Optimization without Gradients: Powell's method

Slides adapted from **Numerical Recipes in C, Second Edition (1992)**

Minimization in one direction

- Assume that we know how to minimize a function of one variable. If we start at a point $\boldsymbol{p}$ in N-dimensional space, and proceed from there in some vector direction $\boldsymbol{n}$, then any function of N variables $f(\boldsymbol{p})$ can be minimized along the line $\boldsymbol{n}$ by our one-dimensional methods.

- Different methods will differ only by how they choose the next direction $\boldsymbol{n}$ to try.

- The line minimization routine `linmin` is a black-box subalgorithm, whose definition is

  `linmin:` Given as input the vectors $\boldsymbol{p}$ and $\boldsymbol{n}$, and the function f, find the scalar λ that minimizes $f(\boldsymbol{p}+\lambda\boldsymbol{n})$. Replace $\boldsymbol{p}$ by $\boldsymbol{p}+\lambda\boldsymbol{n}$. Replace $\boldsymbol{n}$ by $\lambda\boldsymbol{n}$. Done.

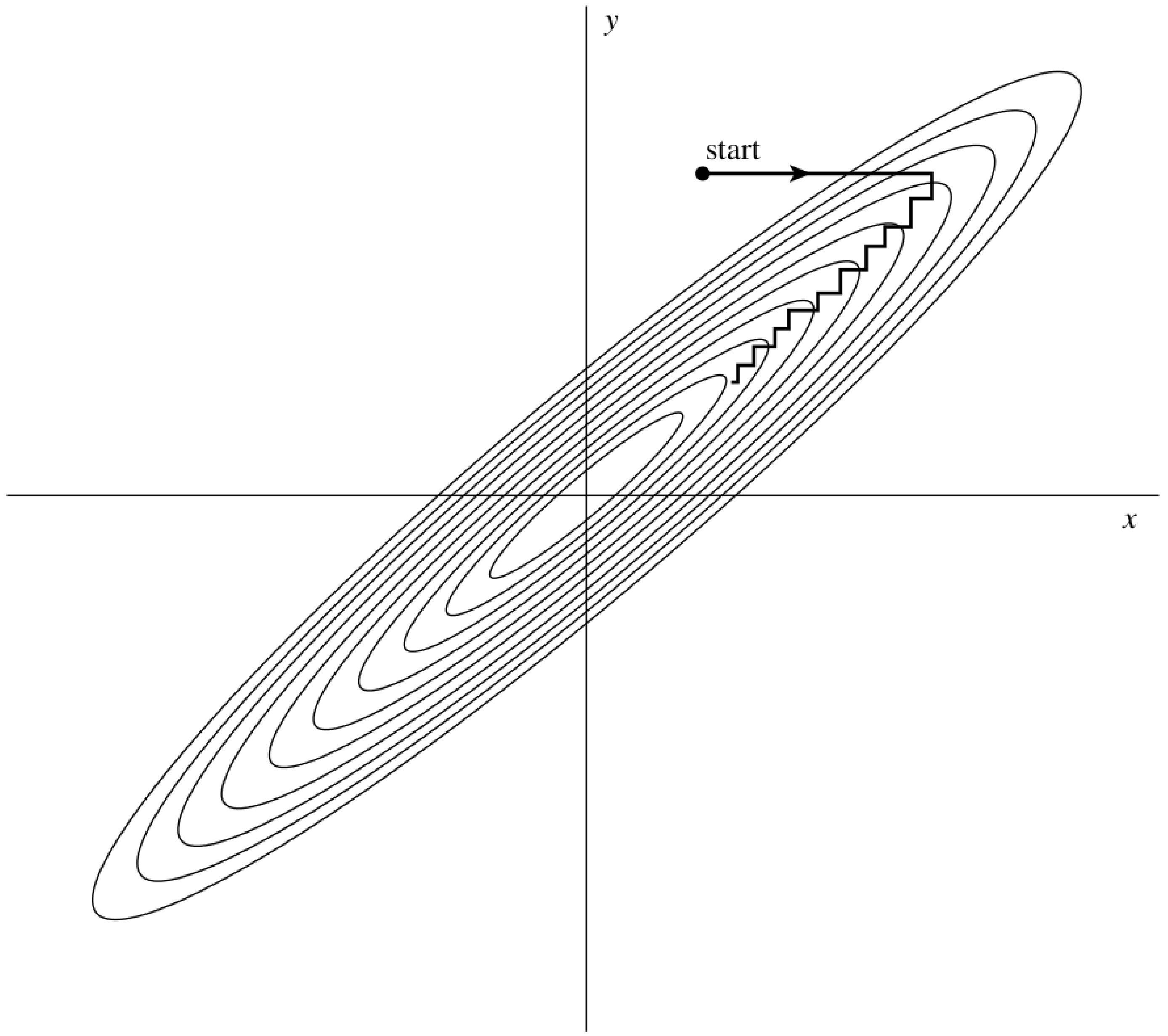
A simple method for general minimization

- Take the unit vectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_N$ as a set of directions. Using `linmin`, move along the first direction to its minimum, then from there along the second direction to its minimum, and so on, cycling through the whole set of directions as many times as necessary, until the function stops decreasing.

- This simple method is actually not too bad for many functions.

- Even more interesting is why it **is** bad, i.e. very inefficient, for some other functions.

A simple method for general minimization



Direction set methods

- Obviously what we need is a better set of directions than the e_i's. All direction set methods consist of prescriptions for updating the set of directions as the method proceeds, attempting to come up with a set which either
  - includes some very good directions that will take us far along narrow valleys, or else (more subtly)
  - includes some number of "non-interfering" directions with the special property that minimization along one is not "spoiled" by subsequent minimization along another, so that interminable cycling through the set of directions can be avoided.

