Dynamic programming
COMS20010 (Algorithms II)

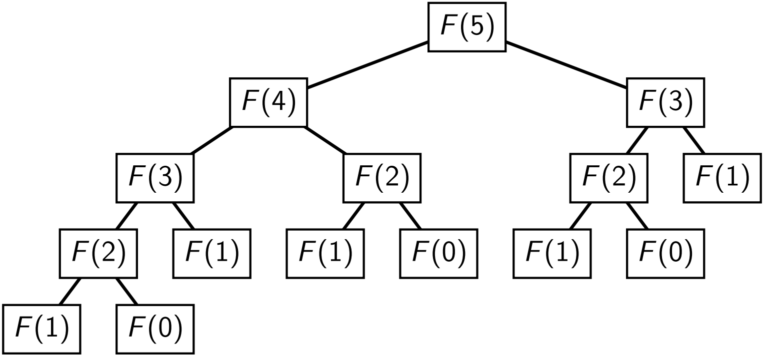
John Lapinskas, University of Bristol

Reminder from COMS10007: The Fibonacci sequence

The **Fibonacci sequence** is given by

$$F(0) = 0; \qquad F(1) = 1; \qquad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2.$$

Trying to use this recurrence to calculate it directly takes $\Theta(\phi^n)$ time:

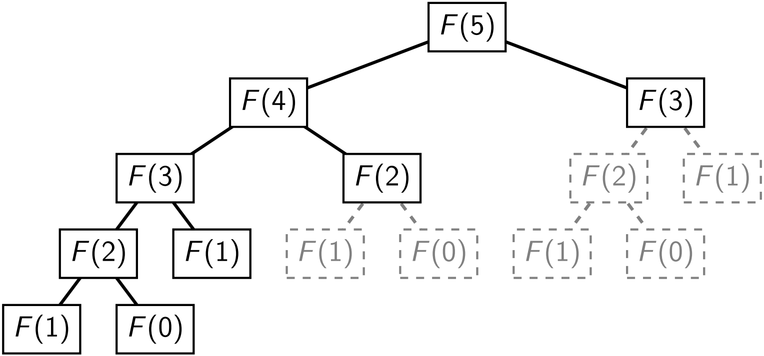


Reminder from COMS10007: The Fibonacci sequence

The **Fibonacci sequence** is given by

$$F(0) = 0; \qquad F(1) = 1; \qquad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2.$$

Trying to use this recurrence to calculate it directly takes $\Theta(\phi^n)$ time:



But if we remember the results of each F call, it takes only $\Theta(n)$ time!

Reminder from COMS10007: Memoisation

We can do this literally, e.g. via static variables or a cache argument. This is called **memoization**, and any language can do it. E.g. for Python:

```
def fibonacci(n):
    if n in fibonacci.cache:
        return fibonacci.cache[n]
    fibonacci.cache[n] = fibonacci(n-1) + fibonacci(n-2)
    return fibonacci.cache[n]
fibonacci.cache = {0:0, 1:1}
```

Alternatively, and **optionally**, we can "unroll the recurrence" into an iterative algorithm that fills out the cache from the bottom up:

```
def fibonacci(n):
    cache = [0,1]+[-1]*(n-1)
    for i in range(2, n+1):
        cache[i] = cache[i-1] + cache[i-2]
    return cache[n]
```

0	1	-1	-1	-1	-1

Reminder from COMS10007: Memoisation

We can do this literally, e.g. via static variables or a cache argument. This is called **memoization**, and any language can do it. E.g. for Python:

```
def fibonacci(n):
    if n in fibonacci.cache:
        return fibonacci.cache[n]
    fibonacci.cache[n] = fibonacci(n-1) + fibonacci(n-2)
    return fibonacci.cache[n]
fibonacci.cache = {0:0, 1:1}
```

Alternatively, and **optionally**, we can "unroll the recurrence" into an iterative algorithm that fills out the cache from the bottom up:

```
def fibonacci(n):
    cache = [0,1]+[-1]*(n-1)
    for i in range(2, n+1):
        cache[i] = cache[i-1] + cache[i-2]
    return cache[n]
```

0	1	1	-1	-1	-1

Reminder from COMS10007: Memoisation

We can do this literally, e.g. via static variables or a cache argument. This is called **memoization**, and any language can do it. E.g. for Python:

```
def fibonacci(n):
    if n in fibonacci.cache:
        return fibonacci.cache[n]
    fibonacci.cache[n] = fibonacci(n-1) + fibonacci(n-2)
    return fibonacci.cache[n]
fibonacci.cache = {0:0, 1:1}
```

