)$ with $f(n) \in \Theta(n)$.

Details for example of mergesort

- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n)$ with $f(n) \in \Theta(n)$.
- Prove by induction that T is increasing.

Details for example of mergesort

- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n)$ with $f(n) \in \Theta(n)$.
- Prove by induction that T is increasing.
- Observe that $2T(\lfloor n/2 \rfloor) + f(n) \leq T(n) \leq 2T(\lceil n/2 \rceil) + f(n)$.

Details for example of mergesort

- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n)$ with $f(n) \in \Theta(n)$.
- Prove by induction that T is increasing.
- Observe that $2T(\lfloor n/2 \rfloor) + f(n) \leq T(n) \leq 2T(\lceil n/2 \rceil) + f(n)$.
- Define $\check{T}(n) = 2\check{T}(\lfloor n/2 \rfloor) + f(n)$, $\hat{T}(n) = 2\hat{T}(\lceil n/2 \rceil) + f(n)$.
Show that $\check{T}(n) \leq T(n) \leq \hat{T}(n)$.

Details for example of mergesort

- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n)$ with $f(n) \in \Theta(n)$.
- Prove by induction that T is increasing.
- Observe that $2T(\lfloor n/2 \rfloor) + f(n) \leq T(n) \leq 2T(\lceil n/2 \rceil) + f(n)$.
- Define $\check{T}(n) = 2\check{T}(\lfloor n/2 \rfloor) + f(n)$, $\hat{T}(n) = 2\hat{T}(\lceil n/2 \rceil) + f(n)$.
Show that $\check{T}(n) \leq T(n) \leq \hat{T}(n)$.
- Show that $\hat{T}(n) \in O(n \lg n)$ and $\check{T}(n) \in \Omega(n \lg n)$ **conditional on** $n = 2^k$.

Details for example of mergesort

- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n)$ with $f(n) \in \Theta(n)$.
- Prove by induction that T is increasing.
- Observe that $2T(\lfloor n/2 \rfloor) + f(n) \leq T(n) \leq 2T(\lceil n/2 \rceil) + f(n)$.
- Define $\check{T}(n) = 2\check{T}(\lfloor n/2 \rfloor) + f(n)$, $\hat{T}(n) = 2\hat{T}(\lceil n/2 \rceil) + f(n)$.
Show that $\check{T}(n) \leq T(n) \leq \hat{T}(n)$.
- Show that $\hat{T}(n) \in O(n \lg n)$ and $\check{T}(n) \in \Omega(n \lg n)$ **conditional on** $n = 2^k$.
- Apply the smoothness rule to conclude that $T(n) \in \Theta(n \lg n)$.

Issues arising from analysis

- The relation between subproblem sizes and the number of subproblems is crucial.

Issues arising from analysis

- The relation between subproblem sizes and the number of subproblems is crucial.
- Reducing a and increasing b are equally important.

Issues arising from analysis

- The relation between subproblem sizes and the number of subproblems is crucial.
- Reducing a and increasing b are equally important.
- Fine-tuning: how to determine the threshold n_0 ? Note order of asymptotics don't depend on n_0 . Need to consider lower order terms, and the implied constants can't be ignored.

Issues arising from analysis

- The relation between subproblem sizes and the number of subproblems is crucial.
- Reducing a and increasing b are equally important.
- Fine-tuning: how to determine the threshold n_0 ? Note order of asymptotics don't depend on n_0 . Need to consider lower order terms, and the implied constants can't be ignored.
- One way to determine threshold; choose n_0 to be minimal such that on size n_0 input, the direct algorithm is no faster than applying recursion once and then using the direct algorithm.

D & C polynomial multiplication I

- Given (integer) polynomials $p(x), q(x)$ of degree n, m , compute the product $r(x) = p(x)q(x)$ efficiently. Integer multiplication is a special case!

D & C polynomial multiplication I

- Given (integer) polynomials $p(x), q(x)$ of degree n, m , compute the product $r(x) = p(x)q(x)$ efficiently. Integer multiplication is a special case!
- Divide and conquer: suppose first that $n = m$ is exact power of 2. Write $p(x) = p_0(x) + x^{n/2}p_1(x), q(x) = q_0(x) + x^{n/2}q_1(x)$ where $\deg p_i, \deg q_i \leq n/2$. So $r(x) = a_0(x) + x^{n/2}a_1(x) + x^n a_2(x)$ where $a_0 = p_0q_0, a_1 = p_0q_1 + p_1q_0, a_2 = p_1q_1$.