Conjugate Directions

- This concept of "non-interfering" directions, called **conjugate directions**, can be made mathematically explicit.
- Assume f is differentiable. If we minimize f along the direction $\boldsymbol{u}$, then the gradient must be perpendicular to $\boldsymbol{u}$ at the line minimum.
- Take some particular point $\boldsymbol{p}$ as the origin of the coordinate system. Then any function f can be approximated by its Taylor series

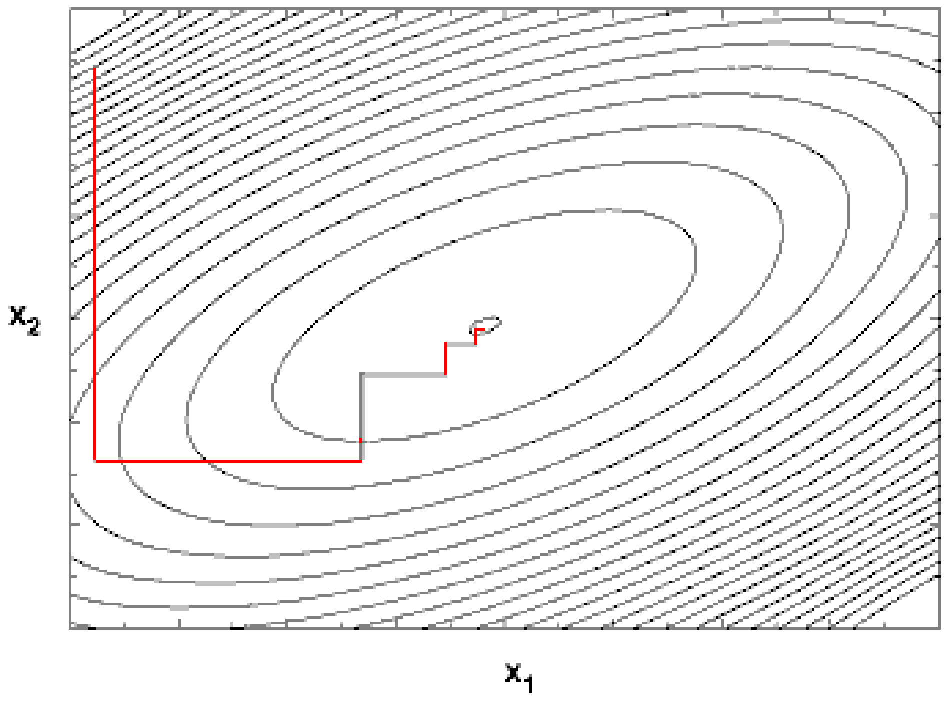
$$\begin{aligned} f(\boldsymbol{x}) &= f(\boldsymbol{p}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \dots \\ &\approx c - \boldsymbol{b}^t \boldsymbol{x} + \frac{1}{2} \boldsymbol{x}^t A \boldsymbol{x} \end{aligned}$$

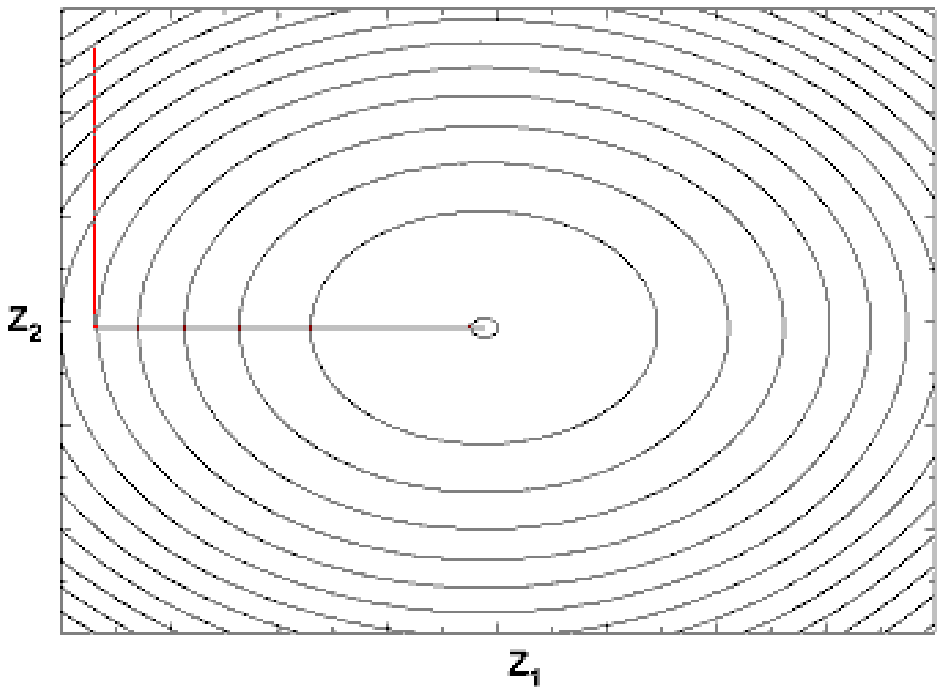
where $c = f(\mathbf{p})$, $\boldsymbol{b} = -\nabla f|_{\boldsymbol{p}}$, $A =$ Hessian matrix.

Conjugate Directions

- In the approximation, the gradient is easily calculated as $\nabla f = A\boldsymbol{x} - \boldsymbol{b}$.
- This implies that the gradient will vanish - the function will be at an extremum - at a value of $\boldsymbol{x}$ obtained by solving $A\boldsymbol{x} = \boldsymbol{b}$.
- How does the gradient f change as we move along some direction $\boldsymbol{v}$? $\boldsymbol{\delta}(\nabla f) = A\boldsymbol{v}$.
- Suppose that we have moved along some direction $\boldsymbol{u}$ to a minimum and now propose to move along some new direction $\boldsymbol{v}$. The condition that motion along $\boldsymbol{v}$ not spoil our minimization along $\boldsymbol{u}$ is just that the gradient stay perpendicular to $\boldsymbol{u}$, i.e., that the change in the gradient be perpendicular to $\boldsymbol{u}$: $0 = \boldsymbol{u}^t \boldsymbol{\delta}(\nabla f) = \boldsymbol{u}^t A\boldsymbol{v}$
- If this holds for two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, they are said to be **conjugate**.