Alternatively, and **optionally**, we can "unroll the recurrence" into an iterative algorithm that fills out the cache from the bottom up:

```
def fibonacci(n):
    cache = [0,1]+[-1]*(n-1)
    for i in range(2, n+1):
        cache[i] = cache[i-1] + cache[i-2]
    return cache[n]
```

0	1	1	2	-1	-1

Reminder from COMS10007: Memoisation

We can do this literally, e.g. via static variables or a cache argument. This is called **memoization**, and any language can do it. E.g. for Python:

```python
def fibonacci(n):
    if n in fibonacci.cache:
        return fibonacci.cache[n]
    fibonacci.cache[n] = fibonacci(n-1) + fibonacci(n-2)
    return fibonacci.cache[n]
fibonacci.cache = {0:0, 1:1}
```

Alternatively, and **optionally**, we can “unroll the recurrence” into an iterative algorithm that fills out the cache from the bottom up:

```python
def fibonacci(n):
    cache = [0,1]+[-1]*(n-1)
    for i in range(2, n+1):
        cache[i] = cache[i-1] + cache[i-2]
    return cache[n]
```

0	1	1	2	3	5

⟵ Return cache[5].

Either way, we turn a $\Theta(\phi^n)$-time algorithm for calculating F_n into a $\Theta(n)$-time algorithm. This technique is called **dynamic programming**.

Dynamic programming for weighted interval scheduling

In weighted interval scheduling, we have a slow recursive algorithm:

- Pick an arbitrary interval I;
- Recursively find the best schedule containing I;
- Recursively find the best schedule not containing I;
- Return whichever is better.

But almost every recursive call will be different. Memoisation doesn't help.

So we need to choose I in such a way as to **make** almost all the recursive calls the same!

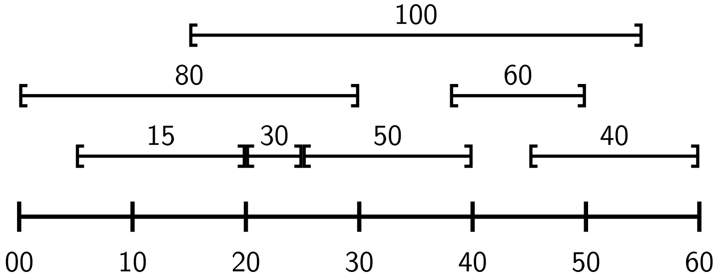
If our recursive algorithm is built around "try all possible options of a choice", like "is I in the schedule or not?" then one trick is to impose an order on the choices so that each choice only has a "local" effect.

Here, if we take I to be the interval with the latest finish time, rather than choosing it arbitrarily, things will work out nicely!

Why "fastest-finishing" works fast

Key point: Say our intervals are $\mathcal{R} = \{(s_1, f_1), \ldots, (s_n, f_n)\}$, where $f_1 \leq \cdots \leq f_n$. Then the slowest-finishing interval (s_n, f_n) only overlaps with intervals finishing later than s_n.

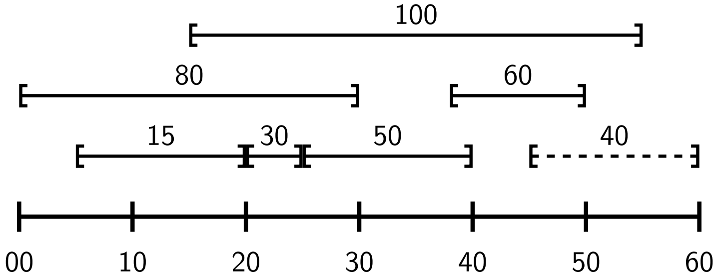
So our recursive calls always take $\mathcal{R} = \{(s_1, f_1), \ldots, (s_i, f_i)\}$ for some i!



Why "fastest-finishing" works fast

Key point: Say our intervals are $\mathcal{R} = \{(s_1, f_1), \dots, (s_n, f_n)\}$, where $f_1 \le \cdots \le f_n$. Then the slowest-finishing interval (s_n, f_n) only overlaps with intervals finishing later than s_n.

So our recursive calls always take $\mathcal{R} = \{(s_1, f_1), \dots, (s_i, f_i)\}$ for some i!