D & C polynomial multiplication I

- Given (integer) polynomials $p(x), q(x)$ of degree n, m , compute the product $r(x) = p(x)q(x)$ efficiently. Integer multiplication is a special case!
- Divide and conquer: suppose first that $n = m$ is exact power of 2. Write $p(x) = p_0(x) + x^{n/2}p_1(x), q(x) = q_0(x) + x^{n/2}q_1(x)$ where $\deg p_i, \deg q_i \leq n/2$. So $r(x) = a_0(x) + x^{n/2}a_1(x) + x^n a_2(x)$ where $a_0 = p_0q_0, a_1 = p_0q_1 + p_1q_0, a_2 = p_1q_1$.
- Using distributive rule naively gives 4 multiplications of half the size plus linear overhead, no improvement. Key idea: reduce to 3 multiplications of half size at cost of more (still linear) overhead. How??

D & C polynomial multiplication II

- The trick: we only need $p_0q_1 + p_1q_0$, not p_0q_1 and p_1q_0 . Note that $p_0q_1 + p_1q_0 = (p_0 + p_1)(q_0 + q_1) - p_0q_0 - p_1q_1$. So we only need 3 multiplications of half size polynomials!

D & C polynomial multiplication II

- The trick: we only need $p_0q_1 + p_1q_0$, not p_0q_1 and p_1q_0 . Note that $p_0q_1 + p_1q_0 = (p_0 + p_1)(q_0 + q_1) - p_0q_0 - p_1q_1$. So we only need 3 multiplications of half size polynomials!
- Recurrence looks like $T(n) \leq 3T(n/2) + g(n)$, with $g(n) \in \Theta(n)$. Solution $T(n) \in O(n^{\lg 3})$, $\lg 3 = 1.59\dots$, provided we can deal with odd length numbers!

D & C polynomial multiplication II

- The trick: we only need $p_0q_1 + p_1q_0$, not p_0q_1 and p_1q_0 . Note that $p_0q_1 + p_1q_0 = (p_0 + p_1)(q_0 + q_1) - p_0q_0 - p_1q_1$. So we only need 3 multiplications of half size polynomials!
- Recurrence looks like $T(n) \leq 3T(n/2) + g(n)$, with $g(n) \in \Theta(n)$. Solution $T(n) \in O(n^{\lg 3})$, $\lg 3 = 1.59\dots$, provided we can deal with odd length numbers!
- Note that if we replace $x^{n/2}$ by a variable t , then $r = a_0 + a_1t + a_2t^2$. We can determine r by evaluating at any 3 points, (interpolation). This generalizes (see assignment).

D & C polynomial multiplication II

- The trick: we only need $p_0q_1 + p_1q_0$, not p_0q_1 and p_1q_0 . Note that $p_0q_1 + p_1q_0 = (p_0 + p_1)(q_0 + q_1) - p_0q_0 - p_1q_1$. So we only need 3 multiplications of half size polynomials!
- Recurrence looks like $T(n) \leq 3T(n/2) + g(n)$, with $g(n) \in \Theta(n)$. Solution $T(n) \in O(n^{\lg 3})$, $\lg 3 = 1.59\dots$, provided we can deal with odd length numbers!
- Note that if we replace $x^{n/2}$ by a variable t , then $r = a_0 + a_1t + a_2t^2$. We can determine r by evaluating at any 3 points, (interpolation). This generalizes (see assignment).
- If $m < n$, use distributivity to break q into blocks of degree m and multiply each by p as above, give $O(nm^{\lg 3/2})$ running time.

D & C polynomial multiplication II

- The trick: we only need $p_0q_1 + p_1q_0$, not p_0q_1 and p_1q_0 . Note that $p_0q_1 + p_1q_0 = (p_0 + p_1)(q_0 + q_1) - p_0q_0 - p_1q_1$. So we only need 3 multiplications of half size polynomials!
- Recurrence looks like $T(n) \leq 3T(n/2) + g(n)$, with $g(n) \in \Theta(n)$. Solution $T(n) \in O(n^{\lg 3})$, $\lg 3 = 1.59\dots$, provided we can deal with odd length numbers!
- Note that if we replace $x^{n/2}$ by a variable t , then $r = a_0 + a_1t + a_2t^2$. We can determine r by evaluating at any 3 points, (interpolation). This generalizes (see assignment).
- If $m < n$, use distributivity to break q into blocks of degree m and multiply each by p as above, give $O(nm^{\lg 3/2})$ running time.
- Note: the Fast Fourier Transform does multiplication in $O(n \log n)$ time. It is also a D & C algorithm.