Conjugate Directions





The procedure of searching for an optimum holding all independent variables except one constant is not necessarily efficient (left panel).

It would be much more efficient to search along a set of directions that is conjugate to the objective function, as represented by Z_1, Z_2 in the figure on the right.

Conjugate Sets

- When the relation holds pairwise for all members of a set of vectors, they are said to be a **conjugate set**.
- If you do successive line minimization of a function along a conjugate set of directions, then you don't need to redo any of those directions.
- The idea behind a direction set method is to come up with a set of N linearly independent, mutually conjugate directions.
- Then, one pass of N line minimizations will put it exactly at the minimum of a quadratic form. For functions f that are not exactly quadratic forms, it won't be exactly at the minimum; but repeated cycles of N line minimizations will in due course converge quadratically to the minimum.

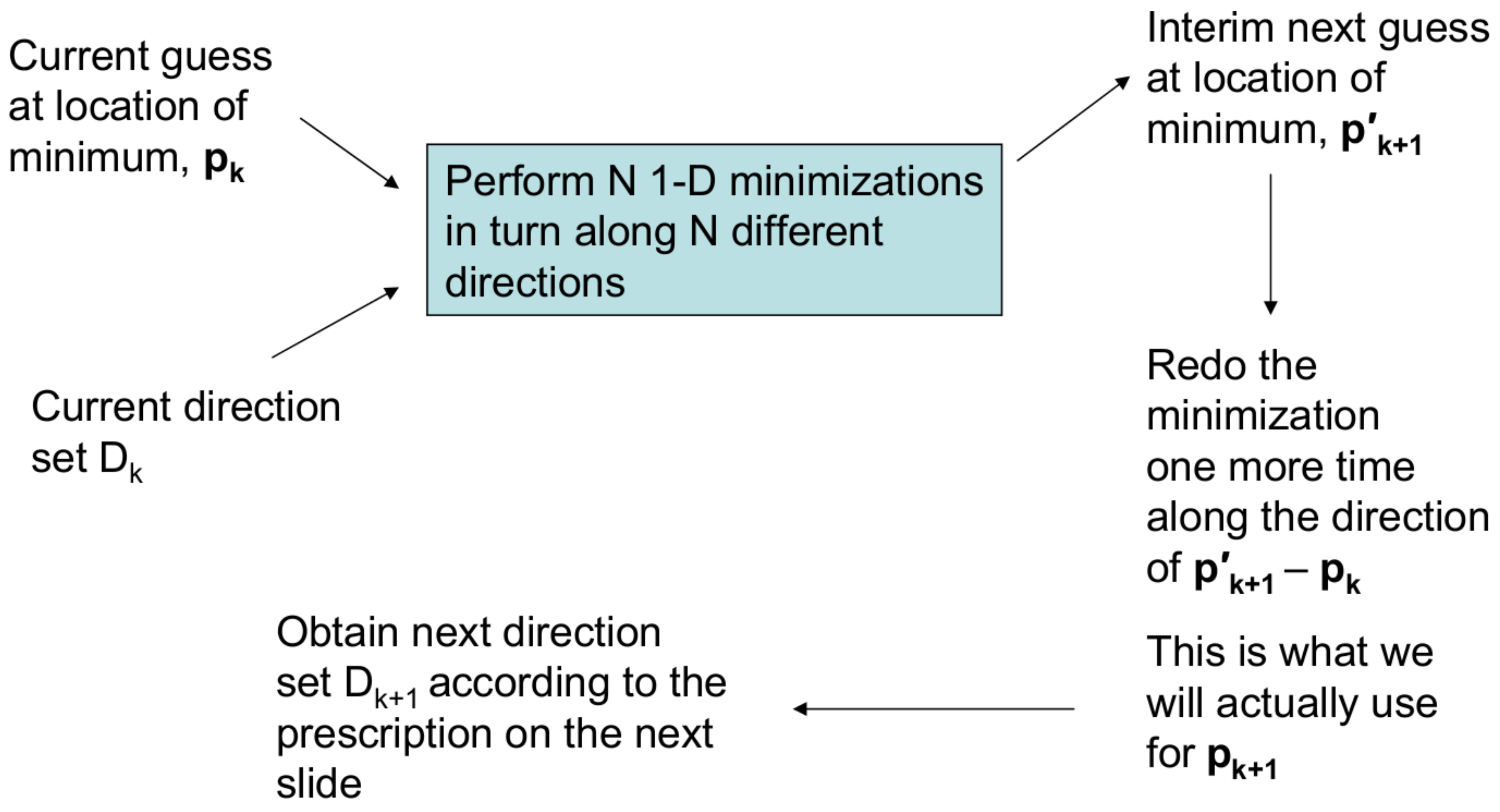
Powell's Method

Powell first discovered a direction set method that does produce N mutually conjugate directions (doesn't require us to know any derivatives!).

Initialize the set of directions $\boldsymbol{u}_i$ to the basis vectors, $\boldsymbol{u}_i = \boldsymbol{e}_i, i = 1, \ldots, N$
Now repeat the following sequence of steps until f stops decreasing:

- Save your starting position as $\boldsymbol{p}_0$.
- For $i = 1, \ldots, N$, move $\boldsymbol{p}_{i-1}$ to the minimum along direction $\boldsymbol{u}_i$ and call this point $\boldsymbol{p}_i$.
- For $i = 1, \ldots, N-1$, set $\boldsymbol{u}_i \leftarrow \boldsymbol{u}_{i+1}$.
- Set $\boldsymbol{u}_N \leftarrow \boldsymbol{p}_N - \boldsymbol{p}_0$.
- Move p_N to the minimum along direction $\boldsymbol{u}_N$ and call this point $\boldsymbol{p}_0$.