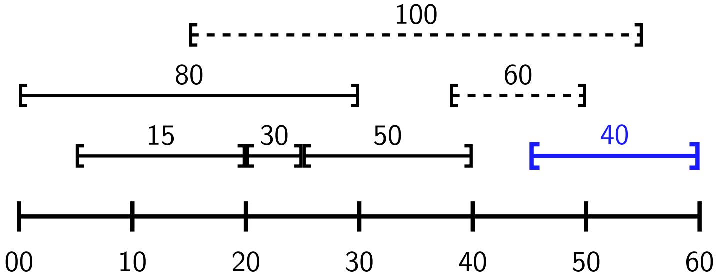


(45, 60) not in schedule: Recurse on $(5, 20), (20, 25), \dots, (15, 55)$.

Why "fastest-finishing" works fast

Key point: Say our intervals are $\mathcal{R} = \{(s_1, f_1), \ldots, (s_n, f_n)\}$, where $f_1 \leq \cdots \leq f_n$. Then the slowest-finishing interval (s_n, f_n) only overlaps with intervals finishing later than s_n.

So our recursive calls always take $\mathcal{R} = \{(s_1, f_1), \ldots, (s_i, f_i)\}$ for some i!



(45, 60) is in schedule: Recurse on $(5, 20), (20, 25), \ldots, (25, 40)$.

Choosing I to be the fastest-starting interval works too — see quiz!

The recursive (memoised) version

Algorithm: WIS

Input : A **sorted** array $\mathcal{R}$ of n requests and a weight function w.
Output : A maximum-weight compatible subset of $\mathcal{R}$.

begin
  Write $\mathcal{R} = (s_1, f_1), \dots, (s_n, f_n)$ with $f_1 \leq \dots \leq f_n$.
  if $\mathcal{R} = \emptyset$ **then**
    Return $\emptyset$.
  else if $\mathcal{R}$ *is in* cache **then**
    Return cache$[\mathcal{R}]$.
  else
    Let $X \leftarrow \{(s_i, f_i) \colon f_i > s_n\}$ be the set of intervals in $\mathcal{R}$ incompatible with (s_n, f_n).
    $S_{\text{out}} \leftarrow \text{WIS}(\mathcal{R} \setminus \{(s_n, f_n)\}, w)$.
    $S_{\text{in}} \leftarrow \{I\} \cup \text{WIS}(\mathcal{R} \setminus (\{(s_n, f_n)\} \cup X), w)$.
    if $w(S_{\text{out}}) > w(S_{\text{in}})$ **then** output $\leftarrow S_{\text{out}}$, **else** output $\leftarrow S_{\text{in}}$.
    cache$[\mathcal{R}] \leftarrow$ output.
    Return output.

Here cache is a static dictionary. Any sensible implementation (e.g. a hash table) will take $O(\log n)$ time or $O(1)$ time per access. We can find X in $O(\log n)$ time with binary search. So each call takes $O(\log n)$ time.

The recursive (memoised) version

Algorithm: WIS

Input : A **sorted** array $\mathcal{R}$ of n requests and a weight function w.
Output : A maximum-weight compatible subset of $\mathcal{R}$.

begin
    Write $\mathcal{R} = (s_1, f_1), \dots, (s_n, f_n)$ with $f_1 \leq \cdots \leq f_n$.
    if $\mathcal{R} = \emptyset$ **then**
        Return $\emptyset$.
    else if $\mathcal{R}$ *is in* `cache` **then**
        Return `cache`$[\mathcal{R}]$.
    else
        Let $X \leftarrow \{(s_i, f_i) \colon f_i > s_n\}$ be the set of intervals in $\mathcal{R}$ incompatible with (s_n, f_n).
        $S_{\mathrm{out}} \leftarrow \mathrm{WIS}(\mathcal{R} \setminus \{(s_n, f_n)\}, w)$.
        $S_{\mathrm{in}} \leftarrow \{I\} \cup \mathrm{WIS}(\mathcal{R} \setminus (\{(s_n, f_n)\} \cup X), w)$.
        if $w(S_{\mathrm{out}}) > w(S_{\mathrm{in}})$ **then** `output` $\leftarrow S_{\mathrm{out}}$, **else** `output` $\leftarrow S_{\mathrm{in}}$.
        `cache`$[\mathcal{R}] \leftarrow$ `output`.
        Return `output`.

Each call takes $O(\log n)$ time, and there are $O(n)$ total calls, for a total of $O(n \log n)$ time. We also need to sort $\mathcal{R}$ before calling WIS for the first time, which takes $O(n \log n)$ time.