D & C for order statistics

- Given an array T of size n and integer $s \in 1 \dots n$, find the s -th smallest element. If $s = \lceil n/2 \rceil$, this is the *median*.

D & C for order statistics

- Given an array T of size n and integer $s \in 1 \dots n$, find the s -th smallest element. If $s = \lceil n/2 \rceil$, this is the *median*.
- Use quicksort approach: pivot around element p , partitioning T into elements that are $\leq p$, $= p$, $\geq p$. Desired element lies in one of these; it is trivial to find which one (as for binary search).

D & C for order statistics

- Given an array T of size n and integer $s \in 1 \dots n$, find the s -th smallest element. If $s = \lceil n/2 \rceil$, this is the *median*.
- Use quicksort approach: pivot around element p , partitioning T into elements that are $\leq p$, $= p$, $\geq p$. Desired element lies in one of these; it is trivial to find which one (as for binary search).
- Problem: if we use a fixed choice for pivot, then worst case, as with quicksort, is quadratic, as subproblems can be very unbalanced. How to choose pivot? Need good approximation to median.

D & C for order statistics

- Given an array T of size n and integer $s \in 1 \dots n$, find the s -th smallest element. If $s = \lceil n/2 \rceil$, this is the *median*.
- Use quicksort approach: pivot around element p , partitioning T into elements that are $\leq p$, $= p$, $\geq p$. Desired element lies in one of these; it is trivial to find which one (as for binary search).
- Problem: if we use a fixed choice for pivot, then worst case, as with quicksort, is quadratic, as subproblems can be very unbalanced. How to choose pivot? Need good approximation to median.
- One idea: divide T into subarrays $Z[i]$, $1 \leq i \leq z := \lceil n/5 \rceil$ of fixed size, say 5. Form the median of each sample, and then the median of these. This is the median.

Analysis of order statistics algorithm

- Time required to find pivot is $t(\lfloor n/5 \rfloor) + O(n)$. Pivoting step also in $O(n)$.

Analysis of order statistics algorithm

- Time required to find pivot is $t(\lfloor n/5 \rfloor) + O(n)$. Pivoting step also in $O(n)$.
- At least 3 elements of each $Z[i]$ are \leq its median. At least $\lceil z/2 \rceil$ of these medians are $\leq p$. So at least $3z/2$ elements of T are $\leq p$.

Analysis of order statistics algorithm

- Time required to find pivot is $t(\lfloor n/5 \rfloor) + O(n)$. Pivoting step also in $O(n)$.
- At least 3 elements of each $Z[i]$ are \leq its median. At least $\lceil z/2 \rceil$ of these medians are $\leq p$. So at least $3z/2$ elements of T are $\leq p$.
- So subproblem sizes are somewhat balanced. In worst case we get, for example, $t(n) \leq dn + t(\lceil n/5 \rceil) + t(7n/10)$. Can prove by constructive induction that $t(n) \in O(n)$.

Modular exponentiation

- We wish to compute $a^n \bmod z$, where $a \geq 0, z \geq 1$ are fixed integers and n is large. Obvious iterative method is too slow.

Modular exponentiation

- We wish to compute $a^n \bmod z$, where $a \geq 0, z \geq 1$ are fixed integers and n is large. Obvious iterative method is too slow.
- Divide and conquer: $a^n = (a^{n/2})^2$. Let $N(n)$ be the number of multiplications. For n even get $N(n) = N(n/2) + 1$. Otherwise $N(n) = N(n-1) + 1$ so $N(\lfloor n/2 \rfloor) + 1 \leq N(n) \leq N(\lfloor n/2 \rfloor) + 2$. Smoothness rule yields $N(n) \in \Theta(\log(n))$.

Modular exponentiation

- We wish to compute $a^n \bmod z$, where $a \geq 0, z \geq 1$ are fixed integers and n is large. Obvious iterative method is too slow.
- Divide and conquer: $a^n = (a^{n/2})^2$. Let $N(n)$ be the number of multiplications. For n even get $N(n) = N(n/2) + 1$. Otherwise $N(n) = N(n-1) + 1$ so $N(\lfloor n/2 \rfloor) + 1 \leq N(n) \leq N(\lfloor n/2 \rfloor) + 2$. Smoothness rule yields $N(n) \in \Theta(\log(n))$.
- Useful for **public key cryptography**: send message $c := a^n \bmod z$, eavesdropper knows z, n, c , but no known fast algorithm for finding a ("finding n th root").

The Fast Fourier Transform