Powell's Method



Current guess at location of minimum, $\mathbf{p}_k$

Current direction set D_k

Perform N 1-D minimizations in turn along N different directions

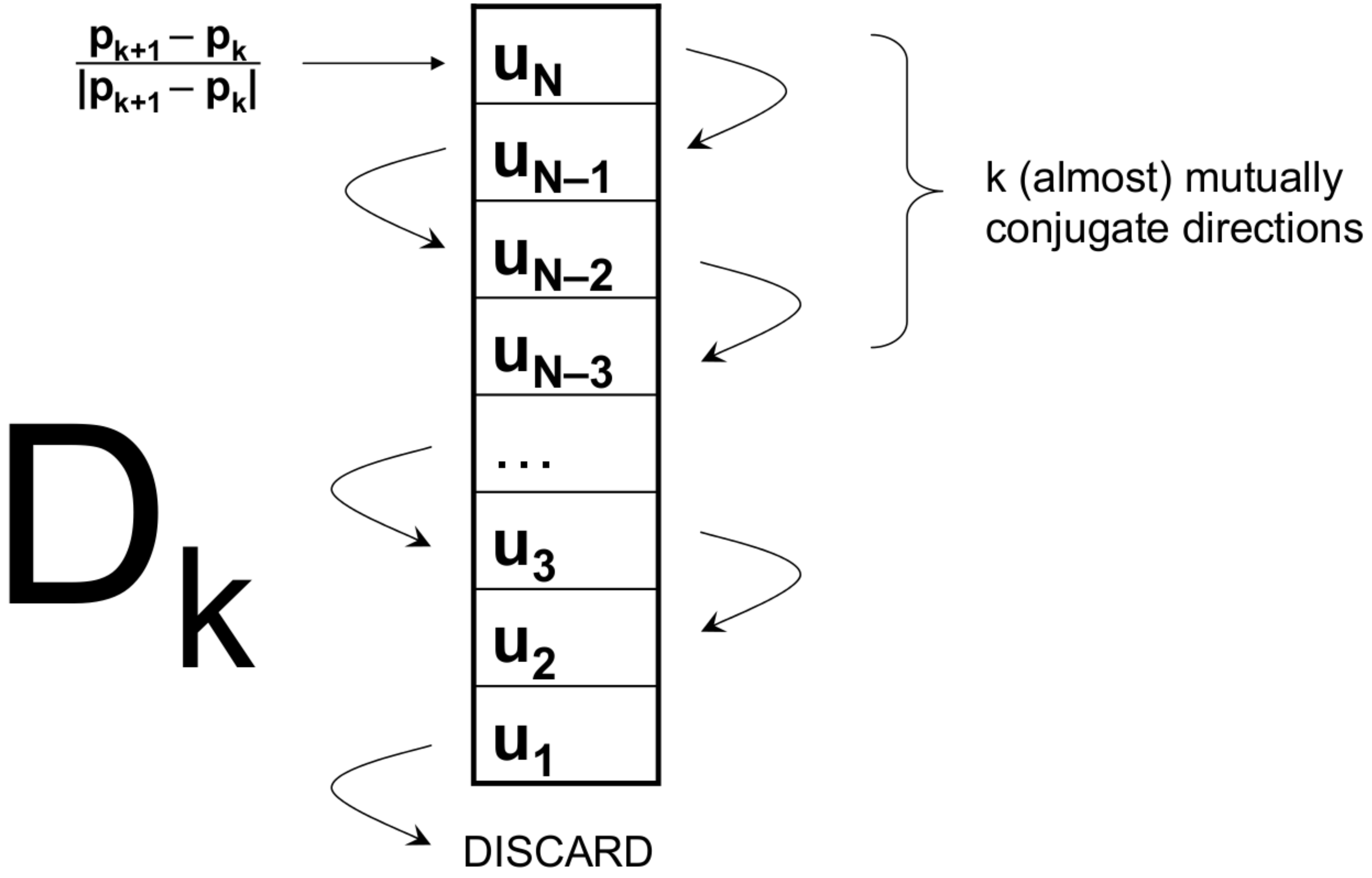
Interim next guess at location of minimum, $\mathbf{p}'_{k+1}$

Redo the minimization one more time along the direction of $\mathbf{p}'_{k+1} - \mathbf{p}_k$