The recursive (memoised) version

Algorithm: WIS

Input : A **sorted** array $\mathcal{R}$ of n requests and a weight function w.
Output : A maximum-weight compatible subset of $\mathcal{R}$.

begin
- Write $\mathcal{R} = (s_1, f_1), \dots, (s_n, f_n)$ with $f_1 \leq \cdots \leq f_n$.
- **if** $\mathcal{R} = \emptyset$ **then**
  - Return $\emptyset$.
- **else if** $\mathcal{R}$ *is in* `cache` **then**
  - Return `cache`$[\mathcal{R}]$.
- **else**
  - Let $X \leftarrow \{(s_i, f_i) \colon f_i > s_n\}$ be the set of intervals in $\mathcal{R}$ incompatible with (s_n, f_n).
  - $S_{\mathrm{out}} \leftarrow \mathrm{WIS}(\mathcal{R} \setminus \{(s_n, f_n)\}, w)$.
  - $S_{\mathrm{in}} \leftarrow \{I\} \cup \mathrm{WIS}(\mathcal{R} \setminus (\{(s_n, f_n)\} \cup X), w)$.
  - **if** $w(S_{\mathrm{out}}) > w(S_{\mathrm{in}})$ **then** `output` $\leftarrow S_{\mathrm{out}}$, **else** `output` $\leftarrow S_{\mathrm{in}}$.
  - `cache`$[\mathcal{R}] \leftarrow$ `output`.
  - Return `output`.

So overall, the running time is $O(n \log n)$!

The iterative version

Algorithm: WIS

Input : An **unsorted** array $\mathcal{R}$ of n requests and a weight function w.
Output : A maximum-weight compatible subset of $\mathcal{R}$.

begin
  Sort $\mathcal{R} \leftarrow (s_1, f_1), \dots, (s_n, f_n)$ with $f_1 \le \cdots \le f_n$.
  $\texttt{cache} \leftarrow [\texttt{Null}] \times (n+1)$.
  $\texttt{cache}[0] \leftarrow \emptyset$.
  for $i = 1$ *to* n **do**
    Let $p(i) \leftarrow \max\{\{0\} \cup \{1 \le j \le i-1 \colon f_j \le s_i\}\}$.
    $S_{\mathrm{out}} \leftarrow \texttt{cache}[i-1]$.
    $S_{\mathrm{in}} \leftarrow \texttt{cache}[p(i)] \cup \{(s_i, f_i)\}$.
    if $w(S_{\mathrm{out}}) > w(S_{\mathrm{in}})$ **then** $\texttt{cache}[i] \leftarrow S_{\mathrm{out}}$, **else** $\texttt{cache}[i] \leftarrow S_{\mathrm{in}}$.
  Return $\texttt{cache}[n]$.

This algorithm is doing the same thing as the recursive algorithm, working from the base case $\mathcal{R} = \emptyset$ (corresponding to $\texttt{cache}[0]$) upwards.

Again, we can find $p(i)$ in $O(\log n)$ time with binary search, so the overall running time is $O(n \log n)$ — the same as the recursive version!

The iterative version

Algorithm: WIS

Input : An **unsorted** array $\mathcal{R}$ of n requests and a weight function w.

Output : A maximum-weight compatible subset of $\mathcal{R}$.

begin

- Sort $\mathcal{R} \leftarrow (s_1, f_1), \dots, (s_n, f_n)$ with $f_1 \le \dots \le f_n$.
- $\texttt{cache} \leftarrow [\texttt{Null}] \times (n+1)$.
- $\texttt{cache}[0] \leftarrow \emptyset$.
- **for** $i = 1$ *to* n **do**
    - Let $p(i) \leftarrow \max\{\{0\} \cup \{1 \le j \le i-1 \colon f_j \le s_i\}\}$.
    - $S_{\mathrm{out}} \leftarrow \texttt{cache}[i-1]$.
    - $S_{\mathrm{in}} \leftarrow \texttt{cache}[p(i)] \cup \{(s_i, f_i)\}$.
    - **if** $w(S_{\mathrm{out}}) > w(S_{\mathrm{in}})$ **then** $\texttt{cache}[i] \leftarrow S_{\mathrm{out}}$, **else** $\texttt{cache}[i] \leftarrow S_{\mathrm{in}}$.
- Return $\texttt{cache}[n]$.

It's generally good practice to make your dynamic programming algorithms iterative, since it often has lower constant overhead, and it can help you identify more significant savings. (See video 11-4!) But it is **not** necessary.

Unless you already know it's a performance bottleneck, do whichever you find easiest — premature optimisation creates bugs!