This is what we will actually use for $\mathbf{p}_{k+1}$

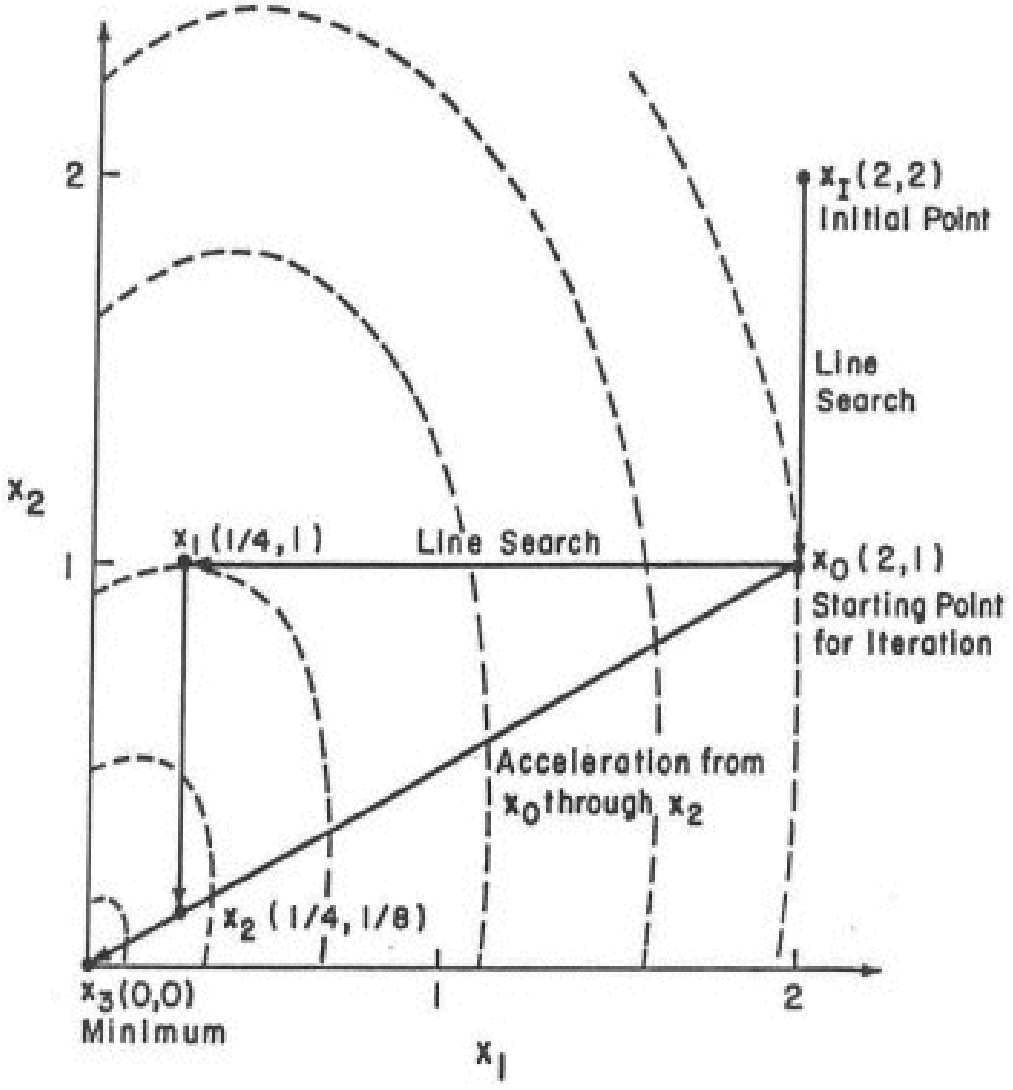
Obtain next direction set D_{k+1} according to the prescription on the next slide

Powell's Method



$\frac{p_{k+1} - p_k}{|p_{k+1} - p_k|}$
u_N
u_{N-1}
u_{N-2}
u_{N-3}
...
u_3
u_2
u_1
DISCARD
D_k
k (almost) mutually conjugate directions

Powell's Method



Powell's Method

- Powell (1964) showed that, for a quadratic form, k iterations of the above basic procedure produce a set of directions $\boldsymbol{u}_i$ whose last k members are mutually conjugate. Therefore, N iterations of the basic procedure, amounting to $N(N+1)$ line minimizations in all, will exactly minimize a quadratic form. [Brent, 1973] gives proofs of these statements in accessible form.

- Unfortunately, there is a problem with Powell's algorithm. The procedure of throwing away, at each stage, $\boldsymbol{u}_N$ in favor of $\boldsymbol{p}_N - \boldsymbol{p}_0$ tends to produce sets of directions that "fold up on each other" and become linearly dependent. Once this happens, then the procedure finds the minimum of the function f only over a subspace of the full N-dimensional case; in other words, it gives the wrong answer.

Powell's Method in Practice

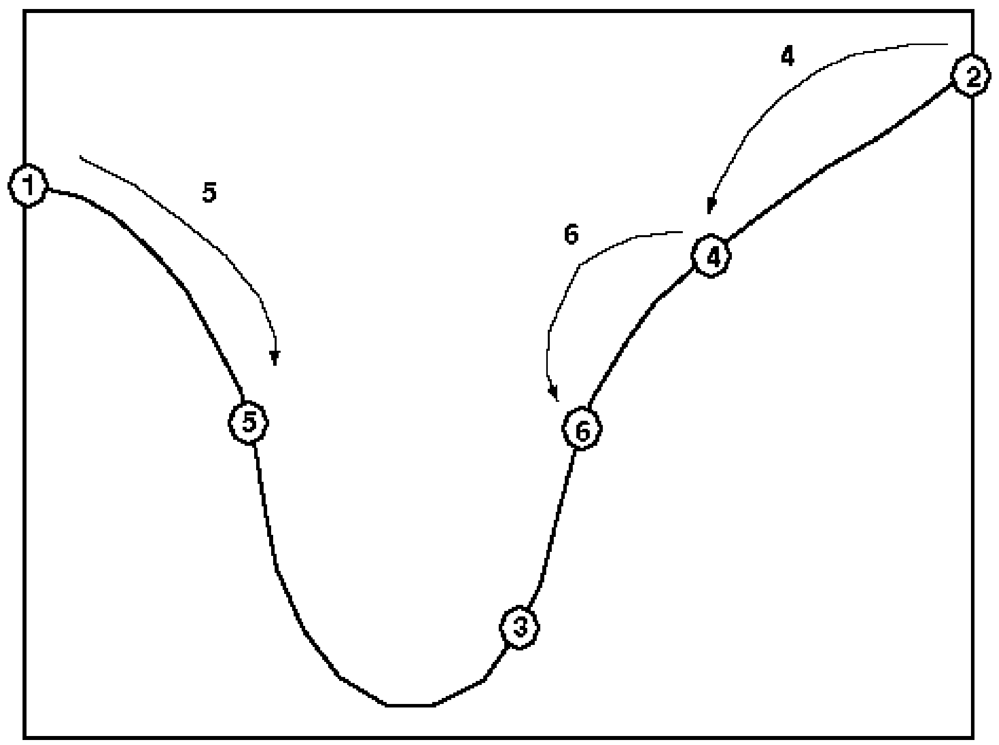
There are a number of ways to fix up the problem of linear dependence in Powell's algorithm, among them:

- You can reinitialize the set of directions $\boldsymbol{u}_i$ to the basis vectors $\boldsymbol{e}_i$ after every N or $N+1$ iterations of the basic procedure. This produces a serviceable method, which we commend to you if quadratic convergence is important for your application (i.e., if your functions are close to quadratic forms and if you desire high accuracy).

- You can give up the property of quadratic convergence in favor of a more heuristic scheme (due to Powell) which tries to find a few good directions along narrow valleys instead of N necessarily conjugate directions.

One-dimensional Minimization: Golden Section Search

Idea: successive bracketing of a minimum of a function f.



Minimum is originally bracketed by triplet $\{\mathbf{1},\mathbf{3},\mathbf{2}\}$. f is evaluated at $\mathbf{4}$, which replaces $\mathbf{2}$; than at $\mathbf{5} \rightsquigarrow$ replaces $\mathbf{1}$; then at $\mathbf{6} \rightsquigarrow$ replaces $\mathbf{4}$.

Rule: keep a center point that is lower than the two outside points.

Bracketing a Minimum

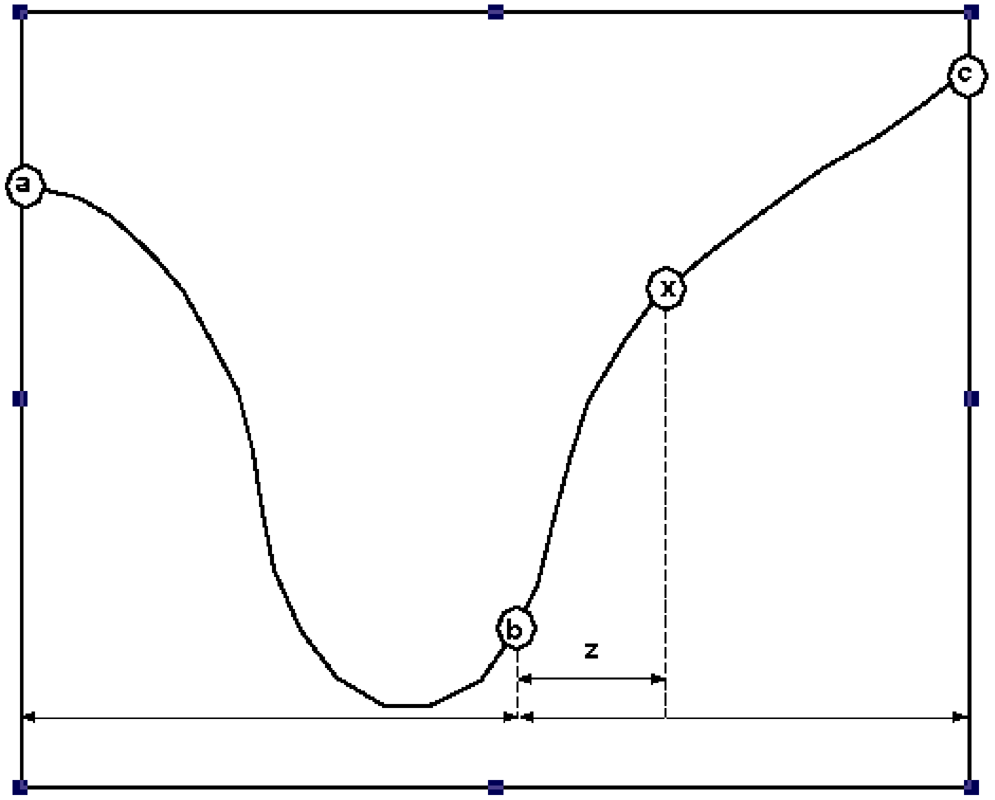
A minimum is bracketed when there is a triplet of points $a < b < c$ such that $f(c) > f(b) < f(a)$. In the Figure above: $\{a, b, c\} = \{\mathbf{1}, \mathbf{3}, \mathbf{2}\}$.
Successively improve this bracketing:

- Evaluate f at new x, between $\mathbf{1}$ and $\mathbf{3}$, or between $\mathbf{3}$ and $\mathbf{2}$.
- Choosing a point $\mathbf{4}$ between $\mathbf{3}$ and $\mathbf{2}$, we evaluate $f(4)$.
- $f(3) < f(4)$, so the new bracketing triplet of points is $\{\mathbf{1}, \mathbf{3}, \mathbf{4}\}$.
- Choose another point at $\mathbf{5}$, and evaluate $f(5)$.
- $f(3) < f(5)$, so the new bracketing triplet is $\{\mathbf{5}, \mathbf{3}, \mathbf{4}\}$.

Middle point of new triplet is the best minimum achieved so far.

Bracketing a Minimum

Important question: "How do we choose the new point x each time?"



Let b be some fraction w of the way between a and c:

$$w = \frac{b-a}{c-a}, \text{ and } 1 - w = \frac{c-b}{c-a}.$$

Bracketing a Minimum

- If x is some additional fraction z beyond b, then $z = \frac{x-b}{c-a}$.
- The next bracketing segment will either be of length $w + z$ relative to the current one, or of length $1 - w$.
- To minimize the worst case, choose z to make these equal: $z = 1 - 2w$.
- This gives us the symmetric point to b in the interval (a, c) i.e. $|b - a| = |x - c|$. This implies that x lies in the larger of the two segments ($z > 0$ only if $w < 1/2$).
- But where in the larger segment? Where did the value of w itself come from?

Bracketing a Minimum

- ...presumably from the previous stage of applying the same strategy: if z is optimal, then so was w.

- **Scale similarity:** z should be the same fraction of the way from b to c (if that is the bigger segment) as was b from a to c i.e. $w = \frac{z}{1-w}$.

- $w^2 - 3w + 1 = 0 \rightsquigarrow w \approx 0.38197$, and $1 - w \approx 0.61803$.

- Optimal interval has its middle point a fractional distance 0.38197 from one end, and 0.61803 from the other $\rightsquigarrow$ **golden section** fractions.

In each step, we select a point a fraction 0.38197 into the larger of the two intervals (from the central point of the triplet).

Chapter 5

Dynamic Programming

Dynamic Programming

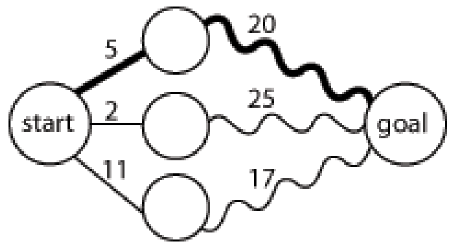
- **Simplex for LP:** Greedy algorithm, makes a **locally optimal** choice.
- For many problems we need a different approach called **Dynamic Programming**
- Finds effcient solutions for problems with lots of **overlapping sub-problems**. Essentially, we try to solve each sub-problem **only once**.
- The name is a little misleading, but comes historically from **Richard Bellman**, a professor of mathematics at Stanford University around 1950. In those days **programming** had a meaning closer to **planning**.

Dynamic Programming

The main ideas behind dynamic programming are:

- **Optimal substructure:** optimal solutions of **subproblems** can be used to find the optimal solutions of the **overall problem**.

  Example: shortest path can be found by first computing the shortest path from all adjacent vertices, and then picking the best overall path.

- **Overlapping subproblems:** Same subproblems are used to solve many different larger problems. A naive approach may **waste time** recomputing optimal solutions to subproblems.

- **Idea:** save the solutions to problems we have already solved...
  $\rightsquigarrow$ **Memoization.**

Dynamic Programming

Typically, a dynamic programming solution is constructed using a **series of steps:**

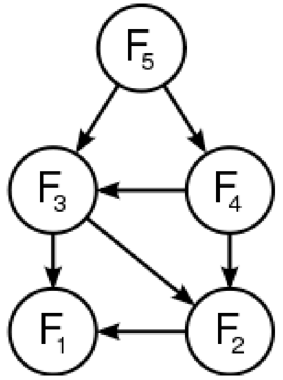
1. Characterise the **structure** of an optimal solution.
2. **Recursively** define the value of an **optimal solution**.
3. Compute the value of an optimal solution in a **bottom-up** or **top-down** fashion. That is, build it from the results of **smaller solutions** either **iteratively** from the bottom or **recursively** from the top.

A Simple Example: Fibonacci numbers

Fibonacci sequence: The n-th number is the sum of the previous two. This can be implemented using a simple recursive algorithm:

```
**function** FIBONACCI($n$)
    **if** $n = 0$ then
        **return** 0
    **if** $n = 1$ **then**
        **return** 1
    **return** FIBONACCI($n-1$) + FIBONACCI($n-2$)
```

Relation to dynamic programming: overlapping sub-problems, for example computing FIBONACCI($n-1$) over-laps FIBONACCI($n-2$).



A Simple Example (2)

Now, suppose we have a simple MAP object, m, which maps each value of FIBONACCI that has already been calculated to its result, and we modify our function to use it and update it.

The resulting function requires only $O(n)$ time:

```
**var** $m = \text{MAP}(0 \to 1, 1 \to 1)$
**function** FIBONACCI($n$)
    **if** $m$ does not contain key $n$
        $m[n] = \text{FIBONACCI}(n-1) + \text{FIBONACCI}(n-2)$
    **return** $m[n]$
```

A Simple Example (3)

We can make an array f and compute the n-th Fibonacci number in a bottom-up way:

```
*Pre-computation*
$f[0] = 0$
$f[1] = 1$
*Main Algorithm*
FIBONACCI($n$)
begin
    for $i = 2$ upto $n$ step $1$ do
        $f[i] = f[i-1] + f[i-2]$
    return $f[n]$
end
```

Recursion (top-down) $\Rightarrow$ **iteratation** (bottom-up)

Another Example: Optimal Binary Search Trees

Problem: Construction of **O**ptimal **B**inary **S**earch **T**rees.

BST: Tree where the key values are stored in the internal nodes, the external nodes (leaves) are null nodes, and the keys are ordered lexicographically.

For each internal node all keys in the left subtree are less than the keys in the node, and all the keys in the right subtree are greater.

Knowing the probabilities of searching each one of the keys makes it easy to compute the expected cost of accessing the tree.

An **OBST** is a BST with *minimal expected costs*.

OBST

- **Keys** $k_1, \ldots, k_n$ in lexicographical order,
- **Probabilities** of acessing keys $p_1, \ldots, p_n$.
- **Depth** $D_T(k_m)$ of node k_m in tree T. $D_T(\text{root}) = 0$
- T^{ij}: tree constructed from keys $k_i, \ldots, k_j$
- **Costs:** number of comparisons done in a search.
- **Expected costs:** expected number of comparisons done during search in tree, given the acess probabilities p_i

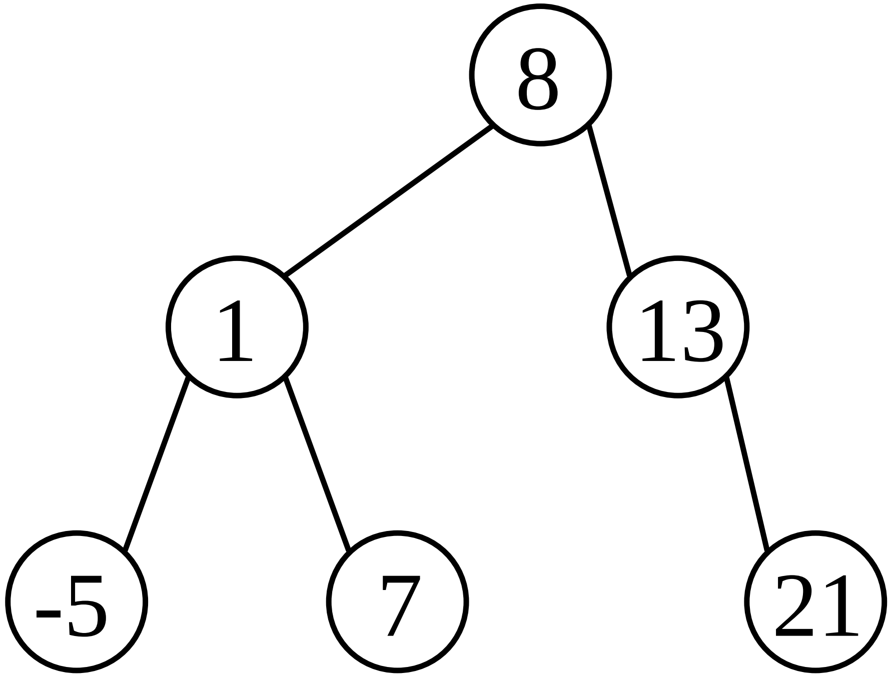
OBST: Expected costs

Definiton of expected costs of tree constructed from keys $k_i, \dots, k_j$:

$$
\begin{aligned}
C_{i,j} &:= E[cost(T^{ij})] \\
&= \sum_{\text{all keys in } T} \text{prob. of key} \times (\text{depth of key} \overbrace{+1}^{\text{one comparison for root}}) \\
&= \sum_{m=i}^{j} p_m (D_T(k_m) + 1)
\end{aligned}
$$

Example

Keys	-5	1	8	7	13	21
Probabilities	1/8	1/32	1/16	1/32	1/4	1/2

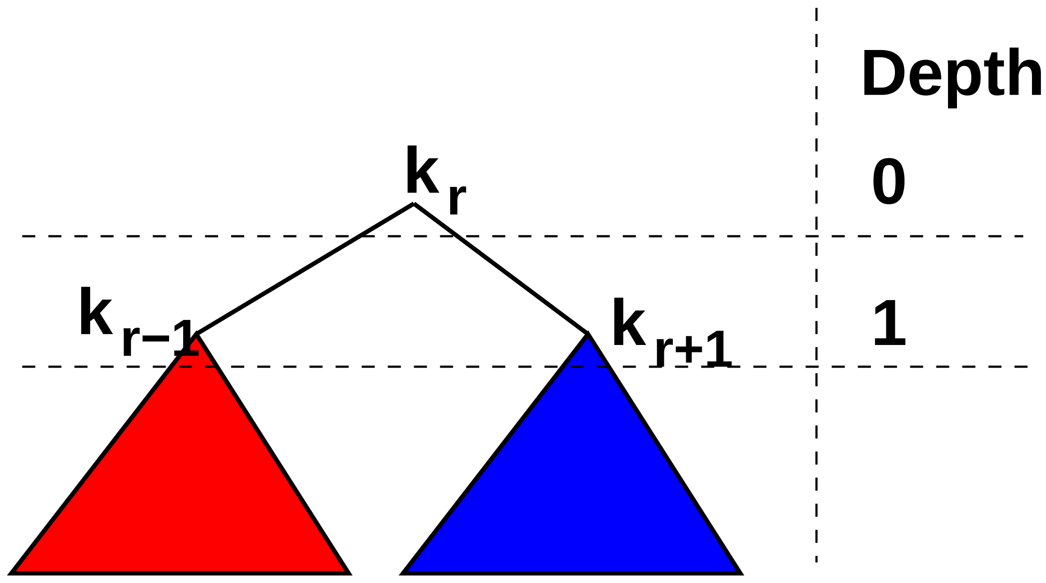


$E[cost] = 1 \cdot 1/16 + 2 \cdot (1/32 + 1/4) + 3 \cdot (1/8 + 1/32 + 1/2)$

Tree is **not optimal:** 21 closer to root $\Rightarrow$ lower $E[cost]$.

OBST

- **Key observation:** each subtree of an optimal tree is itself optimal (replacing a subtree with a better one lowers the costs of entire tree)
- Consider tree T^{ij} with root node k_r.



Expected costs of tree

$$
\begin{aligned}
C_{i,j} &= \sum_{m=i}^{j} p_m(D_T(k_m)+1) \\
&= \underbrace{\sum_{m=i}^{r-1} p_m(D_T(k_m)+1)}_{\text{left subtree}} + \underbrace{p_r}_{\text{root}} + \underbrace{\sum_{m=r+1}^{j} p_m(D_T(k_m)+1)}_{\text{right subtree}} \\
&= C(T_L) + \sum_{m=i}^{r-1} p_m + p_r + C(T_R) + \sum_{m=r+1}^{j} p_m \\
&= C(T_L) + C(T_R) + \sum_{m=i}^{j} p_m
\end{aligned}
$$

OBST: algorithm

Recursive algorithm:

- consider every node as being the root
- split rest of the keys into left and right subtrees and recursively calculate their costs.

$$C_{i,i} = p_i$$

$$C_{i,j} = 0 \, \forall \, j < i \quad \text{(tree with no nodes)}$$

$$C_{i,j} = \sum_{m=i}^{j} p_m + \min_{i \le r \le j} \left[C_{i,r-1} + C_{r+1,j} \right]$$

DP for an OBST

$$(\mathbf{C})_{ij} = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & \frac{1}{8} & \frac{3}{16} & \frac{9}{32} & \frac{15}{32} & \frac{31}{32} & \frac{63}{32} \\ 2 & 0 & \frac{1}{32} & \frac{3}{32} & \frac{7}{32} & \frac{19}{32} & \frac{47}{32} \\ 3 & & 0 & \frac{1}{32} & \frac{1}{8} & \frac{15}{32} & \frac{21}{16} \\ 4 & & & 0 & \frac{1}{16} & \frac{3}{8} & \frac{19}{16} \\ 5 & & & & 0 & \frac{1}{4} & 1 \\ 6 & & & & & 0 & \frac{1}{2} \end{array} = \frac{1}{32}\begin{pmatrix} 4 & 6 & 9 & 15 & 31 & 63 \\ & 1 & 3 & 7 & 19 & 47 \\ & & 1 & 4 & 15 & 42 \\ & & & 2 & 12 & 38 \\ & & & & 8 & 32 \\ & & & & & 16 \end{pmatrix}$$

Comments:

- The cost of the $OBST$ is in $C_{1,n}$ ($C_{1,6}$ in our example).
- It is practical to work with frequencies rather than with probabilities to avoid fractions as matrix-entries.